

Ingegneria del software

Decima edizione

Ian Sommerville

The background of the cover features a complex network of red dashed lines connecting numerous small, semi-transparent red circles of varying sizes. This pattern creates a sense of interconnectedness and data flow, typical of a software engineering or network analysis context.

MyLab Codice per accedere alla piattaforma



Pearson

INGEGNERIA DEL SOFTWARE

Decima edizione

INGEGNERIA DEL SOFTWARE

Decima edizione

Ian Sommerville



Pearson

© 2017 Pearson Italia, Milano – Torino

*Authorized translation from the English language edition, entitled: **SOFTWARE ENGINEERING, 10th Edition**, by IAN SOMMERVILLE published by Pearson Education, Inc, publishing as Pearson, Copyright © 2016.*

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Italian language edition published by Pearson Italia S.p.A., Copyright © 2017.

Per i passi antologici, per le citazioni, per le riproduzioni grafiche, cartografiche e fotografiche appartenenti alla proprietà di terzi, inseriti in quest'opera, l'editore è a disposizione degli aventi diritto non potuti reperire nonché per eventuali non volute omissioni e/o errori di attribuzione nei riferimenti.

È vietata la riproduzione, anche parziale o ad uso interno didattico, con qualsiasi mezzo, non autorizzata.

Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le fotocopie effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEAREDì, Centro Licenze e Autorizzazioni per le Riproduzioni Editoriali, Corso di Porta Romana 108, 20122 Milano, e-mail autorizzazioni@clearedi.org e sito web www.clearedi.org.

Traduzione e impaginazione: Carmelo Giarratana

Grafica di copertina: Maurizio Garofalo

Stampa: Tip.Le.Co – S. Bonico (PC)

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

9788891902245

Printed in Italy

1^a edizione: gennaio 2017

Ristampa	Anno
00 01 02 03 04	17 18 19 20 21

LIBRI DI TESTO E SUPPORTI DIDATTICI

Il sistema di gestione per la qualità della Casa Editrice è certificato in conformità alla norma UNI EN ISO 9001:2008 per l'attività di progettazione, realizzazione e commercializzazione di prodotti editoriali scolastici, lessicografici, universitari e di varia.



Sommario

Prefazione	XI
------------	----

Parte I Descrizione

Capitolo 1 Introduzione	3
1.1 Sviluppo di software professionale	5
1.2 Etica dell'ingegneria del software	15
1.3 Casi di studio	19
Punti chiave	28
Esercizi	29
Ulteriori letture	30
Capitolo 2 Processi software	31
2.1 Modelli dei processi software	33
2.2 Attività di processo	42
2.3 Far fronte ai cambiamenti	50
2.4 Miglioramento dei processi	55
Punti chiave	58
Esercizi	59
Ulteriori letture	60
Capitolo 3 Sviluppo agile del software	61
3.1 Metodi agili	64
3.2 Tecniche di sviluppo agile	66
3.3 Gestione agile della progettazione	75
3.4 Scalabilità dei metodi agili	79
Punti chiave	91
Esercizi	91
Ulteriori letture	92
Capitolo 4 Ingegneria dei requisiti	93
4.1 Requisiti funzionali e non funzionali	97
4.2 Processi di ingegneria dei requisiti	104

4.3 Deduzione dei requisiti	105
4.4 Specifica dei requisiti	114
4.5 Convalida dei requisiti	124
4.6 Modifica dei requisiti	126
Punti chiave	130
Esercizi	130
Ulteriori letture	131
Capitolo 5 Modelli di sistema	133
5.1 Modelli contestuali	136
5.2 Modelli di interazione	139
5.3 Modelli strutturali	145
5.4 Modelli comportamentali	150
5.5 Architettura guidata da modelli	156
Punti chiave	160
Esercizi	161
Ulteriori letture	162
Capitolo 6 Progettazione architettonurale	163
6.1 Decisioni di progettazione architettonurale	167
6.2 Viste architettonurali	169
6.3 Schemi architettonurali	172
6.4 Architetture applicative	182
Punti chiave	191
Esercizi	192
Ulteriori letture	193
Capitolo 7 Progettazione e implementazione	195
7.1 Progettazione orientata agli oggetti tramite UML	197
7.2 Design pattern (schemi di progettazione)	209
7.3 Problemi di implementazione	213
7.4 Sviluppo open-source	220
Punti chiave	224
Esercizi	225
Ulteriori letture	226
Capitolo 8 Test del software	227
8.1 Test di sviluppo	233
8.2 Sviluppo guidato da test	246
8.3 Test della release	248
8.4 Test degli utenti	253
Punti chiave	256
Esercizi	257
Ulteriori letture	258

Capitolo 9 Evoluzione del software	259
9.1 Processi evolutivi	263
9.2 Sistemi ereditati	267
9.3 Manutenzione del software	277
Punti chiave	288
Esercizi	289
Ulteriori letture	289

Parte II Fidatezza e protezione (online)

Capitolo e10 Sistemi fidati (online)	293
10.1 Proprietà della fidatezza	
10.2 Sistemi sociotecnici	
10.3 Ridondanza e diversità	
10.4 Processi fidati	
10.5 Metodi formali e fidatezza	
Capitolo e11 Ingegneria dell'affidabilità (online)	295
11.1 Disponibilità e affidabilità	
11.2 Requisiti di affidabilità	
11.3 Architetture fault-tolerant	
11.4 Programmare per l'affidabilità	
11.5 Misura dell'affidabilità	
Capitolo e12 Ingegneria della sicurezza (online)	297
12.1 Sistemi a sicurezza critica	
12.2 Requisiti di sicurezza	
12.3 Ingegneria della sicurezza	
12.4 Casi di sicurezza	
Capitolo e13 Ingegneria della protezione (online)	299
13.1 Protezione e fidatezza	
13.2 Protezione e organizzazioni	
13.3 Requisiti di protezione	
13.4 Progettare sistemi protetti	
13.5 Test e garanzia della protezione	
Capitolo e14 Ingegneria della resilienza (online)	301
14.1 Protezione informatica	
14.2 Resilienza socio-tecnica	
14.3 Progettare sistemi resilienti	

Parte III Ingegneria avanzata del software

Capitolo 15 Riutilizzo del software	305
15.1 Panoramica sul riutilizzo	307
15.2 Framework applicativi	312
15.3 Linee di prodotti software	316
15.4 Riutilizzo dei sistemi applicativi	323
Punti chiave	332
Esercizi	333
Ulteriori letture	334
Capitolo 16 Ingegneria del software basato sui componenti	335
16.1 Componenti e modelli di componenti	338
16.2 Processi CBSE	346
16.3 Composizione dei componenti	354
Punti chiave	361
Esercizi	362
Ulteriori letture	363
Capitolo 17 Ingegneria del software distribuito	365
17.1 Sistemi distribuiti	367
17.2 Calcolo client-server	375
17.3 Schemi architetturali per sistemi distribuiti	377
17.4 Software come servizio	391
Punti chiave	396
Esercizi	397
Ulteriori letture	398
Capitolo 18 Ingegneria del software orientato ai servizi	399
18.1 Architettura orientata ai servizi	404
18.2 Servizi RESTful	409
18.3 Ingegneria dei servizi	413
18.4 Composizione dei servizi	423
Punti chiave	430
Esercizi	431
Ulteriori letture	432

Parte IV Gestione del software

Capitolo 19 Gestione della progettazione	435
19.1 Gestione dei rischi	439
19.2 Gestione del personale	447
19.3 Lavoro di squadra	452

Punti chiave	461
Esercizi	461
Ulteriori letture	462
Capitolo 20 Pianificazione della progettazione	463
20.1 Prezzo del software	466
20.2 Sviluppo guidato da piani	468
20.3 Tempistica dei progetti	472
20.4 Pianificazione agile	478
20.5 Tecniche di stima	480
20.6 Modelli di costi COCOMO	485
Punti chiave	495
Esercizi	496
Ulteriori letture	497
Capitolo 21 Gestione della qualità	499
21.1 Qualità del software	502
21.2 Standard del software	505
21.3 Revisioni e ispezioni	511
21.4 Gestione della qualità e sviluppo agile	516
21.5 Misure del software	518
Punti chiave	531
Esercizi	531
Ulteriori letture	532
Capitolo 22 Gestione della configurazione	533
22.1 Gestione delle versioni	537
22.2 Costruzione dei sistemi	545
22.3 Gestione delle modifiche	551
22.4 Gestione delle release	557
Punti chiave	560
Esercizi	561
Ulteriori letture	562
Glossario	563
Bibliografia	577
Indice analitico	593

Prefazione

Il progresso dell’ingegneria del software negli ultimi 50 anni è stato sbalorditivo. Le nostre società non potrebbero funzionare senza sistemi software professionali. I servizi e le infrastrutture – energia, comunicazioni e trasporti – si affidano a sistemi computerizzati complessi e abbastanza affidabili. Il software ci ha permesso di esplorare lo spazio e di creare il World Wide Web – il più significativo sistema informatico nella storia del genere umano. Smartphone e tablet sono ovunque e negli ultimi cinque anni è emersa una fiorente “industria delle app” che sviluppa software per questi dispositivi.

L’umanità oggi deve affrontare numerose sfide – variazioni climatiche, condizioni ambientali estreme, riduzione delle risorse naturali, aumento della popolazione, terrorismo internazionale e la necessità di aiutare gli anziani a condurre una vita soddisfacente e dignitosa. Occorrono nuove tecnologie per vincere queste sfide e, certamente, il software avrà un ruolo centrale in queste tecnologie. L’ingegneria del software è, quindi, criticamente importante per il futuro del nostro pianeta. Dobbiamo continuare a istruire gli ingegneri del software e sviluppare le materie informatiche in modo da soddisfare la richiesta di ulteriore software e creare sistemi sempre più complessi in grado di soddisfare le nostre esigenze.

Ovviamente, ci sono ancora vari problemi nella progettazione del software. I sistemi a volte vengono ancora consegnati in ritardo e costano più del previsto. Stiamo realizzando sistemi software sempre più complessi e, quindi, non dovremmo sorprenderci se durante lo sviluppo incontriamo qualche ostacolo. Tuttavia, non dovremmo permettere che questi problemi oscurino i significativi successi che sono stati conseguiti dall’ingegneria del software.

Questo libro, nelle varie edizioni, è in uso da oltre 30 anni; questa edizione si basa sui principi essenziali che furono stabiliti nella prima edizione.

1. Ho scelto di trattare l’ingegneria del software così come viene applicata nell’industria, senza assumere posizioni profetiche su particolari approcci, come lo sviluppo agile o i metodi formali. In realtà, l’industria è solita mescolare varie tecniche, come lo sviluppo agile e quello basato sui piani, e questo si riflette nel libro.
2. Ho scritto ciò che conosco e capisco. Ho ricevuto molti suggerimenti su ulteriori argomenti che potrebbero essere trattati più dettagliatamente, come lo sviluppo open-source, l’uso dell’UML e l’ingegneria del software mobile. In effetti, non conosco molto bene questi temi. Il mio lavoro si è rivolto soprattutto all’affidabilità e all’ingegneria dei sistemi, e questo si riflette sulla scelta dei temi avanzati per il libro.

Credo che gli argomenti chiave della moderna ingegneria del software siano la gestione della complessità del software, l’integrazione dei metodi agili con altre tecniche di sviluppo e l’adozione di tecniche per garantire che i sistemi siano protetti e resilienti. Questi temi sono stati i motivi ispiratori per le modifiche e le novità di questa nuova edizione del libro.

Modifiche rispetto alla precedente edizione

In sintesi, le principali modifiche e novità di questo libro rispetto alla precedente edizione sono:

- ho ampiamente aggiornato il capitolo sull'ingegneria del software agile, con nuovo materiale sul metodo Scrum. Ho aggiornato altri capitoli come richiesto per riflettere il crescente diffondersi dei metodi agili dell'ingegneria del software;
- ho aggiunto nuovi capitoli sull'ingegneria della resilienza, sull'ingegneria dei sistemi e sui sistemi di sistemi;
- ho riorganizzato completamente tre capitoli che trattano l'affidabilità, la sicurezza e la protezione dei sistemi;
- ho aggiunto nuovi argomenti sui servizi RESTful nel capitolo che tratta l'ingegneria del software orientata ai servizi;
- ho modificato e aggiornato il capitolo sulla gestione della configurazione con nuovo materiale sul controllo distribuito delle versioni;
- ho spostato sul sito web alcuni capitoli sul miglioramento dei processi e dell'ingegneria del software;
- ho aggiunto nuovo materiale nel sito web, inclusi alcuni video di supporto. Ho spiegato alcuni temi chiave sui video e sui video correlati di YouTube.

La struttura in quattro parti del libro, introdotta nelle precedenti edizioni, è stata mantenuta, ma ho apportato modifiche significative in ciascuna parte.

1. Nella Parte I ho riscritto completamente il Capitolo 3 (metodi agili) e l'ho aggiornato per riflettere il crescente utilizzo del metodo Scrum. Nel Capitolo 1 ho aggiunto un nuovo caso di studio su un ambiente di apprendimento digitale, che viene utilizzato in altri capitoli. I sistemi ereditati sono trattati più dettagliatamente nel Capitolo 9. Altre modifiche e novità secondarie sono state introdotte in tutti gli altri capitoli.
2. La Parte II, che tratta la fidatezza e la protezione dei sistemi, è stata rivista e ristrutturata. Anziché seguire un approccio orientato alle attività, dove le informazioni sulla sicurezza, protezione e affidabilità dei sistemi sono distribuite su vari capitoli, ho riorganizzato tutto in modo che ogni argomento sia trattato in un proprio capitolo. Questo agevola la trattazione di un argomento specifico, come la protezione, come parte di una materia più generale. Ho aggiunto un capitolo completamente nuovo sull'ingegneria della resilienza, che tratta la sicurezza cibernetica, la resilienza aziendale e la progettazione di sistemi resilienti.
3. Nella Parte III ho rivisto completamente il materiale sull'ingegneria dei sistemi orientati ai servizi per riflettere il crescente utilizzo dei servizi RESTful. Il capitolo sull'ingegneria del software orientata agli aspetti è stato eliminato dalla versione stampata ed è stato inserito online nella versione digitale del testo.

4. Nella Parte IV ho aggiornato il materiale sulla gestione della configurazione per riflettere il crescente diffondersi degli strumenti di controllo distribuito delle versioni, come Git. Il capitolo sul miglioramento dei processi è stato eliminato dalla versione stampata ed è stato messo online.

Un'importante modifica nel materiale supplementare del libro è l'aggiunta di video di suggerimenti per tutti i capitoli. Ho realizzato oltre 40 video su una serie di argomenti che sono disponibili sul mio canale YouTube e collegati alle pagine web del libro all'indirizzo iansommerville.com/software-engineering-book/videos/. Nei casi in cui non ho creato video, consiglio di vedere i video di YouTube che potrebbero essere utili.

A chi si rivolge il libro

Il libro si rivolge principalmente agli studenti universitari che intraprendono corsi preliminari e avanzati di ingegneria del software e dei sistemi. Si suppone che i lettori abbiano le conoscenze di base sulla programmazione e sulle principali strutture di dati.

Gli ingegneri del software possono servirsi del libro per avere una visione panoramica sulla materia e per aggiornare le loro conoscenze su argomenti quali il riutilizzo del software, la progettazione architettonica, la fidatezza e la protezione dei sistemi software.

Usare il libro nei corsi di ingegneria del software

Ho ideato il libro in modo che possa essere utilizzato in tre differenti tipi di corsi di ingegneria del software.

1. *Corsi introduttivi generali sull'ingegneria del software.* La prima parte del libro è stata ideata come supporto a un corso semestrale di ingegneria del software. Ci sono nove capitoli che trattano argomenti fondamentali nell'ingegneria del software. Se il vostro corso prevede una componente pratica, è possibile utilizzare i capitoli sulla gestione nella Parte IV.
2. *Corso introattivo o intermedio su argomenti specifici di ingegneria del software.* È possibile creare una serie di corsi più avanzati utilizzando i capitoli delle Parti II-IV. Per esempio, io ho tenuto un corso sui sistemi critici utilizzando i capitoli della Parte II più i capitoli sull'ingegneria dei sistemi e sulla gestione della qualità. In un corso che tratta l'ingegneria dei sistemi con uso intensivo del software, ho utilizzato i capitoli che trattano l'ingegneria del software, l'ingegneria dei requisiti, i sistemi dei sistemi, l'ingegneria del software distribuito, il software integrato, la gestione e la pianificazione dei progetti.
3. *Corsi più avanzati su argomenti specifici di ingegneria del software.* In questo caso, i capitoli del libro possono essere utilizzati per formare la base di corsi più avanzati, che possono essere integrati da ulteriori letture che trattano più dettagliatamente gli argomenti. Per esempio, un corso sul riutilizzo del software potrebbe basarsi sui Capitoli 15-18.

Ringraziamenti

Numerose persone hanno contribuito negli anni all’evoluzione di questo libro. Vorrei ringraziare tutti: revisori, studenti e lettori che hanno commentato le precedenti edizioni proponendo modifiche costruttive. In particolare vorrei ringraziare la mia famiglia, Anne, Ali e Jane per il loro affetto, aiuto e supporto mentre lavoravo su questo libro (e su tutte le precedenti versioni).

Ian Sommerville

Pearson Learning Solution

L'attività didattica e di apprendimento del corso è proposta all'interno di un ambiente digitale per lo studio, che ha l'obiettivo di completare il libro offrendo risorse didattiche fruibili in modo autonomo o per assegnazione del docente.

La piattaforma digitale **MyLab** – accessibile per diciotto mesi – integra e monitora l'attività individuale di studio con risorse multimediali: strumenti per l'autovalutazione e per il ripasso dei concetti chiave, esercitazioni interattive, gruppi di studio e aule virtuali animate da strumenti per l'apprendimento collaborativo (chat, forum, wiki, blog).

Le risorse multimediali sono costruite per rispondere a un preciso obiettivo formativo e sono organizzate attorno all'indice del manuale.

The screenshot shows the MyLab platform interface. At the top, it displays 'MyLab | Italia', 'Area personale', 'SOMMERVILLE_ITY', 'Altre classi ▾', and a 'Disconnecti' button. On the left, there's a sidebar with links: 'Home', 'Pearson eText', 'Materiale didattico', 'Risorse docente', 'Elenco studenti', 'Informazioni sulla classe', and 'Impostazioni'. The main content area is titled 'Ingegneria del software - 10/Ed.'. It features a thumbnail image of the book 'Ingegneria del software' by Sommerville. Below the thumbnail, there's a brief description of the course: 'L'attività didattica e di apprendimento del corso è proposta all'interno di un ambiente digitale per lo studio, che ha l'obiettivo di completare il libro offrendo risorse didattiche fruibili in modo autonomo o per assegnazione del docente.' Further down, it says: 'La piattaforma MyLab – accessibile per diciotto mesi – integra e monitora l'attività individuale di studio con risorse multimediali: strumenti per l'autovalutazione e per il ripasso dei concetti chiave, esercitazioni interattive, gruppi di studio e aule virtuali animate da strumenti per l'apprendimento collaborativo (chat, forum, wiki, blog).'. Another section discusses the eText version: 'Le risorse multimediali sono costruite per rispondere ad un preciso obiettivo formativo e sono organizzate attorno all'indice del manuale.' A third section mentions integrative materials: 'All'interno della piattaforma è possibile accedere all'edizione digitale del libro (eText), arricchita da funzionalità che permettono di personalizzarne la lettura, inserire segnalibri, studiare e condividere note anche su tablet.' A bulleted list follows: '• **Capitoli aggiuntivi** su: Sistemi fidati, Ingegneria dell'affidabilità, Ingegneria della sicurezza, Ingegneria della protezione, Ingegneria della resilienza; • **Video e Case studies** tratti dal sito web dell'autore (in inglese); • una raccolta di **esercizi di fine capitolo**.'. At the bottom, it states: 'Nelle risorse per il docente è disponibile una raccolta di **slide delle lezioni** relative ai singoli capitoli: un valido strumento che delinea i concetti chiave del capitolo e offre spunti per la discussione in aula.'

All'interno della piattaforma è possibile accedere all'edizione digitale del libro, arricchita da funzionalità che permettono di personalizzare la lettura, evidenziare il testo, inserire segnalibri e annotazioni, studiare e condividere note anche su tablet.

The screenshot shows the MyLab Italia interface. At the top, there's a navigation bar with 'Area personale', 'SOMMERVILLE_ITY' (highlighted in a black box), 'Altre classi', and 'Disconnecti'. Below the bar, a user profile for 'Emiliano Biondo' is shown. The main content area is titled 'Materiale didattico' and features a sub-section for 'Sommerville - INGEGNERIA DEL SOFTWARE'. The sidebar on the left lists chapters: 'Sommario', 'Materiale didattico: video', 'Materiale didattico: case studies', '1. Introduzione', '2. Processi software', and '3. Sviluppo agile del software'. The main content area has a heading 'Materiale didattico: video' with a sub-sub-section 'Materiale didattico: case studies'. It lists five video topics with icons: 'Video: Software engineering', 'Video: Agile methods', 'Video: Requirements and design', 'Video: Implementation and evolution', and 'Video: Critical systems'. A thumbnail for the book 'Ingegneria del software' by Sommerville is displayed on the right.

Tra i materiali integrativi sono disponibili:

■ **cinque capitoli aggiuntivi:**

- Capitolo e10 – Sistemi fidati
- Capitolo e11 – Ingegneria dell'affidabilità
- Capitolo e12 – Ingegneria della sicurezza
- Capitolo e13 – Ingegneria della protezione
- Capitolo e14 – Ingegneria della resilienza

■ un set di **esercizi di fine capitolo** con le relative soluzioni.

Materiali aggiuntivi dal sito web dell'autore

È inoltre possibile accedere ai seguenti materiali:

- presentazioni in formato PowerPoint per tutti i capitoli del libro;
- una serie di video che trattano alcuni argomenti sull'ingegneria del software. Sono consigliati anche alcuni video di supporto all'apprendimento sul canale YouTube;
- una guida per i docenti che offre alcuni suggerimenti su come utilizzare il libro nell'insegnamento di vari corsi;
- ulteriori informazioni sui casi di studio presentati nel libro (pompa di insulina, sistema di gestione di pazienti psichiatrici, sistema di gestione di una stazione meteorologica, sistema di apprendimento digitale) e su altri casi di studio, come il fallimento del lancio di Ariane 5;
- sei capitoli in lingua inglese trattano i seguenti argomenti: miglioramento dei processi, metodi formali, progettazione delle interazioni, architetture delle applicazioni, documentazione e sviluppo orientato agli aspetti;
- paragrafi addizionali che arricchiscono i contenuti di ciascun capitolo. Il link di questi paragrafi si trovano all'interno di appositi riquadri ombreggiati;
- presentazioni PowerPoint aggiuntive che trattano vari temi di ingegneria dei sistemi.

Descrizione

Lo scopo di questa parte del libro è fornire un'introduzione generale all'ingegneria del software. I capitoli di questa parte sono stati progettati per supportare un primo corso semestrale di ingegneria del software. Vengono presentati importanti concetti, quali i processi software e i metodi agili, e descritte le attività fondamentali di sviluppo del software, dalla specifica dei requisiti all'evoluzione dei sistemi.

Il Capitolo 1 è un'introduzione generale all'ingegneria del software professionale. Include anche una breve descrizione dei problemi di etica professionale. È importante per gli ingegneri del software riflettere sulle più ampie implicazioni del loro lavoro. Il capitolo presenta anche quattro casi di studio che saranno utilizzati nel libro; questi sono un sistema informativo per la gestione delle cartelle cliniche di pazienti psichiatrici (Mentcare), un sistema di controllo per una pompa di insulina portatile, un sistema integrato per una stazione meteorologica e un ambiente di apprendimento digitale (iLearn).

I Capitoli 2 e 3 trattano i processi di ingegneria del software e lo sviluppo agile. Nel Capitolo 2 sono introdotti i modelli di processi software, come il modello a cascata, e sono descritte le attività di base che fanno parte di questi processi. Il Capitolo 3 completa tutto questo con una descrizione dei metodi di sviluppo agile per l'ingegneria del software. Questo capitolo è cambiato notevolmente rispetto alle precedenti edizioni con una particolare attenzione allo sviluppo agile mediante l'approccio Scrum e ai metodi agili, quali le storie per le definizioni dei requisiti e lo sviluppo guidato da test.

Gli altri capitoli di questa parte sono descrizioni estese delle attività dei processi software che sono stati introdotti nel Capitolo 2. Il Capitolo 4 tratta l'importantissimo tema dell'ingegneria dei requisiti, dove vengono definiti i requisiti che caratterizzano un sistema. Il Capitolo 5 spiega la modellazione dei sistemi tramite UML, con particolare riguardo all'uso dei diagrammi use-case, diagrammi di classe, diagrammi di sequenza e diagrammi di stato per modellare un sistema software. Il Capitolo 6 evidenzia l'importanza dell'architettura software e l'uso degli schemi architettonici nella progettazione del software.

Il Capitolo 7 descrive la progettazione orientata agli oggetti e l'uso degli schemi di progettazione. Vengono inoltre presentati alcuni importanti temi di implementazione – riutilizzo, gestione della configurazione, sviluppo host-target e sviluppo open-source. Il Capitolo 8 è dedicato al test del software, dal test

delle unità durante lo sviluppo dei sistemi al test delle release software. Il capitolo tratta anche le tecniche di sviluppo guidato dai test – un approccio già adottato nei metodi agili, ma che ha un'ampia applicabilità. Infine, il Capitolo 9 presenta una panoramica sui temi dell'evoluzione del software; sono trattati i processi evolutivi, la manutenzione del software e la gestione dei sistemi ereditati.

1

Introduzione

Questo capitolo si propone di presentare l'ingegneria del software e fornire le basi per capire il resto del libro. Dopo aver letto questo capitolo:

- capirete cos'è l'ingegneria del software e perché è importante;
- saprete che lo sviluppo di differenti tipi di un sistema software potrebbe richiedere tecniche differenti di ingegneria del software;
- conoscerete i problemi etici e professionali che sono rilevanti per gli ingegneri del software;
- conoscerete quattro diversi tipi di sistemi che saranno utilizzati come esempi nel libro.

- 1.1 Sviluppo di software professionale
- 1.2 Etica dell'ingegneria del software
- 1.3 Casi di studio

L'ingegneria del software è essenziale per il funzionamento dei governi, della società, delle aziende e delle istituzioni nazionali e internazionali. Il mondo moderno non potrebbe funzionare senza software. Infrastrutture e aziende sono controllate da sistemi basati su computer, e molti prodotti elettrici includono un computer e un software di controllo. La produzione e la distribuzione industriale sono completamente computerizzate, come il sistema finanziario. L'intrattenimento – che include i giochi per computer, la musica, il cinema e la televisione – fa un uso intensivo del software. Oltre il 75% della popolazione mondiale ha un telefonino e, nel 2016, quasi tutti questi avranno accesso a Internet.

I sistemi software sono astratti e intangibili; non sono vincolati da materiali, governati da leggi fisiche o da processi di produzione. In qualche modo questo semplifica l'ingegneria del software, perché non ci sono limitazioni fisiche sul potenziale del software; d'altra parte la mancanza di vincoli naturali indica che il software può facilmente diventare estremamente complesso, difficile da capire e costoso da modificare.

Ci sono vari tipi di sistemi software, che vanno dai semplici sistemi integrati ai complessi sistemi informativi. Non esistono notazioni, metodi o tecniche universali per l'ingegneria del software, in quanto tipi differenti di software richiedono approcci differenti. Lo sviluppo di un sistema informativo organizzativo è completamente diverso dallo sviluppo di un controllore per uno strumento scientifico. Nessuno di questi sistemi ha molto in comune con un gioco di computer grafica. Tutte queste applicazioni richiedono l'ingegneria del software, ma non gli stessi metodi e le stesse tecniche di ingegneria del software.

Ancora oggi sentiamo parlare di progetti software che non funzionano correttamente e di “fallimenti del software”. L'ingegneria del software viene criticata perché inadeguata al moderno sviluppo del software. Secondo me, invece, molti dei cosiddetti fallimenti del software sono conseguenza di due fattori:

1. *Complessità crescente dei sistemi.* Mentre le nuove tecniche di ingegneria del software ci aiutano a creare sistemi più grandi e complessi, le esigenze cambiano. I sistemi devono essere creati e rilasciati più rapidamente; sono richiesti sistemi più grandi e perfino più complessi; e i sistemi devono avere nuove capacità che prima si ritenevano impossibili. Nuove tecniche di ingegneria del software devono essere sviluppate per affrontare la nuova sfida di un software più complesso.
2. *Fallimento nell'utilizzo dei metodi di ingegneria del software.* È relativamente facile scrivere programmi per computer senza utilizzare metodi e tecniche di ingegneria del software. Molte società sono finite alla deriva nello sviluppo del software mentre i loro prodotti e servizi si evolvevano. Esse non usano i metodi dell'ingegneria del software nel loro lavoro quotidiano. Di conseguenza, il loro software è spesso più costoso e meno affidabile di come dovrebbe essere. Abbiamo bisogno di una migliore istruzione e formazione sull'ingegneria del software per affrontare questo problema.

Gli ingegneri del software possono essere giustamente orgogliosi dei traguardi raggiunti. Naturalmente, abbiamo ancora qualche problema nello sviluppo di software complesso, ma senza l'ingegneria del software non avremmo esplorato lo spazio, non avremmo Internet e le moderne telecomunicazioni; tutte le forme di viaggio sarebbero più pericolose e costose. Le sfide per l'umanità nel XXI secolo sono le variazioni climatiche, la riduzione delle risorse naturali, i cambiamenti demografici e la crescita della popolazione mondiale. Ci affideremo all'ingegneria del software per sviluppare i sistemi che ci servono per affrontare questi problemi.

Storia dell'ingegneria del software

L'idea di ingegneria del software fu proposta per la prima volta nel 1968 a una conferenza tenuta per discutere quella che fu poi chiamata la "crisi del software" (Naur e Randell 1969). Divenne chiaro che i singoli approcci allo sviluppo del software non potevano tradursi in grandi e complessi sistemi software. Questi erano inaffidabili, costavano più del previsto e venivano rilasciati in ritardo.

Negli anni '70 e '80 furono sviluppati numerosi metodi e tecniche di ingegneria del software, quali la programmazione strutturata, l'information hiding (nascondere le informazioni) e lo sviluppo orientato agli oggetti. Gli strumenti e le notazioni standard allora sviluppati sono la base dell'attuale ingegneria del software.

<http://software-engineering-book.com/web/history/>

1.1 Sviluppo di software professionale

Molte persone scrivono programmi. Nelle aziende le persone scrivono programmi per fogli di calcolo per semplificare i loro compiti; scienziati e ingegneri scrivono programmi per elaborare i loro dati sperimentali; gli hobbisti scrivono programmi per i loro personali interessi e divertimento. Tuttavia, gran parte dello sviluppo del software è un'attività professionale in cui il software è sviluppato per scopi aziendali, per includerlo in altri componenti o come prodotti software, come i sistemi informativi o i sistemi di progettazione assistita dal computer. Le differenze chiave consistono nel fatto che il software professionale è destinato ad essere utilizzato da persone diverse dai loro sviluppatori e che di solito sono i team, anziché un singolo individuo, a sviluppare il software; per tutta la sua vita è soggetto a revisione e manutenzione.

L'ingegneria del software ha lo scopo di supportare lo sviluppo di software professionale, anziché la programmazione individuale. Include tecniche che supportano le specifiche, la progettazione e l'evoluzione dei programmi, nessuna delle quali è normalmente rilevante per lo sviluppo di software personale. Per aiutare il lettore ad avere una panoramica dell'ingegneria del software, ho sintetizzato le domande più frequenti su questo argomento nella Figura 1.1.

Domanda	Risposta
Cos'è il software?	Programmi per computer e relativa documentazione. I prodotti software possono essere sviluppati per un particolare cliente o per il mercato in generale.
Quali sono le caratteristiche di un buon software?	Un buon software deve fornire all'utente le funzionalità e le prestazioni richieste; deve essere mantenibile, affidabile e usabile.
Cos'è l'ingegneria del software?	L'ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione del software, dalla nascita al funzionamento e alla manutenzione.
Quali sono le attività fondamentali dell'ingegneria del software?	La specifica, lo sviluppo, la convalida e l'evoluzione del software.
Qual è la differenza tra l'ingegneria del software e l'informatica?	L'informatica si occupa della teoria e dei fondamenti; l'ingegneria del software degli aspetti pratici che riguardano lo sviluppo di software di qualità.
Qual è la differenza tra l'ingegneria del software e l'ingegneria dei sistemi?	L'ingegneria dei sistemi ha come oggetto tutti gli aspetti dello sviluppo di sistemi informatici, inclusi quelli hardware, software e di processo. L'ingegneria del software è una parte di questo processo più generale.
Quali sono le sfide chiave che l'ingegneria del software si pone?	Affrontare la crescente diversità e sviluppare software affidabile in tempi sempre più ridotti.
Quali sono i costi dell'ingegneria del software?	All'incirca il 60% dei costi è legato allo sviluppo, il 40% alle prove. Per il software personalizzato, i costi di evoluzione spesso superano quelli di sviluppo.
Quali sono le tecniche e i metodi migliori dell'ingegneria del software?	Mentre tutti i progetti software devono essere gestiti e sviluppati professionalmente, tecniche differenti sono appropriate per differenti tipi di sistemi. Per esempio, i giochi devono essere sempre sviluppati utilizzando una serie di prototipi, mentre i sistemi di controllo della sicurezza richiedono lo sviluppo di una specifica completa e chiara. Non ci sono metodi e tecniche che vanno bene per tutto.
Quali differenze ha apportato Internet all'ingegneria del software?	Internet ha consentito non soltanto lo sviluppo di complessi sistemi basati su servizi, largamente distribuiti, ma ha anche supportato la creazione di un'industria di "app" per cellulari e dispositivi mobili che ha cambiato l'economia del software.

Figura 1.1 Domande più frequenti sull'ingegneria del software.

Molti ritengono che il software sia semplicemente un altro modo di chiamare i programmi per computer. Tuttavia, quando si parla di ingegneria del software, il software non identifica soltanto i programmi, ma anche tutta la documentazione associata, le librerie, i siti web di supporto e i dati di configurazione che sono richiesti per rendere utili questi programmi. Un sistema software sviluppato professionalmente spesso è composto da più di un singolo programma. Un sistema può essere formato da vari programmi indipendenti e da file di configurazione che sono utilizzati per impostare tali programmi. Può includere la documentazione del sistema (che descrive la struttura del sistema), la documentazione per gli utenti (che spiega come utilizzare il sistema) e i siti web (dai quali gli utenti possono scaricare informazioni aggiornate sul prodotto).

Questa è una delle più importanti differenze tra sviluppo di software professionale e sviluppo di software amatoriale. Se state scrivendo un programma per conto vostro, nessun altro lo utilizzerà e, quindi, non dovete preoccuparvi di scrivere una guida al programma, che ne descrive il progetto o altro. Se, invece, state sviluppando un software che sarà utilizzato da altre persone e che sarà modificato da altri ingegneri, allora dovete fornire informazioni aggiuntive oltre al codice del programma.

Gli ingegneri del software si occupano dello sviluppo dei prodotti come, per esempio, il software che può essere venduto a un cliente. Esistono due tipi fondamentali di software.

1. *Prodotti generici*: sistemi autosufficienti prodotti da un'organizzazione che si occupa di sviluppo e disponibili sul mercato per i clienti che desiderano acquistarli. Esempi di questo tipo di prodotto includono le applicazioni per i dispositivi mobili, il software per PC come i database, gli elaboratori di testo, i programmi di grafica e gli strumenti di gestione dei progetti. Questo tipo di software include anche applicazioni “verticali” progettate per uno specifico mercato, quali i sistemi informatici delle biblioteche, i sistemi di contabilità e i sistemi di memorizzazione delle impronte dentali.
2. *Prodotti personalizzati (o su richiesta)*: sistemi commissionati a un fornitore di software da uno specifico cliente. Esempi di questo tipo di software includono sistemi di controllo per dispositivi elettronici, sistemi che supportano un determinato processo aziendale e sistemi di controllo del traffico aereo.

Una differenza sostanziale tra questi due tipi di software è che nei prodotti generici l'organizzazione che sviluppa il software controlla le sue specifiche. Questo significa che si incontrano dei problemi durante lo sviluppo del software, l'organizzazione può ripensare a ciò che deve essere sviluppato. Per i prodotti personalizzati, invece, gli sviluppatori devono lavorare sulle specifiche indicate e controllate da chi sta acquistando il software; gli sviluppatori devono attenersi a quelle specifiche.

Tuttavia la linea di demarcazione tra questi tipi di prodotti si sta assottigliando sempre più: sono in aumento le società che partono con un sistema generico e lo adattano alle necessità di un particolare cliente. I sistemi ERP (Enterprise Resource Planning, pianificazione delle risorse aziendali), come i sistemi SAP e Oracle, sono i migliori esempi di questo approccio: un sistema grande e complesso viene adattato a una società inserendo le informazioni relative alle regole e ai processi aziendali, ai rapporti richiesti e così via.

Quando si parla della qualità del software professionale, si deve considerare che il software è utilizzato e modificato da persone diverse dagli sviluppatori. La qualità quindi non riguarda semplicemente quello che il software fa, ma piuttosto deve includere il comportamento del software durante la sua esecuzione, e la struttura e l'organizzazione dei programmi sorgenti e della documentazione associata. Questi caratteri qualitativi, anche detti attributi non funzionali, sono, per esempio, il tempo di risposta del software a una richiesta dell'utente o la comprensibilità del codice del programma.

L'insieme specifico dei caratteri qualitativi di un sistema software dipende, ovviamente, dalla sua applicazione, pertanto un sistema di controllo aereo deve essere sicuro, un gioco interattivo deve rispondere velocemente, un sistema di commutazione telefonica deve essere affidabile e così via. Questi caratteri sono stati sintetizzati nella Figura 1.2, che mostra le caratteristiche essenziali di un sistema software professionale.

Caratteristica del prodotto	Descrizione
Accettabilità	Il software deve essere accettabile per il tipo di utenti per i quali è progettato. Questo significa che deve essere comprensibile, usabile e compatibile con gli altri sistemi che essi usano.
Fidatezza e protezione	La fidatezza del software include varie caratteristiche, quali l'affidabilità, la protezione e la sicurezza. Un software fidato non dovrebbe causare danni fisici o economici nel caso di malfunzionamento del sistema. Il software deve essere protetto in modo tale che utenti malintenzionati non possano accedere o danneggiare il sistema.
Efficienza	Il software non dovrebbe fare un uso dispendioso delle risorse del sistema, come la memoria o i cicli di processore. L'efficienza include la risposta alle sollecitazioni, i tempi di elaborazione, l'uso della memoria ecc.
Mantenibilità	Il software dovrebbe essere scritto in modo da evolversi in rapporto alle nuove richieste dei clienti. Questo è un attributo critico perché la modifica del software è una conseguenza inevitabile della variabilità dell'ambiente aziendale.

Figura 1.2 Caratteristiche essenziali di un software di qualità.

1.1.1 Ingegneria del software

L'ingegneria del software è una disciplina ingegneristica che riguarda tutti gli aspetti della produzione del software, dalle prime fasi della specifica di un sistema fino alla manutenzione del sistema dopo che è entrato in uso. In questa definizione ci sono due frasi chiave.

1. *Disciplina ingegneristica*: gli ingegneri fanno funzionare le cose, applicano teorie, metodi e strumenti dove sono necessari, ma li usano selettivamente e cercano sempre di trovare soluzioni ai problemi, anche quando non ci sono teorie e metodi adatti. Gli ingegneri sanno anche che devono lavorare sotto vincoli organizzativi e finanziari e devono cercare soluzioni che soddisfino questi vincoli.
2. *Tutti gli aspetti della produzione del software*: l'ingegneria del software non si occupa soltanto dei processi tecnici dello sviluppo, ma anche di attività come la gestione dei progetti e degli strumenti di sviluppo, dei metodi e delle teorie che supportano la produzione di software.

L'ingegneria aiuta a ottenere risultati con la qualità richiesta ed entro i limiti di tempo e di costo. Questo spesso richiede compromessi – gli ingegneri non sono perfezionisti. Le persone che scrivono programmi per se stessi, invece, possono impiegare tutto il tempo che vogliono per sviluppare un programma.

In generale, gli ingegneri del software adottano un approccio sistematico e organizzato al proprio lavoro, visto che, spesso, è il metodo più efficace per produrre software di alta qualità. Tuttavia l'ingegneria, scegliendo il metodo più adatto a determinate circostanze, può preferire un approccio più creativo e meno formale perché più efficace, per esempio quando si stanno sviluppando sistemi basati sul Web, che richiedono una fusione di capacità tecniche e grafiche.

L'ingegneria del software è importante per due motivi.

1. Singoli individui e organizzazioni si affidano sempre di più a sistemi software avanzati. Dobbiamo essere capaci di produrre sistemi affidabili e sicuri in modo economico e rapido.
2. Nel lungo termine, di solito, è economicamente più conveniente utilizzare i metodi e le tecniche di ingegneria del software per i sistemi software professionali, anziché scrivere semplicemente programmi come un progetto di programmazione personale. Il mancato utilizzo dei metodi di ingegneria del software comporta costi più elevati per il test, la garanzia di qualità e la manutenzione del software.

L'approccio sistematico che viene utilizzato nell'ingegneria del software è anche detto processo software. Un processo software è un insieme di attività che porta alla creazione di un prodotto software. Quattro attività fondamentali sono comuni a tutti i processi software.

1. *Specifiche del software*: clienti e ingegneri definiscono le funzionalità e i vincoli operativi del software da produrre.

2. *Sviluppo del software*: viene progettato e pianificato il software.
3. *Convalida del software*: il software viene convalidato per garantire ciò che il cliente richiede.
4. *Evoluzione del software*: il software viene modificato per soddisfare eventuali cambiamenti dei requisiti del cliente e del mercato.

Differenti tipi di sistemi richiedono differenti approcci di sviluppo, come vedremo nel Capitolo 2. Per esempio, il software real-time (letteralmente “in tempo reale”) nei velivoli deve essere specificato in maniera completa prima che inizi lo sviluppo, mentre nei sistemi di commercio elettronico le specifiche e il programma sono solitamente sviluppati assieme. Di conseguenza, queste attività generali devono essere organizzate in modi diversi e descritte con livelli di dettaglio differenti, in funzione del tipo di software da sviluppare.

L’ingegneria del software è correlata sia all’informatica sia all’ingegneria dei sistemi.

1. L’informatica si occupa delle teorie e dei metodi che stanno alla base dei sistemi software e di quelli informatici, mentre l’ingegneria del software si occupa dei problemi pratici relativi alla produzione del software. Alcune conoscenze di informatica sono essenziali per gli ingegneri del software così come le nozioni di fisica sono essenziali per gli ingegneri elettrotecnici. La teoria dell’informatica di solito si applica meglio allo sviluppo di programmi relativamente piccoli. Le più eleganti teorie dell’informatica sono raramente rilevanti per risolvere grandi e complicati problemi che richiedono una soluzione software.
2. L’ingegneria dei sistemi si occupa di tutti gli aspetti dello sviluppo e dell’evoluzione di sistemi complessi dove il software gioca un ruolo chiave; si occupa quindi dello sviluppo dell’hardware, della progettazione delle politiche e dei processi e dello sviluppo del sistema, come pure dell’ingegneria del software. Gli ingegneri dei sistemi si dedicano alla creazione delle specifiche del sistema, definendo la sua architettura generale e integrando le diverse parti necessarie alla realizzazione del prodotto finito.

Come vedremo nel prossimo paragrafo, ci sono vari tipi di software. Non esistono tecniche e metodi di ingegneria del software che possono essere universalmente utilizzati. Tuttavia, ci sono quattro aspetti correlati che influiscono su molti tipi di software differenti.

1. *Diversità* (o eterogeneità): sempre più spesso ai sistemi è richiesto di operare come sistemi distribuiti attraverso reti che includono tipi diversi di computer e dispositivi mobili. Oltre ad essere eseguito su un classico computer, il software deve poter essere eseguito su cellulari e tablet. Spesso è necessario integrare nuovo software in vecchi sistemi ereditati e scritti in diversi linguaggi di programmazione. La sfida della diversità consiste nel riuscire a sviluppare tecniche per costruire software affidabile ma anche flessibile, per gestire questa eterogeneità.

2. *Variabilità*: la società e le aziende oggi mutano in modo incredibilmente veloce in seguito allo sviluppo delle economie emergenti e alla disponibilità di nuove tecnologie. Occorre essere capaci di modificare il software esistente e sviluppare rapidamente nuovo software. Molte tecniche tradizionali di ingegneria del software richiedono molto tempo e la consegna di nuovi sistemi spesso si protrae più del previsto. Occorre evolversi in modo tale da ridurre significativamente i tempi di consegna del software ai clienti.
3. *Fiducia e protezione*: poiché il software ormai è radicato in ogni aspetto della nostra vita, è essenziale che sia possibile fidarsi di esso, specialmente se riguarda sistemi software remoti a cui si accede tramite una pagina web o un'interfaccia di servizio web. Occorre garantire che il software non possa essere attaccato da utenti malintenzionati e che sia possibile assicurare sempre la protezione delle informazioni.
4. *Scala*: il software deve essere sviluppato per una vasta scala di sistemi, dai piccoli sistemi integrati nelle unità portabili ai più estesi sistemi cloud di Internet che servono una comunità più vasta di utenti.

Per affrontare tutte queste sfide, abbiamo bisogno di nuovi strumenti e tecniche, e anche di vie innovative per combinare e utilizzare i metodi di ingegneria del software esistenti.

1.1.2 Diversità nell'ingegneria del software

L'ingegneria del software è un approccio sistematico alla produzione di software che si occupa di costi, pianificazione e problemi di fidatezza, come pure delle esigenze dei clienti e dei produttori di software. I metodi specifici, le tecniche e gli strumenti utilizzati dipendono dall'organizzazione che sviluppa il software, dal tipo di software e dalle persone coinvolte nel processo di sviluppo. Non ci sono metodi di ingegneria del software universali che sono applicabili a tutti i sistemi e a tutte le aziende; piuttosto, negli ultimi 50 anni sono stati sviluppati vari set di metodi e strumenti di ingegneria del software. Tuttavia, l'iniziativa SEMAT (Jacobson et al. 2013) suppone che ci possa essere un meta-processo di base che può essere istanziato per creare diversi tipi di processi. Questo avviene in una fase iniziale di sviluppo e può essere una base per migliorare gli attuali metodi di ingegneria del software.

Forse il fattore più significativo per determinare quali metodi e tecniche di ingegneria del software siano più rilevanti è il tipo di applicazione che si vuole sviluppare. Ci sono vari tipi di applicazioni differenti.

1. *Applicazioni autonome*. Sono applicazioni che possono essere eseguite su un personal computer o su un dispositivo mobile. Includono tutte le funzionalità necessarie e potrebbero non richiedere una connessione a una rete. Esempi sono le applicazioni per ufficio sui PC, i programmi CAD, il software per l'elaborazione delle immagini, le app di viaggio, le app di produttività e così via.

2. *Applicazioni interattive basate sulle transazioni.* Sono applicazioni che vengono eseguite su un computer remoto; gli utenti vi accedono dai propri computer, cellulari o tablet. Ovviamente, includono le applicazioni web, come quelle per il commercio elettronico dove l’utente interagisce con un sistema remoto per acquistare merci e servizi. Questa categoria di applicazioni include anche i sistemi aziendali, dove un’azienda fornisce l’accesso ai suoi sistemi tramite un browser web, un programma client o un servizio cloud, come la condivisione di foto o e-mail. Le applicazioni interattive spesso incorporano un grosso database che viene consultato e aggiornato in ogni transazione.
3. *Sistemi di controllo integrati.* Sono sistemi che controllano e gestiscono i dispositivi hardware. Numericamente, esistono più sistemi integrati che altri tipi di sistemi. Esempi di sistemi integrati sono il software in un cellulare, il software che controlla l’impianto antibloccaggio dei freni di un’automobile, e il software in un forno a microonde che controlla il processo di cottura.
4. *Sistemi di elaborazione batch.* Sono sistemi aziendali progettati per elaborare grandi blocchi di dati in sequenza. Questi sistemi elaborano un gran numero di singoli input per creare i corrispondenti output. Tipici esempi sono i sistemi per la fatturazione periodica, come la fatturazione telefonica, e i sistemi per il calcolo degli stipendi.
5. *Sistemi di intrattenimento.* Si tratta di sistemi per uso personale progettati per il divertimento dell’utente. Molti di questi sistemi sono giochi di vario genere, che possono essere eseguiti su una speciale console hardware. La qualità dell’interazione con l’utente è la caratteristica più importante di questi sistemi.
6. *Sistemi per la modellazione e la simulazione.* Sono sistemi sviluppati da scienziati e ingegneri per creare modelli di situazioni o processi fisici, che includono molti oggetti distinti che interagiscono tra di loro. Spesso si tratta di sistemi eseguiti in parallelo che richiedono alte prestazioni e capacità di calcolo.
7. *Sistemi per la raccolta e l’analisi dei dati.* I sistemi di raccolta dei dati sono sistemi che raccolgono i dati dai loro ambienti e li inviano ad altri sistemi per l’elaborazione. Il software può essere chiamato a interagire con sensori e spesso è installato in un ambiente ostile, per esempio all’interno di un motore o in un luogo remoto. Analisi di “big data” possono interessare sistemi cloud che eseguono analisi statistiche e ricercano relazioni fra i dati raccolti.
8. *Sistemi di sistemi.* Si tratta di sistemi, utilizzati nelle aziende e in altre grandi organizzazioni, che sono composti da un certo numero di altri sistemi software. Alcuni di questi possono essere prodotti software generici, come

un sistema ERP. Altri sistemi del gruppo possono essere scritti esplicitamente per quell'ambiente.

Ovviamente, i confini tra questi tipi di sistemi sono sfocati. Se sviluppare un gioco per un cellulare, dovete prendere in considerazione gli stessi vincoli (potenza, interazione hardware) degli sviluppatori del software per il cellulare. I sistemi di elaborazione batch sono spesso utilizzati in combinazione con i sistemi per le transazioni web. Per esempio, in un'azienda, i rimborsi delle spese di viaggio possono essere presentate tramite un'applicazione web, ma elaborati da un'applicazione batch per il pagamento mensile.

Ciascun tipo di sistema richiede tecniche di ingegneria del software specializzate, in quanto il software ha caratteristiche differenti. Per esempio, un sistema di controllo integrato in un'automobile è a sicurezza critica e viene incorporato nella ROM (read-only memory) quando viene installato nel veicolo; è quindi molto costoso modificarlo. Tale sistema richiede verifiche e convalide rigorose in modo che le probabilità di dover richiamare le automobili dopo la vendita per correggere il software siano ridotte al minimo. L'interazione con l'utente è minima (o forse inesistente), quindi non c'è bisogno di utilizzare un processo di sviluppo basato su un prototipo dell'interfaccia utente.

Per le applicazioni o i sistemi interattivi basati sulle transazioni, sviluppo e consegna iterativi sono l'approccio migliore, essendo il sistema composto da componenti riusabili. Tuttavia, tale approccio potrebbe risultare impraticabile per un sistema di sistemi, dove le specifiche dettagliate delle interazioni del sistema devono essere specificate in anticipo, in modo che ciascun sistema possa essere sviluppato separatamente.

Nonostante ciò, esistono dei concetti fondamentali di ingegneria del software che si applicano a tutti i tipi di sistemi software.

1. I sistemi devono essere sviluppati utilizzando un processo di sviluppo chiaro e accuratamente pianificato. L'organizzazione che sviluppa il software dovrà pianificare il processo di sviluppo e avere idee chiare su cosa sarà prodotto e quando sarà completato. Ovviamente, il processo specifico che dovrà essere utilizzato dipenderà dal tipo di software che sarà sviluppato.
2. La fidatezza e le prestazioni sono importanti per tutti i tipi di sistemi. Il software dovrà comportarsi come previsto, senza fallimenti, e dovrà essere disponibile all'uso quando richiesto. Il suo funzionamento dovrà essere sicuro e protetto il più possibile contro attacchi esterni. Il sistema dovrà essere eseguito con efficienza, senza spreco di risorse.
3. È importante capire e gestire la specifica e i requisiti del software (cosa deve fare il software). Occorre sapere che cosa si aspettano i differenti clienti e utenti del sistema, e occorre gestire le loro aspettative in modo che possa essere rilasciato un sistema utile entro i costi e i tempi previsti.

4. Occorre utilizzare con efficienza le risorse esistenti. Questo significa che, ove appropriato, occorre riusare il software che è già stato sviluppato, anziché scrivere un nuovo software.

Questi concetti fondamentali di processo, fidatezza, requisiti, gestione e riuso sono argomenti importanti di questo libro. Metodi differenti li rispecchiano in maniera differente, ma essi sono alla base di qualsiasi sviluppo di software professionale.

Tali fondamentali sono indipendenti dal linguaggio di programmazione utilizzato per sviluppare il software. Questo libro non tratta tecniche di programmazione specifiche, in quanto queste variano significativamente da un tipo di sistema all’altro. Per esempio, un linguaggio dinamico, come Ruby, è il tipo appropriato di linguaggio per sviluppare un sistema interattivo, ma non lo è per l’ingegneria dei sistemi integrati.

1.1.3 Ingegneria del software di Internet

Lo sviluppo di Internet e del World Wide Web ha avuto un profondo effetto su tutte le nostre vite. Inizialmente, il Web era soprattutto un magazzino di informazioni accessibile a tutti, ed ebbe scarsa influenza sui sistemi software. Questi sistemi erano eseguiti su computer locali ed erano accessibili soltanto dall’interno di un’organizzazione. Intorno al 2000 il Web iniziò ad evolversi e furono aggiunte molte nuove funzionalità ai browser. Ciò implicava che i sistemi basati sul Web potevano essere sviluppati dove (al posto di una speciale interfaccia utente) era possibile accedere a tali sistemi tramite un browser. Questo portò allo sviluppo di una vasta gamma di nuovi prodotti di sistema che fornivano servizi innovativi, accessibili sul Web. Questi servizi erano spesso finanziati dai messaggi pubblicitari che apparivano sugli schermi degli utenti e che non comportavano alcuna spesa diretta per gli utenti.

Oltre a questi prodotti di sistema, lo sviluppo dei browser che potevano eseguire piccoli programmi e svolgere qualche elaborazione locale portò a un’evoluzione del software per le aziende e le organizzazioni. Anziché scrivere software e distribuirlo nei PC degli utenti, il software veniva sviluppato su un server del Web. Ciò rese più economiche le operazioni di modifica e aggiornamento del software, in quanto non era necessario installarlo su ciascun PC. Furono anche ridotti i costi, in quanto lo sviluppo delle interfacce utente era particolarmente costoso. Ogni volta che era possibile farlo, le aziende passarono alle interazioni basate sul Web tramite i sistemi software.

Il concetto di software come servizio (Capitolo 17) fu proposto agli inizi del XXI secolo. Oggi questo è diventato l’approccio standard per la distribuzione dei prodotti di sistema basati sul Web, come Google Apps, Microsoft Office 365 e Adobe Creative Suite. Un numero sempre crescente di applicazioni software vengono eseguite su “cloud” remoti (anziché su server locali), cui è possibile accedere tramite Internet. Il calcolo cloud è svolto da un gran numero di computer

collegati che sono condivisi da molti utenti. Gli utenti non comprano il software, ma lo pagano in base a quanto ne utilizzano, oppure hanno accesso gratuito al software a patto di guardare gli annunci pubblicitari che appaiono sui loro schermi. Se utilizzate servizi come mail, storage o video basati sul Web, state utilizzando un sistema cloud.

L'avvento del Web ha portato a un drastico cambiamento del modo in cui viene organizzato il software aziendale. Prima del Web, le applicazioni aziendali erano monolitiche, singoli programmi che giravano su singoli computer o su gruppi di computer. Le comunicazioni erano locali, all'interno di un'organizzazione. Adesso il software è molto distribuito, a volte in tutto il mondo. Le applicazioni aziendali non vengono programmate da zero, ma riutilizzano molto spesso componenti e programmi esistenti.

Questo cambiamento nell'organizzazione del software ha avuto un significativo effetto sull'ingegneria del software per i sistemi basati sul Web. Per esempio:

1. il riutilizzo del software è diventato l'approccio dominante per costruire sistemi basati sul Web. Quando costruire questi sistemi, pensate a come potete assemblarli utilizzando componenti e sistemi software preesistenti, di solito raggruppati in una stessa struttura;
2. adesso è generalmente riconosciuto che è impraticabile specificare in anticipo tutti i requisiti per tali sistemi. I sistemi basati sul Web vengono sempre sviluppati e distribuiti in modo incrementale;
3. il software può essere implementato utilizzando l'ingegneria del software orientata ai servizi, dove i componenti software sono servizi web indipendenti. Questo approccio all'ingegneria del software sarà approfondito nel Capitolo 18;
4. si sono affermate alcune tecnologie di sviluppo delle interfacce, come AJAX (Holdener 2008) e HTML5 (Freeman 2011), che supportano la creazione di ricche interfacce all'interno di un browser web.

I concetti fondamentali dell'ingegneria del software, descritti nel precedente paragrafo, si applicano al software basato sul Web, come ad altri tipi di software. I sistemi basati sul Web stanno diventando sempre più grandi, quindi le tecniche di ingegneria del software che riguardano la scala e la complessità sono importanti per questi sistemi.

1.2 Etica dell'ingegneria del software

Come altre discipline ingegneristiche, anche l'ingegneria del software viene svolta in un ambito legale e sociale che limita la libertà degli ingegneri: questi devono comprendere che il loro lavoro comporta responsabilità più vaste della semplice applicazione di capacità tecniche, e devono agire in maniera eticamente e moralmente responsabile se vogliono essere rispettati come professionisti.

È naturale che bisogna sempre sottostare ai normali standard di onestà e integrità, non usare le proprie abilità e capacità in maniera disonesta o in modo da rovinare la reputazione dei colleghi; ci sono, però, delle aree in cui lo standard comportamentale non è limitato da leggi, ma da più tenui nozioni di responsabilità professionale, come per esempio:

1. *riservatezza*: occorre rispettare la riservatezza dei propri datori di lavoro e dei propri clienti, che sia stato firmato un accordo formale o meno;
2. *competenza*: non bisogna mentire sul proprio livello di competenza, né accettare lavori al di sopra delle proprie capacità;
3. *diritti di proprietà intellettuale*: occorre conoscere le leggi locali che governano l'uso delle proprietà intellettuali, come i brevetti e il copyright, in modo da proteggere quelle del proprio datore di lavoro e dei propri clienti;
4. *uso improprio del computer*: non bisogna sfruttare le proprie capacità tecniche per un cattivo uso dei computer altrui. L'uso improprio del computer varia dal banale (giocare con il computer di lavoro) all'estremamente serio (disseminazione di virus).

Società professionali e istituzioni hanno un ruolo importante nell'impostare gli standard etici: organizzazioni come ACM (*Association for Computing Machinery*, Associazione dei macchinari di calcolo), IEEE (*Institute of Electrical and Electronic Engineers*, Istituto degli ingegneri elettrotecnici ed elettronici) e BCS (*British Computer Society*, società informatica inglese) pubblicano un codice di condotta professionale o codice etico, sottoscritto dagli associati all'atto dell'iscrizione. Questi codici di condotta riguardano in genere il comportamento etico fondamentale. ACM e IEEE hanno cooperato per produrre un codice di etica e di pratica professionale unificato, in versione sintetica (Figura 1.3) o estesa (Gotterbarn et al. 1999). Quest'ultima aggiunge dettaglio e sostanza alla prima. Le motivazioni che sostengono questo codice comportamentale sono riassunte nei primi due paragrafi della versione più ampia:

I computer hanno un ruolo centrale e crescente nel commercio, nell'industria, nella pubblica amministrazione, nella medicina, nella formazione, nello spettacolo, e nel complesso della società. Gli sviluppatori software sono coloro che contribuiscono, partecipando direttamente o insegnando, all'analisi, alla specifica, al disegno, allo sviluppo, alla certificazione, alla manutenzione, al test dei sistemi software. Dato il loro ruolo nello sviluppo dei sistemi software, gli sviluppatori software hanno opportunità significative di fare del bene o di causare danni, di mettere in grado altre persone di fare del bene o di causare danni, o di influenzare altri a fare del bene o a causare danni. Per assicurare, per quanto possibile, che i loro sforzi verranno usati a fin di bene, gli sviluppatori software devono impegnarsi a rendere lo sviluppo software una professione rispettata e dagli effetti benefici. In accordo con questo impegno, gli sviluppatori software devono aderire al seguente Codice Etico e di pratica professionale.

Codice etico e di pratica professionale dell'ingegneria del software

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

Premessa

La versione breve del codice fornisce una sintesi delle aspirazioni, a un elevato livello di astrazione. Le clausole incluse nella versione integrale forniscono esempi e dettagli di come queste aspirazioni cambiano il modo in cui agiamo come professionisti dello sviluppo software. Senza le aspirazioni, i dettagli possono diventare legalistici e noiosi; senza i dettagli, le aspirazioni possono diventare belle parole, ma vuote; insieme, aspirazioni e dettagli formano un codice coeso.

Gli sviluppatori software devono impegnarsi a rendere l'analisi, la specifica, il disegno, lo sviluppo, il test e la manutenzione del software una professione rispettata e dagli effetti benefici. In accordo con il loro impegno alla salute, alla sicurezza fisica (*safety*) ed al benessere del pubblico, gli sviluppatori software devono aderire agli otto principi qui elencati.

1. **Pubblico.** Gli sviluppatori software devono agire in linea con l'interesse pubblico.
2. **Cliente e datore di lavoro.** Gli sviluppatori software devono agire in un modo conforme agli interessi del loro cliente e datore di lavoro, restando in accordo con l'interesse pubblico.
3. **Prodotto.** Gli sviluppatori software devono assicurare che i loro prodotti e le modifiche che vi applicano siano al livello di standard professionale più elevato possibile.
4. **Giudizio.** Gli sviluppatori software devono mantenere integrità e indipendenza nel loro giudizio professionale.
5. **Management.** Manager e leader degli sviluppatori software devono sottoscrivere e promuovere un approccio etico al management dello sviluppo e della manutenzione del software.
6. **Professione.** Gli sviluppatori software devono far progredire l'integrità e la reputazione della professione, restando in accordo con l'interesse pubblico.
7. **Colleghi.** Gli sviluppatori software devono essere leali e di supporto nei confronti dei loro colleghi.
8. **Se stessi.** Gli sviluppatori software devono, per tutta la durata della loro attività lavorativa, continuare la propria formazione sulla pratica della professione, e devono promuovere un approccio etico a essa.

Figura 1.3 Codice Etico ACM/IEEE (©IEEE/ACM 1999).

Il codice contiene otto Principi relativi al comportamento e alle decisioni che vengono prese da sviluppatori software professionisti: cioè coloro che svolgono ruoli direttamente legati alla professione, e inoltre educatori, manager, supervisori, decisori di alto livello, così come gli apprendisti e gli studenti della professione. I Principi identificano le relazioni significative dal punto di vista etico nelle quali possono prendere parte gli individui, i gruppi e le organizzazioni, e gli obblighi principali nell'ambito di tali relazioni. Le Clausole di ogni Principio illustrano alcuni degli obblighi compresi in queste relazioni. Questi obblighi trovano il loro fondamento nella natura umana degli sviluppatori software, nella speciale attenzione dovuta

*alle persone la cui vita viene interessata dal lavoro degli sviluppatori software, e negli elementi specifici della pratica dello sviluppo software. Il Codice li prescrive come obblighi per chiunque pretenda di essere, o aspiri a essere, uno sviluppatore software.*¹

Nelle situazioni in cui diverse persone hanno differenti punti di vista e obiettivi, si può avere a che fare con dilemmi etici. Per esempio, se per principio non si è d'accordo con le politiche di un proprio superiore, cosa bisogna fare? Chiaramente dipende dai singoli individui e dalla natura del disaccordo. È meglio discutere un problema relativo alla propria posizione dall'interno dell'azienda, oppure, in linea di principio, dimettersi? Se ci si accorge che ci sono problemi con un progetto, quando bisogna dirlo alla direzione? Se lo si discute quando è solo un sospetto, potrebbe sembrare di reagire più del necessario, se lo si solleva troppo tardi, potrebbe essere impossibile risolverlo.

Tali dilemmi etici devono essere affrontati da ogni ingegnere nella propria vita professionale e, per fortuna, molto spesso sono problemi relativamente poco importanti o che possono essere risolti senza troppa difficoltà. Quando questo non accade l'ingegnere deve affrontare, probabilmente, un altro problema: l'azione più semplice sarebbe dimettersi dal proprio lavoro, ma questo potrebbe influenzare la vita di altre persone come il partner o i figli.

Una situazione particolarmente difficile per un ingegnere professionista è quella in cui il datore di lavoro si comporta in modo non etico: per esempio, se un'azienda che sviluppa sistemi a sicurezza critica, per rispettare i tempi di consegna, falsifica i dati della convalida di sicurezza, la responsabilità dell'ingegnere è mantenere la riservatezza, avvisare il cliente o rendere pubblico, in qualche maniera, che il sistema distribuito potrebbe non essere sicuro?

Il problema è che non ci sono certezze quando si parla di sicurezza: per quanto un sistema possa non essere stato convalidato secondo i criteri predefiniti, questi ultimi potrebbero essere troppo rigidi, e il sistema potrebbe, in realtà, funzionare correttamente per tutto il tempo necessario. Potrebbe anche succedere che, pur essendo stato convalidato a dovere, il sistema fallisca e causi un incidente; divulgare troppo presto i problemi può causare danni al datore di lavoro e agli impiegati, mentre non rivelarli potrebbe creare danni ad altri.

Bisogna farsi un'opinione personale su questi argomenti; la giusta posizione etica dipende interamente dalla visione degli individui coinvolti. In questo caso dipende da tipo e dimensione dei danni potenziali e dalle persone che sarebbero danneggiate. Se la situazione è molto pericolosa, potrebbe essere giustificato rendere pubblica la notizia, per esempio, sulla stampa nazionale, sempre tentando di risolvere la situazione rispettando i diritti del proprio datore di lavoro.

¹ ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, versione sintetica della premessa. <http://www.acm.org/about/se-code> Copyright © 1999 dell'Association for Computing Machinery, Inc. e dell'Institute for Electrical and Electronics Engineers, Inc.

Un altro problema etico riguarda la partecipazione allo sviluppo di sistemi militari e nucleari: alcune persone non vogliono parteciparvi; altre lavorano a sistemi militari, ma non a quelli che hanno a che fare con armi; altre ancora pensano che la sicurezza nazionale sia un principio prioritario e non hanno obiezioni etiche a lavorare su sistemi di difesa.

In questa situazione è importante che, sia il datore di lavoro sia l'impiegato conoscano in precedenza le rispettive opinioni. Quando un'azienda è coinvolta nello sviluppo militare o nucleare, dovrebbe specificare agli impiegati che devono essere disposti ad accettare qualsiasi lavoro venga loro assegnato. Allo stesso modo, se un impiegato è stato assunto e ha dichiarato di non voler lavorare su simili sistemi, il datore di lavoro non dovrebbe fare successivamente pressioni perché questi cambi idea.

L'area generale dell'etica e della responsabilità professionale è una di quelle che ha ricevuto maggiore attenzione negli ultimi anni, ed è impossibile analizzarla da un punto di vista filosofico, approccio usato da Laudon (Laudon 1995) e Johnson (Johnson 2001). Alcuni testi più recenti, come quello di Tavani (Tavani 2013), introducono il concetto di cibernetica e trattano sia i temi filosofici di fondo sia i problemi legali e pratici. Sono descritti anche i problemi etici per gli utenti e gli sviluppatori di tecnologie.

Questa impostazione, tuttavia, appare troppo astratta e difficile da collegare all'esperienza quotidiana, ed è quindi preferibile un approccio più concreto rappresentato dai codici di condotta professionale (Bott 2005; Duquenoy 2007). Io credo che sia meglio discutere di etica in un contesto di ingegneria del software, anziché come un argomento a se stante; pertanto, in questo libro non sono incluse discussioni etiche astratte, ma esempi negli esercizi, che potranno costituire punti di partenza per una discussione di gruppo sui problemi etici.

1.3 Casi di studio

Per illustrare i concetti dell'ingegneria del software, utilizzerò quattro esempi di differenti tipi di sistema. Non ho utilizzato deliberatamente un solo caso di studio, in quanto uno dei messaggi chiave di questo libro è che la pratica dell'ingegneria del software dipende dal tipo di sistema che si sta producendo. Pertanto ho scelto degli esempi appropriati per i concetti di sicurezza, fidatezza, riutilizzo, modellazione dei sistemi ecc.

I tipi di sistemi utilizzati come casi di studio sono i seguenti.

1. *Un sistema integrato.* Si tratta di un sistema dove il software controlla un dispositivo hardware ed è integrato nel dispositivo. I problemi di un sistema integrato tipicamente includono la dimensione fisica, la rapidità di risposta e la gestione della potenza. L'esempio di sistema integrato scelto è un sistema software che controlla una pompa di insulina utilizzata nel trattamento di pazienti diabetici.

2. *Un sistema informativo.* Lo scopo principale di questo tipo di sistema è gestire e fornire l'accesso a un database di informazioni. I problemi di un sistema informativo includono la protezione, l'usabilità, la riservatezza e l'integrità dei dati. L'esempio di sistema informativo scelto è un sistema di registrazione medica.
3. *Un sistema di raccolta dati basata su sensori.* Si tratta di un sistema i cui obiettivi primari sono la raccolta dei dati da un insieme di sensori e l'elaborazione di tali dati. I requisiti chiave di questo tipo di sistema sono l'affidabilità, anche in condizioni ambientali ostili, e la mantenibilità. L'esempio scelto è una stazione meteorologica in un'area selvaggia.
4. *Un ambiente di supporto.* Questo sistema è formato da un insieme integrato di strumenti software che sono utilizzati per supportare qualche tipo di attività. Gli ambienti di programmazione, come Eclipse (Vogel 2012), saranno i tipi di ambienti più familiari per i lettori di questo libro. L'esempio scelto è un ambiente di apprendimento digitale che viene utilizzato per supportare l'apprendimento degli studenti nelle scuole.

In questo capitolo presenterò ciascuno di questi sistemi; maggiori informazioni sono disponibili nel sito software-engineering-book.com.

~~X~~ 1.3.1 Sistema di controllo di una pompa di insulina

Una pompa di insulina è un sistema medico che simula il funzionamento del pancreas (un organo interno). Il software che controlla questo sistema è un sistema integrato che riceve le informazioni da un sensore e controlla una pompa che rilascia una dose controllata di insulina a un paziente.

Le persone che soffrono di diabete usano questo sistema. Il diabete è una condizione relativamente comune in cui il pancreas non è capace di produrre sufficienti quantità di un ormone, chiamato insulina, che metabolizza il glucosio nel sangue. Il trattamento convenzionale del diabete richiede iniezioni regolari di insulina prodotta geneticamente. I diabetici attraverso un misuratore esterno rilevano il livello di zuccheri nel sangue e poi calcolano la dose di insulina necessaria.

Il problema di questo trattamento è che il livello di insulina nel sangue non dipende solo dal livello degli zuccheri, ma è anche funzione del momento in cui è stata fatta l'iniezione di insulina. Controlli irregolari possono portare gli zuccheri nel sangue a livelli molto bassi (se c'è troppa insulina) o molto alti (se ce n'è troppo poca). La prima situazione è peggiore, nel breve termine, perché livelli troppo bassi di zuccheri possono causare malfunzionamenti temporanei del cervello, fino alla perdita di conoscenza e alla morte; nel lungo termine, continui livelli troppo alti di zuccheri possono portare danni agli occhi, ai reni e problemi al cuore.

Grazie all'attuale avanzamento nello sviluppo di sensori miniaturizzati, è possibile sviluppare sistemi automatici di rifornimento di insulina, che controllano i

livelli degli zuccheri nel sangue e forniscono la giusta dose di insulina quando richiesto. Sistemi di rifornimento di questo genere esistono già negli ospedali per la cura dei pazienti. In futuro potrebbe essere possibile per un diabetico averne uno costantemente collegato al proprio corpo.

Un sistema di rifornimento di insulina controllato da un software può funzionare utilizzando un micro-sensore inserito nel paziente per misurare un certo parametro del sangue, proporzionale al livello degli zuccheri, che viene inviato alla centralina della pompa; questa calcola la quantità di insulina necessaria e invia il segnale alla micro-pompa che rilascia tale quantità attraverso l'ago costantemente inserito.

La Figura 1.4 mostra i componenti hardware e l'organizzazione della pompa per l'insulina. Per capire gli esempi di questo libro, basta sapere che il sensore del sangue misura la conducibilità elettrica del sangue in condizioni differenti e che i valori misurati possono essere correlati al livello degli zuccheri. La pompa rilascia una unità di insulina a ogni impulso del sistema di controllo; quindi, per rilasciare 10 unità di insulina, il sistema di controllo invia 10 impulsi alla pompa. La Figura 1.5 è un modello UML (*Unified Modeling Language*, linguaggio di modellazione unificato) che illustra come il livello iniziale degli zuccheri nel sangue si trasforma in una sequenza di comandi di controllo per la pompa.

Chiaramente, questo è un sistema a sicurezza critica. Se la pompa si guasta o non funziona correttamente, il paziente può subire gravi danni o entrare in coma, in quanto il livello degli zuccheri nel sangue è troppo basso o troppo alto. Il sistema deve quindi soddisfare due requisiti essenziali di alto livello:

1. il sistema deve essere disponibile a fornire l'insulina quando richiesta;
2. il sistema deve comportarsi in modo affidabile e fornire il giusto quantitativo di insulina per correggere il livello corrente di zuccheri nel sangue.

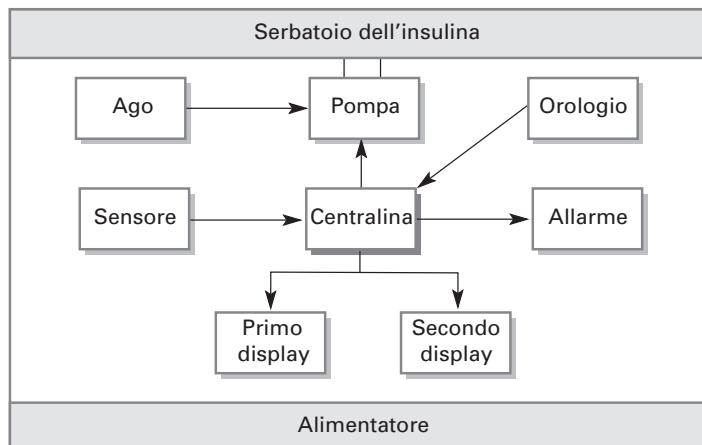


Figura 1.4 Struttura della pompa di insulina.

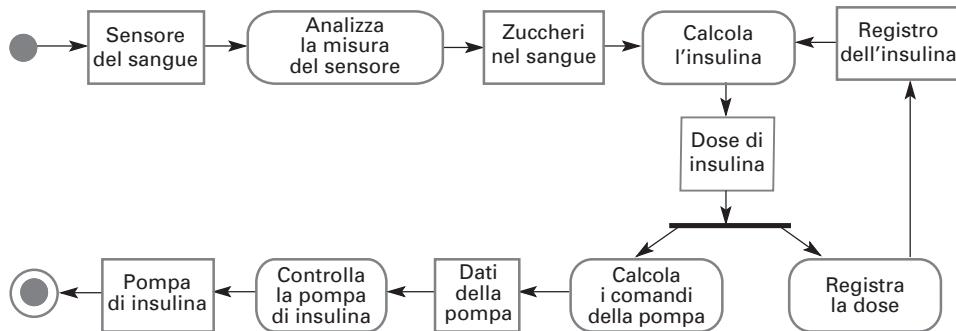


Figura 1.5 Modello delle attività della pompa di insulina.

Il sistema deve essere progettato e implementato in modo che questi due requisiti siano sempre garantiti. Altri dettagli su come assicurare che il sistema sia sicuro sono descritti nei capitoli successivi.

~~1.3.2 Sistema informativo di una clinica psichiatrica~~

Mentcare è un sistema informativo medico che registra informazioni su pazienti con problemi di salute mentale e le cure cui sono sottoposti. Molti di questi pazienti non richiedono cure ospedaliere dedicate, ma hanno bisogno di frequentare regolarmente delle cliniche specialistiche, dove possono incontrare un medico che ha una conoscenza approfondita dei loro problemi. Per agevolare l’assistenza dei pazienti, queste cliniche non sono all’interno di ospedali; gli incontri con il medico possono svolgersi presso studi medici locali o centri comunitari.

Il sistema Mentcare (Figura 1.6) è stato ideato per essere utilizzato nelle cliniche; si serve di un database centralizzato con le informazioni sui pazienti; può essere eseguito anche da un computer portatile, in modo che possa essere utilizzato da siti che non hanno una connessione di rete protetta. Quando i sistemi locali hanno un accesso protetto alla rete, utilizzano le informazioni sui pazienti registrate nel database, ma possono scaricare copie delle cartelle cliniche dei pazienti che possono utilizzare localmente quando non sono collegati. Mentcare non è un sistema medico completo e, quindi, non contiene informazioni su altre condizioni mediche dei pazienti; tuttavia, può interagire e scambiare dati con altri sistemi informativi medici.

Questo sistema ha due obiettivi:

1. generare le informazioni che consentono ai responsabili dei servizi sanitari di valutare le condizioni cliniche rispetto agli obiettivi locali o governativi;
2. fornire al personale medico informazioni tempestive per definire le cure dei pazienti.

I pazienti che soffrono di disturbi mentali a volte sono irrazionali e disorganizzati, quindi possono mancare agli appuntamenti, perdere prescrizioni e farmaci delibera-

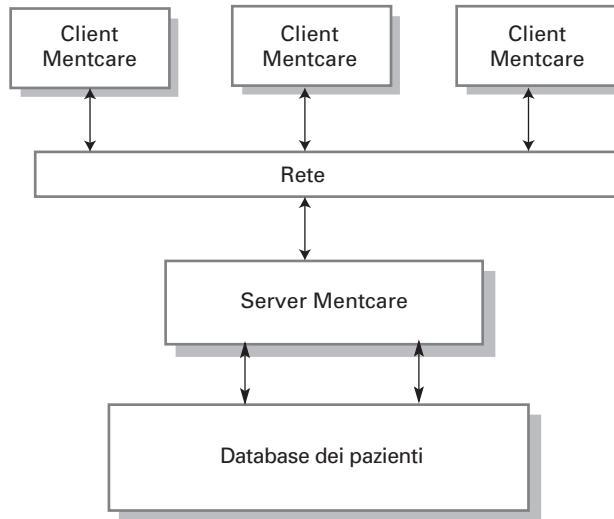


Figura 1.6 L'organizzazione del sistema Mentcare.

ratamente o accidentalmente, dimenticare le istruzioni e avere pretese irragionevoli dal personale medico. Possono presentarsi nelle cliniche in modo inaspettato. In pochi casi, possono essere un pericolo per se stessi o per altre persone. Possono cambiare regolarmente indirizzo o essere senza casa per breve o lungo tempo. Se i pazienti sono pericolosi, potrebbe essere necessario “isolarli”, ovvero ospitarli in una struttura sanitaria sicura dove possano essere osservati e curati.

Gli utenti di Mentcare includono lo staff medico, come dottori, infermieri e assistenti sanitari a domicilio (infermieri che visitano i pazienti a casa per controllarne le cure). Gli utenti non medici includono gli addetti alla reception che fissano gli appuntamenti, i tecnici che gestiscono il sistema informativo delle cartelle cliniche e il personale amministrativo che redige i rapporti.

Il sistema è utilizzato per registrare le informazioni sui pazienti (nome, indirizzo, età, parenti più prossimi ecc.), consultazioni mediche (data, visita, impressioni soggettive del paziente ecc.), condizioni e cure. I rapporti vengono redatti a intervalli regolari per lo staff medico e i dirigenti sanitari. Tipicamente, i rapporti per lo staff medico riguardano le informazioni su singoli pazienti, mentre i rapporti per i dirigenti sanitari sono in forma anonima e si occupano dei costi delle cure, condizioni sanitarie ecc.

Le caratteristiche chiave del sistema sono:

1. *gestione delle cure individuali.* Il personale medico può creare cartelle cliniche per i pazienti, modificare le informazioni registrate nel sistema, esaminare la storia clinica dei pazienti e così via. Il sistema fornisce riepiloghi di dati, in modo che i dottori che non hanno incontrato precedentemente un paziente possano conoscere rapidamente i problemi chiave e le cure prescritte;

2. *monitoraggio dei pazienti.* Il sistema controlla regolarmente le cartelle cliniche dei pazienti che sono in cura ed emette degli avvisi se vengono rilevati determinati problemi. Per esempio, se un paziente non ha incontrato un dottore per un certo tempo, viene generato un avviso. Uno dei più importanti elementi del sistema di monitoraggio è quello di tenere traccia dei pazienti che sono stati isolati e garantire che i controlli previsti dalla legge siano effettuati al momento giusto;
3. *report amministrativo.* Il sistema genera mensilmente dei rapporti gestionali che mostrano il numero di pazienti trattati in ogni clinica, il numero di pazienti che hanno iniziato o finito una cura, il numero di pazienti isolati, i farmaci prescritti e i loro costi, e così via.

Due leggi diverse governano il sistema: la legge sulla protezione dei dati che disciplina la riservatezza delle informazioni personali e la legge sulla salute mentale che disciplina la detenzione obbligatoria dei pazienti considerati un pericolo per se stessi o per altri. La salute mentale è unica in questo senso, in quanto è l'unica specializzazione che autorizza i medici a raccomandare la detenzione di un paziente contro la sua volontà. Tutto questo è soggetto a rigorose garanzie legislative. Uno degli obiettivi del sistema Mentcare è garantire che tutto il personale medico agisca sempre in accordo con le leggi e che le loro decisioni siano registrate per un eventuale esame della magistratura.

Come in tutti i sistemi medici, la riservatezza è un requisito critico. È essenziale che le informazioni sui pazienti siano confidenziali e non siano mai divulgata a nessuno, tranne al personale medico autorizzato e ai pazienti stessi. Mentcare è anche un sistema a sicurezza critica. Alcune malattie mentali potrebbero portare i pazienti al suicidio o ad arrecare danni ad altri. Ove possibile, il sistema dovrebbe segnalare al personale medico i pazienti potenzialmente suicidi o pericolosi.

Il progetto complessivo del sistema deve tenere in considerazione i requisiti di riservatezza e sicurezza. Il sistema deve essere disponibile quando richiesto, altrimenti la sicurezza potrebbe essere compromessa e potrebbe essere impossibile prescrivere le cure appropriate ai pazienti. Qui c'è un potenziale conflitto. La riservatezza è più facile da mantenere quando c'è una singola copia dei dati del sistema. Tuttavia, per garantire la disponibilità nel caso di guasto del server o di disconnessione da una rete, dovrebbero essere mantenute più copie dei dati. Tratterò i compromessi tra questi requisiti nei successivi capitoli.

1.3.3 Stazione meteorologica in un'area selvaggia

Per agevolare il monitoraggio delle variazioni climatiche e migliorare l'accuratezza delle previsioni meteorologiche in aree remote, il governo di una nazione con vaste aree selvagge ha deciso di installare centinaia di stazioni meteorologiche in queste zone. Le stazioni raccolgono dati da una serie di strumenti che misurano

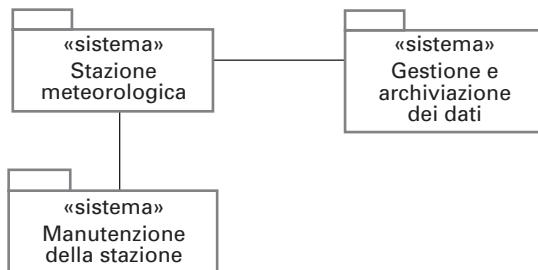


Figura 1.7 Ambiente della stazione meteorologica.

la temperatura, la pressione, la quantità di pioggia, la radiazione solare, la velocità e la direzione del vento.

Le stazioni fanno parte di un sistema meteorologico più complesso (Figura 1.7), che raccoglie i dati dalle stazioni e li mette a disposizione di altri sistemi dove vengono elaborati. I sistemi nella Figura 1.7 sono:

1. *sistema di stazioni meteorologiche*. Questo sistema ha il compito di raccogliere i dati meteorologici, effettuare una prima elaborazione di questi dati e trasmetterli a un sistema di gestione;
2. *sistema di gestione e archiviazione dei dati*. Questo sistema riceve i dati da tutte le stazioni meteorologiche, elabora e analizza i dati, e li archivia in un formato che può essere letto da altri sistemi, come quelli che effettuano le previsioni meteorologiche;
3. *sistema di manutenzione delle stazioni*. Questo sistema può comunicare via satellite con tutte le stazioni meteorologiche per monitorarne il corretto funzionamento e fornire un rapporto su eventuali problemi. Può aggiornare il software integrato nelle stazioni. Nel caso di problemi, questo sistema può essere utilizzato anche per controllare in modo remoto le stazioni.

Nella Figura 1.7 è stato utilizzato il simbolo di package UML per indicare che ciascun sistema è un insieme di componenti e i singoli sistemi sono identificati tramite lo stereotipo UML «sistema». Le associazioni tra i package indicano che c’è uno scambio di informazioni ma, a questo livello, non occorre definirle più dettagliatamente.

Le stazioni meteorologiche includono gli strumenti che misurano i parametri atmosferici, quali la velocità e la direzione del vento, le temperature del suolo e dell’aria, la pressione barometrica e la quantità di pioggia nel periodo di 24 ore. Ciascuno di questi strumenti è controllato da un sistema software che riceve periodicamente i valori dei parametri e gestisce i dati forniti dagli strumenti.

Il sistema delle stazioni opera raccogliendo le misurazioni meteorologiche a intervalli frequenti; per esempio, le temperature sono misurate ogni minuto. Tuttavia, poiché la larghezza di banda con il satellite è relativamente stretta, la stazione meteorologica effettua localmente una prima elaborazione e aggregazione

dei dati; poi trasmette questi dati aggregati quando richiesto dal sistema di raccolta dei dati. Se è impossibile stabilire una connessione, la stazione conserva i dati localmente, finché non sarà ripristinato il collegamento.

Ciascuna stazione è alimentata da batterie e deve essere completamente autosufficiente; non ci sono sorgenti esterne di energia elettrica né cavi di rete. Tutte le comunicazioni avvengono tramite un collegamento satellitare relativamente lento. La stazione deve includere qualche meccanismo (energia solare o eolica) per caricare le sue batterie. Poiché le stazioni sono installate in aree selvagge, possono essere esposte a condizioni ambientali ostili ed essere danneggiate dagli animali. Il software di una stazione meteorologica non deve quindi occuparsi soltanto della raccolta dei dati, ma deve anche:

1. monitorare gli strumenti, la carica delle batterie e l'hardware delle comunicazioni e riportare eventuali guasti al sistema di gestione;
2. gestire l'alimentazione elettrica del sistema, assicurando che le batterie vengano caricate ogni volta che le condizioni ambientali lo permettano, ma anche che i generatori vengano spenti quando le condizioni atmosferiche diventano potenzialmente pericolose, come nel caso di forti raffiche di vento;
3. consentire la riconfigurazione dinamica quando alcune parti del software vengono sostituite con nuove versioni o quando intervengono gli strumenti di backup nel caso di guasto del sistema.

Poiché le stazioni meteorologiche devono essere autosufficienti e non sono custodite, questo implica che il software installato è complesso, anche se la raccolta dei dati funzionalmente è un'operazione abbastanza semplice.

1.3.4 Ambiente di apprendimento digitale

Molti insegnanti sostengono che l'utilizzo di sistemi software interattivi a supporto dell'insegnamento possa migliorare le motivazioni degli studenti ed elevare il loro livello di conoscenza e comprensione. Tuttavia, non c'è una convergenza generale sulla strategia "migliore" da seguire per l'apprendimento supportato dai computer, e gli insegnanti in pratica usano una vasta gamma di strumenti interattivi basati sul Web a supporto dell'apprendimento. Gli strumenti utilizzati dipendono dall'età degli studenti, dal loro background culturale, dalla familiarità con i computer e i dispositivi informatici, e dalle preferenze degli insegnanti.

Un ambiente di apprendimento digitale è un sistema che si serve di una serie di strumenti generici e specifici per l'apprendimento, oltre a una serie di applicazioni che sono adattate alle esigenze degli studenti che usano il sistema. Questo sistema di apprendimento fornisce i servizi generali, quali il servizio di autenticazione, i servizi di comunicazione sincrona e asincrona, e un servizio di immagazzinamento dei dati.

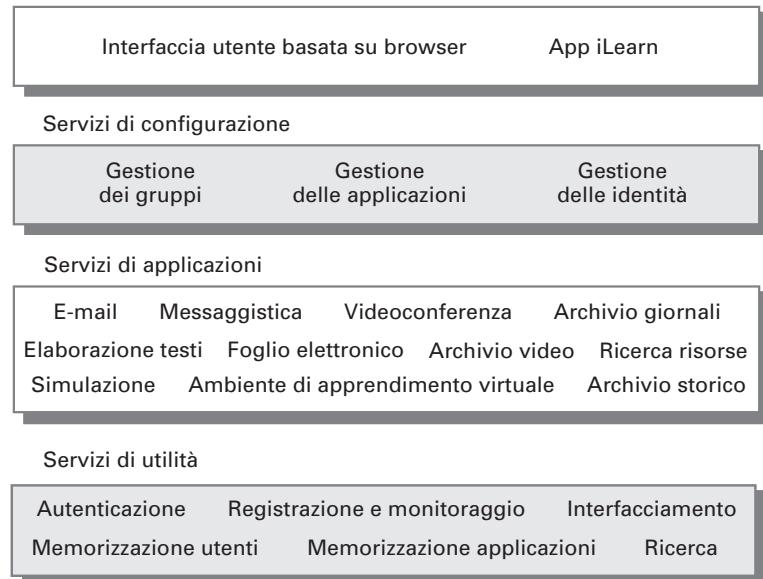


Figura 1.8 Architettura di un ambiente di apprendimento digitale (iLearn).

Gli strumenti inclusi in ciascuna versione dell’ambiente di apprendimento sono scelti dagli insegnanti e dagli studenti per soddisfare le loro specifiche esigenze. Tali strumenti possono essere applicazioni generiche, come i fogli elettronici, le applicazioni per l’apprendimento digitale, come VLE (Virtual Learning Environment), per la gestione della presentazione e della valutazione dei compiti a casa, dei giochi e delle simulazioni. Possono anche includere documenti specifici, come quelli sulla guerra civile americana, e le applicazioni per visualizzare e commentare tali documenti.

La Figura 1.8 è un modello architettonico ad alto livello di un ambiente di apprendimento digitale (iLearn) che è stato progettato per scuole di studenti di età compresa fra 3 e 18 anni. Si tratta di un sistema distribuito in cui tutti i componenti dell’ambiente sono servizi accessibili da qualsiasi collegamento Internet. Non c’è alcun obbligo che tutti gli strumenti di apprendimento siano concentrati in un unico luogo.

Il sistema è orientato ai servizi e tutti i suoi componenti sono considerati servizi sostituibili. Ci sono tre tipi di servizi nel sistema.

1. *Servizi di utilità.* Forniscono le funzionalità di base indipendenti dalle applicazioni e che possono essere utilizzate da altri servizi del sistema. I servizi di utilità di solito sono sviluppati o adattati specificatamente per questo sistema.

2. *Servizi di applicazioni.* Forniscono applicazioni specifiche, quali e-mail, condivisione di foto e videoconferenza, e consentono di accedere a materiali specifici, quali filmati scientifici o documenti storici. I servizi di applicazioni sono servizi esterni espressamente acquistati per il sistema o accessibili gratuitamente da Internet.
3. *Servizi di configurazione.* Sono utilizzati per adattare l’ambiente di apprendimento a specifici servizi di applicazioni e per definire come i servizi devono essere condivisi tra insegnanti, studenti e loro genitori.

L’ambiente è stato progettato in modo tale che ogni servizio possa essere sostituito quando si rende disponibile un nuovo servizio e per fornire versioni differenti del sistema che si adattano all’età degli utenti. Questo significa che il sistema deve supportare due livelli di integrazione dei servizi.

1. *Servizi integrati.* Sono i servizi che offrono un’API (Application Programming Interface, interfaccia per programmi applicativi) e che sono accessibili da altri servizi tramite tale API. È quindi possibile una comunicazione diretta tra un servizio all’altro. Un servizio di autenticazione è un esempio di servizio integrato. Anziché utilizzare i loro meccanismi di autenticazione, altri servizi possono chiamare un servizio di autenticazione per autenticare gli utenti. Se gli utenti sono già autenticati, allora il servizio di autenticazione può passare le informazioni di autenticazione direttamente a un altro servizio, tramite un’API, senza bisogno che gli utenti si autentichino di nuovo.
2. *Servizi indipendenti.* Sono servizi che sono accessibili semplicemente tramite un browser e che funzionano indipendentemente da altri servizi. Le informazioni possono essere condivise con altri servizi tramite esplicite azioni dell’utente, come la richiesta di copia e incolla; potrebbe essere necessario ripetere l’autenticazione per ogni servizio indipendente.

Se un servizio indipendente viene largamente utilizzato, il team di sviluppo potrebbe integrarlo in modo che diventi un servizio integrato supportato dal sistema.

Punti chiave

- L’ingegneria del software è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione di software.
- Il software non è un semplice programma o un gruppo di programmi, ma include anche tutta la documentazione elettronica che serve agli utenti dei sistemi, agli sviluppatori e ai responsabili della garanzia della qualità. Le caratteristiche essenziali di un prodotto software sono la mantenibilità, la fidatezza, l’efficienza e l’accettabilità.

- Il processo software include tutte le attività coinvolte nello sviluppo del software. Le attività di alto livello delle specifiche, dello sviluppo, della convalida e dell’evoluzione del software sono parte di tutti i processi software.
- Ci sono vari tipi di sistemi software; ciascuno di essi richiede appropriati strumenti e tecniche di ingegneria del software per il suo sviluppo. Poche tecniche di implementazione e progettazione sono applicabili a tutti i tipi di sistemi.
- I concetti fondamentali dell’ingegneria del software sono applicabili a tutti i tipi di sistemi software. Questi concetti includono la pianificazione dei processi software, la fidatezza e la protezione del software, l’ingegneria dei requisiti e il riutilizzo del software esistente.
- Gli ingegneri del software hanno responsabilità sia verso la loro professione sia verso la società; non dovrebbero occuparsi soltanto di problemi tecnici, ma dovrebbero occuparsi anche di problemi etici che influiscono sul loro lavoro.
- Gli ordini professionali pubblicano codici di condotta che delineano gli standard di comportamento cui i loro iscritti devono attenersi.

Esercizi

- 1.1 Spiegate perché il software professionale sviluppato per un cliente non è formato semplicemente dai programmi che sono stati sviluppati e consegnati.
- * 1.2 Qual è la differenza più importante tra lo sviluppo di software generico e quello personalizzato? Che cosa potrebbe significare questo in pratica per gli utenti di prodotti software generici?
- * 1.3 Quali sono le quattro caratteristiche più importanti che tutti i prodotti software professionali dovrebbero avere? Suggerite altre quattro caratteristiche che in alcuni casi potrebbero essere importanti.
- * 1.4 Oltre alle sfide di eterogeneità, dei continui mutamenti della società e delle aziende, della fiducia e della protezione, identificate altri problemi e sfide che l’ingegneria del software dovrà probabilmente affrontare nel XXI secolo (*suggerimento*: pensate all’ambiente).
- * 1.5 Sulla base delle vostre conoscenze di alcuni tipi di applicazioni descritti nel Paragrafo 1.1.2, spiegate con alcuni esempi perché differenti tipi di applicazioni richiedono specifiche tecniche di ingegneria del software per supportarne la progettazione e lo sviluppo.
- 1.6 Spiegate perché i principi fondamentali dell’ingegneria del software di processo, fidatezza, gestione dei requisiti e riutilizzo sono importanti per tutti i tipi di sistemi software.
- 1.7 Spiegate perché l’uso universale del Web ha cambiato i sistemi software e l’ingegneria dei sistemi software.
- * 1.8 Discutete se gli ingegneri professionisti dovrebbero avere l’abilitazione, come i dottori e gli avvocati.
- 1.9 Indicate un esempio appropriato che illustri ogni singola norma del Codice Etico ACM/IEEE riportato nella Figura 1.3.

- 1.10 Per favorire la lotta al terrorismo, molte nazioni stanno pianificando lo sviluppo di sistemi informatici che tengano sotto controllo gran parte dei propri cittadini e delle loro azioni, con ovvie conseguenze sulla privacy. Discutete l’etica dello sviluppo di questo tipo di sistemi.
- * *Gli esercizi contrassegnati con l’asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Software Engineering Code of Ethics Is Approved.” Un articolo che tratta il background di sviluppo del codice etico di ACM/IEEE e che include la forma breve e completa del codice. (Comm. ACM, D. Gotterbarn, K. Miller e S. Rogerson, ottobre 1999.) <http://dx.doi.org/10.1109/MC.1999.796142>

“A View of 20th and 21st Century Software Engineering.” Uno sguardo al passato e al futuro dell’ingegneria del software da parte di uno dei primi e più illustri ingegneri del software. Barry Boehm identifica i principi intramontabili dell’ingegneria del software, ma indica anche quelle pratiche comuni che sono diventate obsolete. (B. Boehm, Proc. 28th Software Engineering Conf., Shanghai 2006.) <http://dx.doi.org/10.1145/1134285.1134288>

“Software Engineering Ethics.” Un numero speciale di *IEEE Computer*, con diversi articoli sull’etica dell’ingegneria del software (*IEEE Computer*, 42 (6), giugno 2009).

Ethics for the Information Age. Un libro che tratta tutti gli aspetti dell’etica dell’informatica, non soltanto dell’ingegneria del software. Credo che questo sia l’approccio corretto per capire i problemi etici dell’ingegneria del software all’interno di un contesto etico più ampio (M. J. Quinn, 2013, Addison-Wesley).

The Essence of Software Engineering: Applying the SEMAT kernel. Questo libro espone il concetto di un contesto universale che può stare alla base di tutti i metodi di ingegneria del software. Può essere adattato e utilizzato per tutti i tipi di sistemi e organizzazioni. Personalmente, sono scettico sull’idea che un approccio universale possa essere realistico nella pratica, ma il libro presenta qualche interessante idea che vale la pena esaminare (I. Jacobsen, P-W Ng, P. E. McMahon, I. Spence e S. Lidman, 2013, Addison-Wesley.)

2

Processi software

L'obiettivo di questo capitolo è spiegare il concetto di processo software. Dopo aver letto questo capitolo:

- capirete i concetti di processo software e modello di processo software;
- conoscerete i tre modelli di processi software generici e quando possono essere utilizzati;
- conoscerete le attività fondamentali coinvolte nell'ingegneria dei requisiti, nello sviluppo, nel test e nell'evoluzione del software;
- capirete perché i processi devono essere organizzati per far fronte ai cambiamenti dei requisiti e della progettazione del software;
- conoscerete il concetto di miglioramento dei processi software e i fattori che influiscono sulla qualità dei processi.

- 2.1 Modelli dei processi software
- 2.2 Attività di processo
- 2.3 Far fronte ai cambiamenti
- 2.4 Miglioramento dei processi

Un processo software è un insieme di attività che porta alla creazione di un prodotto software. Come detto nel Capitolo 1, ci sono vari tipi di sistemi software e non esiste un unico metodo di ingegneria del software che può essere applicato a tutti questi sistemi. Di conseguenza, non esiste un processo software universalmente applicabile. Il processo utilizzato in aziende differenti dipende dal tipo di software che si sta sviluppando, dalle richieste del cliente e dalle capacità delle persone che scrivono il software.

Benché ci siano molti processi software differenti, tuttavia tutti devono includere in qualche misura le quattro attività fondamentali dell'ingegneria del software, che sono state introdotte nel Capitolo 1.

1. *Specifica del software*: vengono definite le funzionalità del software e fissati i vincoli operativi.
2. *Sviluppo del software*: deve essere prodotto il software che realizza i requisiti definiti nella specifica.
3. *Convalida del software*: il software deve essere convalidato per garantire ciò che il cliente richiede.
4. *Evoluzione del software*: il software deve evolversi per soddisfare le necessità e i cambiamenti dei requisiti dell'utente.

Queste attività sono complesse di per sé e includono sottoattività, quali la convalida dei requisiti, la progettazione architettonica e il test delle unità. I processi includono anche altre attività, quali la gestione della configurazione del software e la pianificazione del progetto che supportano le attività di produzione.

Quando descriviamo e analizziamo i processi, di solito parliamo della attività svolte in questi processi, come specificare un modello di dati, progettare un'interfaccia utente e ordinare tali attività. Tutto può essere correlato a ciò che fanno le persone per sviluppare il software. Tuttavia, quando si descrivono i processi, è anche importante descrivere le persone che sono coinvolte, che cosa viene prodotto e le condizioni che influiscono sulla sequenza delle attività.

1. Prodotti o consegne sono il risultato di un'attività di processo. Per esempio, il risultato dell'attività di progettazione architettonica potrebbe essere un modello dell'architettura del software.
2. I ruoli riflettono le responsabilità delle persone coinvolte nel processo. Esempi di ruoli sono il gestore del progetto, il gestore della configurazione e il programmatore.
3. Precondizioni e postcondizioni sono le condizioni che devono essere soddisfatte prima e dopo che un'attività di un processo è stata svolta o che un prodotto è stato realizzato. Per esempio, prima che inizi la progettazione architettonica, una precondizione potrebbe essere che il client approvi tutti i requisiti; finita questa attività, una postcondizione potrebbe essere che siano rivisti i modelli UML che descrivono l'architettura.

I processi software sono complessi e, come tutti i processi intellettuali e creativi, si basano sulle decisioni e sui giudizi delle persone. Dal momento che non esiste un processo universale che sia valido per tutti i tipi di software, molte società di software hanno ideato i loro processi di sviluppo. I processi si sono evoluti per sfruttare le capacità degli sviluppatori di software in una particolare organizzazione e le caratteristiche dei sistemi che si stanno sviluppando. Per i sistemi a sicurezza critica, è richiesto l'uso di un processo di sviluppo molto strutturato, dove vengono mantenute registrazioni dettagliate; per i sistemi aziendali, con requisiti che cambiano velocemente, sarebbe più appropriato un processo più agile e flessibile.

Come detto nel Capitolo 1, lo sviluppo del software è un'attività professionale, quindi la pianificazione è una parte intrinseca di tutti i processi. I processi guidati da piani (*plan-driven*) sono processi dove tutte le attività sono pianificate in anticipo e il loro avanzamento è misurato rispetto a quanto previsto dal piano. Nei processi agili, descritti nel Capitolo 3, la pianificazione è incrementale e continua durante lo sviluppo del software. È quindi più facile modificare il processo per adeguarsi alle modifiche dei requisiti del cliente o del prodotto. Come hanno spiegato Boehm e Turner (Boehm e Turner 2004), ogni approccio è adatto a differenti tipi di software. In generale, per i grandi sistemi occorre trovare un compromesso tra processi pianificati e processi agili.

Per quanto non esista un processo software universale, in molte organizzazioni c'è la possibilità di migliorarlo. I processi possono comprendere tecniche superate o non trarre vantaggio dalle migliori pratiche dell'ingegnerizzazione industriale del software, anzi molte organizzazioni ancora non sfruttano i metodi dell'ingegneria del software nello sviluppo del software. Possono migliorare i loro processi introducendo tecniche come la modellazione UML e lo sviluppo guidato da test. Descriverò brevemente il miglioramento dei processi software più avanti in questo capitolo e più dettagliatamente nel Capitolo 26.

2.1 Modelli dei processi software

Un modello di processo software – detto anche modello SDLC (Software Development Life Cycle) – è una rappresentazione semplificata di un processo software. Ogni modello rappresenta un processo da una particolare prospettiva, e quindi fornisce solo delle informazioni parziali sul processo. In questo paragrafo saranno introdotti alcuni modelli di processo molto generici (chiamati a volte *paradigmi di processo*) e saranno presentati da una prospettiva architettonica; ovvero si vedranno le strutture del processo, ma non i dettagli sulle specifiche attività.

Questi modelli generici sono descrizioni astratte di alto livello dei processi software, che possono essere utilizzate per spiegare i diversi approcci allo sviluppo del software. Possono essere considerate come strutture di processo da estendere e adattare per creare processi di ingegneria del software più specifici.

I modelli di processo generici qui trattati sono i seguenti:

1. *modello a cascata*. Le attività di processo fondamentali (specificazione, sviluppo, convalida ed evoluzione) sono rappresentate come fasi distinte del processo, per esempio la specifica dei requisiti, la progettazione del software, l'implementazione, il test e così via;
2. *sviluppo incrementale*. Questo approccio intreccia le attività di specificazione, sviluppo e convalida. Il sistema viene sviluppato come una serie di versioni (incrementi), ciascuna delle quali aggiunge nuove funzionalità alla versione precedente;
3. *integrazione e configurazione*. Questo approccio si basa sulla disponibilità di un gran numero di componenti o sistemi riutilizzabili. Il processo di sviluppo si basa sulla configurazione di questi componenti per utilizzarli in una nuova disposizione e integrarli in un sistema.

Come detto in precedenza, non esiste un modello di processo universale che si possa applicare appropriatamente a tutti i tipi di sviluppo del software. Il processo appropriato dipende dai requisiti dei clienti e delle politiche normative, dall'ambiente in cui il software sarà utilizzato e dal tipo di software che si intende sviluppare. Per esempio, il software a sicurezza critica di solito è sviluppato utilizzando un processo a cascata in quanto sono richieste molte analisi e documentazione prima di iniziare l'implementazione. I prodotti software adesso sono sempre sviluppati utilizzando un modello di processo incrementale. I sistemi aziendali vengono sempre più sviluppati configurando e integrando i sistemi esistenti per creare un nuovo sistema con le funzionalità richieste.

Gran parte dello sviluppo pratico del software si basa su un modello generale, ma spesso incorpora caratteristiche di altri modelli. Questo è particolarmente vero per l'ingegnerizzazione di grandi sistemi, per i quali ha senso combinare alcune delle migliori caratteristiche di tutti i processi generali. È necessario avere informazioni sui requisiti essenziali dei sistemi per progettare un'architettura software a supporto di tali requisiti; non è possibile effettuare questo sviluppo in modo incrementale. I sottosistemi all'interno di un sistema più grande possono essere sviluppati utilizzando approcci differenti. Le parti del sistema che sono ben chiare possono essere specificate e sviluppate utilizzando il modello a cascata o incluse come sistemi pronti all'uso da configurare. Altre parti del sistema, che sono difficili da specificare preventivamente, dovrebbero essere sempre sviluppate utilizzando un approccio incrementale. In entrambi i casi, i componenti software con molta probabilità saranno riutilizzabili.

Sono stati fatti vari tentativi per sviluppare modelli di processi “universali” che si ispirano a tutti questi modelli generali. Uno dei più noti di questi modelli universali è RUP (Rational Unified Process) (Krutchen 2003), che è stato sviluppato da Rational, una società di ingegneria del software degli USA. RUP è un

Rational Unified Process (RUP)

Il modello RUP (Rational Unified Process) mette insieme gli elementi di tutti i modelli di processo generici qui descritti, e supporta la prototipazione e la consegna incrementale del software (Krutchen 2003). Questo modello di solito è descritto secondo tre prospettive: una prospettiva dinamica che mostra le fasi del modello nel tempo; una prospettiva statica che mostra le attività di processo coinvolte; una prospettiva pratica che suggerisce le buone prassi da seguire durante il processo. Le fasi del modello RUP sono quattro: avvio, dove viene definito un business case per il sistema; elaborazione, dove vengono sviluppati i requisiti e l'architettura; costruzione, dove viene implementato il software; transizione, dove viene installato il sistema.

<http://software-engineering-book.com/web/rup/>

modello flessibile che può essere istanziato in vari modi per creare processi che somigliano ai modelli generali qui descritti. RUP è stato adottato da grandi società di software (in particolare IBM), ma non ha avuto una vasta diffusione.

2.1.1 Modello a cascata

Il primo modello di processo di sviluppo software pubblicato fu derivato dai modelli utilizzati nell'ingegnerizzazione di grandi sistemi militari (Royce 1970). In questo modello il processo di sviluppo del software è costituito da un certo numero di stadi, come illustra la Figura 2.1. A causa del susseguirsi di una fase dopo l'altra, il modello è conosciuto come modello a cascata o come ciclo di vita del software. Il modello a cascata è un esempio di processo guidato da un piano. Almeno in linea di principio, occorre pianificare tutte le attività di processo prima di iniziare lo sviluppo del software.

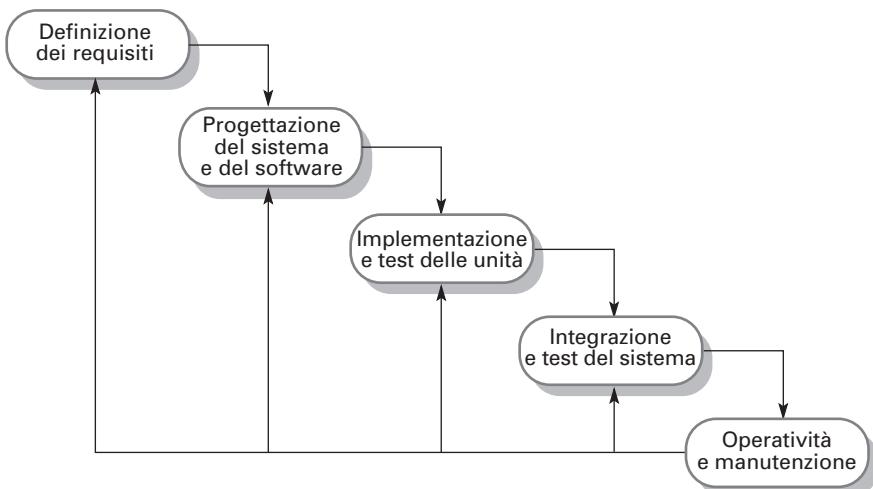


Figura 2.1 Modello a cascata.

I principali stadi del modello a cascata riflettono direttamente le attività di sviluppo fondamentali del software.

1. *Analisi e definizione dei requisiti*: si determinano i servizi del sistema, i suoi vincoli e obiettivi attraverso incontri con gli utenti del sistema. I requisiti vengono definiti in dettaglio e servono come specifica del sistema.
2. *Progettazione del sistema e del software*: il processo di progettazione del sistema suddivide i requisiti sul sistema hardware o sul software e stabilisce l'architettura generale del sistema. La progettazione coinvolge l'identificazione e la descrizione delle astrazioni fondamentali del sistema e le reciproche relazioni.
3. *Implementazione e test delle unità*: in questa fase il progetto del software è realizzato come un insieme di programmi o di unità del programma. Il test delle unità verifica che ognuna soddisfi le proprie specifiche.
4. *Integrazione e test del sistema*: le singole unità di programma o i programmi sono integrati e testati come un sistema completo per accertare che i requisiti del software siano soddisfatti. Dopo il test finale, il sistema è consegnato al cliente.
5. *Operatività e manutenzione*: normalmente (anche se non è obbligatoria), questa è la fase più lunga dell'intero ciclo di vita. Il sistema è installato e messo in opera, si correggono gli errori non scoperti nei primi stadi del ciclo di vita, si migliora l'implementazione delle unità di sistema e si incrementano i servizi del sistema quando vengono evidenziati nuovi requisiti.

In teoria, il risultato di ogni stadio nel modello a cascata è costituito da uno o più documenti approvati (*signed off*, letteralmente “firmati”): la fase successiva non dovrebbe partire prima che sia finita quella precedente. Nello sviluppo dell'hardware, dove i costi di fabbricazione sono elevati, questo ha senso. Nello sviluppo del software, invece, questi stadi si sovrappongono e si scambiano informazioni vicendevolmente. Durante la progettazione vengono identificati problemi e requisiti; durante la codifica vengono individuati i problemi di progettazione e così via. In pratica, il processo software non è mai un modello lineare semplice, ma richiede una sequenza di feedback da uno stadio all'altro.

Quando emergono nuove informazioni in uno stadio del processo, i documenti prodotti negli stadi precedenti devono essere modificati per rispecchiare le modifiche richieste. Per esempio, se si scopre che un requisito è troppo costoso da implementare, il documento dei requisiti deve essere modificato per eliminare tale requisito; ma ciò richiede l'approvazione del cliente e ritarda il processo di sviluppo complessivo.

Di conseguenza, sia i clienti sia gli sviluppatori potrebbero congelare prematuramente la specifica del software in modo che nessuna modifica possa essere apportata. Sfortunatamente, questo significa che i problemi vengono ignorati o sospesi per essere risolti in un successivo momento. Un congelamento prematuro

della specifica potrebbe significare che il sistema non eseguirà ciò che vuole il cliente; inoltre, ciò potrebbe portare a sistemi mal strutturati in quanto i problemi di progettazione vengono elusi dai trucchi di implementazione.

Durante la fase finale del ciclo di vita (operatività e manutenzione), il software viene messo in funzione; vengono scoperti errori e omissioni nei requisiti originali del software; emergono gli errori di programmazione e progettazione e si identificano nuove funzionalità. Il sistema deve essere modificato per rimanere utile; le modifiche necessarie (manutenzione del software) possono richiedere la ripetizione dei precedenti stadi del processo.

In realtà, il software deve essere flessibile e recepire le modifiche richieste durante il suo sviluppo. Il fatto di dover stabilire inizialmente gli obiettivi e di dover rilavorare il sistema quando vengono apportate le modifiche implica che il modello a cascata è appropriato soltanto per alcuni tipi di sistemi:

1. i sistemi integrati, dove il software deve interfacciarsi con i sistemi hardware. A causa della inflessibilità dell'hardware, di solito non è possibile rinviare le decisioni sulle funzionalità del software, finché esso è in fase di implementazione;
2. i sistemi critici, dove occorre un'analisi approfondita della sicurezza e della protezione del software. In questi sistemi, i documenti della specifica e del progetto devono essere completi, in modo che questa analisi sia possibile. La risoluzione dei problemi relativi alla sicurezza di solito è molto costosa nella fase di implementazione.
3. grandi sistemi software che fanno parte di sistemi più complessi sviluppati da più società. L'hardware nei sistemi può essere sviluppato utilizzando un modello simile, e le società trovano più semplice utilizzare un modello comune per l'hardware e il software. Inoltre, quando sono coinvolte più società, potrebbe essere necessario disporre di specifiche complete per consentire lo sviluppo autonomo di sottosistemi differenti.

Il modello a cascata non è appropriato in quei casi in cui sia consentita una comunicazione informale nei team e quando i requisiti del software cambiano rapidamente. Lo sviluppo iterativo e i metodi agili sono più indicati in questi sistemi.

Una variante importante del modello a cascata è lo sviluppo di sistemi formali, dove viene creato un modello matematico della specifica del sistema. Questo modello viene successivamente perfezionato in un codice eseguibile, utilizzando trasformazioni matematiche che ne preservano la coerenza. I processi di sviluppo formale, come quelli basati sul metodo B (Abrial 2005, 2010), sono utilizzati principalmente nello sviluppo di sistemi software che hanno requisiti severi di sicurezza, affidabilità e protezione. L'approccio formale semplifica la produzione di test per dimostrare ai clienti o agli enti di certificazione che il sistema soddisfa veramente tali requisiti. Tuttavia, a causa degli alti costi di sviluppo di una specifica formale, questo modello di sviluppo viene raramente utilizzato, tranne nell'ingegnerizzazione dei sistemi critici.

Modello di processo a spirale di Boehm

Barry Boehm, uno dei pionieri nell'ingegneria del software, ha proposto un modello di processo incrementale che è guidato dai rischi (*risk-driven*). Il processo è rappresentato come una spirale, anziché come una sequenza di attività (Boehm 1988).

Ogni giro della spirale corrisponde a una fase del processo software. Pertanto, il giro più interno può occuparsi della fattibilità del sistema, il successivo della definizione dei requisiti, poi della progettazione del sistema e così via. Il modello a spirale combina la necessità di escludere le modifiche con la tolleranza verso di esse. Il modello suppone che le modifiche siano il risultato dei rischi della progettazione e include esplicite attività di gestione dei rischi per ridurli al minimo.

<http://software-engineering-book.com/web/spiral-model/>

2.1.2 Sviluppo incrementale

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, finché non si ottiene il sistema richiesto (Figura 2.2). Le attività di specifica, sviluppo e convalida sono intrecciate anziché separate, con feedback veloci tra le varie attività.

Lo sviluppo incrementale in qualche forma è adesso l'approccio più comune per sviluppare sistemi di applicazioni e prodotti software. Questo approccio può essere plan-driven, agile o una combinazione di questi approcci. In un approccio plan-driven, gli incrementi del sistema sono stabiliti in anticipo; in un approccio agile, vengono identificati gli incrementi iniziali, ma lo sviluppo dei successivi incrementi dipende dall'avanzamento del lavoro e dalle priorità del cliente.

Lo sviluppo incrementale del software, che è una parte fondamentale dei metodi di sviluppo agili, è migliore dell'approccio a cascata per quei sistemi i cui requisiti è probabile che cambino durante il processo di sviluppo. È questo il caso di molti sistemi aziendali e prodotti software. Lo sviluppo incrementale riflette il modo in cui risolviamo i problemi. Raramente troviamo in anticipo la soluzione

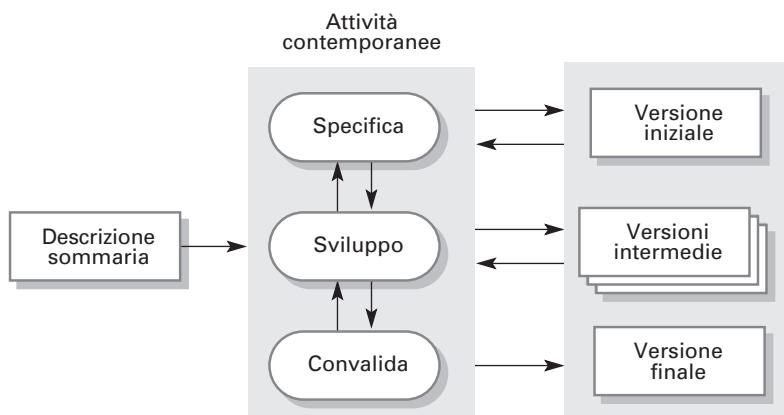


Figura 2.2 Sviluppo incrementale.

finale di un problema, ma arriviamo alla soluzione tramite una serie di passaggi, tornando indietro quando ci accorgiamo di aver commesso un errore. Sviluppando il software in modo incrementale, è più economico e semplice apportare modifiche al software mentre viene sviluppato.

Ciascun incremento o versione del sistema incorpora qualcuna delle funzionalità richieste dal cliente. In generale, gli incrementi iniziali del sistema includono la funzionalità più importante e urgente richiesta dal cliente. Questo significa che il cliente o l'utente può valutare il sistema nelle prime fasi dello sviluppo per vedere se esso realizza ciò che è stato richiesto. In caso contrario, deve essere modificato soltanto l'incremento corrente del sistema e, possibilmente, dovrà essere definita una nuova funzionalità per i successivi incrementi.

Lo sviluppo incrementale offre tre vantaggi principali rispetto al modello a cascata.

1. Il costo di implementazione delle modifiche dei requisiti è ridotto. L'analisi e la documentazione che devono essere ripetute sono significativamente minori rispetto a quelle richieste dal modello a cascata.
2. È più facile ottenere il feedback del cliente sul lavoro di sviluppo che è stato fatto. Il cliente può fare i suoi commenti durante le dimostrazioni del software e vedere quanto è già stato implementato. È difficile per un cliente giudicare l'avanzamento del lavoro dai documenti della progettazione del software.
3. È possibile consegnare in anticipo al cliente una versione utilizzabile del software, anche se non sono state incluse tutte le funzionalità. I clienti sono in grado di utilizzare e valutare il software prima di quanto sia possibile con un processo a cascata.

Da un punto di vista gestionale, l'approccio incrementale ha due problemi.

1. Il processo non è visibile. I manager devono avere delle consegne regolari per misurare i progressi. Se il sistema è sviluppato velocemente, non è economico produrre documentazione che rifletta ogni versione del sistema.
2. La struttura dei sistemi tende a degradarsi quando vengono aggiunti nuovi incrementi. Ogni volta che viene aggiunta una nuova funzionalità, il codice diventa sempre più complicato, quindi diventa sempre più difficile e costoso aggiungere nuove funzionalità a un sistema. Per ridurre il livello di degrado strutturale e di complicazione del codice, i metodi agili suggeriscono un costante refactoring (miglioramento della struttura) del software.

I problemi dello sviluppo incrementale diventano particolarmente gravi per sistemi grandi, complessi e di lunga durata, dove diversi team sviluppano parti differenti di uno stesso sistema. I grandi sistemi richiedono contesti o architetture stabili, e le responsabilità dei vari team che lavorano su differenti parti del sistema devono essere definite in modo chiaro. Questo deve essere pianificato in anticipo, anziché svilupparlo in modo incrementale.

Problemi con lo sviluppo incrementale

Sebbene lo sviluppo incrementale offra molti vantaggi, tuttavia non è completamente privo di problemi. La causa principale delle difficoltà è il fatto che le grandi organizzazioni hanno procedure burocratiche che si evolvono nel tempo e ci possono essere discordanze tra queste procedure e un processo iterativo o agile più informale.

A volte queste procedure sono create per buoni motivi. Per esempio, alcune procedure servono a garantire che il software sia conforme a normative esterne (come la legge federale Sarbanes-Oxley sulla contabilità aziendale negli Stati Uniti). Modificare queste procedure potrebbe non essere possibile e, quindi, sarebbe impossibile evitare conflitti tra processi.

<http://software-engineering-book.com/web/incremental-development/>

Sviluppo incrementale non significa che bisogna consegnare ciascun incremento al cliente del sistema. È possibile sviluppare un sistema in modo incrementale e presentarlo al cliente per avere i suoi commenti, senza necessariamente consegnarlo e installarlo nell’ambiente del cliente. La consegna incrementale (trattata nel Paragrafo 2.3.2) significa che il software viene utilizzato in processi operativi reali, in modo che il feedback del cliente sia più realistico. Purtroppo non è sempre possibile ottenere il feedback, in quanto la sperimentazione del nuovo software potrebbe intralciare i normali processi aziendali.

2.1.3 Integrazione e configurazione

Nella maggior parte dei progetti software si è soliti riutilizzare il software. Ciò avviene spesso in maniera informale; quando le persone che lavorano al progetto si accorgono che un codice già prodotto è simile a quanto richiesto, lo cercano, lo modificano e lo integrano nel proprio sistema.

Questo riutilizzo informale del software avviene indipendentemente dal processo di sviluppo utilizzato. Tuttavia, a partire dal 2000, i processi di sviluppo del software che sfruttano il riutilizzo di software esistente si stanno diffondendo sempre più. L’approccio orientato al riutilizzo si fonda su una larga base di componenti software riutilizzabili e su un framework di integrazione per comporre questi componenti.

Tre tipi di componenti software sono frequentemente utilizzati.

1. Sistemi di applicazioni indipendenti che sono configurati per essere utilizzati in un particolare ambiente. Si tratta di sistemi generici che hanno numerose funzionalità, che devono essere adattate a una specifica applicazione.
2. Collezioni di oggetti che sono sviluppate come un componente o un pacchetto da integrare tramite un framework di integrazione dei componenti, come Java Spring (Wheeler e White 2013).
3. Servizi web che sono sviluppati in conformità agli standard e che sono disponibili per essere utilizzati da siti remoti su Internet.

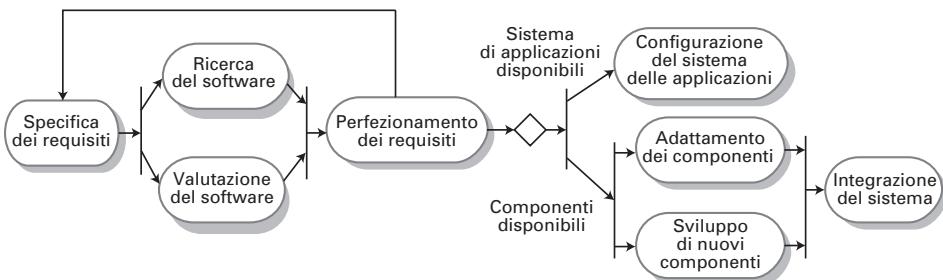


Figura 2.3 Ingegneria del software orientata al riutilizzo.

La Figura 2.3 mostra un modello di processo generico per lo sviluppo basato sul riutilizzo tramite l'integrazione e la configurazione dei componenti. Le fasi di questo processo sono:

1. *specifiche dei requisiti*. Vengono definiti i requisiti iniziali del sistema. Non occorre un'elaborazione dettagliata, ma una breve descrizione dei requisiti e delle funzionalità essenziali del sistema;
2. *ricerca e valutazione del software*. Data la descrizione dei requisiti del software, vengono ricercati i componenti e i sistemi che possono fornire le funzionalità richieste. I componenti e i sistemi candidati vengono valutati per vedere se soddisfano i requisiti essenziali e se sono disponibili per essere utilizzati.
3. *perfezionamento dei requisiti*. Durante questa fase, i requisiti vengono perfezionati utilizzando le informazioni sulle applicazioni e sui componenti riutilizzabili che sono stati trovati. I requisiti vengono modificati in funzione dei componenti disponibili, e la specifica del sistema viene ridefinita. Se le modifiche non sono possibili, l'attività di analisi dei componenti può essere ripetuta per cercare soluzioni alternative.
4. *configurazione del sistema delle applicazioni*. Se è disponibile un sistema di applicazioni pronto all'uso che soddisfa i requisiti, esso può essere configurato per creare il nuovo sistema.
5. *adattamento e integrazione dei componenti*. Se non è disponibile un sistema pronto all'uso, i singoli componenti riutilizzabili possono essere modificati per sviluppare i nuovi componenti, che vengono poi integrati per creare il sistema.

L'ingegneria del software orientata al riutilizzo, basata sulla configurazione e sull'integrazione, ha l'ovvio vantaggio di ridurre la quantità di software da sviluppare, riducendo così i costi e i rischi, e portando anche a consegne più veloci. Sono però inevitabili dei compromessi nei requisiti che possono produrre un sistema che non soddisfa le reali necessità degli utenti. Inoltre si perde il controllo sull'evoluzione del sistema, in quanto le nuove versioni dei componenti riutilizzati non sono sotto il controllo dell'organizzazione che li usa.

Strumenti di sviluppo del software

Gli strumenti di sviluppo del software sono programmi che vengono usati per supportare le attività dei processi di ingegneria del software. Fra essi figurano gli strumenti di gestione dei requisiti, gli editor di testo, gli strumenti di refactoring, i compilatori, i debugger, gli strumenti di bug-tracking (rilevamento di bug) e gli strumenti di building dei sistemi.

Gli strumenti software forniscono il supporto al processo automatizzandone alcune attività e fornendo informazioni sul software che si sta sviluppando. Alcuni esempi di attività che possono essere automatizzate:

- sviluppo di modelli grafici di sistema come parte della specifica dei requisiti o del progetto del software;
- generazione di codice da questi modelli grafici;
- generazione di interfacce utente a partire da una descrizione dell'interfaccia grafica che viene creata interattivamente dall'utente;
- debug di un programma attraverso la raccolta di informazioni sul programma in esecuzione;
- traduzione automatica dei programmi da una vecchia versione di un linguaggio di programmazione a una versione più recente.

Gli strumenti possono essere combinati all'interno di un ambiente di sviluppo interattivo, detto IDE (Interactive Development Environment). Questo ambiente offre una serie di funzionalità che gli strumenti possono utilizzare per semplificare e integrare la loro comunicazione e il loro funzionamento.

<http://software-engineering-book.com/web/software-tools/>

Poiché il riutilizzo del software è molto importante, ho dedicato a questo argomento vari capitoli nella Parte III del libro. I temi generali del riutilizzo del software sono trattati nel Capitolo 15; l'ingegneria del software basato sui componenti è descritta nei Capitoli 16 e 17; i sistemi orientati ai servizi sono descritti nel Capitolo 18.

2.2 Attività di processo

I processi software reali sono sequenze intrecciate di attività tecniche, collaborative e manageriali con l'obiettivo comune di specificare, progettare, implementare e provare un sistema software. In generale, i processi adesso sono supportati da strumenti. Questo significa che gli sviluppatori di software possono usare una ricca gamma di strumenti software come aiuto nel loro lavoro, come i sistemi di gestione dei requisiti, editor dei modelli di progettazione, editor dei programmi, strumenti di prova automatizzati e debugger.

Le quattro attività fondamentali di specifica, sviluppo, convalida ed evoluzione sono organizzate in modo diverso in processi di sviluppo diversi: nel modello a cascata sono organizzate in sequenza, mentre nello sviluppo incrementale sono

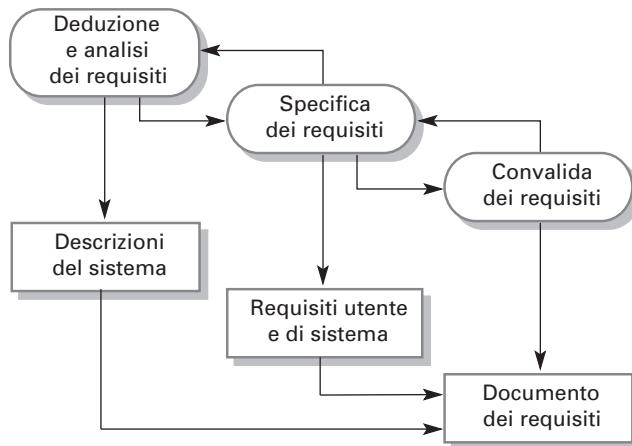


Figura 2.4 Il processo di ingegneria dei requisiti.

intrecciate. Come sono svolte queste attività dipende dal tipo di software che si sta sviluppando, dall’esperienza e la competenza degli sviluppatori e dal tipo di organizzazione che sviluppa il software.

2.2.1 Specifica del software

La creazione delle specifiche del software, o ingegneria dei requisiti, è il processo per capire e definire quali servizi sono richiesti dal sistema, e per identificare i vincoli all’operatività e allo sviluppo del sistema. L’ingegneria dei requisiti è uno stadio particolarmente critico del processo software, poiché gli errori in questa fase portano inevitabilmente a problemi successivi nella progettazione e implementazione del sistema.

Prima di avviare il processo di ingegneria dei requisiti, una società può effettuare uno studio di fattibilità o di mercato per stabilire se esiste un mercato per il software e se sia tecnicamente ed economicamente realistico sviluppare il software richiesto. Uno studio di fattibilità dovrebbe essere rapido e poco costoso; il risultato dovrebbe permettere di decidere se continuare o meno con un’analisi più dettagliata.

Il processo di ingegneria dei requisiti (illustrato nella Figura 2.4) porta alla produzione di un documento di caratteristiche concordate che specifica un sistema che soddisfa i requisiti degli stakeholder.¹ I requisiti di solito sono presentati in due livelli di dettaglio: gli utenti finali e i clienti hanno bisogno di una formulazione dei requisiti di alto livello, mentre gli sviluppatori del sistema devono avere una specifica più dettagliata.

¹ Il termine *stakeholder* identifica ogni persona o gruppo che sarà influenzato dal sistema.

Le fasi principali del processo di ingegneria dei requisiti sono tre.

1. *Deduzione e analisi dei requisiti*: è il processo di deduzione dei requisiti di sistema attraverso l’osservazione di sistemi già esistenti, la discussione con i possibili utenti e acquirenti, l’analisi dei task (compiti) e via dicendo. Può coinvolgere lo sviluppo di uno o più modelli e prototipi, che aiutano gli analisti a capire il sistema da specificare.
2. *Specifica dei requisiti*: è l’attività di tradurre le informazioni raccolte durante la fase di analisi in un documento che definisce un insieme di requisiti. Il documento può includere due tipi di requisiti: i requisiti utente che sono proposizioni astratte dei requisiti del sistema per i clienti e gli utenti finali e i requisiti di sistema che sono una descrizione più dettagliata delle funzionalità che devono essere fornite.
3. *Convalida dei requisiti*: quest’attività controlla che i requisiti siano realistici, coerenti e completi. Durante questo processo vengono inevitabilmente scoperti gli errori nel documento dei requisiti, che dovrà essere modificato per correggere tali errori.

L’analisi dei requisiti continua durante la definizione e la specifica, e nuovi requisiti possono venire alla luce durante il processo; per questo le attività di analisi, definizione e specifica sono intrecciate.

Nei metodi agili, la specifica dei requisiti non è un’attività separata, ma è considerata come parte dello sviluppo del sistema. I requisiti sono specificati in modo informale per ciascun incremento del sistema, appena prima che l’incremento sia sviluppato. I requisiti sono specificati in base alle priorità dell’utente. La deduzione dei requisiti proviene dagli utenti che appartengono al team di sviluppo o lavorano strettamente con esso.

2.2.2 Progettazione e implementazione del software

Lo stadio di implementazione dello sviluppo del software è il processo di conversione delle specifiche in un sistema eseguibile da consegnare al cliente. A volte include processi di progettazione e programmazione; tuttavia, se viene adottato un approccio agile nello sviluppo, la progettazione e l’implementazione sono intrecciate, senza produrre documenti formali di progettazione durante il processo. Ovviamente, il software continua a essere progettato, ma il progetto viene registrato informalmente sulle lavagne e sui notebook del programmatore.

Il progetto del software è una descrizione della struttura del software che si deve implementare, dei modelli e delle strutture di dati usati dal sistema, delle interfacce tra i componenti del sistema e, a volte, tra gli algoritmi usati. I progettisti non ottengono immediatamente un risultato completo, ma sviluppano il progetto in varie fasi; aggiungono dettagli mentre sviluppano il loro progetto, con controlli incrociati costanti per correggere i progetti iniziali.

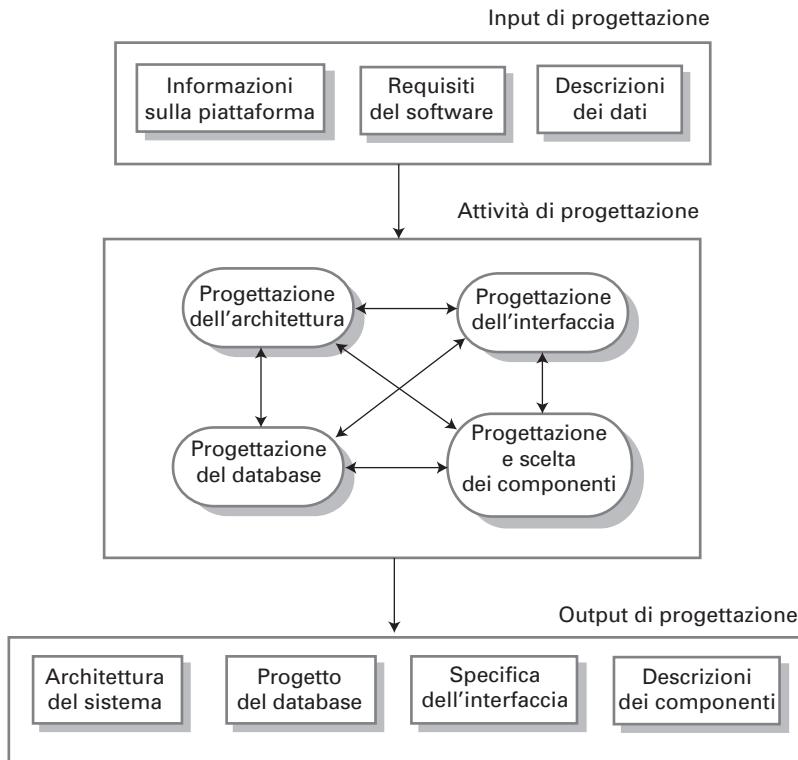


Figura 2.5 Modello generico del processo di progettazione.

La Figura 2.5 è un modello astratto del processo di progettazione e mostra gli input, le attività e gli output del processo. Le attività sono sia intrecciate sia interdipendenti. Nuove informazioni sul progetto vengono costantemente generate, e questo influenza sulle precedenti scelte del progetto; quindi è inevitabile la revisione del progetto.

Molte interfacce software si interfacciano con altri sistemi software. Questi altri sistemi includono il sistema operativo, il database, il middleware (letteralmente “software di mezzo”) e altri sistemi di applicazioni. Tutto questo costituisce la “piattaforma software”, l’ambiente in cui il software sarà eseguito. Le informazioni su questa piattaforma sono l’input essenziale per il processo di progettazione, in quanto i progettisti devono decidere come integrare meglio i loro prodotti con l’ambiente operativo. Se il sistema deve elaborare dati esistenti, allora la loro descrizione può essere inclusa nella specifica della piattaforma; altrimenti, la descrizione dei dati deve essere un input per il processo di progettazione, in modo che possa essere definita l’organizzazione dei dati del sistema.

Le attività nel processo di progettazione variano, in funzione del tipo di sistema che si sta sviluppando. Per esempio, i sistemi in tempo reale richiedono una fase aggiuntiva di progettazione temporale, ma potrebbe non includere un databa-

se, quindi manca la fase di progettazione del database. La Figura 2.5 mostra le quattro attività che possono far parte del processo di progettazione per i sistemi informativi.

1. *Progettazione dell’architettura.* Consiste nell’identificazione della struttura complessiva del sistema, dei componenti principali (detti anche sottosistemi o moduli), delle loro relazioni e della loro distribuzione.
2. *Progettazione del database.* Vengono progettate le strutture dei dati del sistema e come devono essere rappresentate in un database.
3. *Progettazione dell’interfaccia.* Viene definita l’interfaccia tra i componenti del sistema. La specifica dell’interfaccia non deve essere ambigua. Con una definizione chiara dell’interfaccia, un componente può essere usato da altri componenti senza conoscerne la sua implementazione. Una volta approvata la specifica dell’interfaccia, i componenti possono essere progettati e sviluppati separatamente.
4. *Progettazione e scelta dei componenti.* Vengono ricercati i componenti riutilizzabili e, se non sono disponibili componenti idonei, vengono progettati nuovi componenti. Il progetto in questa fase può essere una semplice descrizione di componenti, lasciando al programmatore i dettagli della loro implementazione. In alternativa, potrebbe essere una lista di modifiche da apportare a un componente riutilizzabile o un modello dettagliato di progettazione espresso in UML. Il modello di progettazione può essere successivamente utilizzato per generare automaticamente una implementazione.

Queste attività portano agli output di progettazione, anch’essi mostrati nella Figura 2.5. Per i sistemi critici, gli output del processo di progettazione sono documenti dettagliati che descrivono in modo accurato il sistema. Se si adotta un approccio guidato da modelli (Capitolo 5), gli output sono diagrammi di progettazione. Se si usano metodi agili si sviluppo, gli output potrebbero non essere documenti di specifiche separate, ma potrebbero essere rappresentati nel codice del programma.

Lo sviluppo di un programma per implementare un sistema segue naturalmente la progettazione del sistema. Sebbene alcune classi di programmi, come i sistemi a sicurezza critica, di solito sono progettate dettagliatamente prima di iniziare l’implementazione, è più comune che la progettazione e lo sviluppo del programma siano intrecciati. Gli strumenti di sviluppo del software possono essere utilizzati per generare uno scheletro del programma a partire da un progetto, che include il codice per definire e implementare le interfacce, e al programmatore spesso non resta che aggiungere dettagli all’operatività di ciascun componente del programma.

La programmazione è un’attività individuale e non c’è un processo generico che viene seguito abitualmente. Alcuni programmatore iniziano con i componenti che capiscono, li sviluppano e poi passano ai componenti che comprendono meno; altri usano l’approccio opposto, lasciando i componenti noti alla fine, per-

ché sanno come svilupparli. Ad alcuni sviluppatori piace definire i dati nelle prime fasi del processo per poi usarli come guida nello sviluppo; altri lasciano i dati indefiniti più a lungo possibile.

Normalmente i programmatore testano il codice che hanno sviluppato e in questo modo mettono in luce difetti che devono essere rimossi dal programma; questa procedura è chiamata debug.² I processi di test e di debug sono diversi: il primo stabilisce l'esistenza dei difetti, il secondo si occupa di localizzarli e correggerli.

Quando si esegue il debug, si fanno ipotesi sul comportamento visibile del programma, che vengono poi verificate nella speranza di trovare la causa del problema riscontrato. La verifica delle ipotesi può richiedere il tracciamento manuale del codice del programma, oppure la necessità di scrivere alcuni test case (letteralmente “casi di test”) per localizzare il problema. Il processo di debug può essere aiutato da strumenti interattivi che mostrano i valori intermedi delle variabili del programma e tracciano le istruzioni eseguite.

2.2.3 Convalida del software

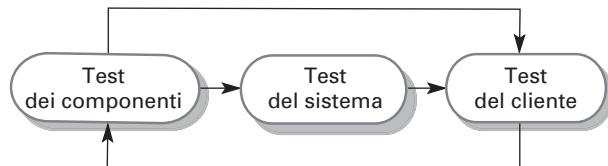
La convalida del software o, più genericamente, la verifica e la convalida (V&V, *Verification and Validation*) è intesa a mostrare che un sistema è conforme alle sue specifiche e soddisfa le aspettative del cliente. Il test dei programmi, dove il sistema viene eseguito utilizzando dati di prova simulati, è la tecnica principale di convalida del software. La convalida può richiedere anche processi di controllo, come le ispezioni e le revisioni, a ogni stadio del processo software, dalla definizione dei requisiti dell'utente allo sviluppo del programma. Tuttavia, la maggior parte del tempo di V&V viene spesa nel test dei programmi.

Tranne che per piccoli programmi, i sistemi non dovrebbero essere testati come un'unica unità monolitica. La Figura 2.6 mostra un processo di test a tre stadi, dove sono testati prima i singoli componenti del sistema e poi il sistema integrato. Per il software di un cliente, il terzo stadio richiede che il sistema sia testato con i dati reali del cliente. Per i prodotti venduti come applicazioni, il terzo stadio, chiamato a volte “alpha test”, prevede che alcuni utenti selezionati provino e commentino il software.

Gli stadi del processo di test sono tre.

1. *Test dei componenti (o delle unità)*. I componenti che formano il sistema vengono testati dalle persone che sviluppano il sistema. Ciascun componente è testato separatamente, senza altri componenti del sistema. I componenti possono essere entità semplici, come funzioni o classi di oggetti, oppure gruppi coerenti di queste entità. Di solito si usano strumenti di automazione, come JUnit per Java, che possono ripetere questi test quando vengono create nuove versioni di un componente (Koskela 2013).

² Letteralmente “rimuovere gli insetti”, si riferisce al termine *bug* che viene usato per indicare i difetti o gli errori di un programma (N.d.T.).

**Figura 2.6** Stadi di test.

2. *Test del sistema.* Si integrano i componenti per formare il sistema completo. Questo processo si occupa di trovare gli errori causati da interazioni impreviste tra i componenti e i problemi con le relative interfacce. Provvede anche a convalidare la conformità dei requisiti, funzionali e non, e a verificare le proprietà significative del sistema. Per i sistemi più grandi, può essere un processo articolato in più stadi, dove i componenti vengono integrati per formare dei sottosistemi testati individualmente, prima di essere integrati per realizzare il sistema finale.
3. *Test del cliente.* È lo stadio finale del processo di test, prima che il sistema venga approvato per la sua messa in uso. Il sistema viene testato dal cliente (o da un potenziale cliente) con i suoi dati, anziché con dati di simulazione. Per il software creato per un particolare cliente, questo test può rivelare errori e omissioni nella definizione dei requisiti del sistema, in quanto i dati reali fanno lavorare il sistema in modo diverso dai dati di simulazione. Il test può anche rivelare problemi con i requisiti, laddove le funzionalità del sistema non soddisfino le necessità dell’utente o le prestazioni siano inaccettabili. Per i prodotti, questo test mostra in quale misura il software soddisfa le esigenze del cliente.

Teoricamente, i difetti dei componenti vengono scoperti nelle prime fasi di test, mentre i problemi con le interfacce vengono scoperti durante l’integrazione del sistema. I difetti scoperti richiedono un debug del programma, e ciò può comportare la ripetizione di altri stadi del processo di test. Gli errori nei componenti del programma, per esempio, possono apparire durante il test del sistema. Il processo è dunque iterativo, con informazioni che dagli stadi finali vengono riportate a quelli iniziali.

Di solito, il test dei componenti è semplicemente una parte del normale processo di sviluppo. I programmatore creano i propri dati di test e verificano il codice in modo incrementale durante lo sviluppo. Il programmatore conosce il componente ed è quindi la persona più adatta a generare i *test case*.

Se viene usato un approccio di sviluppo incrementale, ogni incremento deve essere testato quando viene sviluppato, con alcuni test basati sui requisiti dell’incremento. Nello sviluppo guidato da test, che è una parte normale dei processi agili, i test sono sviluppati insieme ai requisiti, prima che inizi lo sviluppo. Questo aiuta coloro che eseguono i test (*tester*) e gli sviluppatori a capire i requisiti e ad assicurarsi che non ci siano ritardi dovuti alla creazione dei test case.

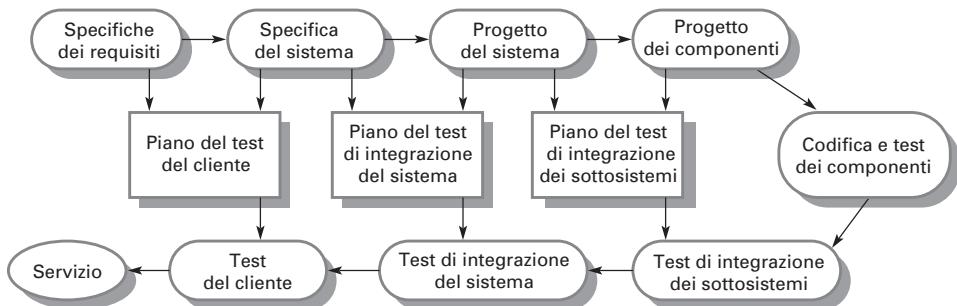


Figura 2.7 Fasi di test in un processo guidato da piani di prova.

Se si usa un processo guidato da piani (per esempio, nello sviluppo di sistemi critici), il test è guidato da una serie di piani di prova. Un team indipendente di tester lavora seguendo questi piani di prova, che sono stati sviluppati dalle specifiche e dalla progettazione del sistema. La Figura 2.7 mostra come questi piani sono collegati alle attività di sviluppo e di test. Questo è anche detto modello-V di sviluppo (la lettera V si vede ruotando lateralmente lo schema). Il modello-V mostra le attività di convalida del software che corrispondono a ciascuna fase del modello a cascata.

Quando un sistema viene venduto come prodotto software, si utilizza un processo di test chiamato “beta test”, che comporta la consegna del sistema a una serie di potenziali clienti che accettano di usarlo per un certo periodo e riferiscono i problemi agli sviluppatori. Questo sottopone il prodotto all’uso reale e svela errori che potrebbero non essere stati previsti dagli sviluppatori del software. Dopo questa fase di feedback, il prodotto viene modificato e rilasciato per ulteriori beta test o per la vendita.

2.2.4 Evoluzione del software

La flessibilità dei sistemi software è una delle ragioni principali per cui sempre più il software viene incorporato in sistemi grandi e complessi. Una volta presa una decisione sulla fabbricazione di un componente hardware, successivamente è molto costoso apportare delle modifiche al progetto di tale componente. Il software, invece, può essere modificato in qualsiasi momento, durante o dopo lo sviluppo del sistema; anche le grandi modifiche sono sempre molto più economiche delle corrispondenti modifiche dell’hardware.

Storicamente c’è sempre stata una divisione tra i processi di sviluppo e di evoluzione (o manutenzione) del software. Le persone pensano che lo sviluppo del software sia un’attività creativa, nella quale partendo da un concetto iniziale si arriva allo sviluppo di un sistema funzionante. A volte si ritiene che la manutenzione del software sia più semplice e meno interessante dello sviluppo originario del software.

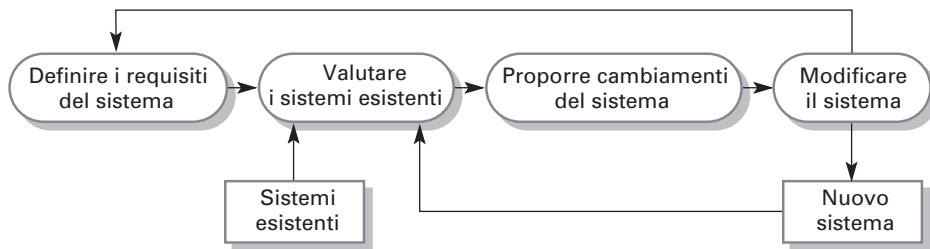


Figura 2.8 Evoluzione di un sistema software.

Questa distinzione tra sviluppo e manutenzione del software sta diventando sempre più irrilevante. Sono pochi i sistemi software completamente nuovi ed è quindi più sensato vedere lo sviluppo e la manutenzione come un tutt'uno. Anziché come due processi distinti, è più realistico considerare l'ingegneria del software come un unico processo evolutivo (Figura 2.8), in cui il software viene modificato continuamente nel corso della sua vita per adeguarlo ai cambiamenti dei requisiti e delle necessità dei clienti.

2.3 Far fronte ai cambiamenti

I cambiamenti sono inevitabili in tutti i grandi progetti software. I requisiti del sistema cambiano quando la società che ha acquistato il sistema reagisce alle pressioni esterne, alla concorrenza e alle nuove priorità di gestione. Cambiano i progetti e le implementazioni con l'arrivo di nuove tecnologie. Ne consegue che, indipendentemente dal modello di processo utilizzato, è essenziale che esso consenta di adeguare il software che si sta sviluppando ai cambiamenti in corso.

I cambiamenti aggiungono costi allo sviluppo del software, perché di solito essi richiedono che il lavoro svolto deve essere rifatto; tutto questo è chiamato *rilavorazione* del progetto. Per esempio, se le relazioni fra i requisiti in un sistema sono state analizzate e sono stati identificati nuovi requisiti, una parte o tutta l'analisi dei requisiti deve essere ripetuta. Potrebbe essere necessario riprogettare il sistema per soddisfare i nuovi requisiti, modificare tutti i programmi che sono stati sviluppati e ripetere i test del sistema.

Per ridurre i costi di rilavorazione si usano due approcci correlati:

1. *Anticipazione dei cambiamenti*, quando il processo software include attività che possono anticipare o predire possibili variazioni, prima che sia richiesta una significativa rilavorazione. Per esempio, si potrebbe sviluppare un sistema prototipo per mostrare alcune caratteristiche chiave del sistema ai clienti, che possono provare il prototipo e perfezionare i loro requisiti, prima di impegnarsi in alti costi di produzione del software.
2. *Tolleranza ai cambiamenti*, quando il processo e il software sono progettati in modo che le modifiche possano essere facilmente apportate al sistema.

Questo di solito implica qualche forma di sviluppo incrementale. Le modifiche proposte possono essere implementate tramite incrementi che non sono stati ancora sviluppati. Se questo non è possibile, allora potrebbe essere necessario cambiare soltanto un singolo incremento (una piccola parte del sistema) per includere la modifica.

In questo paragrafo descriverò due metodi per far fronte ai cambiamenti dei requisiti del sistema.

1. *Prototipazione del sistema.* Una versione del sistema o una sua parte vengono sviluppate rapidamente per verificare i requisiti del cliente e la fattibilità delle scelte di progettazione. Questo è un metodo di anticipazione dei cambiamenti, in quanto consente agli utenti di provare il sistema prima della consegna e, quindi, di perfezionare i loro requisiti. Il numero delle proposte di modifica dei requisiti fatte dopo la consegna è quindi molto probabile che si riduca.
2. *Consegna incrementale.* Gli incrementi del sistema vengono consegnati al cliente per essere commentati e provati. Questo combina la necessità di escludere le modifiche con la tolleranza verso di esse, in quanto evita l'approvazione prematura dei requisiti per l'intero sistema e consente alle modifiche di essere incluse in successivi incrementi a costi relativamente bassi.

Il refactoring (miglioramento della struttura e dell'organizzazione di un programma) è un altro importante meccanismo che supporta la tolleranza ai cambiamenti. Il concetto di refactoring è trattato nel Capitolo 3.

2.3.1 Prototipazione

Un prototipo è la versione iniziale di un sistema software che viene utilizzata per dimostrare concetti, provare opzioni di progettazione e scoprire più informazioni sui problemi e sulle possibili soluzioni. Lo sviluppo rapido e iterativo del prototipo è essenziale per tenere sotto controllo i costi e permettere agli stakeholder del sistema di sperimentarlo nelle fasi iniziali del processo software.

Un prototipo può essere utilizzato nel processo di sviluppo software come aiuto per anticipare le modifiche che potrebbero essere richieste.

1. Nel processo di ingegneria dei requisiti, un prototipo può aiutare a identificare e convalidare i requisiti del sistema.
2. Nel processo di progettazione del sistema, un prototipo può essere utilizzato per esplorare particolari soluzioni software e durante lo sviluppo di un'interfaccia utente per il sistema.

I prototipi permettono agli utenti di vedere come il sistema supporta il loro lavoro; possono suggerire nuove idee per i requisiti e facilitare la scoperta dei punti di forza e di debolezza del software. Possono quindi scaturire proposte di nuovi requisiti. Inoltre, durante lo sviluppo del prototipo, possono emergere gli errori e le omissioni nei requisiti proposti. Una funzione descritta nelle specifiche potreb-

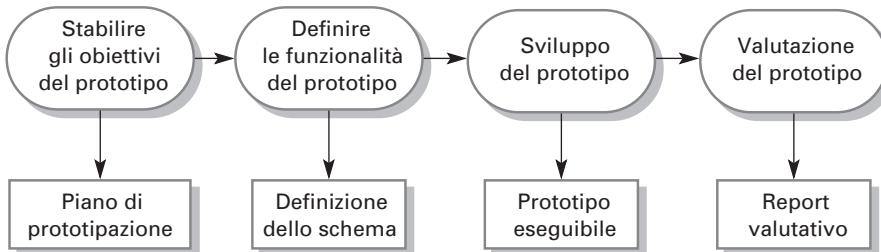


Figura 2.9 Sviluppo dei prototipi.

be sembrare utile e ben definita, ma quando questa funzione viene unita ad altre funzioni, gli utenti potrebbero accorgersi che il loro giudizio iniziale era sbagliato o incompleto. Le specifiche del sistema possono essere dunque modificate per riflettere la mutata comprensione dei requisiti.

Un prototipo può essere utilizzato mentre si sta sviluppando il sistema per eseguire esperimenti di progettazione e per verificare la fattibilità di un progetto proposto. Per esempio il progetto di un database può essere prototipato e testato per verificare che permetta un accesso efficiente ai dati per le interrogazioni più comuni da parte degli utenti. Per questo la prototipazione rapida, con il coinvolgimento degli utenti finali, è l'unico modo sicuro per sviluppare interfacce grafiche utente. A causa della natura dinamica delle interfacce utente, le descrizioni testuali e i diagrammi non sono così efficaci per esprimere i requisiti di interfaccia utente.

Un modello di processo per lo sviluppo di prototipi è mostrato nella Figura 2.9. Gli obiettivi della prototipazione dovrebbero essere resi esplicativi all'inizio del processo. Questi obiettivi possono essere sviluppare un'interfaccia utente, sviluppare un sistema per convalidare i requisiti funzionali di sistema o sviluppare un sistema per mostrare l'applicazione ai manager. Lo stesso prototipo di solito non raggiunge tutti questi obiettivi. Se gli obiettivi non sono dichiarati, il management o gli utenti finali possono fraintendere la funzione del prototipo e, di conseguenza, non possono ricevere dallo sviluppo del prototipo i benefici attesi.

Il passo successivo è decidere cosa inserire e, soprattutto, cosa lasciar fuori dal prototipo del sistema. Per ridurre i costi di prototipazione e accelerare i tempi di consegna, si possono trascurare alcune funzionalità, per esempio i requisiti non funzionali, come i tempi di risposta e l'utilizzazione della memoria. La gestione degli errori può essere ignorata, a meno che l'obiettivo del prototipo non sia stabilire un'interfaccia utente. Possono essere ridotti gli standard di affidabilità e di qualità del programma.

Lo stadio finale del processo è la valutazione del prototipo. Si deve provvedere alla formazione degli utenti e utilizzare gli obiettivi del prototipo per definire un piano di valutazione. I potenziali utenti hanno bisogno di tempo per trovarsi a loro

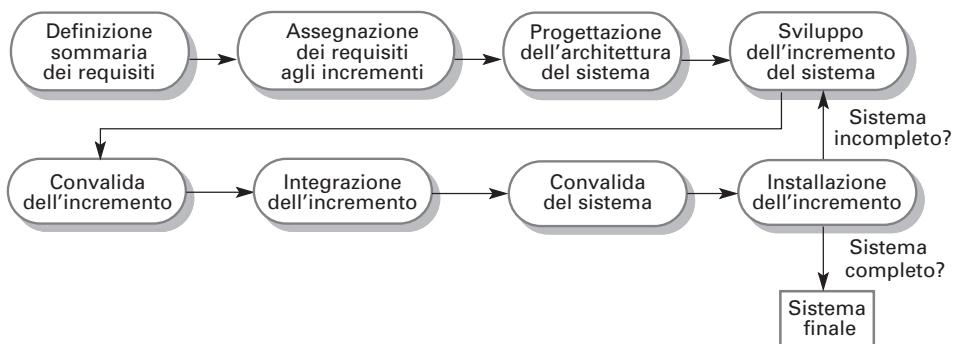


Figura 2.10 Consegna incrementale.

agio con il nuovo sistema e per abituarsi a un normale schema di utilizzo. Una volta che utilizzano il sistema normalmente, gli utenti possono scoprire errori e omissioni nei requisiti. Un problema generale della prototipazione è che gli utenti non possono usare il prototipo nello stesso modo in cui usano il sistema finale. Chi prova il prototipo potrebbe non essere un tipico utente del sistema. Il periodo di formazione degli utenti durante la valutazione del prototipo potrebbe essere insufficiente. Se il prototipo è lento, gli utenti potrebbero correggere il loro modo di lavorare ed evitare quelle funzionalità del sistema che hanno tempi di risposta lenti. Quando gli utenti riceveranno risposte migliori nel sistema finale, potranno utilizzarlo in modo diverso.

2.3.2 Consegnare incrementalmente

La consegna incrementale (Figura 2.10) è un approccio allo sviluppo del software dove alcuni degli incrementi sviluppati sono consegnati al cliente e installati per essere usati nel loro ambiente di lavoro. In un processo di consegna incrementale, il cliente definisce i servizi che il sistema deve fornire, indicando quali sono più importanti e quali meno. Viene poi definito un numero di incrementi, in modo che ogni incremento fornisca un sottoinsieme delle funzionalità del sistema. L'assegnazione dei servizi agli incrementi dipende dalla priorità del servizio: un servizio con priorità più alta sarà implementato e consegnato prima.

Una volta identificati gli incrementi del sistema, vengono definiti in dettaglio i requisiti del primo incremento, che viene subito sviluppato. Durante lo sviluppo viene svolta l'analisi dei requisiti per gli incrementi successivi, ma non vengono accettati cambiamenti per i requisiti dell'incremento in corso di sviluppo.

Dopo che l'incremento è completato e consegnato, può essere installato nell'ambiente usuale di lavoro del cliente; in questo modo, il cliente può provarlo nel sistema e definire i requisiti per i successivi incrementi. Quando i nuovi incrementi sono completati, vengono integrati con quelli esistenti in modo da migliorare le funzionalità del sistema ad ogni consegna.

La consegna incrementale ha diversi vantaggi.

1. I clienti possono usare i primi incrementi come prototipi e acquisire esperienza per perfezionare i requisiti degli incrementi successivi. Diversamente dai prototipi, gli incrementi sono parti del sistema reale, quindi non occorre addestrare di nuovo l’utente quando il sistema completo è pronto.
2. I clienti non devono aspettare che il sistema completo sia consegnato prima di poterne trarre vantaggio. Il primo incremento soddisfa i loro requisiti più critici, quindi possono utilizzare il software immediatamente.
3. Il processo conserva i benefici dello sviluppo incrementale, in quanto è relativamente facile incorporare le modifiche nel sistema.
4. Poiché i servizi con priorità più alta sono consegnati prima e gli incrementi successivi sono integrati con questi, i servizi di sistema più importanti sono testati più intensamente degli altri. Questo significa che i clienti hanno meno probabilità di sperimentare fallimenti del software nelle parti più importanti del sistema.

La consegna incrementale presenta anche alcuni problemi. In pratica, essa funziona soltanto nei casi in cui si vuole introdurre un sistema completamente nuovo e le persone che valutano il sistema hanno tempo sufficiente per sperimentare il nuovo sistema. I problemi chiave di questo approccio sono i seguenti.

1. La consegna iterativa è problematica quando un nuovo sistema deve rimpiazzare un sistema esistente. Gli utenti hanno bisogno di tutte le funzionalità del vecchio sistema e di solito sono riluttanti a sperimentare un sistema completamente nuovo. Spesso non è praticabile usare insieme il vecchio e il nuovo sistema, in quanto è probabile che usino differenti database e interfacce utente.
2. Molti sistemi richiedono una serie di funzionalità di base che sono utilizzate da parti differenti del sistema. Poiché i requisiti non sono definiti nei dettagli finché c’è un incremento da implementare, può essere difficile identificare le funzionalità comuni che sono richieste da tutti gli incrementi.
3. L’essenza dei processi iterativi è che la specifica viene sviluppata insieme al software. Tuttavia, questo è in conflitto con il modello di acquisizione di molte organizzazioni, dove la specifica del sistema completo è parte del contratto di sviluppo del software. Nell’approccio incrementale, non c’è la specifica del sistema completo finché non viene definito l’ultimo incremento. Ciò richiede una nuova forma di contratto, che i grandi clienti, come le agenzie governative, difficilmente riescono ad accettare.

Per alcuni tipi di sistemi, lo sviluppo e la consegna incrementalni non sono l’approccio migliore. Per esempio i sistemi molto grandi dove lo sviluppo può richiedere team che lavorano in posti diversi, oppure alcuni sistemi integrati dove il software dipende dallo sviluppo hardware, e alcuni sistemi critici dove tutti i

requisiti devono essere analizzati per verificare le interazioni che possono compromettere la sicurezza o la protezione del sistema.

Ovviamente, anche questi sistemi hanno gli stessi problemi di incertezza e di requisiti mutevoli. Per risolverli e trarre alcuni benefici dello sviluppo incrementale, si può sviluppare un prototipo e utilizzare una piattaforma per sperimentare i requisiti e la progettazione del sistema. Con l'esperienza acquisita con questo prototipo è possibile concordare i requisiti definitivi.

2.4 Miglioramento dei processi

Al giorno d'oggi, c'è una costante richiesta di un software migliore e più economico, che deve essere consegnato a scadenze sempre più stringenti. Di conseguenza molte società di software sono particolarmente interessate al miglioramento dei processi software come strumento per migliorare la qualità del proprio software, per ridurre i costi o per accelerare i processi di sviluppo. Miglioramento dei processi significa comprendere i processi esistenti e modificarli per aumentare la qualità dei prodotti e/o diminuire i costi e i tempi di sviluppo. Il Capitolo 26 tratta dettagliatamente i problemi generali connessi alle misurazioni e al miglioramento dei processi.

Per cambiare e migliorare i processi si usano due differenti approcci.

1. *Approccio di maturità del processo.* Si basa sul miglioramento della gestione dei processi e dei progetti e sull'introduzione di buone pratiche di ingegneria del software nelle organizzazioni. Il livello di maturità dei processi riflette la misura in cui le buone pratiche tecniche e gestionali sono state adottate nei processi organizzativi di sviluppo del software. Gli obiettivi principali di questo approccio sono migliorare la qualità del prodotto e la prevedibilità dei processi.
2. *Approccio agile.* Si basa sullo sviluppo iterativo e sulla riduzione degli overhead nei processi software. Le caratteristiche principali dei metodi agili sono una rapida consegna di funzionalità e una prontezza di risposta ai cambiamenti dei requisiti del cliente. La filosofia del miglioramento qui consiste nel fatto che i migliori processi sono quelli con i più bassi overhead, e gli approcci agili consentono di fare questo. I metodi agili sono descritti nel Capitolo 3.

Le persone che sono entusiaste di uno di questi approcci sono generalmente scettiche sui vantaggi dell'altro approccio. L'approccio di maturità del processo è radicato nello sviluppo guidato da piani e di solito richiede maggiori "overhead", nel senso che vengono introdotte attività che non sono direttamente rilevanti per lo sviluppo del programma. Gli approcci agili si focalizzano sul codice da sviluppare e riducono al minimo deliberatamente le formalità e la documentazione.

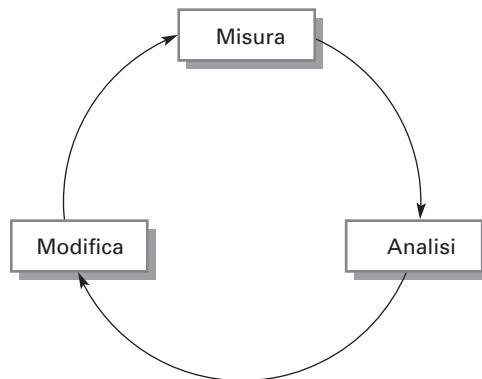


Figura 2.11 Il ciclo di miglioramento dei processi.

Il processo generale di miglioramento dei processi che sta alla base dell’approccio di maturità è un’attività ciclica, come mostra la Figura 2.11. Gli stadi in questo processo sono:

1. *misurazione del processo*: si misurano uno o più attributi del prodotto o processo software. Queste misure formano la base per decidere se i miglioramenti sono stati efficaci. Ogni volta che vengono introdotti dei miglioramenti, si misurano di nuovo gli stessi attributi, che dovrebbero essere migliori dei precedenti;
2. *analisi del processo*: viene valutato il processo corrente e vengono identificati i punti deboli e i colli di bottiglia; durante questo stadio di solito vengono sviluppati i modelli di processo che descrivono il processo. L’analisi potrebbe focalizzarsi su determinate caratteristiche dei processi, quali la rapidità e la robustezza;
3. *modifica del processo*: vengono proposte alcune modifiche del processo per eliminare i punti deboli identificati durante l’analisi. Una volta apportate le modifiche, il ciclo ricomincia, e si raccolgono i dati per valutare l’efficacia delle modifiche.

Senza dati concreti su un processo o sul software sviluppato che usa tale processo, è impossibile stabilire l’entità del miglioramento del processo. Purtroppo, le aziende che avviano il processo di miglioramento difficilmente hanno a disposizione i dati sul processo che servono come base iniziale di valutazione del miglioramento. Pertanto, come parte del primo ciclo di modifiche, potrebbe essere necessario acquisire i dati sul processo software e misurare le caratteristiche del prodotto software.

Il miglioramento dei processi è un’attività a lungo termine, quindi ogni stadio del processo di miglioramento potrebbe durare parecchi mesi. È anche un’attività continua poiché, indipendentemente dal nuovo processo introdotto, l’ambiente

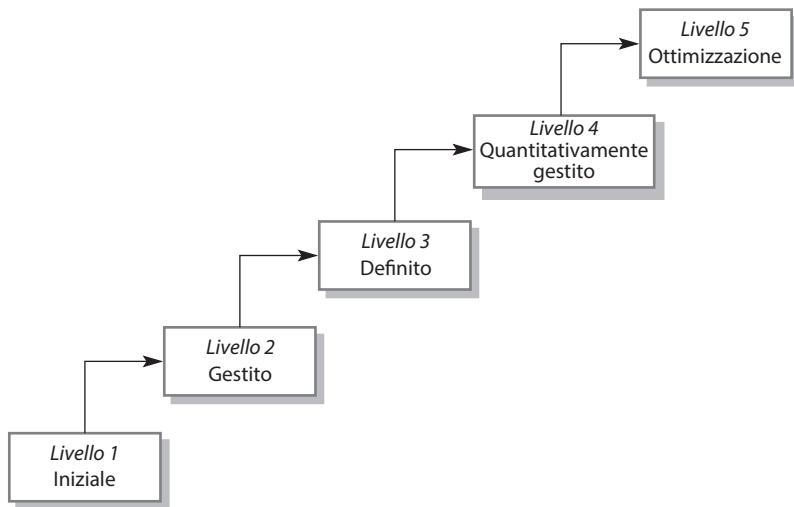


Figura 2.12 Livelli della maturità delle capacità dei processi.

operativo delle aziende cambia e, quindi, anche i nuovi processi dovranno evolversi per tenere conto di questi cambiamenti.

Il concetto di maturità dei processi fu introdotto alla fine degli anni '80, quando l'Istituto di ingegneria del software (SEI, *Software Engineering Institute*) di Pittsburgh propose il suo modello di maturità delle capacità dei processi (Humphrey 1988). La maturità dei processi di una società di software rispecchia la gestione e la misurazione dei processi, e l'uso di buone pratiche di ingegneria del software all'interno della società. Questa idea venne introdotta in modo che il Dipartimento della Difesa U.S.A. potesse valutare le capacità di ingegneria del software dei suoi appaltatori, con l'obiettivo di limitare i contratti soltanto a quegli appaltatori che avessero raggiunto un determinato livello di maturità dei processi. Furono proposti cinque livelli di maturità dei processi, come mostra la Figura 2.12. Questi livelli si sono evoluti negli ultimi 25 anni (Chrissis, Konrad e Shrum 2011), ma i concetti fondamentali del modello di Humphrey sono ancora alla base del processo di valutazione della maturità dei processi software.

I livelli del modello di maturità delle capacità dei processi sono:

1. *iniziale*: gli obiettivi associati all'area dei processi sono soddisfatti, e per tutti i processi lo scopo del lavoro da svolgere è esplicitamente definito e comunicato a membri del team;
2. *gestito*: a questo livello, gli obiettivi associati all'area dei processi sono soddisfatti, e le politiche organizzative in atto definiscono quando ciascun processo deve essere utilizzato. Ci devono essere dei piani di progettazione documentati che definiscono gli obiettivi dei progetti. Le procedure per la gestione delle risorse e il monitoraggio dei processi devono essere in atto in tutta l'istituzione;

3. *definito*: questo livello si focalizza sulla standardizzazione e sull'installazione dei processi organizzativi. Ciascun progetto ha un processo gestito che viene adattato ai requisiti dei progetti sulla base di una serie definita di processi organizzativi. I giudizi e le misurazioni dei processi devono essere raccolti e utilizzati per i miglioramenti futuri dei processi;
4. *quantitativamente gestito*: a questo livello c'è una responsabilità organizzativa nell'uso di metodi statistici o quantitativi per controllare i sottoprocessi; ovvero le misurazioni effettuate su processi e prodotti devono essere utilizzate nella gestione di processi;
5. *ottimizzazione*: a questo livello massimo l'organizzazione deve utilizzare le misurazioni su processi e prodotti per guidare il miglioramento dei processi. Devono essere analizzati i trend, e i processi devono essere adattati alle mutate esigenze delle aziende.

Il lavoro sui livelli di maturità dei processi ha avuto un forte impatto sull'industria del software. Ha focalizzato l'attenzione sui processi e sulle pratiche di ingegneria del software e ha portato significativi miglioramenti nelle capacità dell'ingegneria del software. Tuttavia, ci sono troppi overhead nel miglioramento formale dei processi per le piccole società di software, ed è difficile stimare la maturità con i processi agili. Di conseguenza, soltanto le grandi società di software adesso usano questo approccio focalizzato sulla maturità per migliorare i processi software.

Punti chiave

- I processi software sono le attività coinvolte nella produzione di un sistema software. I modelli di processi software sono rappresentazioni astratte di tali processi.
- I modelli generici dei processi descrivono l'organizzazione dei processi software. Esempi di tali modelli includono il modello a cascata, lo sviluppo incrementale, e la configurazione e l'integrazione dei componenti riutilizzabili.
- L'ingegneria dei requisiti è il processo che sviluppa una specifica del software. Le specifiche servono a comunicare le esigenze del cliente agli sviluppatori del sistema software.
- I processi di progettazione e implementazione si occupano di trasformare le specifiche dei requisiti in un sistema software eseguibile.
- La convalida del software è il processo che verifica il rispetto delle specifiche di sistema e delle effettive necessità degli utenti del sistema.
- L'evoluzione del software si occupa di modificare i sistemi esistenti per soddisfare nuovi requisiti. I cambiamenti sono continui, e il software deve evolversi per conservare la sua utilità.
- I processi devono includere speciali attività che fanno fronte ai cambiamenti. Ciò potrebbe richiedere una fase di prototipazione che aiuta a evitare decisioni inappro-

priate sui requisiti e i progetti. I processi possono essere strutturati per lo sviluppo e la consegna iterativi in modo che le modifiche possano essere apportate senza interferire con l'intero sistema.

- Il miglioramento dei processi è un processo che permette di migliorare la qualità del software esistente, ridurre costi e tempi di sviluppo. È un'attività ciclica che richiede misurazioni, analisi e modifiche dei processi.

Esercizi

- * 2.1 Motivando la vostra risposta in base al tipo di sistema che si sta sviluppando, suggerite i modelli generici di processo software più adatti per gestire lo sviluppo dei seguenti sistemi:
 - un sistema per controllare l'antibloccaggio dei freni in un'autovettura;
 - un sistema di realtà virtuale per supportare la manutenzione del software;
 - un sistema di contabilità universitaria che sostituisce il sistema esistente;
 - un sistema interattivo di pianificazione dei viaggi che permette agli utenti di programmare i viaggi con il minore impatto ambientale.
- 2.2 Spiegate perché lo sviluppo incrementale è l'approccio più efficace per sviluppare sistemi software aziendali. Perché questo modello è il meno appropriato per l'ingegnerizzazione dei sistemi in tempo reale?
- * 2.3 Considerate il modello dei processi di integrazione e configurazione illustrato nella Figura 2.3. Spiegate perché questo modello è essenziale per ripetere l'attività di ingegneria dei requisiti nel processo.
- * 2.4 Indicate perché, nel processo di ingegneria dei requisiti, è importante distinguere lo sviluppo dei requisiti dell'utente da quello del sistema.
- 2.5 Spiegate con un esempio perché le attività di progettazione dell'architettura, del database, dell'interfaccia e dei componenti del sistema sono interdipendenti.
- 2.6 Spiegate perché il test del software dovrebbe essere sempre un'attività incrementale in più fasi. I programmati sono le persone più adatte a provare i programmi che hanno sviluppato?
- * 2.7 Spiegate perché i cambiamenti sono inevitabili nei sistemi complessi e indicate alcuni esempi (a parte la prototipazione e la consegna incrementale) di attività di processi software che possono aiutare a prevedere i possibili cambiamenti e rendere il software che si sta sviluppando più resiliente alle modifiche.
- 2.8 Avete sviluppato il prototipo di un sistema software e il vostro direttore è rimasto particolarmente impressionato da esso. Vi propone di metterlo subito in uso come sistema di produzione, aggiungendo le nuove funzionalità richieste. In questo modo si evitano le spese di sviluppo del sistema e si rende il sistema immediatamente utile. Scrivete un breve rapporto per spiegare al vostro direttore perché i prototipi dei sistemi di solito non dovrebbero essere utilizzati come sistemi di produzione.
- * 2.9 Indicate due vantaggi e due svantaggi dell'approccio alla valutazione e al miglioramento dei processi che è incluso nel modello di maturità delle capacità creato dal SEI.

2.10 Storicamente, l'introduzione della tecnologia ha causato profondi cambiamenti nel mercato del lavoro e, almeno temporaneamente, ha fatto perdere il lavoro ad alcune persone. Discutete se l'introduzione di una tecnologia avanzata di automazione dei processi avrebbe le stesse conseguenze per gli ingegneri del software. Se la risposta è negativa, spiegate perché. Se invece pensate che potrebbero essere ridotti le opportunità di lavoro, spiegate se è etico per gli ingegneri coinvolti resistere, passivamente o attivamente, all'introduzione di questa nuova tecnologia.

* *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Process Models in Software Engineering.” Un'eccellente panoramica sui modelli di ingegneria del software che sono stati proposti. (W. Scacchi, *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley & Sons, 2001) <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>

Software Process Improvement: Results and Experience from the Field. Questo libro è una raccolta di articoli che trattano casi di studio di miglioramento dei processi in varie aziende norvegesi di piccola e media dimensione. Include anche una buona introduzione ai problemi generali del miglioramento dei processi (Conradi, R., Dybå, T., Sjøberg, D. e Ulsund, T. (eds.), Springer, 2006).

“Software Development Life Cycle Models and Methodologies.” Questo post è una breve sintesi di numerosi modelli di processi software che sono stati proposti e utilizzati. Descrive vantaggi e svantaggi di ciascuno di questi modelli (M. Sami 2012). <http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/>

3

Sviluppo agile del software

L'obiettivo di questo capitolo è descrivere i metodi di sviluppo agile del software. Dopo aver letto questo capitolo:

- capirete la logica dei metodi di sviluppo agile del software, il manifesto per lo sviluppo agile e le differenze tra sviluppo agile e sviluppo guidato da piani;
- conoscerete le principali pratiche di sviluppo agile, quali le storie utente, il refactoring, la programmazione a coppie e lo sviluppo con test iniziali;
- capirete l'approccio Scrum per la gestione agile della progettazione;
- capirete i problemi per scalare i metodi di sviluppo agile e per combinare gli approcci agili con quelli guidati da piani nello sviluppo di grandi sistemi software.

- 3.1 Metodi agili
- 3.2 Tecniche di sviluppo agile
- 3.3 Gestione agile della progettazione
- 3.4 Scalabilità dei metodi agili

Le aziende oggi lavorano in un ambiente globale dal cambiamento rapido; devono cogliere nuove opportunità, e rispondere a nuovi mercati, alle condizioni economiche variabili e alla presenza di prodotti e servizi concorrenti. Il software è parte di quasi tutte le operazioni aziendali, quindi è essenziale che quello nuovo sia sviluppato velocemente per trarre vantaggio dalle nuove opportunità e per rispondere alle pressioni concorrenziali. Oggi la rapidità dello sviluppo e della consegna è quindi il requisito più critico per la maggior parte dei sistemi software aziendali. In effetti, molte aziende sono disposte ad accettare compromessi sulla qualità e sui requisiti pur di avere una consegna rapida del software.

Poiché queste aziende operano in un ambiente in continua evoluzione, spesso è praticamente impossibile ottenere un insieme completo di requisiti stabili. I requisiti cambiano perché per i clienti è impossibile prevedere come un sistema influenzerà le pratiche operative, come interagirà con gli altri sistemi e quali operazioni degli utenti dovranno essere automatizzate. I requisiti reali diventano chiari solo dopo che il sistema è stato consegnato e utilizzato dagli utenti; ma, successivamente, fattori esterni inducono a modificare i requisiti.

I processi di sviluppo del software guidati da piani, che si basano sulla completa specifica dei requisiti e sulla progettazione, costruzione e verifica del sistema, non sono adatti allo sviluppo rapido del software, poiché se i requisiti cambiano o creano problemi, la progettazione o l'implementazione del sistema deve essere eseguita e verificata nuovamente. Di conseguenza, un comune processo a cascata o basato su specifiche, di solito, si prolunga fino a determinare la consegna del software finale al cliente oltre i tempi inizialmente previsti.

Per alcuni tipi di software, come i sistemi di controllo a sicurezza critica, dove è essenziale un'analisi completa del sistema, questo approccio guidato da piani è quello appropriato. Tuttavia, in un ambiente aziendale in rapida evoluzione, questo può creare problemi. Nel momento in cui il software è disponibile per l'uso, la motivazione iniziale del suo acquisto può essere mutata in modo così radicale da rendere il software effettivamente inutile. Per questa ragione, specialmente per i sistemi aziendali, sono essenziali i processi che si concentrano sulla rapidità di sviluppo e consegna del software.

L'esigenza di uno sviluppo rapido del software e di processi che siano in grado di gestire la variabilità dei requisiti è nota già da molti anni (Larman e Basili 2003). Lo sviluppo rapido del software è effettivamente decollato alla fine degli anni '90 con l'idea dei "metodi agili", come Extreme Programming (Beck 1999), Scrum (Schwaber e Beedle 2001) e DSDM (Stapleton 2003).

Lo sviluppo rapido del software acquisì notorietà come "sviluppo agile" e con i "metodi agili"; questi metodi sono progettati per produrre rapidamente software utile. Tutti i metodi agili che sono stati proposti hanno alcune caratteristiche comuni.

1. I processi di specifica, progettazione e implementazione sono intrecciati. Non c'è una specifica di sistema dettagliata, e la documentazione dei pro-

getti è minimizzata o prodotta automaticamente dall’ambiente di programmazione utilizzato per implementare il sistema. Il documento dei requisiti degli utenti definisce solo le caratteristiche più importanti del sistema.

2. Il sistema viene sviluppato in una serie di incrementi. Gli utenti finali e gli altri stakeholder del sistema sono coinvolti nella specifica e nella valutazione di ogni incremento. Possono proporre modifiche al software e nuovi requisiti che dovranno essere implementati in una versione successiva del sistema.
3. Una ricca gamma di strumenti supporta il processo di sviluppo; per esempio, gli strumenti per i test automatici, gli strumenti per la gestione della configurazione, gli strumenti per l’integrazione del sistema e per la produzione automatica delle interfacce utente.

I metodi agili sono metodi dello sviluppo incrementale, nei quali gli incrementi sono piccoli e, tipicamente, le nuove release del sistema vengono create e messe a disposizione dei clienti ogni due o tre settimane. I clienti vengono coinvolti nel processo di sviluppo per ottenere un rapido feedback sulle modifiche dei requisiti. La documentazione viene ridotta al minimo ricorrendo a comunicazioni informali, anziché a riunioni formali con documenti scritti.

Gli approcci agili allo sviluppo del software considerano la progettazione e l’implementazione attività centrali nel processo software. Incorporano altre attività, come la deduzione e la verifica dei requisiti, nella progettazione e implementazione. Al contrario, un approccio guidato da piani identifica delle fasi distinte nel processo software con output associati a ciascuna fase. Gli output di una fase sono utilizzati come base per pianificare la successiva attività del processo.

La Figura 3.1 mostra le differenze essenziali tra approcci agili e approcci guidati da piani per la specifica del sistema. In un processo di sviluppo del software guidato da piani, l’iterazione si svolge all’interno delle attività, con documenti formali utilizzati per comunicare tra le fasi del processo. Per esempio, i requisiti cambieranno e, alla fine, sarà prodotta una specifica dei requisiti; questa è quindi un input per il processo di progettazione e implementazione. In un approccio agile, l’iterazione si svolge tra le attività, quindi, i requisiti e la progettazione sono sviluppati insieme, anziché separatamente.

In pratica, come spiegherò nel Paragrafo 3.4.1, i processi guidati da piani sono spesso utilizzati insieme a pratiche agili di programmazione, e i metodi agili possono includere alcune attività pianificate, a parte la programmazione e il test. È perfettamente fattibile, in un processo guidato da piani, allocare i requisiti e pianificare la fase di progettazione e sviluppo come una serie di incrementi. Un processo agile non è inevitabilmente incentrato sul codice, e può produrre qualche documento di progettazione. Gli sviluppatori agili possono decidere che un’iterazione non deve produrre nuovo codice, ma modelli del sistema e documenti.

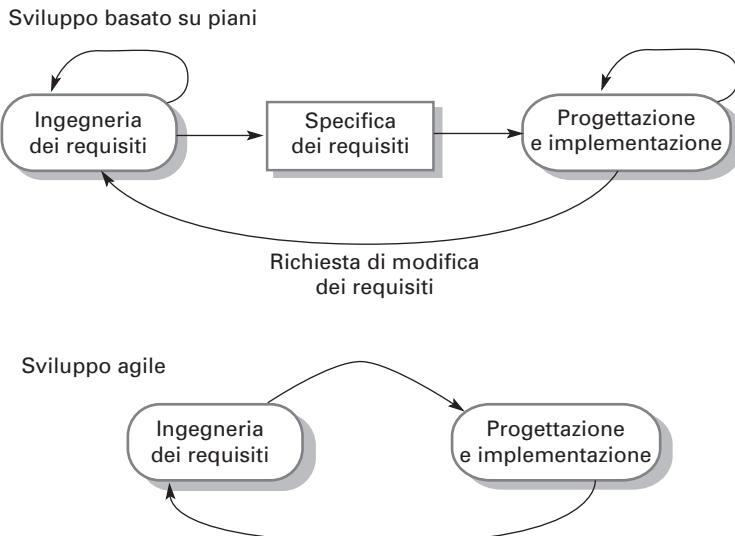


Figura 3.1 Sviluppo guidato da piani e sviluppo agile.

3.1 Metodi agili

Negli anni '80-'90 si diffuse l'idea che si potesse avere software migliore attraverso l'attenta pianificazione dei progetti, la garanzia di qualità formale, l'uso di metodi di analisi e progettazione supportati da strumenti software e processi di sviluppo software controllati e rigorosi. Questa idea arrivava dalla comunità di ingegneri del software che era responsabile dello sviluppo di grandi sistemi di lunga durata, come i sistemi aerospaziali e governativi.

Questo approccio guidato da piani fu ideato per il software sviluppato da grandi team, che a volte lavoravano per diverse società. I team spesso erano geograficamente lontani e lavoravano al software per lunghi periodi: per esempio i sistemi di controllo per i velivoli moderni richiedono fino a 10 anni di sviluppo, dalla specifica iniziale alla consegna. Gli approcci guidati da piani richiedono overhead significativi per la pianificazione, la progettazione e la documentazione del sistema. Tali overhead sono giustificati quando il lavoro di più team di sviluppo deve essere coordinato, quando il sistema è critico e quando diverse persone saranno coinvolte nella manutenzione del software durante la sua vita.

Quando questo gravoso approccio di sviluppo guidato da piani fu applicato ai sistemi di aziende di piccole e medie dimensioni, gli overhead richiesti erano così alti che a volte superavano il processo di sviluppo del sistema. Si impiegava più tempo per decidere come il sistema sarebbe stato sviluppato piuttosto che per sviluppare e provare il programma. Cambiando i requisiti, il sistema doveva essere rielaborato e, in teoria almeno, anche le specifiche e la progettazione dovevano essere modificate con il programma.

L'insoddisfazione con questi gravosi approcci all'ingegneria del software portò allo sviluppo di metodi agili alla fine degli anni '90. Questi metodi permettevano al team di sviluppo di concentrarsi sul software stesso anziché sulla progettazione e sulla documentazione. I metodi agili sono particolarmente indicati per sviluppare applicazioni nelle quali i requisiti del sistema cambiano rapidamente durante il processo di sviluppo. Sono stati ideati per consentire una consegna rapida del software ai clienti, che possono quindi proporre requisiti nuovi o modificati da includere in successive iterazioni del sistema. I metodi agili hanno anche lo scopo di ridurre la burocrazia nei processi di sviluppo, evitando tutto quel lavoro di dubbia validità a lungo termine ed eliminando la documentazione che probabilmente non sarà mai utilizzata.

La filosofia che sta alla base dei metodi agili è illustrata nel manifesto per lo sviluppo agile di software (<http://agilemanifesto.org>), pubblicato dai principali ideatori di questi metodi. Il manifesto stabilisce:

Stiamo scoprendo modi migliori di creare software, sviluppandolo e aiutando gli altri a fare lo stesso. Grazie a questa attività siamo arrivati a considerare importanti:

Gli individui e le interazioni più che i processi e gli strumenti

Il software funzionante più che la documentazione esaustiva

La collaborazione col cliente più che la negoziazione dei contratti

Rispondere al cambiamento più che seguire un piano

Ovvero, fermo restando il valore delle voci a destra, consideriamo più importanti le voci a sinistra.¹

Tutti i metodi agili suggeriscono che il software deve essere sviluppato e consegnato in modo incrementale. Questi metodi si basano su processi agili differenti, ma condividono alcuni principi ispirati al manifesto e dunque hanno molto in comune. Tali principi sono elencati nella Figura 3.2.

I metodi agili sono stati particolarmente utili per due tipi di sviluppo di sistemi.

1. Lo sviluppo di prodotti in cui una società di software sta sviluppando un prodotto di piccole o medie dimensioni. Virtualmente, tutti i prodotti software e le applicazioni oggi vengono sviluppati utilizzando un approccio agile.
2. Lo sviluppo personalizzato di sistemi all'interno di un'organizzazione, dove c'è un chiaro impegno da parte del cliente di essere coinvolto nel processo di sviluppo e dove ci sono pochi stakeholder e regolamenti esterni che possono influire sul software.

¹ <http://www.agilemanifesto.org/iso/it/>

Principio	Descrizione
Coinvolgimento del cliente	I clienti devono essere strettamente coinvolti in tutto il processo di sviluppo. Il loro ruolo è fornire nuovi requisiti del sistema, dar loro priorità e valutare le iterazioni del sistema.
Accettare i cambiamenti	Prevedere i requisiti di sistema che potranno cambiare; quindi progettare il sistema in modo che possa accogliere tali cambiamenti.
Consegna incrementale	Il software viene sviluppato per incrementi; il cliente specifica i requisiti da includere in ogni incremento.
Mantenere la semplicità	Concentrarsi sulla semplicità sia nel software che si sviluppa sia nel processo di sviluppo. Se possibile, lavorare attivamente per eliminare le complessità del sistema.
Persone, non processi	Le capacità del team di sviluppo devono essere riconosciute e sfruttate. I membri del team devono essere lasciati liberi di sviluppare il software secondo i loro metodi di lavoro, senza processi prescrittivi.

Figura 3.2 I principi dei metodi agili.

I metodi agili funzionano bene in queste situazioni, in quanto è possibile stabilire una comunicazione continua tra il responsabile del prodotto o il cliente e il team di sviluppo. Il software stesso è un sistema autonomo, anziché strettamente integrato in altri sistemi che si stanno sviluppando contemporaneamente. Di conseguenza, non occorre coordinare i flussi di sviluppo in parallelo. I sistemi di piccole e medie dimensioni possono essere sviluppati da team fisicamente vicini, in modo che le comunicazioni informali fra i membri dei team possano svolgersi facilmente.

3.2 Tecniche di sviluppo agile

Le idee che stanno alla base dei metodi agili furono sviluppate quasi contemporaneamente da persone differenti negli anni '90. La programmazione estrema (XP, *eXtreme Programming*) è forse l'approccio più significativo all'innovazione dello sviluppo del software. Il nome fu coniato da Kent Beck (Beck 1998) perché l'approccio fu sviluppato spingendo le normali pratiche, come lo sviluppo iterativo, a livelli "estremi". Per esempio, in XP, diverse nuove versioni di un sistema possono essere sviluppate da differenti programmatore, integrate e provate in un solo giorno. La Figura 3.3 illustra il processo XP per produrre un incremento del sistema che si sta sviluppando.

Nella programmazione estrema, i requisiti sono espressi come scenari (o *storie utente*), che sono implementati direttamente come una serie di compiti (*task*).

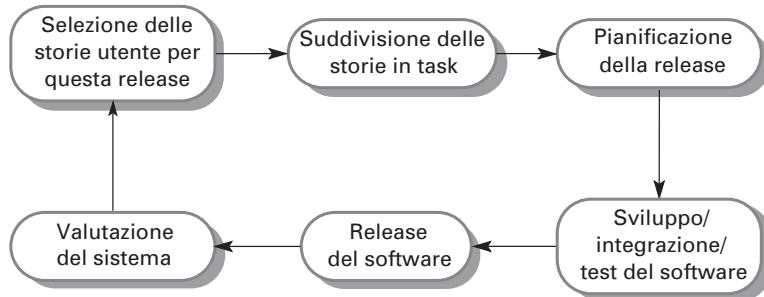


Figura 3.3 Il ciclo di rilascio della programmazione estrema.

I programmatore lavorano a coppie e sviluppano test per ogni compito prima di scrivere il codice. Tutti i test devono essere stati eseguiti con successo quando il nuovo codice viene integrato nel sistema. C'è un breve scarto temporale tra le release del sistema.

La programmazione estrema fu una tecnica controversa, in quanto introduceva un certo numero di pratiche agili che erano molto diverse da quelle tradizionali dell'epoca. Queste pratiche sono riassunte nella Figura 3.4 e rispecchiano i principi del manifesto per lo sviluppo agile.

1. Lo sviluppo incrementale è supportato attraverso piccole e frequenti release del sistema. I requisiti si basano su semplici scenari, che sono utilizzati come base per decidere quale funzionalità deve essere inclusa in un incremento del sistema.
2. Il coinvolgimento dell'utente è supportato attraverso l'impegno costante del cliente nel team di sviluppo; anche i rappresentanti del cliente prendono parte allo sviluppo e hanno la responsabilità di definire i test di accettabilità del sistema.
3. Le persone, non il processo, sono supportate dalla programmazione in coppia, dal possesso collettivo del codice del sistema, e da un processo di sviluppo sostenibile che non richiede periodi di lavoro eccessivamente lunghi.
4. Le modifiche sono supportate da regolari release del sistema, dallo sviluppo preceduto da test, dal refactoring per evitare la degenerazione del codice, e dall'integrazione continua di nuove funzionalità.
5. Il mantenimento della semplicità è supportato dal costante refactoring che migliora la qualità del codice e dall'uso di semplici progetti che non necessariamente prevedono future modifiche del sistema.

Nella pratica reale, l'applicazione della programmazione estrema, come originalmente proposta, si è rivelata più difficile del previsto. Non può essere immediatamente integrata con le pratiche di gestione e le tradizioni di molte aziende. Per questo, le società che adottano i metodi agili selezionano quelle pratiche XP

Principio o pratica	Descrizione
Proprietà collettiva	Le coppie di sviluppatori lavorano su tutte le aree del sistema, in modo che non ci siano esperti isolati che sviluppano; ogni sviluppatore è responsabile di tutto il codice; chiunque può cambiare qualsiasi cosa.
Integrazione continua	Appena un task è concluso, viene integrato nel sistema. Dopo ogni integrazione tutti i test sulle unità del sistema devono essere superati.
Pianificazione incrementale	I requisiti sono registrati su "story card" e le storie da includere in una release sono determinate dal tempo disponibile e dalla loro priorità relativa. Gli sviluppatori suddividono queste storie in "task" di sviluppo (Figure 3.5 e 3.6).
Cliente on-site	Un rappresentante dell'utente finale del sistema (il cliente) dovrebbe essere sempre disponibile per i membri del team XP. In un processo di programmazione estrema, il cliente è un membro del team di sviluppo e ha la responsabilità di consegnare i requisiti del sistema al team per l'implementazione.
Programmazione a coppie	Gli sviluppatori lavorano a coppie, verificando reciprocamente il loro lavoro e fornendo il supporto per fare sempre un buon lavoro.
Refactoring	Tutti gli sviluppatori effettuano in continuazione il refactoring del codice appena si trovano miglioramenti al codice stesso. Questo rende il codice semplice e mantenibile.
Progettazione semplice	Deve essere eseguita la progettazione sufficiente a soddisfare i requisiti correnti, niente di più.
Piccole release	Inizialmente deve essere sviluppato il minimo numero di funzionalità utili. Le release del sistema sono frequenti e aggiungono, in modo incrementale, nuove funzionalità alla release iniziale.
Ritmo sostenibile	Non sono considerati accettabili grandi ritardi, poiché spesso l'effetto finale è la riduzione della qualità del codice e della produttività a medio termine.
Sviluppo con test iniziali	Viene utilizzato un ambiente automatico di test delle unità per provare una nuova parte di funzionalità, prima che questa sia implementata.

Figura 3.4 Pratiche di programmazione estrema.

che sono più appropriate al loro modo di lavorare. A volte, queste pratiche vengono incorporate nei loro processi di sviluppo, ma, più comunemente, vengono utilizzate congiuntamente a un metodo agile incentrato sulla gestione, come Scrum (Rubin 2013).

Non credo che XP in sé sia un metodo agile pratico per molte società, ma il suo più importante contributo è probabilmente l'insieme delle pratiche di sviluppo agile che sono state introdotte nella comunità. In questo paragrafo tratto la più importante di queste pratiche.

3.2.1 Storie utente

I requisiti del software cambiano sempre. Per gestire questi cambiamenti, i metodi agili non hanno un'apposita attività di ingegneria dei requisiti, ma integrano la deduzione dei requisiti con lo sviluppo. Per semplificare questo, fu sviluppata l'idea delle “storie utente”, dove una storia utente è uno scenario d'uso in cui potrebbe trovarsi un utente del sistema.

Per quanto possibile, il cliente del sistema lavora a stretto contatto con il team di sviluppo e discute questi scenari con altri membri del team. Insieme sviluppano una “carta della storia” (*story card*) che raccoglie le esigenze dell'utente. Il team di sviluppo implementa tale scenario in una release successiva del software. Un esempio di story card per il sistema Mentcare è illustrato nella Figura 3.5. Si tratta di una breve descrizione di uno scenario per prescrivere un farmaco a un paziente.

Prescrizione dei farmaci
<p>Kate è un dottore che desidera prescrivere un farmaco a un paziente che frequenta una clinica. La cartella clinica del paziente è già visualizzata sul suo computer, quindi fa clic sul campo del farmaco e può selezionare “farmaco corrente”, “nuovo farmaco” o “formulario”. Se sceglie “farmaco corrente”, il sistema le chiede di controllare la dose. Se Kate vuole cambiare la dose, digita la nuova dose e conferma la prescrizione.</p> <p>Se sceglie “nuovo farmaco”, il sistema suppone che Kate conosca quale farmaco prescrivere. Kate digita le prime lettere del nome del farmaco. Il sistema visualizza una lista di possibili farmaci che iniziano con quelle lettere. Kate seleziona il farmaco richiesto e il sistema risponde chiedendole di verificare se il farmaco selezionato è corretto. Lei digita la dose e conferma la prescrizione.</p> <p>Se sceglie “formulario”, il sistema visualizza una finestra di ricerca per il formulario approvato. Kate può cercare il farmaco richiesto e poi lo seleziona. Il sistema le chiede di verificare se il farmaco scelto è corretto. Lei digita la dose e conferma la prescrizione.</p> <p>Il sistema controlla sempre che la dose sia entro i limiti consentiti; in caso contrario, chiede a Kate di cambiare la dose.</p> <p>Dopo che Kate ha confermato la prescrizione, dovrà controllarla e poi fare clic su “OK” o su “Cambia”. Se fa clic su “OK”, la prescrizione viene memorizzata nel database di controllo. Se fa clic su “Cambia”, il sistema riavvia il processo di prescrizione dei farmaci.</p>

Figura 3.5 Story card per la prescrizione dei farmaci.

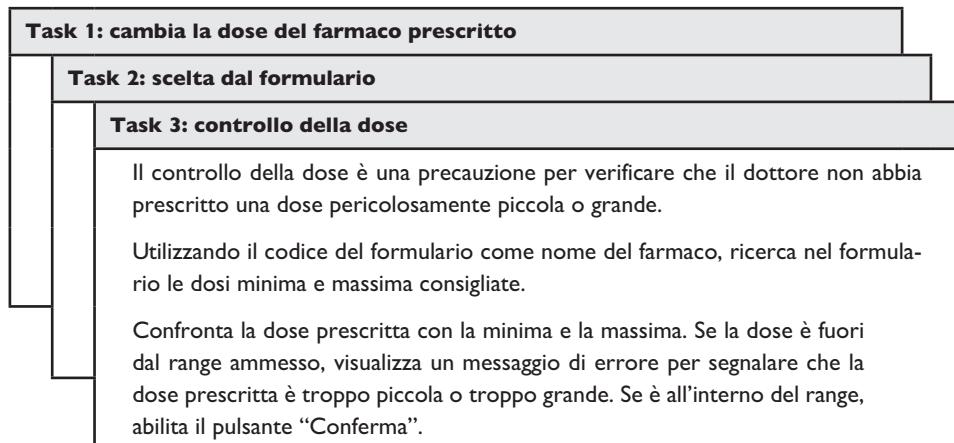


Figura 3.6 Esempi di carte di task per prescrivere i farmaci.

Le storie utente possono essere incluse nella pianificazione delle iterazioni del sistema. Una volta sviluppate le story card, il team di sviluppo le suddivide in task (Figura 3.6) e stima le risorse e gli sforzi richiesti per implementare ciascun task. Per fare questo, di solito, occorre discutere con il cliente in modo da perfezionare i requisiti. Il cliente poi elenca in ordine di priorità le storie, iniziando da quelle che possono fornire immediatamente un supporto utile all’azienda. L’obiettivo è identificare le funzionalità utili che possono essere implementate in due settimane circa, quando la successiva release del sistema sarà presentata al cliente.

Ovviamente, se i requisiti variano, le storie non implementate possono cambiare o essere scartate. Se sono richieste modifiche per un sistema che è già stato consegnato, vengono sviluppate nuove story card e, ancora una volta, il cliente decide se queste modifiche devono avere priorità sulle nuove funzionalità.

L’idea delle storie utente è molto efficace – le persone trovano più semplice relazionarsi con queste storie, anziché con un tradizionale documento di requisiti o con i casi d’uso. Le storie utente possono essere utili per coinvolgere gli utenti nel suggerire requisiti durante una preliminare attività di deduzione dei requisiti. Questo sarà descritto meglio nel Capitolo 4.

Il problema principale delle storie utente è la completezza. È difficile stabilire se sono state sviluppate storie utente sufficienti a trattare tutti i requisiti essenziali di un sistema. È anche difficile stabilire se una singola storia fornisce la rappresentazione vera di un’attività. Gli utenti esperti sono così familiari con il loro lavoro che spesso omettono alcuni particolari quando lo descrivono.

3.2.2 Refactoring

Un precezzo fondamentale dell’ingegneria del software tradizionale è che si dovrebbe progettare per il cambiamento; questo significa che dovremmo essere in grado di prevedere i cambiamenti futuri dei progetti e del software, in modo che

tali cambiamenti possano essere facilmente implementati. La programmazione estrema, purtroppo, ha scartato questo principio, considerando che la progettazione per il cambiamento spesso è uno sforzo inutile. Non vale la pena impiegare il tempo per rendere più generico un programma in modo da far fronte ai cambiamenti. Spesso le modifiche previste non si avverano mai oppure vengono richieste modifiche completamente diverse.

Ovviamente, in pratica, le modifiche dovranno essere sempre apportate al codice che si sta sviluppando. Per semplificare queste modifiche, gli sviluppatori XP suggeriscono che il codice che si sta sviluppando debba essere costantemente rifattorizzato. La *rifattorizzazione* o *refactoring* (Fowler et al. 1999) richiede che il team di programmazione ricerchi possibili miglioramenti del software e li implementi immediatamente. Quando i membri del team vedono che il codice può essere migliorato, realizzano questi miglioramenti anche nei casi in cui non ci sia un'immediata necessità di farlo.

Un problema fondamentale dello sviluppo incrementale è che le modifiche locali tendono a deteriorare la struttura del software. Di conseguenza, le modifiche future diventano sempre più difficili da implementare. In essenza, lo sviluppo procede in modo da trovare dei modi per aggirare i problemi, con il risultato che il codice spesso è duplicato, parti del software vengono riutilizzate in modi inappropriati, e la struttura complessiva del software si deteriora ogni volta che viene aggiunto nuovo codice. Il refactoring migliora la struttura e la leggibilità del software, evitando così il deterioramento strutturale che si verifica naturalmente quando si modifica il software.

Esempi di refactoring sono la riorganizzazione della gerarchia delle classi per eliminare il codice duplicato, il riordino e la ridenominazione degli attributi e dei metodi, e la sostituzione di sezioni di codice simili, con chiamate ai metodi definite in una libreria di programma. Gli ambienti di sviluppo dei programmi di solito includono strumenti di refactoring. Essi semplificano la ricerca delle dipendenze tra sezioni di codice e la modifica del codice globale.

In teoria, quando il refactoring è parte del processo di sviluppo, il software dovrebbe essere sempre facile da capire e modificare quando vengono proposti nuovi requisiti. In pratica, questo non è sempre possibile. A volte, le pressioni che gravano sul processo di sviluppo fanno sì che il refactoring sia rinviato, perché il tempo viene dedicato all'implementazione di nuove funzionalità. Alcune caratteristiche e modifiche nuove non possono essere realizzate immediatamente dal refactoring a livello codice, in quanto richiedono che sia modificata l'architettura del sistema.

3.2.3 Sviluppo con test iniziali

Come detto all'inizio di questo capitolo, una delle più importanti differenze tra lo sviluppo incrementale e lo sviluppo guidato da piani è il modo in cui il sistema viene testato. Nello sviluppo incrementale non c'è una specifica del sistema che

può essere utilizzata da un team esterno per sviluppare i test del sistema. Di conseguenza, alcuni approcci allo sviluppo incrementale hanno un processo di test molto più informale, se paragonato ai test guidati da piani.

La programmazione estrema ha sviluppato un nuovo approccio al test dei programmi per superare alcune difficoltà tipiche dei test senza specifica. I test sono automatizzati e sono centrali nel processo di sviluppo, e lo sviluppo non può procedere finché tutti i test non sono stati superati con successo. Gli elementi caratteristici del test nella programmazione estrema sono:

1. sviluppo con test iniziali;
2. sviluppo di test incrementale dagli scenari;
3. coinvolgimento dell’utente nello sviluppo e nella convalida dei test;
4. uso di strutture automatiche per i test.

La filosofia dei test iniziali della programmazione estrema adesso si è evoluta in tecniche di sviluppo più generali guidate da test (Jeffries e Melnik 2007). Io credo che lo sviluppo guidato da test sia una delle più importanti innovazioni nell’ingegneria del software. Anziché scrivere il codice e poi i test per il codice, scriviamo i test prima del codice. Questo significa che potete eseguire i test mentre viene scritto il codice e scoprire eventuali problemi durante lo sviluppo. Nel Capitolo 8 descriverò dettagliatamente lo sviluppo guidato da test.

La scrittura dei test implica la definizione di un’interfaccia e di una specifica comportamentale per le funzionalità da sviluppare. I problemi con i requisiti e le incomprensioni dell’interfaccia si riducono. Lo sviluppo con test iniziali richiede che ci sia una relazione chiara tra i requisiti del sistema e il codice che li implementa. Nella programmazione estrema si può sempre vedere questa relazione perché le story card che rappresentano i requisiti sono suddivise in task, e i task sono le unità principali dell’implementazione.

Nello sviluppo con test iniziali, chi implementa i task deve capire bene le specifiche, in modo che possa scrivere test per il sistema. Ciò significa che le ambiguità e le omissioni nelle specifiche devono essere chiarite prima che inizi l’implementazione, evitando così il problema del ritardo dei test (*test-lag*). Questo può accadere quando lo sviluppatore del sistema lavora più velocemente di chi esegue i test. L’implementazione è sempre più veloce del processo di test, e c’è la tendenza a tralasciare i test per poter rispettare la tempistica dello sviluppo.

L’approccio con test iniziali della programmazione estrema suppone che le storie utente siano state sviluppate, e che queste siano state suddivise in una serie di carte di task, come mostrato nella Figura 3.6. Ogni task genera uno o più test che controllano l’implementazione descritta nella carta del task. La Figura 3.7 è una breve descrizione di un test case che è stato sviluppato per verificare che la dose prescritta del farmaco rientri nei limiti di sicurezza consentiti.

Il ruolo del cliente nel processo di test consiste nell’aiutare a sviluppare i test di accettazione delle storie che devono essere implementate nella successiva release del sistema. Come dirò nel Capitolo 8, i test di accettazione sono quelle atti-

Task 4: controllo della dose
Input:
1. Un numero in mg rappresenta una singola dose di farmaco. 2. Un numero rappresenta il numero di singole dosi giornaliere.
Test:
1. Controlla l'input nel quale la singola dose è corretta, ma la frequenza è troppo alta. 2. Controlla l'input nel quale la singola dose * la frequenza è troppo grande o troppo piccola. 3. Controlla l'input nel quale la singola dose * la frequenza è nel range consentito.
Output:
OK o messaggio di errore per segnalare che la dose è fuori dal range consentito.

Figura 3.7 Descrizione del test case per controllare la dose del farmaco.

vità in cui il sistema viene provato utilizzando i dati del cliente per verificare che esso soddisfa le sue reali esigenze.

L'automazione dei test è essenziale per lo sviluppo con test iniziali. I test vengono scritti come componenti eseguibili prima che il task sia implementato. Questi componenti devono essere autonomi, devono simulare gli input da provare e devono controllare che il risultato soddisfi la specifica degli output. Un framework automatico di testing è un sistema che semplifica la scrittura di test eseguibili e propone una serie di test da eseguire. Junit (Tahchiev et al. 2010) è un esempio molto diffuso di framework automatico di testing per i programmi Java.

Una volta che il testing è automatizzato, ci sarà sempre una serie di test che potranno essere eseguiti rapidamente e facilmente. Ogni volta che una nuova funzionalità viene aggiunta al sistema, è possibile eseguire i test e identificare immediatamente i problemi introdotti dal nuovo codice.

Lo sviluppo con test iniziali e l'automazione dei test di solito implicano un gran numero di test da scrivere ed eseguire. Tuttavia, è difficile garantire che la serie dei test sia completa:

1. I programmatore preferiscono la programmazione al testing, e a volte scrivono test incompleti che non sono in grado di verificare tutti i casi insoliti che potrebbero verificarsi.
2. Alcuni test possono essere molto difficili da scrivere immediatamente; per esempio in un'interfaccia utente complessa spesso è difficile scrivere test per il codice che implementa la logica di visualizzazione e il flusso di lavoro tra le schermate.

È difficile giudicare la completezza di un insieme di test. Pur disponendo di molti test, il vostro insieme di test potrebbe non fornire una copertura completa. Le parti cruciali del sistema potrebbero non essere eseguite e quindi non essere testate. Ne consegue che, sebbene si eseguano numerosi test, il sistema non è corretto e completo. Se i test non vengono aggiornati e non vengono scritti ulteriori test dopo lo sviluppo, potrebbero nascondersi dei bug nella release del sistema.

3.2.4 Programmazione a coppie

Un'altra pratica innovativa introdotta nella programmazione estrema consiste nel fatto che i programmatore operano in coppia per sviluppare il software. Coppie di programmatore siedono realmente alla stessa postazione di lavoro per sviluppare il software. Ciò non significa che la coppia sia sempre formata dalle stesse persone; piuttosto l'idea è che le coppie siano create dinamicamente in modo che tutti i membri del team possano lavorare in coppia con altri durante il processo di sviluppo. La programmazione a coppie offre una serie di vantaggi.

1. Supporta l'idea della proprietà e della responsabilità comune del sistema. Questo riflette l'idea di Weinberg sulla programmazione impersonale (*ego-less programming*) (Weinberg 1971), dove il software è di proprietà dell'intero team e i singoli non sono ritenuti responsabili dei problemi riscontrati nel codice. È il team che ha la responsabilità collettiva della risoluzione di questi problemi.
2. Funge da processo di revisione informale, in quanto ogni linea di codice è visionata almeno da due persone. L'ispezione e la revisione del codice (Capitolo 21) sono efficaci nello scoprire un'alta percentuale di errori del software, ma richiedono tempo per l'organizzazione e, di solito, causano ritardi nel processo di sviluppo. Sebbene la programmazione a coppie sia un processo meno formale che probabilmente non trova tanti errori quanti ne trova l'ispezione del codice, tuttavia è più semplice ed economico da organizzare dell'ispezione formale dei programmi.
3. Incentiva il refactoring per migliorare la struttura del software. Il problema di chiedere ai programmatore di eseguire il refactoring in un normale ambiente di sviluppo è che lo sforzo richiesto produce benefici nel lungo termine. Uno sviluppatore che esegue il refactoring potrebbe essere giudicato meno efficiente di uno che semplicemente continua a sviluppare codice. Applicando la programmazione a coppie e la proprietà collettiva, altri possono beneficiare immediatamente del refactoring, quindi è più probabile che questi supportino il processo.

Si potrebbe pensare che la programmazione a coppie sia meno efficiente della programmazione individuale, e che una coppia di sviluppatori produca la metà del codice rispetto a due individui che lavorano da soli. Molte società che hanno adottato i metodi agili sono sospettose verso la programmazione a coppie e non la usano. Altre società combinano la programmazione individuale e a coppie con un programmatore esperto che lavora con un collega meno esperto quando ci sono problemi.

Studi formali sul valore della programmazione a coppie hanno riportato risultati contrastanti. Utilizzando studenti volontari, Williams e i suoi collaboratori (Williams et al. 2000) hanno scoperto che la produttività con la programmazione a coppie sembra essere confrontabile con quella di due persone che lavorano

indipendentemente, perché le coppie discutono il software prima di svilupparlo, quindi probabilmente hanno meno false partenze e meno necessità di rielaborare il codice; inoltre il numero di errori evitati dall’ispezione informale è tale che occorre meno tempo per risolvere i problemi scoperti durante il processo di test.

Tuttavia, studi con programmatore più esperti non confermano questi risultati (Arisholm et al. 2007). È emerso che c’era una significativa perdita di produttività rispetto a due programmatore che lavoravano da soli. C’erano alcuni benefici per la qualità, ma questi non compensavano completamente gli overhead della programmazione a coppie. Ciononostante, la condivisione delle conoscenze che si realizza durante la programmazione a coppie è molto importante, in quanto essa riduce i rischi di fallimento di un progetto nel caso in cui alcuni membri abbandonino il team di sviluppo. Solo questo rende valida la programmazione a coppie.

3.3 Gestione agile della progettazione

In qualsiasi società di software, i manager hanno bisogno di sapere che cosa sta accadendo, se un progetto potrà raggiungere i suoi obiettivi e se il software sarà consegnato in tempo con il budget previsto. Gli approcci allo sviluppo del software guidati da piani si sono evoluti per soddisfare queste esigenze. Come vedremo nel Capitolo 20, i manager elaborano un piano per il progetto per specificare che cosa deve essere consegnato, quando deve essere consegnato e chi dovrà occuparsi dello sviluppo. Un approccio basato su piani richiede che un manager abbia una visione continua su qualsiasi cosa deve essere sviluppato e sullo sviluppo dei processi.

La pianificazione informale e il controllo del progetto che furono proposti dai primi seguaci dei metodi agili si sono scontrati con questa esigenza di visibilità da parte dei manager. I team avevano organizzazioni autonome, non producevano documentazione e pianificavano lo sviluppo in cicli molto brevi. Sebbene questo possa andare bene per le piccole società che sviluppano prodotti software, tuttavia non è appropriato per le società più grandi che hanno bisogno di sapere cosa sta accadendo nella loro organizzazione.

Come qualsiasi altro processo professionale di sviluppo del software, lo sviluppo agile deve essere gestito in modo che venga fatto il miglior uso del tempo e delle risorse a disposizione del team. Per risolvere questo problema, fu sviluppato il metodo agile Scrum (Schwaber e Beedle 2001; Rubin 2013) che offre un framework per organizzare progetti agili e fornire, in una certa misura, una visibilità esterna su ciò che sta accadendo. Gli sviluppatori di Scrum avevano chiarito che Scrum non era un metodo per la gestione dei progetti nel senso convenzionale, per questo avevano deliberatamente creato una nuova terminologia, come ScrumMaster, che sostituiva termini come “project manager”. La Figura 3.8 riassume i termini di Scrum e il loro significato.

Termine di Scrum	Definizione
Team di sviluppo	Un gruppo di sviluppatori software con organizzazione autonoma, che non dovrebbe avere più di sette persone. Sono responsabili dello sviluppo del software e di altri documenti essenziali di progettazione.
Incremento di un prodotto potenzialmente rilasciabile	L'incremento del software che è consegnato da uno sprint. L'idea è che questo incremento deve essere "potenzialmente rilasciabile", ovvero deve trovarsi in uno stato finito e non occorre altro lavoro, come il testing, per incorporarlo nel prodotto finale. In pratica, questo non è sempre realizzabile.
Product backlog	Una lista di elementi di cui si deve occupare il team di Scrum; per esempio, definizioni di caratteristiche per il software, requisiti del software, storie utente o descrizioni di task supplementari, come la definizione dell'architettura o la documentazione per l'utente.
Product owner	Un individuo (o anche un piccolo gruppo) il cui compito è identificare le caratteristiche o i requisiti del prodotto, stabilirne le priorità e rivedere continuamente il product backlog per garantire che il progetto continui a soddisfare i requisiti critici. Il product owner può essere un cliente, ma anche un product manager in una società di software o un altro rappresentante degli stakeholders.
Scrum	Una riunione giornaliera del team di Scrum che esamina l'avanzamento del lavoro e stabilisce le priorità del lavoro da svolgere in quel giorno. Teoricamente, lo scrum dovrebbe essere un breve incontro faccia a faccia di tutti i membri del team.
ScrumMaster	Lo ScrumMaster ha la responsabilità di garantire che il processo Scrum sia seguito e di guidare il team nell'uso efficiente di Scrum. Deve fungere da interfaccia con il resto della società e garantire che il team non venga svilto da interferenze esterne. Gli sviluppatori Scrum sono fermamente convinti che lo ScrumMaster non deve essere considerato come project manager. Altri, invece, non facilmente riescono a capire la differenza.
Sprint	Una iterazione dello sviluppo. Gli sprint di solito durano da 2 a 4 settimane.
Velocità	Una stima della quantità di lavoro del product backlog che un team può svolgere in un singolo sprint. Conoscere la velocità di un team aiuta a stimare che cosa può essere svolto in uno sprint e fornisce la base per misurare i miglioramenti.

Figura 3.8 Terminologia di Scrum.

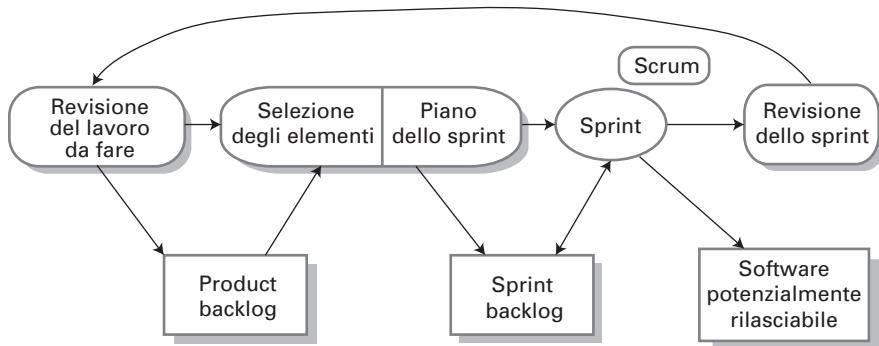


Figura 3.9 Il ciclo degli sprint di Scrum.

Scrum è un metodo agile in quanto segue i principi del manifesto per lo sviluppo agile (riportati nella Figura 3.2), ma è stato ideato soprattutto per fornire un framework per l'organizzazione agile dei progetti; non richiede l'uso di specifiche pratiche di sviluppo, come la programmazione a coppie e lo sviluppo con test iniziali. Questo significa che può essere più facilmente integrato con i metodi esistenti in una società. Di conseguenza, poiché i metodi agili sono diventati l'approccio generale allo sviluppo del software, Scrum è emerso come il metodo più largamente utilizzato.

Il processo Scrum o ciclo degli sprint è illustrato nella Figura 3.9. L'input del processo è il product backlog. Ogni iterazione del processo genera un incremento del prodotto che può essere consegnato al cliente.

Il punto di partenza del ciclo degli sprint di Scrum è il product backlog – la lista degli elementi, come le caratteristiche del prodotto, i requisiti e il miglioramento dell'ingegnerizzazione, di cui si dovrà occupare il team di Scrum. La versione iniziale del product backlog può essere derivata da un documento dei requisiti, da una lista delle storie utente o da un'altra descrizione del software che si sta sviluppando.

Sebbene la maggior parte dei contenuti del product backlog riguardino l'implementazione delle caratteristiche del sistema, possono essere incluse altre attività. A volte, quando si pianifica un'iterazione, vengono alla luce domande alle quali è difficile rispondere, e quindi è necessario svolgere un lavoro addizionale per trovare possibili soluzioni. Per capire i problemi e trovare le soluzioni, il team può creare qualche prototipo e effettuare delle prove. Potrebbe essere necessario definire alcuni elementi di backlog per progettare l'architettura del sistema o per sviluppare la documentazione del sistema.

Il product backlog può essere specificato a vari livelli di dettagli; il product owner ha la responsabilità di garantire che il livello di dettagli nella specifica sia appropriato al lavoro da svolgere. Per esempio, un elemento di backlog potrebbe essere la storia completa di un utente, come quella illustrata nella Figura 3.5, o

una semplice istruzione come “Refactoring del codice dell’interfaccia utente”, che lascia al team la decisione su come effettuare il refactoring.

Ogni ciclo di sprint ha una durata prestabilita, che di solito è compresa tra 2 e 4 settimane. All’inizio di ogni ciclo, il product owner stabilisce le priorità del product backlog per definire quali sono gli elementi più importanti da sviluppare in quel ciclo. Gli sprint non vengono mai prolungati a causa del lavoro non ultimato. Gli elementi che non possono essere completati entro il tempo assegnato allo sprint vengono restituiti al product backlog.

Tutti i membri del team vengono coinvolti nella scelta degli elementi con priorità più alta che dovranno essere completati; poi valutano il tempo richiesto per completare questi elementi. Per fare queste stime, utilizzano la velocità raggiunta nei precedenti sprint, ovvero quanto lavoro del product backlog può essere svolto in un singolo sprint. Questo porta alla creazione di uno sprint backlog – il lavoro da svolgere in quello sprint. Il team sceglie chi dovrà lavorare su determinati elementi, e avvia lo sprint.

Durante lo sprint, il team si riunisce ogni giorno per esaminare l’avanzamento del lavoro e, se necessario, per ridefinire le sue priorità. Durante la riunione quotidiana (scrum), tutti i membri del team condividono le informazioni, descrivono il lavoro svolto dall’ultima riunione, illustrano i problemi che hanno incontrato e stabiliscono cosa deve essere fatto per il giorno successivo. Quindi, ciascun membro del team sa che cosa sta accadendo e, in caso di problemi, può ripianificare il lavoro a breve termine per risolverli. Ognuno partecipa a questa pianificazione a breve termine; non c’è una direzione dall’alto da parte dello ScrumMaster.

Le interazioni giornaliere tra i membri del team possono essere coordinate utilizzando la lavagna di Scrum; si tratta di una lavagna bianca per ufficio che riporta informazioni e note sullo sprint backlog, sul lavoro svolto, sull’indisponibilità delle persone e così via. È una risorsa condivisa per tutto il team, e chiunque può modificare o spostare gli elementi della lavagna. Ciò significa che qualsiasi membro del team può sapere a colpo d’occhio che cosa stanno facendo gli altri e che cosa resta da fare.

Alla fine di ogni sprint, si tiene una riunione di verifica, che coinvolge tutti i membri del team. Questa riunione ha due obiettivi. In primo luogo, è uno strumento per migliorare il processo; il team riesamina il modo in cui è stato svolto il lavoro e riflette su come avrebbe potuto fare meglio le cose. In secondo luogo, nella riunione viene fornito l’input sul prodotto e sul suo stato per la revisione del product backlog che precede il successivo sprint.

Sebbene lo ScrumMaster non sia formalmente un project manager, in pratica svolge questo ruolo in molte organizzazioni che hanno una struttura manageriale convenzionale. Riferisce sull’avanzamento del lavoro al senior manager e prende parte alla pianificazione a lungo termine e al budget del progetto. Può essere coinvolto nell’amministrazione del progetto (programmare le ferie dello staff, collaborare con l’ufficio del personale ecc.) e nell’acquisto di componenti hardware e software.

Nelle varie storie di successo di Scrum (Schatz e Abdelshafi 2005; Mulder e van Vliet 2008; Bellouiti 2009), le cose che gli utenti apprezzano del metodo Scrum sono:

1. il prodotto è suddiviso in parti gestibili e comprensibili alle quali gli stakeholder possono fare riferimento;
2. i requisiti instabili non fanno ritardare l'avanzamento del lavoro;
3. l'intero team ha una visione su tutto e, di conseguenza, la comunicazione e il morale dei suoi membri sono migliori;
4. i clienti ricevono in tempo gli incrementi e hanno un feedback su come funziona il prodotto. Non devono temere sorprese dell'ultimo minuto, quando il team li informa che il software non sarà consegnato come previsto;
5. c'è fiducia tra clienti e sviluppatori, e si genera un'atmosfera positiva, perché tutti si aspettano che il progetto avrà successo.

Scrum originariamente fu progettato per essere utilizzato da team fisicamente vicini, in modo che i membri di tutti i team potessero partecipare alle riunioni giornaliere. Tuttavia, gran parte dello sviluppo del software oggi è svolto da team fisicamente distanti, i cui membri sono distribuiti in vari luoghi del mondo. Questo consente alle società di trarre vantaggi dai costi più bassi dello staff di altre nazioni, di coinvolgere persone particolarmente specializzate, di sviluppare il software 24 ore al giorno, in quanto il lavoro viene svolto in luoghi con fusi orari differenti.

Di conseguenza, ci sono stati casi di sviluppo con Scrum in ambienti distribuiti e con più team. Tipicamente, per lo sviluppo offshore, il product owner si trova in un paese diverso dal team di sviluppo, i cui membri possono essere anche distribuiti in luoghi diversi. La Figura 3.10 mostra i requisiti dello Scrum distribuito (Deemer 2011).

3.4 Scalabilità dei metodi agili

I metodi agili furono sviluppati per piccoli team di programmazione che avrebbero potuto lavorare insieme nella stessa stanza e comunicare in modo informale. Originariamente, furono utilizzati per sviluppare sistemi e prodotti software di piccole e medie dimensioni. Le piccole società, senza processi formali o burocrazia, furono i primi utilizzatori entusiasti di questi metodi.

Ovviamente, l'esigenza di creare un software più adatto alle necessità dei clienti e di doverlo consegnare più velocemente vale anche per i sistemi e le società più grandi. Di conseguenza, negli ultimi anni, sono stati fatti molti sforzi per migliorare i metodi agili in modo da utilizzarli per grandi sistemi software e nelle grandi società.

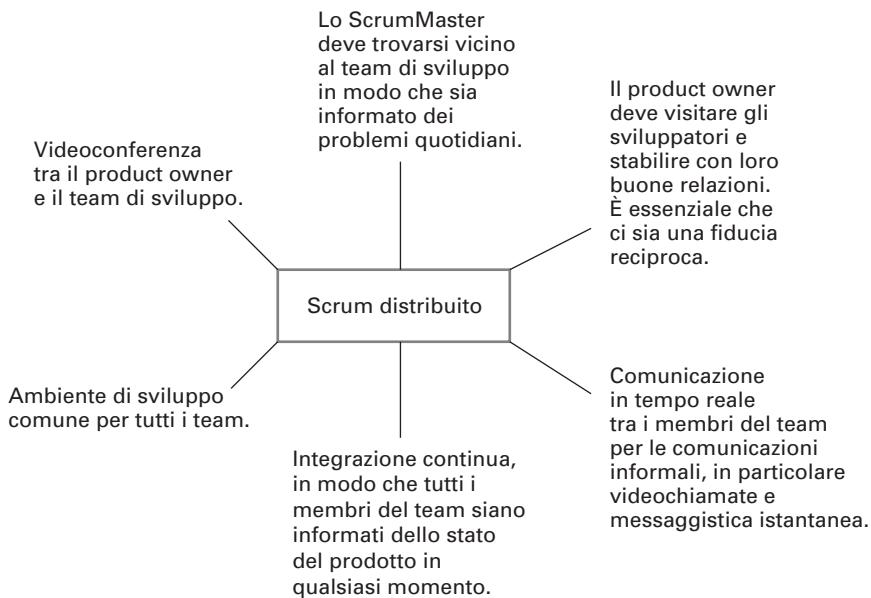


Figura 3.10 Scrum distribuito.

La scalabilità dei metodi agili ha due sfaccettature correlate:

1. potenziare questi metodi per poter gestire lo sviluppo di grandi sistemi, che sono troppo grandi per essere sviluppati da un solo piccolo team;
2. estendere questi metodi da un uso specializzato dei team di sviluppo a un uso più ampio in una grande società che ha molti anni di esperienza nello sviluppo del software.

Ovviamente, il potenziamento e l'estensione dei metodi agili sono strettamente correlati. I contratti per sviluppare grandi sistemi software di solito sono assegnati a grandi organizzazioni, con più team che lavorano sul progetto di sviluppo. Queste grandi società hanno già sperimentato i metodi agili in progetti più piccoli, quindi sono in grado di affrontare sia i problemi di potenziamento dei metodi agili sia quelli di estensione.

Ci sono vari aneddoti circa l'efficacia dei metodi agili, ed è emerso che questi metodi possono portare notevoli miglioramenti della produttività e significative riduzioni dei difetti. Ambler (Ambler 2010), un influente sviluppatore di metodi agili, ritiene che questi miglioramenti della produttività siano esagerati per i grandi sistemi e le grandi organizzazioni. Secondo Ambler, un'organizzazione che passa ai metodi agili può aspettarsi un miglioramento della produttività di circa il 15% in 3 anni, con analoghe riduzioni nel numero dei difetti dei prodotti.

3.4.1 Problemi pratici con i metodi agili

In alcune aree, particolarmente nello sviluppo di prodotti e applicazioni software, lo sviluppo agile ha avuto un successo incredibile. È indubbiamente il miglior approccio da utilizzare per questo tipo di sistemi. Tuttavia, i metodi agili potrebbero non essere appropriati per altri tipi di sviluppo del software, come l'ingegneria dei sistemi integrati o lo sviluppo di sistemi grandi e complessi.

L'uso di un approccio agile per grandi sistemi di lunga durata, che sono sviluppati da una società di software per un cliente esterno, presenta alcuni problemi.

1. L'informalità dello sviluppo agile è incompatibile con l'approccio legale alla definizione dei contratti che di solito si usano nelle grandi società.
2. I metodi agili sono più indicati per lo sviluppo di nuovo software, non per la manutenzione del software; eppure la maggior parte dei costi del software nelle grandi società proviene dalla manutenzione dei loro sistemi software esistenti.
3. I metodi agili sono ideati per piccoli team fisicamente vicini; eppure la maggior parte dello sviluppo del software oggi coinvolge team distribuiti in tutto il mondo.

I problemi contrattuali possono essere un problema importante quando si usano i metodi agili. Se il cliente del sistema usa un'organizzazione esterna per lo sviluppo del sistema, tra queste due parti viene redatto un contratto per lo sviluppo del software. Il documento dei requisiti del software di solito è parte del contratto tra cliente e fornitore. Poiché lo sviluppo intrecciato dei requisiti e del codice è fondamentale nei metodi agili, non è possibile stabilire i requisiti definitivi da includere nel contratto.

Di conseguenza, i metodi agili devono basarsi su contratti dove i clienti pagano il tempo richiesto per lo sviluppo del sistema, anziché lo sviluppo di uno specifico insieme di requisiti. Finché tutto questo procede bene, ne traggono benefici sia il cliente sia lo sviluppatore. Se, invece, sorgono problemi, può essere difficile stabilire chi ne è la causa e chi deve pagare per il tempo e le risorse extra necessarie alla risoluzione dei problemi.

Come dirò nel Capitolo 9, una gran quantità di lavoro di ingegneria del software è impiegata nella manutenzione e nell'aggiornamento dei sistemi software esistenti. Le pratiche agili, come la consegna incrementale e la progettazione per il cambiamento, hanno senso quando il software è in corso di modifica. In effetti, possiamo immaginare un processo di sviluppo agile come un processo che supporta continui cambiamenti. Se i metodi agili sono utilizzati per sviluppare prodotti software, le nuove release di un prodotto o di un'applicazione semplicemente richiedono che si continui ad applicare l'approccio agile.

Tuttavia, se la manutenzione richiede un sistema personalizzato che deve essere modificato perché sono cambiati i requisiti aziendali, non c’è un parere unanime sull’idoneità dei metodi agili alla manutenzione del software (Bird 2011; Kilner 2012). I tipi di problemi che possono nascere sono:

- mancanza di documentazione del prodotto;
- mantenere il coinvolgimento del cliente;
- continuità del team di sviluppo.

La documentazione formale serve a descrivere il sistema e renderlo più comprensibile alle persone che devono modificarlo. In pratica, però, la documentazione formale raramente viene aggiornata e, quindi, non rispecchia esattamente il codice del programma. Per questo motivo, i sostenitori dei metodi agili ritengono che sia una perdita di tempo scrivere questa documentazione e che la chiave per implementare un software mantenibile sia produrre un codice leggibile di alta qualità. La mancanza di documentazione non dovrebbe essere un problema per continuare a sviluppare i sistemi utilizzando un approccio agile.

Tuttavia, la mia esperienza di manutenzione dei sistemi indica che il documento più importante è il documento dei requisiti del sistema, che spiega all’ingegnere del software che cosa dovrebbe fare il sistema. Senza queste conoscenze, è difficile prevedere l’impatto delle modifiche proposte per il sistema. Molti metodi agili raccolgono i requisiti in modo informale e incrementale, e non creano un documento di requisiti coerenti con lo stato reale del sistema. L’uso dei metodi agili può quindi rendere più difficile e costosa la successiva manutenzione del sistema. Questo problema è particolarmente grave se non può essere garantita la continuità del team di sviluppo.

Un fattore chiave per utilizzare con successo un approccio agile nella manutenzione consiste nel mantenere coinvolti i clienti nel processo. Sebbene un cliente possa assicurare il pieno coinvolgimento di un suo rappresentante durante lo sviluppo del sistema, ciò è meno probabile durante la manutenzione quando le modifiche non sono continue. I rappresentanti dei clienti rischiano di perdere interesse nel sistema. Quindi, è probabile che dovranno essere impiegati dei meccanismi alternativi, come le proposte di modifica, descritti nel Capitolo 22, da adattare a un approccio agile.

Un altro potenziale problema che potrebbe presentarsi è mantenere la continuità del team di sviluppo. I metodi agili si basano su membri del team che conoscono gli aspetti del sistema senza bisogno di consultare la documentazione. Se il team di sviluppo agile si scioglie, allora queste conoscenze si perdono ed è difficile per i nuovi membri del team ricostruire le stesse conoscenze del sistema e dei suoi componenti. Molti programmati preferiscono lavorare su un nuovo sviluppo di software, e quindi sono riluttanti a continuare a lavorare su un sistema software dopo che è stata consegnata la prima release. Quindi, anche quando l’intenzione è quella di mantenere unito il team di sviluppo, le persone se ne vanno se ricevono un incarico di manutenzione.

3.4.2 Metodi agili e guidati da piani

Una condizione fondamentale per la scalabilità dei metodi agili è la loro integrazione con gli approcci guidati da piani. Piccole società di startup possono lavorare con una pianificazione informale e a breve termine, ma le società più grandi devono avere piani a lungo termine e budget per gli investimenti, lo staff e lo sviluppo. Lo sviluppo del loro software deve supportare questi piani, quindi è essenziale la pianificazione del software a lungo termine.

I primi utilizzatori dei metodi agili all'inizio del XXI secolo erano entusiasti e rigorosamente fedeli al manifesto dello sviluppo agile. Essi deliberatamente rifiutarono l'approccio guidato da piani all'ingegneria del software ed erano assolutamente riluttanti a cambiare la visione iniziale dei metodi agili. Tuttavia, quando le organizzazioni scoprirono il valore e i benefici di un approccio agile, adottarono questi metodi per adattare la loro cultura e i loro modi di operare. Dovettero fare questo perché i principi che stanno alla base dei metodi agili a volte sono difficili da realizzare nella pratica (Figura 3.11).

Principio	Pratica
Coinvolgimento dell'utente	Questo dipende dalla volontà e dalla disponibilità che ha un cliente di spendere il suo tempo con il team di sviluppo e da chi rappresenta tutti gli stakeholder del sistema. Spesso, i rappresentanti del cliente hanno altre esigenze per il loro tempo e non possono collaborare pienamente allo sviluppo del software. Se ci sono stakeholder esterni, come le autorità di controllo, è difficile presentare i loro punti di vista al team di sviluppo agile.
Accettare i cambiamenti	Stabilire le priorità delle modifiche può essere estremamente difficile, specialmente per quei sistemi che hanno molti stakeholder. Tipicamente, ogni stakeholder fornisce priorità differenti a modifiche differenti.
Consegna incrementale	Iterazioni rapide e pianificazione a breve termine per lo sviluppo non sempre si adatta ai cicli di pianificazione a lungo termine del marketing aziendale. I responsabili del marketing potrebbero aver bisogno di conoscere le caratteristiche del prodotto parecchi mesi in anticipo per preparare una campagna di marketing efficace.
Mantenere la semplicità	Sotto la pressione delle scadenze di consegna, i membri del team potrebbero non avere il tempo di svolgere le semplificazioni desiderabili per il sistema.
Persone, non processi	Alcuni membri del team potrebbero non avere una personalità idonea per il coinvolgimento intenso che è tipico dei metodi agili e, quindi, potrebbero non interagire bene con gli altri membri.

Figura 3.11 Principi dei metodi agili e pratiche organizzative.

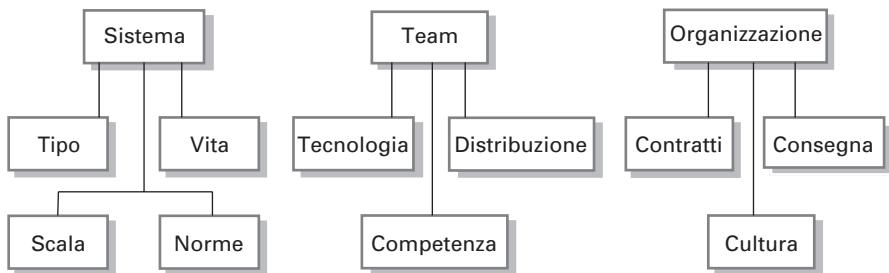


Figura 3.12 Fattori che influenzano la scelta tra sviluppo agile e sviluppo guidato da piani.

Per risolvere questi problemi, molti dei più grandi progetti di sviluppo “agile” del software combinano le pratiche tipiche degli approcci guidati da piani con quelle dei metodi agili. Alcuni di questi progetti sono principalmente agili, altri sono principalmente guidati da piani, ma con qualche pratica agile. Per decidere come bilanciare l’approccio agile con quello guidato da piani, occorre rispondere a una serie di domande tecniche, umane e organizzative. Queste domande riguardano il sistema da sviluppare, il team di sviluppo e le organizzazioni che stanno sviluppando il sistema (Figura 3.12).

I metodi agili furono sviluppati e perfezionati in progetti per sviluppare sistemi aziendali di piccole e medie dimensioni e prodotti software, dove lo sviluppatore del software controlla la specifica del sistema. Altri tipi di sistemi hanno attributi, quali la dimensione, la complessità, la risposta in tempo reale e le normative esterne, che un puro approccio “agile” non sarebbe in grado di gestire. Il processo di ingegneria dei sistemi richiede un certo grado di pianificazione, progettazione e documentazione. Alcuni di questi problemi chiave sono qui elencati.

1. Quanto è grande il sistema da sviluppare? I metodi agili sono più efficaci quando il sistema può essere sviluppato con un team relativamente piccolo, i cui membri possono comunicare in modo informale. Questo potrebbe non essere possibile per i grandi sistemi che richiedono team di sviluppo più grandi, quindi potrebbe essere necessario utilizzare un approccio guidato da piani.
2. Che tipo di sistema si sta sviluppando? I sistemi che richiedono molta analisi prima dell’implementazione (per esempio, i sistemi in tempo reale con requisiti complessi sulla tempistica) di solito hanno bisogno di una progettazione abbastanza dettagliata per svolgere tale analisi. Un approccio guidato da piani potrebbe essere più appropriato in questi casi.
3. Qual è la vita prevista del sistema? I sistemi di lunga durata possono richiedere più documentazione per comunicare le intenzioni originali degli sviluppatori del sistema al team di supporto. Tuttavia, i sostenitori dei metodi

agili giustamente sostengono che la documentazione spesso non viene aggiornata e quindi non è di grande utilità per la manutenzione a lungo termine del sistema.

4. Il sistema è soggetto a norme esterne? Se un sistema deve essere approvato da un'autorità esterna di controllo (per esempio, la Federal Aviation Administration deve approvare il software che è critico per il funzionamento di un aereo), allora sarà necessario produrre una documentazione dettagliata sulla sicurezza del sistema.

I metodi agili pongono sul team di sviluppo una grande responsabilità per cooperare e comunicare durante le fasi di sviluppo del sistema. Si affidano alle competenze ingegneristiche e al supporto dei singoli membri per il processo di sviluppo. In realtà, però, nessuno è un ingegnere di alte competenze professionali, le persone non comunicano in modo efficace, e non sempre è possibile che i membri del team lavorino insieme. Qualche pianificazione potrebbe essere richiesta per organizzare nel modo migliore il lavoro delle persone. I problemi più importanti sono qui elencati.

1. Quanto sono bravi i progettisti e i programmati del team di sviluppo? Qualcuno sostiene che i metodi agili richiedano livelli di competenze professionali più alti degli approcci guidati da piani nei quali i programmati semplicemente traducono un progetto dettagliato in un codice. Se avete un team con livelli di competenze relativamente bassi, potrebbe essere necessario utilizzare le persone più competenti per sviluppare il progetto, lasciando alle altre la responsabilità della programmazione.
2. Come è organizzato il team di sviluppo? Se il team di sviluppo è distribuito o se una parte dello sviluppo è affidata a società esterne, allora potrebbe essere necessario sviluppare un documento di progettazione per favorire la comunicazione fra i team di sviluppo.
3. Quali tecnologie sono disponibili a supporto dello sviluppo del sistema? I metodi agili spesso si basano su buoni strumenti per tenere traccia dell'evoluzione di un progetto. Se state sviluppando un sistema mediante un IDE che non ha buoni strumenti per la visualizzazione e l'analisi dei programmi, allora potrebbero essere necessari altri documenti di progettazione.

La televisione e il cinema presentano le società di software come organizzazioni informali gestite (per lo più) da giovani che offrono un ambiente di lavoro alla moda, con pochissime procedure burocratiche e organizzative. Questo è lontano dalla verità. La maggior parte del software è sviluppata in grandi società che hanno definito le loro pratiche e procedure di lavoro. La gestione in queste società potrebbe essere scomoda senza una documentazione e con le decisioni informali tipiche dei metodi agili. I problemi chiave sono qui elencati.

1. È importante avere una specifica molto dettagliata e un progetto prima di passare all'implementazione, forse per motivi contrattuali? Se sì, probabil-

mente occorrerà adottare un approccio guidato da piani per l'ingegneria dei requisiti, ma si potranno utilizzare i metodi agili di sviluppo durante l'implementazione del sistema.

2. È realistico avere una strategia di sviluppo incrementale, dove il software viene consegnato ai clienti o ad altri stakeholder e dai quali si ottiene un rapido feedback? Saranno disponibili i rappresentanti del cliente, e saranno disposti a collaborare con il membri del team di sviluppo?
3. Ci sono problemi di natura culturali che possono influire sullo sviluppo del sistema? Le organizzazioni di ingegneria tradizionali hanno una cultura di sviluppo basato su piani, in quanto ciò è normale nell'ingegneria. Questo di solito richiede una documentazione estesa di progettazione, anziché la conoscenza informale dei processi agili.

In realtà, il problema se un progetto può essere etichettato come agile o guidato da piani non è molto importante. In essenza, la principale preoccupazione degli acquirenti di un sistema software è se essi hanno oppure no un software eseguibile che risponda alle loro esigenze e che faccia cose utili per il singolo utente o per l'intera organizzazione. Gli sviluppatori di software devono essere pragmatici e scegliere quei metodi che sono più efficaci per il tipo di sistema che stanno sviluppando, indipendentemente dal fatto che i metodi siano agili o guidati da piani.

3.4.3 Metodi agili per grandi sistemi

I metodi agili devono evolversi per essere utilizzati nello sviluppo di software su vasta scala. La ragione principale di questo è che i sistemi software su vasta scala sono molto più complessi e difficili da capire e gestire dei sistemi su piccola scala o dei prodotti software. Sei fattori principali contribuiscono a questa complessità (Figura 3.13).

1. I grandi sistemi di solito sono sistemi di sistemi – collezioni di sistemi distinti in comunicazione fra loro, dove team separati sviluppano il proprio sistema. Spesso questi team lavorano in luoghi distanti, a volte con fusi orari differenti. È praticamente impossibile per ciascun team avere una visione dell'intero sistema. Di conseguenza, la loro priorità di solito è quella di completare la propria parte del sistema senza preoccuparsi dei problemi dei sistemi più grandi.
2. I grandi sistemi sono sistemi *brownfield* (Hopkins e Jenkins 2008), ovvero includono e interagiscono con un certo numero di sistemi esistenti. Molti dei requisiti del sistema riguardano questa interazione e, quindi, non si prestano alla flessibilità e allo sviluppo incrementale. Problemi politici potrebbero essere significativi qui – spesso la soluzione più semplice di un problema è cambiare un sistema esistente. Tuttavia, questo richiede la negoziazione con i gestori di quel sistema per convincerli che le modifiche possono essere implementate senza rischi per il funzionamento del sistema.



Figura 3.13 Caratteristiche di un grande progetto.

3. Se più sistemi sono integrati per creare un sistema più grande, una significativa frazione dello sviluppo riguarda la configurazione dei sistemi, anziché lo sviluppo del codice originale. Questo non è necessariamente compatibile con lo sviluppo incrementale e l'integrazione frequente dei sistemi.
4. I grandi sistemi e i loro processi di sviluppo spesso sono vincolati da regole e norme esterne, che limitano il modo in cui possono essere sviluppati; questo richiede che siano prodotti alcuni tipi di documenti per i sistemi e così via. I clienti potrebbero avere specifici requisiti di conformità che devono essere rispettati, e questo potrebbe richiedere che la documentazione dei processi sia completa.
5. I grandi sistemi hanno tempi lunghi di acquisizione e sviluppo. È difficile mantenere team stabili che conoscano il sistema durante un periodo così lungo, in quanto le persone passano inevitabilmente ad altri compiti e progetti.
6. I grandi sistemi di solito hanno gruppi differenti di stakeholder con prospettive e obiettivi differenti. Per esempio, infermieri e contabili potrebbero essere gli utenti finali di un sistema medico, ma anche il personale medico di alto livello, i dirigenti ospedalieri ed altri ancora sono stakeholder del sistema. È praticamente impossibile coinvolgere tutti questi stakeholder nel processo di sviluppo.

Dean Leffingwell, che ha una grande esperienza nella scalabilità dei metodi agili, ha sviluppato *Scaled Agile Framework* (Leffingwell 2007, 2011) per supportare lo sviluppo del software su vasta scala e con più team. Egli riferisce come questo metodo è stato utilizzato con successo in un certo numero di grandi imprese. Anche IBM ha sviluppato un framework per l'uso su vasta scala di metodi agili: *Agile Scaling Model* (ASM). La Figura 3.14, estratta dal libro bianco di Ambler che tratta il modello ASM (Ambler 2010), mostra una panoramica di questo modello.

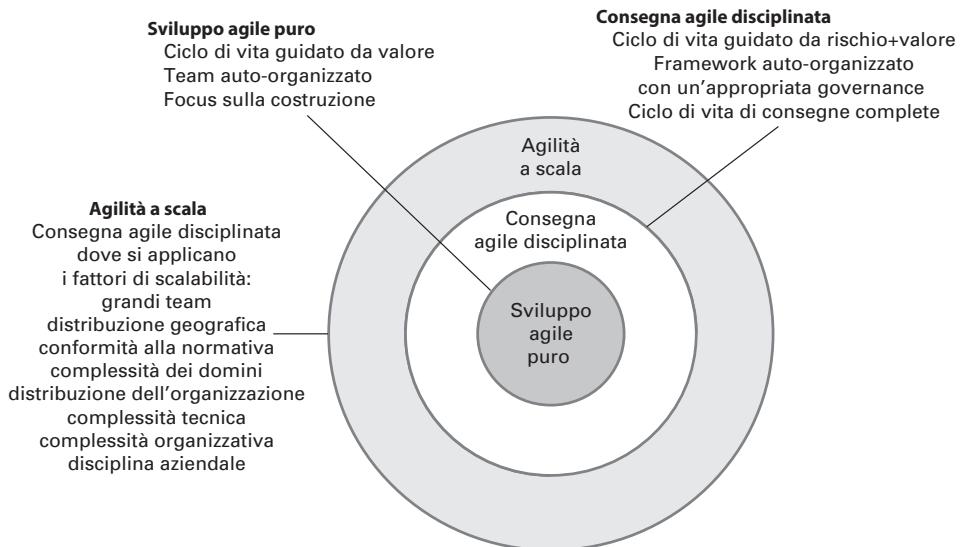


Figura 3.14 Il modello *Agility at Scale* dell'IBM (© IBM 2010).

Il modello ASM riconosce che la scalabilità dei metodi agili è un processo a più stadi, dove i team di sviluppo passano dalle pratiche agili pure (*core agile practices*) qui descritte alla cosiddetta consegna agile disciplinata (*disciplined agile delivery*). Essenzialmente, il processo prevede che queste pratiche si adattino a un ambiente organizzativo disciplinato e che i team non si occupino semplicemente dello sviluppo, ma tengano conto anche di altre fasi del processo di ingegneria del software, quali la definizione dei requisiti e la progettazione architettonica.

Lo stadio finale del modello ASM è il passaggio all'agilità a scala (*agility at scale*), dove viene riconosciuta la complessità che è propria dei grandi progetti. Questo richiede che siano tenuti in considerazione fattori quali lo sviluppo distribuito, ambienti ereditati complessi e requisiti di conformità alle normative. Per tenere conto di questi fattori, potrebbe essere necessario modificare le pratiche utilizzate nella consegna agile disciplinata in funzione del progetto che si sta sviluppando e, in alcuni casi, aggiungere al processo nuove pratiche basate su piani.

Nessun singolo modello è appropriato a tutti i prodotti agili su vasta scala, in quanto il tipo di prodotto, i requisiti del cliente e le persone disponibili sono differenti. Tuttavia, gli approcci alla scalabilità dei metodi agili hanno un certo numero di elementi in comune.

1. Un approccio completamente incrementale all'ingegneria dei requisiti è impossibile. È richiesto un lavoro preliminare sui requisiti del software. Questo lavoro è necessario per identificare le diverse parti del sistema che possono essere sviluppate dai vari team e, spesso, fa parte del contratto di sviluppo del sistema. Tuttavia, questi requisiti non necessariamente devono

essere specificati nei dettagli; i dettagli si sviluppano meglio in modo incrementale.

2. Non ci può essere un singolo product owner o rappresentante del cliente. Persone diverse devono essere coinvolte nelle varie parti del sistema. Queste persone devono comunicare e negoziare continuamente durante il processo di sviluppo.
3. Non è possibile concentrarsi soltanto sul codice del sistema. Occorre più progettazione up-front e documentazione del sistema. L'architettura del software deve essere progettata, e ci deve essere una documentazione che descrive gli aspetti critici del sistema, come gli schemi del database e l'interruzione del lavoro tra i vari team.
4. Devono essere progettati e utilizzati i meccanismi di comunicazione tra i team. Questo implica regolari telefonate e videoconferenze tra i membri dei team e frequenti e brevi riunioni, dove i team si aggiornano sull'avanzamento del proprio lavoro. Per facilitare le comunicazioni, devono essere forniti vari canali di comunicazione, come e-mail, messaggistica istantanea, wiki e social network.
5. L'integrazione continua, nella quale l'intero sistema viene ricostruito ogni volta che uno sviluppatore registra una modifica, è praticamente impossibile quando più programmi distinti devono essere integrati per creare il sistema. Tuttavia, è essenziale ricostruire frequentemente il sistema ed effettuare regolari release del sistema. Sono necessari gli strumenti di gestione della configurazione che supportano lo sviluppo del software con più team.

Il modello Scrum è stato adattato per lo sviluppo su vasta scala. In essenza, il modello del team di Scrum descritto nel Paragrafo 3.3 viene mantenuto, ma vengono definiti più team di Scrum. Le caratteristiche chiave di un modello Scrum multi-team sono qui elencate.

1. *Replica dei ruoli.* Ogni team ha un product owner per la sua componente di lavoro e uno ScrumMaster. Potrebbero esserci un product owner e uno ScrumMaster per l'intero progetto.
2. *Architetti di prodotto.* Ogni team sceglie un architetto di prodotto. Questi architetti collaborano alla progettazione e al progresso dell'intera architettura del sistema.
3. *Allineamento delle release.* Le date delle release del prodotto di ciascun team vengono allineate in modo da produrre un sistema dimostrativo completo.
4. *Scrum di scrum.* Ogni giorno si svolge uno scrum di scrum, dove i rappresentanti di ciascun team si incontrano per discutere l'avanzamento dei lavori, identificare i problemi e pianificare il lavoro da fare in quel giorno. Le riunioni dei singoli team possono essere distribuite nel tempo, in modo che i membri di altri team possano parteciparvi se necessario.

3.4.4 Metodi agili nelle organizzazioni

Le piccole società di software che sviluppano prodotti software sono state fra gli utilizzatori più entusiasti dei metodi agili. Queste società non sono vincolate da standard di processo o dalla burocrazia organizzativa, ma possono cambiare rapidamente per adottare nuove idee. Ovviamente, anche le società più grandi hanno sperimentato i metodi agili in specifici progetti, ma è molto più difficile “estendere” questi metodi a tutta l’organizzazione.

Potrebbe essere difficile introdurre i metodi agili nelle grandi società per vari motivi.

1. I project manager che non hanno esperienza dei metodi agili potrebbero essere riluttanti ad accettare i rischi di un nuovo approccio, in quanto non sanno come tutto questo potrà influire sui loro progetti.
2. Le grandi organizzazioni spesso hanno procedure e standard sulla qualità che tutti i progetti devono rispettare. A causa della natura burocratica delle organizzazioni, è probabile che tali procedure e standard siano incompatibili con i metodi agili. A volte, tutto questo può essere supportato da strumenti software (per esempio, gli strumenti di gestione dei requisiti), e l’utilizzo di questi strumenti diventa obbligatorio per tutti i progetti.
3. I metodi agili operano meglio quando i membri dei team hanno un livello di capacità relativamente alto. Tuttavia, all’interno delle grandi organizzazioni, è facile trovare persone di vari livelli di capacità, e quelle con livelli di capacità più bassi potrebbero non essere membri effettivi dei team nei processi agili.
4. Potrebbe esserci una resistenza culturale ai metodi agili, specialmente in quelle organizzazioni che hanno una lunga esperienza nell’uso dei processi convenzionali di ingegneria dei sistemi.

La gestione delle modifiche e le procedure di test sono esempi di processi aziendali che non possono essere compatibili con i metodi agili. La gestione delle modifiche è il processo che controlla le modifiche apportate a un sistema, in modo che l’impatto delle modifiche sia prevedibile e i costi siano controllati. Tutte le modifiche devono essere approvate in anticipo prima che vengano effettuate, e questo è in conflitto con il concetto di refactoring. Quando il refactoring fa parte di un processo agile, qualsiasi sviluppatore può migliorare qualsiasi codice senza bisogno di un’approvazione esterna. Per i grandi sistemi, ci sono anche gli standard di prova, dove la costruzione di un sistema viene approvata da un team esterno di collaudo. Questo può essere in conflitto con gli approcci con test iniziali utilizzati nei metodi di sviluppo agile.

L’introduzione e il mantenimento dell’uso dei metodi agili in una grande organizzazione è un processo di cambiamento culturale. Il cambiamento culturale richiede tempo e, spesso, implica la sostituzione del management prima che possa effettivamente realizzarsi. Le società che intendono usare i metodi agili hanno bisogno di un evangelist per promuovere il cambiamento. Anziché imporre l’uso

dei metodi agili a sviluppatori recalcitranti, le società hanno scoperto che il modo migliore di introdurre questi metodi è quello del passo dopo passo, partendo da un piccolo gruppo di sviluppatori entusiasti. Il successo di un progetto agile può agire come base di partenza, con il team di progettazione che diffonde le pratiche agili all'interno dell'organizzazione. Una volta che il concetto di sviluppo agile si è largamente diffuso, possono essere intraprese esplicite azioni per affermarlo capillarmente nell'organizzazione.

Punti chiave

- I metodi agili sono metodi di sviluppo iterativo che si concentrano sulla riduzione della documentazione e degli overhead dei processi e sulla consegna incrementale del software. Coinvolgono i rappresentanti del cliente direttamente nel processo di sviluppo.
- La scelta di usare un approccio agile o guidato da piani nello sviluppo del software dipende dal tipo di software da sviluppare, dalle capacità del team di sviluppo e dalla cultura della società che si occupa dello sviluppo. In pratica, si usano tecniche miste di metodi agili e guidati da piani.
- Le pratiche di sviluppo agile includono i requisiti espressi come storie utente, la programmazione a coppie, il refactoring, l'integrazione continua e lo sviluppo con test iniziali.
- Scrum è un metodo agile che fornisce un framework per organizzare i progetti agili. È incentrato su una serie di sprint, che sono periodi di tempo prestabiliti durante viene sviluppato un incremento del sistema. La pianificazione si basa sulla definizione delle priorità di lavoro e sulla scelta dei compiti a priorità più alta per uno sprint.
- La scalabilità dei metodi agili richiede che alcune pratiche basate su piani siano integrate con le pratiche agili. Queste includono i requisiti up-front, più documentazione, più rappresentanti del cliente, strumenti comuni per i team di progettazione e allineamento delle release fra i vari team.

Esercizi

- 3.1 Spiegate perché per le aziende spesso è più importante la consegna rapida dei nuovi sistemi piuttosto che la loro dettagliata funzionalità.
- * 3.2 Spiegate come i principi che stanno alla base dei metodi agili portano allo sviluppo e alla consegna accelerata del software.
- * 3.3 La programmazione estrema esprime i requisiti dell'utente sotto forma di storie, ognuna scritta su una scheda. Discutete i vantaggi e gli svantaggi di tale approccio alla descrizione dei requisiti.
- 3.4 Spiegate perché lo sviluppo con test iniziali aiuta il programmatore a sviluppare una migliore conoscenza dei requisiti di sistema. Quali sono le possibili difficoltà dello sviluppo con test iniziali?

- 3.5 Suggerite quattro ragioni per cui il tasso di produttività dei programmatorei che lavorano in coppia potrebbe essere più della metà di quello di due programmatorei che lavorano singolarmente.
- * 3.6 Confrontate l'approccio Scrum alla gestione della progettazione con gli approcci convenzionali guidati da piani descritti nel Capitolo 20. Il confronto dovrebbe basarsi sull'efficacia di ciascun approccio riguardo alla pianificazione dell'allocatione delle persone ai progetti, alla stima del costo dei progetti, al mantenimento della coesione dei team e alla gestione delle modifiche nei membri dei team di progettazione.
- 3.7 Per ridurre i costi e l'impatto ambientale della trasformazione, la vostra società ha deciso di chiudere un certo numero di uffici e fornire supporto al personale perché possa svolgere il lavoro da casa. Tuttavia, l'alto management che ha preso la decisione non è al corrente che il software è sviluppato tramite il metodo Scrum. Spiegate come utilizzereste la tecnologia per supportare Scrum in un ambiente distribuito per rendere possibile questo. Quali problemi potreste incontrare utilizzando questo approccio?
- * 3.8 Perché è necessario introdurre alcuni metodi e documenti tipici degli approcci guidati da piani quando si estendono i metodi agili ai progetti più grandi che sono sviluppati da team distribuiti?
- 3.9 Spiegate perché potrebbe essere difficile utilizzare i metodi agili in un grande progetto per sviluppare un nuovo sistema informativo che dovrà fare parte di un sistema di sistemi di una organizzazione.
- * 3.10 Uno dei problemi di avere un utente strettamente coinvolto nel team di sviluppo del software è che esso adotta la visione del team di sviluppo e perde di vista le esigenze dei suoi colleghi utenti. Suggerite tre modi per evitare questo problema; discutete i vantaggi e gli svantaggi di ciascun approccio.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Get ready for agile methods, with care”. Un’attenta critica dei metodi agili che discute i loro punti di forza e debolezza, scritta da un ingegnere del software dalla vasta esperienza (B. Boehm, *IEEE Computer*, gennaio 2002) <http://dx.doi.org/10.1109/2.976920>

Extreme Programming Explained. Il primo libro sulla programmazione estrema e, probabilmente, il più leggibile. Spiega l’approccio dalla prospettiva di uno dei suoi inventori, il cui entusiasmo si nota chiaramente (K. Beck e C. Andres, Addison-Wesley, 2004).

Essential Scrum: A Practical Guide to the Most Popular Agile Process. Una descrizione completa e chiara dello sviluppo del metodo Scrum nel 2011 (K.S. Rubin, Addison-Wesley, 2013).

“Agility at Scale: Economic Governance, Measured Improvement and Disciplined Delivery.” Questo articolo descrive l’approccio IBM ai metodi agili, dove si ha un approccio sistematico all’integrazione dello sviluppo agile con quello guidato da piani. È un’eccellente ed esaurente discussione dei temi chiave della scalabilità dei metodi agili (A.W. Brown, S.W. Ambler e W. Royce, *Proc. 35th Int. Conf. on Software Engineering*, 2013) <http://dx.doi.org/10.1145/12944.12948>

4

Ingegneria dei requisiti

Questo capitolo si propone di descrivere i requisiti del software e spiegare i processi coinvolti nella scoperta e nella documentazione di questi requisiti. Dopo aver letto questo capitolo:

- capirete i concetti di requisiti dell'utente e requisiti del sistema, e il motivo per cui devono essere scritti in modi diversi;
- conoscerete le differenze tra requisiti funzionali e non funzionali del software;
- apprenderete le principali attività di ingegneria dei requisiti (deduzione, analisi e convalida) e le loro relazioni;
- capirete perché è necessaria la gestione dei requisiti e come questa supporta le altre attività di ingegneria dei requisiti.

- 4.1 Requisiti funzionali e non funzionali
- 4.2 Processi di ingegneria dei requisiti
- 4.3 Deduzione dei requisiti
- 4.4 Specifica dei requisiti
- 4.5 Convalida dei requisiti
- 4.6 Modifica dei requisiti

I requisiti di un sistema sono la descrizione dei servizi che il sistema deve fornire e dei suoi vincoli operativi. Queste caratteristiche riflettono l'esigenza del cliente di avere un sistema che aiuti a risolvere alcuni problemi, come controllare un dispositivo, effettuare un ordine o trovare informazioni. Il processo di ricerca, analisi, documentazione e verifica di questi servizi e vincoli è chiamato *ingegneria dei requisiti* (RE, Requirements Engineering).

Il termine *requisito* non è usato nell'industria del software in modo coerente. In alcuni casi, un requisito è semplicemente una formulazione astratta e di alto livello di un servizio che il sistema dovrebbe fornire, oppure un vincolo del sistema. Altre volte, può essere una definizione formale e dettagliata di una funzione del sistema. Davis (Davis 1993) spiega perché esistono queste differenze:

*Se una società vuole dare in appalto un grande progetto di sviluppo software, deve definire le sue esigenze in modo abbastanza astratto da non predefinire alcuna soluzione. I requisiti devono essere scritti in modo che diversi appaltatori possano fare le offerte proponendo vari metodi per soddisfare le necessità del cliente. Dopo che l'appalto è stato assegnato, l'appaltatore deve scrivere per il cliente una definizione del sistema molto dettagliata, in modo che il cliente possa capire e verificare che cosa farà il software. Entrambi questi documenti possono essere chiamati documenti dei requisiti del sistema.*¹

Alcuni dei problemi che sorgono durante il processo di ingegneria dei requisiti derivano dalla mancanza di una separazione netta tra questi diversi livelli di descrizione: i requisiti dell'utente indicano i requisiti astratti di alto livello; i requisiti del sistema danno la descrizione dettagliata di quello che il sistema dovrebbe fare.

1. *Requisiti dell'utente*: dichiarano, nel linguaggio naturale e corredati da diagrammi, quali servizi il sistema dovrebbe fornire e i vincoli sotto cui deve operare. Possono variare da descrizione generali delle caratteristiche del sistema a descrizioni dettagliate e precise delle funzionalità del sistema.
2. *Requisiti del sistema*: sono descrizioni più dettagliate delle funzioni, dei servizi e dei vincoli operativi del sistema software. Il documento dei requisiti del sistema (a volte chiamato specifica funzionale) dovrebbe definire esattamente che cosa deve essere implementato. Può essere parte del contratto tra l'acquirente del sistema e gli sviluppatori del software.

Diversi livelli di specifiche del sistema sono utili, perché comunicano le informazioni che riguardano il sistema a diversi tipi di utilizzatori dei documenti o “lettori”. La Figura 4.1 illustra la distinzione tra i requisiti dell'utente e quelli del sistema. Questo esempio, tratto da un sistema informativo medico per la cura di pazienti con problemi di salute mentale (Mentcare), mostra come un requisito

¹ Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.

Definizione dei requisiti dell’utente

- 1.** Il sistema Mentcare genererà mensilmente dei rapporti che riportano il costo dei farmaci prescritti da ciascuna clinica durante il mese.

Specifiche dei requisiti del sistema

- 1.1** Nell’ultimo giorno lavorativo di ogni mese, dovrà essere generata una sintesi dei farmaci prescritti, il loro costo e le cliniche che li hanno prescritti.
- 1.2** Il sistema genererà il rapporto per la stampa dopo le 17:30 dell’ultimo giorno lavorativo di ciascun mese.
- 1.3** Sarà creato un rapporto per ciascuna clinica; dovranno essere elencati i nomi dei singoli farmaci, il numero totale di prescrizioni, il numero delle dosi prescritte e il costo totale dei farmaci prescritti.
- 1.4** Se i farmaci sono disponibili in dosi unitarie differenti (per esempio, 10 mg, 20 mg ecc.), dovranno essere creati dei rapporti separati per ciascuna dose unitaria.
- 1.5** L’accesso ai rapporti dei costi dei farmaci dovrà essere limitato agli utenti autorizzati, come indicato nella lista di controllo degli accessi.

Figura 4.1 Requisiti dell’utente e requisiti del sistema.

dell’utente può essere espanso in diversi requisiti del sistema. Come potete notare dalla Figura 4.1, il requisito dell’utente è generico. I requisiti del sistema forniscono informazioni più specifiche sui servizi e sulle funzioni del sistema che deve essere implementato.

Si dovrebbero scrivere i requisiti con vari livelli di dettaglio, perché tipi diversi di lettori li usano in modo differente. La Figura 4.2 mostra i diversi tipi di lettori dei due tipi di requisiti. I lettori dei requisiti dell’utente solitamente non si occupano del modo in cui il sistema sarà implementato; possono essere manager disinteressati ai dettagli dei servizi del sistema. I lettori dei requisiti del sistema hanno bisogno di sapere con più precisione che cosa il sistema dovrà fare, perché sono interessati all’aiuto che potrà dare ai processi aziendali, oppure perché sono coinvolti nell’implementazione del sistema stesso.

I vari tipi di lettori dei documenti illustrati nella Figura 4.2 sono esempi di stakeholder del sistema. Come gli utenti, anche altre persone hanno un certo interesse verso il sistema. Gli stakeholder del sistema includono chiunque sia influenzato dal sistema in qualche maniera e chiunque abbia un legittimo interesse verso di esso. Gli stakeholder possono essere utenti finali del sistema, manager o persone esterne, come le autorità di controllo che certificano l’accettabilità del sistema. Per esempio, gli stakeholder per il sistema Mentcare includono:

1. i pazienti le cui informazioni sono registrate nel sistema e i parenti di questi pazienti;
2. i dottori che visitano e curano i pazienti;

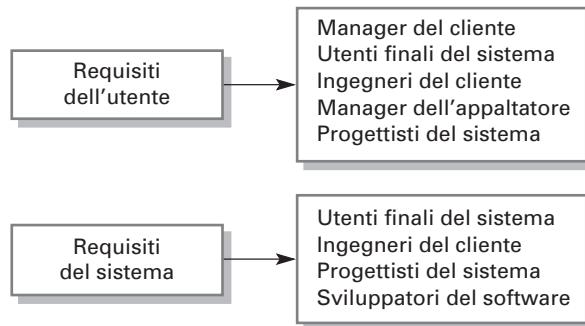


Figura 4.2 I lettori dei diversi tipi di specifica dei requisiti.

3. le infermiere che coordinano i consulti con dottori e somministrano qualche farmaco;
4. il personale medico che gestisce gli appuntamenti con i pazienti;
5. il personale informatico che è responsabile dell'installazione e della manutenzione del sistema;
6. un direttore etico che assicura che il sistema soddisfi la normativa etica per il trattamento dei pazienti;
7. i manager dell'assistenza sanitaria che ottengono dal sistema informazioni gestionali;
8. il personale addetto alle cartelle cliniche che ha la responsabilità di garantire che le informazioni del sistema siano mantenute e preservate, e che le procedure di memorizzazione delle cartelle cliniche siano state appropriatamente implementate.

L'ingegneria dei requisiti di solito è presentata come la prima fase del processo di ingegneria del software. Tuttavia, occorre capire un po' meglio i requisiti del sistema prima di prendere una decisione per procedere con l'acquisizione o lo sviluppo di un sistema. Questa fase iniziale dell'ingegneria dei requisiti fornisce una vista generale di ciò che il sistema dovrebbe fare e dei benefici che dovrebbe apportare. Tutto questo dovrebbe essere poi esaminato in uno studio di fattibilità, che tenterà di stabilire se il sistema è tecnicamente e finanziariamente fattibile. I risultati di tale studio aiutano il management a decidere se procedere con l'acquisizione o lo sviluppo del sistema.

In questo capitolo presento una vista “tradizionale” dei requisiti, anziché i requisiti nei processi agili, che ho descritto nel Capitolo 3. Per la maggior parte dei grandi sistemi, è ancora il caso che ci sia una fase di ingegneria dei requisiti chiaramente identificabile prima che inizi l'implementazione del sistema. Il risultato è un documento dei requisiti, che può essere una parte del contratto di sviluppo del sistema. Ovviamente, le modifiche successive vengono fatte ai requisiti, e

Studi di fattibilità

Uno studio di fattibilità è un breve studio che dovrebbe svolgersi nella prima fase del processo di ingegneria dei requisiti. Dovrebbe a rispondere a tre domande chiave: (1) il sistema contribuirà al raggiungimento degli obiettivi generali dell'organizzazione? (2) il sistema può essere implementato usando la tecnologia attuale nel rispetto dei costi e dei tempi prefissati? (3) il sistema può essere integrato con altri sistemi che sono già utilizzati?

Se la risposta a una di queste domande è no, probabilmente non dovreste procedere con il progetto.

<http://software-engineering-book.com/web/feasibility-study/>

i requisiti dell'utente possono essere espansi in requisiti del sistema più dettagliati. A volte, può essere utilizzato un approccio agile di deduzione dei requisiti mentre il sistema viene sviluppato per aggiungere dettagli e perfezionare i requisiti dell'utente.

4.1 Requisiti funzionali e non funzionali

I requisiti dei sistemi software sono spesso divisi in requisiti funzionali e non funzionali.

1. *Requisiti funzionali*: sono definizioni di servizi che il sistema deve fornire; indicano come il sistema dovrebbe reagire a particolari input e come dovrebbe comportarsi in particolari situazioni. In alcuni casi possono stabilire esplicitamente che cosa il sistema non dovrebbe fare.
2. *Requisiti non funzionali*: sono vincoli sulle funzioni o servizi offerti dal sistema. Includono vincoli temporali e sul processo di sviluppo e vincoli imposti dagli standard. I requisiti non funzionali di solito si applicano al sistema completo, non a singole funzioni o servizi.

In realtà la distinzione tra questi due tipi di requisiti non è così netta come queste semplici definizioni suggeriscono. Un requisito dell'utente riguardante la sicurezza, come una definizione che limita l'accesso agli utenti autorizzati, potrebbe sembrare un requisito non funzionale. Tuttavia, quando questo requisito viene sviluppato nei dettagli, può produrre altri requisiti che sono chiaramente funzionali come, per esempio, la necessità di includere nel sistema funzioni di autenticazione dell'utente.

Questo dimostra che i requisiti non sono indipendenti e che un requisito spesso genera o limita altri requisiti. I requisiti del sistema pertanto non specificano semplicemente le funzioni o i servizi che servono al sistema, ma anche le funzionalità necessarie per garantire che questi servizi/funzioni siano realizzati in modo efficace.

4.1.1 Requisiti funzionali

I requisiti funzionali descrivono ciò che il sistema dovrebbe fare. Questi requisiti dipendono dal tipo di software che si sta sviluppando, dai futuri utenti e dall’approccio generale adottato dall’organizzazione durante la scrittura dei requisiti. Se espressi come requisiti dell’utente, i requisiti funzionali devono essere scritti nel linguaggio naturale, in modo che gli utenti del sistema e i manager possano capirli. I requisiti funzionali del sistema estendono i requisiti dell’utente e sono scritti per gli sviluppatori del sistema; descrivono dettagliatamente le funzioni del sistema, i suoi input e output, e le eccezioni.

I requisiti funzionali variano da descrizioni generiche che riguardano che cosa il sistema deve fare a definizioni specifiche che rispecchiano i modi locali di lavorare o i sistemi esistenti di un’organizzazione. Per esempio, ecco alcuni requisiti funzionali per il sistema Mentcare che vengono utilizzati per registrare le informazioni sui pazienti curati per problemi di salute mentale.

1. Un utente deve essere in grado di cercare gli appuntamenti nelle liste di tutte le cliniche.
2. Il sistema deve generare ogni giorno, per ciascuna clinica, l’elenco dei pazienti che hanno un appuntamento.
3. Ogni membro del personale che usa il sistema dovrà essere identificato in modo univoco mediante un codice di otto cifre.

Questi requisiti funzionali dell’utente definiscono specifiche funzionalità che devono essere incluse nel sistema. I requisiti mostrano che i requisiti funzionali possono essere scritti a diversi livelli di dettaglio (si confrontino i requisiti 1 e 3).

I requisiti funzionali, come suggerisce il nome, tradizionalmente sono incentrati su ciò che il sistema deve fare. Tuttavia, se un’organizzazione decide che un prodotto software esistente può soddisfare le sue necessità, non ha più senso sviluppare una specifica dettagliata delle funzioni. In tali casi, l’attenzione dovrebbe rivolgersi verso lo sviluppo di requisiti che specificano le informazioni che servono alle persone per svolgere il loro lavoro. I requisiti delle informazioni specificano le informazioni necessarie e come devono essere consegnate e organizzate. Un esempio di requisiti delle informazioni per il sistema Mentcare potrebbe essere quello di specificare quali informazioni devono essere incluse nella lista dei pazienti che hanno un appuntamento in un determinato giorno.

L’imprecisione nella specifica dei requisiti può provocare discussioni tra clienti e sviluppatori del software. È naturale per uno sviluppatore di sistema interpretare un requisito ambiguo in un modo che semplifichi la sua implementazione. Spesso, però, questo non è quello che vuole il cliente. Devono essere stabiliti nuovi requisiti e apportate modifiche al sistema. Ovviamente, questo rallenta la consegna del sistema e aumenta i costi.

Per esempio, il primo requisito del sistema Mentcare nella lista precedente stabilisce che un utente dovrà essere in grado di cercare gli appuntamenti nelle

Requisiti di dominio

I requisiti di dominio sono derivati dal dominio di applicazione del sistema, anziché da specifiche esigenze degli utenti del sistema. Possono essere nuovi requisiti funzionali a sé stanti, limitare i requisiti funzionali esistenti o specificare come devono essere eseguiti particolari calcoli.

Il problema dei requisiti di dominio è che gli ingegneri del software potrebbero non capire le caratteristiche del dominio in cui opera il sistema. Ciò significa che questi ingegneri potrebbero non sapere se un requisito di dominio è stato soddisfatto o se è in conflitto con altri requisiti.

<http://software-engineering-book.com/web/domain-requirements/>

liste di tutte le cliniche. La logica di questo requisito è che i pazienti con problemi di salute mentale a volte si confondono; potrebbero avere un appuntamento in una clinica, ma in effetti si recano in un'altra clinica. Se hanno un appuntamento, saranno registrati come pazienti in cura, indipendentemente dalla clinica.

Un membro del personale medico che specifica un requisito di ricerca potrebbe pensare che “ricerca” significhi che, dato il nome di un paziente, il sistema cercherà questo nome in tutte le liste di appuntamenti di tutte le cliniche. Tuttavia, questo non è così esplicito nel requisito. Gli sviluppatori del sistema potrebbero interpretare il requisito in modo che sia più facile da implementare. La loro funzione di ricerca potrebbe richiedere all'utente di scegliere una clinica e poi avviare la ricerca fra i pazienti che sono in cura in quella clinica. Questo richiede più input all'utente e quindi rende più lunga la ricerca.

In linea di principio, le specifiche dei requisiti funzionali di un sistema dovrebbero essere complete e coerenti: complete nel senso che tutti i servizi richiesti dagli utenti devono essere definiti; coerenti in quanto i requisiti non devono avere definizioni contraddittorie.

In pratica, è possibile raggiungere la coerenza e la completezza dei requisiti soltanto per sistemi software molto piccoli. Uno dei motivi è che è facile commettere errori e omissioni quando si scrivono le specifiche di sistemi grandi e complessi. Un altro motivo è che i grandi sistemi hanno molti stakeholder, con background e aspettative differenti. Gli stakeholder di solito hanno anche esigenze differenti, spesso incoerenti. Queste incoerenze possono non essere evidenti nel momento in cui vengono specificati i requisiti e, quindi, possono emergere soltanto dopo un'approfondita analisi o durante lo sviluppo del sistema.

4.1.2 Requisiti non funzionali

I requisiti non funzionali, come suggerisce il nome, sono requisiti che non riguardano direttamente specifici servizi forniti dal sistema ai suoi utenti. Di solito specificano o limitano le caratteristiche del sistema nel suo complesso. Possono riferirsi a proprietà del sistema, come l'affidabilità, i tempi di risposta e l'uso

della memoria; ma possono anche definire vincoli sull’implementazione del sistema, come le capacità dei dispositivi di I/O o la rappresentazione dei dati utilizzata nelle interfacce con altri sistemi.

I requisiti non funzionali sono spesso più critici dei singoli requisiti funzionali. Gli utenti del sistema di solito trovano il modo di aggirare funzioni che non corrispondono appieno alle loro necessità. Tuttavia, se non vengono soddisfatti i requisiti non funzionali, l’intero sistema potrebbe risultare inutilizzabile. Per esempio, se un sistema di controllo degli aerei non soddisfa i requisiti di affidabilità, non potrà ottenere la certificazione di sicurezza necessaria per operare; se un sistema integrato per il controllo in tempo reale non raggiunge le prestazioni richieste, le sue funzioni di controllo non potranno operare correttamente.

Sebbene spesso sia possibile identificare quali componenti del sistema implementino specifici requisiti funzionali (per esempio, ci possono essere componenti di formattazione che implementano i requisiti di reporting), questo è più difficile con i requisiti non funzionali. L’implementazione di questi requisiti può interessare l’intero sistema, per due ragioni:

1. i requisiti non funzionali possono influire sull’intera architettura di un sistema, anziché su singoli componenti. Per esempio, per garantire che i requisiti delle prestazioni siano soddisfatti in un sistema integrato, potrebbe essere necessario organizzare il sistema per minimizzare le comunicazioni tra i componenti;
2. un singolo requisito non funzionale, come quello sulla protezione, potrebbe generare vari requisiti funzionali tra loro correlati che definiscono nuovi servizi del sistema, che si rendono necessari se il requisito non funzionale deve essere implementato. In aggiunta, potrebbe generare requisiti che limitano quelli esistenti; per esempio, potrebbe limitare l’accesso alle informazioni nel sistema.

I requisiti non funzionali derivano dalle necessità degli utenti, dai limiti del budget, dalle politiche organizzative, dalla necessità di interoperabilità con altri sistemi software o hardware, da fattori esterni come norme di sicurezza o legislazioni sulla privacy. La Figura 4.3 mostra una classificazione dei requisiti non funzionali che, come si può vedere dallo schema, possono derivare da caratteristiche richieste dal software (requisiti del prodotto), dall’organizzazione che sta sviluppando il software (requisiti organizzativi) o da sorgenti esterne.

1. *Requisiti del prodotto*: specificano o limitano il comportamento del software; per esempio, i requisiti di prestazioni che definiscono la velocità di esecuzione del programma e la quantità di memoria richiesta, i requisiti di affidabilità che delineano il tasso di fallimenti accettabili, i requisiti di usabilità.
2. *Requisiti organizzativi*: sono requisiti generali del sistema che derivano dalle politiche e dalle procedure delle organizzazioni del cliente e dello

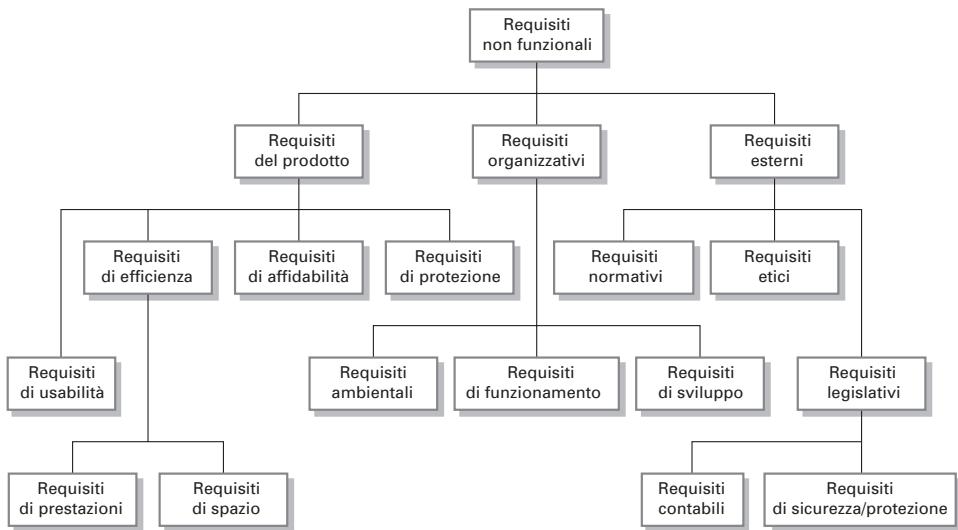


Figura 4.3 Tipi di requisiti non funzionali.

sviluppatore; per esempio, gli standard di lavorazione da seguire, i requisiti del processo di sviluppo che specificano il linguaggio di programmazione, l'ambiente di sviluppo o gli standard dei processi da utilizzare, i requisiti ambientali che specificano il contesto operativo del sistema.

3. *Requisiti esterni*: con questo termine generico si intendono tutti i requisiti che derivano da fattori esterni al sistema e al suo processo di sviluppo; per esempio, i requisiti normativi che stabiliscono cosa deve essere fatto affinché il sistema sia approvato da un ente di controllo, come l'autorità sulla sicurezza nucleare, i requisiti legislativi che devono essere rispettati affinché il sistema operi all'interno delle leggi, i requisiti etici che assicurano che il sistema sia accettato dai suoi utenti e da un pubblico più vasto.

La Figura 4.4 mostra alcuni esempi di requisiti di prodotto, organizzativi ed esterni che potrebbero essere inseriti nel sistema Mentcare. I requisiti del prodotto sono requisiti di disponibilità che stabiliscono quando il sistema deve essere disponibile e il periodo di inattività consentito ogni giorno. Non dicono nulla sulla funzionalità del sistema Mentcare e identificano chiaramente un limite che deve essere considerato dai progettisti del sistema.

I requisiti organizzativi specificano come gli utenti devono identificarsi al sistema. L'autorità sanitaria che controlla il sistema vuole adottare a una procedura standard di autenticazione per tutto il software secondo la quale gli utenti, anziché digitare il loro nome, dovranno strisciare in un lettore magnetico la carta di identità del servizio sanitario per identificarsi. I requisiti esterni derivano dalla necessità per il sistema di essere conforme alle leggi sulla privacy. Ovviamente,

Requisiti del prodotto

Il sistema Mentcare dovrà essere disponibile a tutte le cliniche durante l'orario normale di lavoro (lun-ven, 08:30-17:30). Il periodo di inattività all'interno dell'orario normale di lavoro non dovrà superare 5 secondi durante la giornata.

Requisiti organizzativi

Gli utenti del sistema Mentcare dovranno identificarsi utilizzando la carta di identità rilasciata dall'autorità del servizio sanitario.

Requisiti esterni

Il sistema dovrà implementare le disposizioni sulla privacy dei pazienti definite nello standard HStan-03-2006-priv.

Figura 4.4 Esempi di possibili requisiti non funzionali per il sistema Mentcare.

la privacy è un problema molto importante nei sistemi di assistenza sanitaria, e i requisiti specificano che il sistema deve essere sviluppato in accordo a uno standard nazionale sulla privacy.

Un problema comune dei requisiti non funzionali è che gli stakeholder propongono i requisiti come obiettivi generici, per esempio facilità d'uso, capacità di ripristino del sistema dopo un malfunzionamento o rapida risposta agli utenti. Questi obiettivi vaghi causano problemi agli sviluppatori, perché danno adito alla libera interpretazione e alla successiva discussione quando il sistema viene consegnato. Per esempio, il seguente obiettivo del sistema è tipico di come un manager potrebbe esprimere i requisiti di usabilità:

Il sistema deve essere facile da usare per il personale medico e deve essere organizzato in modo tale che gli errori commessi dall'utente siano ridotti al minimo.

Ho riscritto questo per mostrare come l'obiettivo potrebbe essere espresso come un requisito non funzionale “verificabile”. È impossibile verificare oggettivamente l'obiettivo del sistema, ma nella seguente descrizione si potrebbe almeno includere una strumentazione software per contare gli errori fatti dagli utenti quando provano il sistema.

Il personale medico dovrà essere in grado di usare tutte le funzioni del sistema dopo due ore di addestramento; dopodiché, il numero medio di errori commessi dagli utenti esperti non dovrà maggiore di due, per ogni ora di utilizzo del sistema.

Quando è possibile, bisognerebbe descrivere i requisiti non funzionali quantitativamente, in modo che possano essere verificati in maniera oggettiva; la Figura 4.5 mostra una serie di possibili misurazioni da usare per specificare le proprietà non funzionali di un sistema. È possibile misurare queste caratteristiche quando il

Proprietà	Misura
Velocità	Transazioni elaborate al secondo Tempi di risposta a utenti/eventi Tempo di refresh dello schermo
Dimensione	Megabyte/Numero di chip ROM
Facilità d'uso	Tempo di addestramento Numero di maschere d'aiuto
Affidabilità	Tempo medio di malfunzionamento Probabilità di indisponibilità Tasso di malfunzionamenti Disponibilità
Robustezza	Tempo per il riavvio dopo un malfunzionamento Percentuale di eventi che causano malfunzionamenti Probabilità di corruzione dei dati dopo un malfunzionamento
Portabilità	Percentuale di istruzioni dipendenti dal sistema target Numero di sistemi target

Figura 4.5 Misure per specificare i requisiti non funzionali.

sistema viene testato per controllare se soddisfa o no i suoi requisiti non funzionali.

Nella pratica, però, i clienti di un sistema spesso trovano difficile tradurre i propri obiettivi in requisiti misurabili. Per alcuni obiettivi, come la mantenibilità, non ci sono semplici misurazioni che possano essere effettuate. In altri casi, anche quando una specifica quantitativa è possibile, i clienti potrebbero non essere in grado di correlare le loro necessità con queste specifiche: non riescono a scegliere un numero che possa definire l'affidabilità del sistema (per esempio) in termini della loro esperienza quotidiana con i computer. Inoltre, poiché i costi della verifica oggettiva di requisiti non funzionali misurabili possono essere molto alti, il cliente potrebbe considerarli ingiustificati.

I requisiti non funzionali spesso contrastano o interagiscono con altri requisiti funzionali o non funzionali. Per esempio, i requisiti organizzativi nella Figura 4.4 richiedono che un lettore magnetico sia installato in ogni computer che si collega al sistema. Tuttavia, un altro requisito potrebbe richiedere che dottori e infermieri debbano avere la possibilità di accedere al sistema tramite tablet o smartphone. Questi dispositivi di solito non sono equipaggiati con i lettori magnetici, quindi, in questi casi, dovrà essere supportato un altro metodo di identificazione.

Nel documento dei requisiti è difficile separare i requisiti funzionali da quelli non funzionali. Se i requisiti non funzionali vengono dichiarati separatamente da quelli funzionali, potrebbe essere difficile capire le loro relazioni. Teoricamente,

bisognerebbe evidenziare quei requisiti che sono chiaramente correlati a proprietà visibili del sistema, come le prestazioni o l'affidabilità, per esempio creando una sezione separata del documento dei requisiti o distinguendoli in qualche maniera dagli altri requisiti del sistema.

I requisiti non funzionali, come i requisiti di sicurezza, protezione e riservatezza, sono particolarmente importanti per i sistemi critici. Tratterò questi requisiti di fidatezza nella Parte II, che descrive i modi di specificare i requisiti di affidabilità, sicurezza e protezione.

4.2 Processi di ingegneria dei requisiti

Come detto nel Capitolo 2, l'ingegneria dei requisiti è formata da tre attività chiave; scoperta dei requisiti attraverso l'interazione con gli stakeholder (deduzione e analisi), conversione dei requisiti in una forma standard (specificata) e controllo che i requisiti definiscano realmente il sistema voluto dal cliente (convalida). Nella Figura 2.4 sono illustrate queste attività come processi sequenziali. In pratica, però, l'ingegneria dei requisiti è un processo iterativo in cui queste attività sono interacciate.

La Figura 4.6 mostra come queste attività sono interacciate. Le attività sono organizzate come un processo iterativo attorno a una spirale. L'output del processo di ingegneria dei requisiti è un documento dei requisiti del sistema. La quantità di tempo e gli sforzi dedicati a ogni attività in una iterazione dipende dallo stadio del processo generale, dal tipo di sistema che si sta sviluppando e dal budget disponibile.

Nelle prime fasi del processo saranno spesi più sforzi per comprendere i requisiti aziendali di alto livello, i requisiti non funzionali e i requisiti dell'utente del sistema. Nelle fasi finali, i giri più esterni della spirale, gli sforzi saranno dedicati principalmente alla deduzione e alla comprensione dei requisiti non funzionali e alla definizione più dettagliata dei requisiti del sistema.

Il modello a spirale si adatta agli approcci in cui i requisiti sono sviluppati con diversi livelli di dettaglio: il numero di iterazioni attorno alla spirale può variare, quindi si può uscire dalla spirale dopo aver raccolto tutti o alcuni dei requisiti dell'utente. Si potrebbe utilizzare lo sviluppo agile, anziché la prototipazione, in modo che i requisiti e l'implementazione del sistema possano essere sviluppati insieme.

Teoricamente, i requisiti cambiano in tutti i sistemi. Le persone coinvolte migliorano le conoscenze su ciò che il software debba effettivamente fare; l'organizzazione che acquista il sistema cambia; l'hardware, il software e l'ambiente organizzativo cambiano. Questi cambiamenti devono essere gestiti per capire il loro impatto su altri requisiti, sui costi e sul sistema. Il Paragrafo 4.6 descrive il processo di gestione dei requisiti.

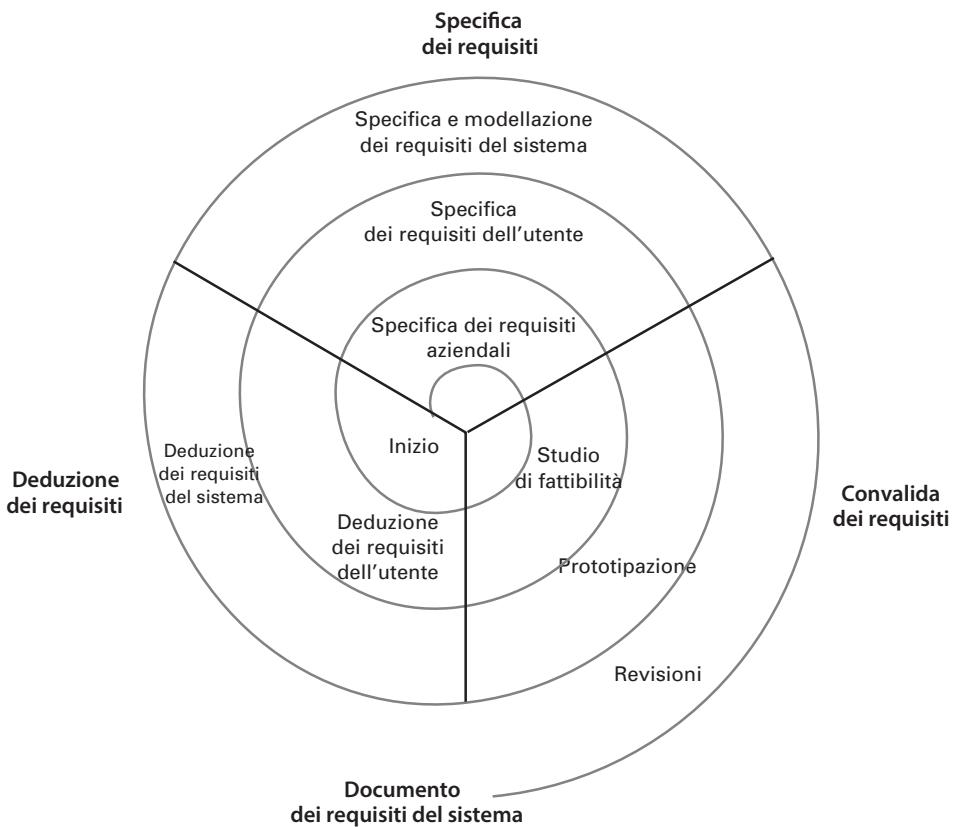


Figura 4.6 Il modello a spirale del processo di ingegneria dei requisiti.

4.3 Deduzione dei requisiti

L’obiettivo principale del processo di deduzione dei requisiti consiste nel capire il lavoro svolto dagli stakeholder e come essi potrebbero utilizzare un nuovo sistema a supporto del loro lavoro. Durante la fase di deduzione dei requisiti, gli ingegneri del software lavorano con gli stakeholder per scoprire informazioni sul dominio di applicazione, sulle attività, sui servizi e sulle caratteristiche che il sistema dovrebbe fornire loro, sulle prestazioni richieste dal sistema, sui vincoli hardware e così via.

La deduzione e l’analisi dei requisiti degli stakeholder è un processo difficile per varie ragioni.

1. Gli stakeholder spesso non sanno cosa vogliono da un sistema informatico, se non in termini molto generici; potrebbero trovare difficoltà nell’esprimere che cosa intendono far fare al sistema o potrebbero fare richieste irrealizzabili perché non sanno che cosa può essere realmente fatto con un sistema software.

2. Gli stakeholder naturalmente esprimono i requisiti nei propri termini e con la conoscenza implicita del proprio lavoro. Gli ingegneri dei requisiti che non hanno esperienza delle attività del cliente potrebbero non capire questi requisiti.
3. Diversi stakeholder hanno requisiti diversi che possono essere espressi in modi differenti. Gli ingegneri dei requisiti devono considerare tutte le possibili fonti dei requisiti e scoprire le parti in comune e quelle in conflitto.
4. Fattori politici possono influenzare i requisiti del sistema; per esempio i manager potrebbero inserire requisiti che aumentano la loro influenza nell’organizzazione.
5. L’ambiente economico e aziendale in cui si analizzano i requisiti è dinamico; esso cambia inevitabilmente durante il processo di analisi, quindi l’importanza di un particolare requisito può cambiare oppure possono emergere nuovi requisiti da nuovi stakeholder che non sono stati consultati inizialmente.

Un modello del processo di deduzione e analisi è mostrato nella Figura 4.7. Ogni organizzazione ha una propria versione o rappresentazione di questo modello generale, a seconda dei fattori locali, come l’esperienza del personale, il tipo di sistema che si sta sviluppando e gli standard utilizzati. Le attività del processo sono le seguenti:

1. *scoperta e comprensione dei requisiti*: il processo di interazione con gli stakeholder del sistema per scoprire i loro requisiti; in questa attività vengono scoperti anche i requisiti di dominio degli stakeholder e la loro documentazione;
2. *classificazione e organizzazione dei requisiti*: l’attività che, partendo dalla raccolta non strutturata dei requisiti, raggruppa quelli correlati e li organizza in gruppi coerenti;
3. *negoziazione e priorità dei requisiti*: inevitabilmente, quando sono coinvolti più stakeholder, i requisiti possono essere in conflitto. Questa attività si occupa di dare una priorità ai requisiti, trovare e risolvere i conflitti attraverso la negoziazione;
4. *documentazione dei requisiti*: i requisiti vengono documentati e diventano l’input del successivo giro della spirale. In questa fase possono essere prodotte le prime bozze dei documenti dei requisiti software, oppure i requisiti possono essere mantenuti in modo informale su lavagne bianche, wiki o altri spazi condivisi.

La Figura 4.7 mostra che la deduzione e l’analisi dei requisiti sono un processo iterativo con un continuo feedback da ciascuna attività alle altre. Il ciclo del processo inizia con la scoperta dei requisiti e finisce con la loro documentazione. La comprensione dei requisiti da parte dell’analista migliora a ogni giro del ciclo. Il ciclo termina quando il documento dei requisiti è completato.



Figura 4.7 Il processo di deduzione e analisi dei requisiti.

Per semplificare l'analisi dei requisiti, è utile organizzare e raggruppare le informazioni degli stakeholder. Il modo più comune per farlo è considerare ciascun gruppo di stakeholder come un punto di vista e raccogliere tutti i requisiti di tale gruppo nel punto di vista. È anche possibile includere punti di vista che rappresentano i requisiti dei domini e i vincoli di altri sistemi. In alternativa, è possibile utilizzare un modello di architettura del sistema per identificare i sottosistemi e associare i requisiti a ciascun sottosistema.

Inevitabilmente, stakeholder differenti hanno punti di vista differenti sull'importanza e la priorità dei requisiti, e a volte questi punti di vista sono in contrasto. Se uno stakeholder ritiene che il suo punto di vista non è stato appropriatamente considerato, potrebbe deliberatamente minare il processo di ingegneria dei requisiti. Pertanto, è importante organizzare periodiche riunioni con gli stakeholder, in modo che essi abbiano l'opportunità di esprimere le loro preoccupazioni e raggiungere un compromesso sui requisiti.

Nello stadio di documentazione dei requisiti, è importante usare frasi e diagrammi semplici per descrivere i requisiti, in modo che gli stakeholder possano capirli e commentarli. Per agevolare la condivisione delle informazioni, è consigliabile utilizzare un documento condiviso (per esempio, su Google Docs o Office 365) o un wiki che sia accessibile a tutti gli stakeholder interessati.

Punti di vista

Un punto di vista è un modo per raccogliere e organizzare una serie di requisiti da un gruppo di stakeholder che hanno qualcosa in comune. Ciascun punto di vista include una serie di requisiti del sistema. I punti di vista possono provenire da utenti, manager o altri; aiutano a identificare le persone che possono fornire informazioni sui loro requisiti e a predisporre i requisiti all'analisi.

4.3.1 Tecniche di deduzione dei requisiti

La deduzione dei requisiti richiede che ci si incontri con vari stakeholder per scoprire le informazioni sul sistema proposto. Queste informazioni possono essere integrate con la conoscenza dei sistemi esistenti, del loro uso e delle loro informazioni ricavate da documenti di vario tipo. Occorre spendere un po' di tempo per capire come lavorano le persone, che cosa producono, come usano altri sistemi e come devono cambiare per accogliere un nuovo sistema.

Ci sono due tecniche fondamentali per la deduzione dei requisiti.

1. Interviste, ovvero parlare con le persone per sapere che cosa fanno.
2. Etnografia, ovvero osservare le persone mentre svolgono il loro lavoro per vedere quali elementi utilizzano, come li utilizzano e così via.

È importante adottare un mix di queste due tecniche per raccogliere informazioni e, da queste, derivare i requisiti, che saranno la base per le successive discussioni.

Interviste

Le interviste formali o informali con gli stakeholder del sistema sono parte della gran parte dei processi di ingegneria dei requisiti. In queste interviste i team di ingegneria fanno domande agli stakeholder sul sistema che usano e sul sistema che deve essere sviluppato; dalle loro risposte ottengono i requisiti. Le interviste possono essere di due tipi:

1. interviste chiuse, dove lo stakeholder risponde a un insieme predefinito di domande;
2. interviste aperte, dove non c'è niente di predefinito. Il team di ingegneria dei requisiti esamina vari problemi degli stakeholder del sistema e quindi migliora la conoscenza delle loro necessità.

In pratica, le interviste con gli stakeholder sono normalmente un mix di entrambi i tipi. Le risposte ad alcune domande possono portare ad altri problemi che sono discussi in maniera meno strutturata. Le discussioni completamente aperte raramente funzionano bene. Molte interviste iniziano da alcune domande preliminari e poi si focalizzano sul sistema che deve essere sviluppato.

Le interviste servono per avere una comprensione generale di quello che fanno gli stakeholder, di come possono interagire con il sistema e delle difficoltà che affrontano con i sistemi già esistenti. Alle persone piace parlare del loro lavoro e solitamente sono felici di essere intervistate. Tuttavia, a meno che non abbiate un prototipo da far provare, non aspettatevi che gli stakeholder suggeriscano requisiti specifici e dettagliati. Tutti hanno difficoltà nel descrivere come un sistema dovrebbe essere, per questo dovrete analizzare le informazioni raccolte e da queste derivare i requisiti.

Dedurre le conoscenze del dominio di applicazione attraverso le interviste può essere difficile, per due ragioni.

1. Tutti gli specialisti dell'applicazione usano una terminologia e un gergo specifici del loro dominio. È impossibile per loro discutere i requisiti del dominio senza usare questa terminologia. Utilizzano termini così precisi e minuziosi che gli ingegneri dei requisiti spesso non riescono a capire.
2. Alcune conoscenze del dominio sono così familiari agli stakeholder che trovano difficile spiegarle, o pensano che siano così importanti che è inutile citarle. Per esempio, per un bibliotecario è ovvio che tutti i nuovi libri debbano essere catalogati prima di aggiungerli alla biblioteca, ma potrebbe non esserlo per l'intervistatore tanto da non includerlo tra i requisiti.

Le interviste non sono una tecnica efficace per dedurre conoscenze sui requisiti e sui vincoli organizzativi, perché ci sono delle sottili relazioni di potere tra i vari stakeholder dell'organizzazione. Le strutture organizzative ufficiali raramente rispecchiano la realtà di chi veramente prende le decisioni, ma gli intervistati potrebbero non voler rivelare le vere strutture a un estraneo. In generale, molte persone sono riluttanti a discutere di problemi politici e organizzativi che possono influenzare i requisiti.

Gli intervistatori efficaci hanno due caratteristiche.

1. Sono di larghe vedute, evitano le idee preconcette sui requisiti e sono disposti ad ascoltare gli stakeholder. Se lo stakeholder salta fuori con un requisito sorprendente, sono pronti a cambiare idea sul sistema.
2. Inducono l'intervistato a iniziare la discussione con una domanda, una proposta di requisiti o suggeriscono di lavorare insieme a un prototipo del sistema. Dicendo a una persona "dimmi quello che vuoi", difficilmente si otterranno informazioni utili. Molti trovano più semplice discutere in un contesto definito piuttosto che in termini generali.

Le informazioni ottenute dalle interviste si aggiungono ad altre informazioni acquisite dai documenti che descrivono i processi o i sistemi esistenti, dalle osservazioni degli utenti e dall'esperienza degli sviluppatori. A volte, oltre alle informazioni ottenute dai documenti del sistema, le interviste possono essere l'unica sorgente di informazioni sui requisiti del sistema. Tuttavia, la semplice intervista può portare alla mancanza di informazioni essenziali, quindi questa tecnica di deduzione dovrebbe essere sempre utilizzata insieme ad altre.

Etnografia

I sistemi software non sono mai isolati; vengono utilizzati in un contesto sociale e organizzativo, e i requisiti del sistema possono essere derivati o vincolati da questo contesto. Una ragione per la quale molti sistemi software sono consegnati, ma mai utilizzati, è che non si valuta nel modo giusto l'importanza di come i fattori sociali e organizzativi influenzano il funzionamento pratico di un sistema. È quindi molto importante, durante il processo di ingegneria del software, cercare di capire i problemi sociali e organizzativi che influiscono sull'uso del sistema.

L’etnografia è una tecnica di osservazione che può essere utilizzata per capire i processi operativi e per derivare i requisiti del software a supporto di tali processi. Un analista si immerge nell’ambiente di lavoro in cui il sistema sarà usato; osserva il lavoro quotidiano, prendendo nota dei compiti in cui i partecipanti sono coinvolti. Il valore dell’etnografia è l’aiuto che fornisce all’analista nella scoperta dei requisiti impliciti del sistema che riflettono i processi reali in cui le persone sono coinvolte, anziché i processi formali definiti dall’organizzazione.

Le persone spesso trovano difficile descrivere i dettagli del proprio lavoro perché per loro è una seconda natura; capiscono il loro lavoro, ma non la relazione con il resto del lavoro dell’organizzazione. I fattori sociali organizzativi non sono ovvi ai singoli, ma possono diventare chiari solo quando sono notati da un osservatore neutrale. Per esempio, un gruppo di lavoro può avere un’organizzazione autonoma, nel senso che ciascun membro conosca il lavoro degli altri e possa svolgerlo se qualche membro è assente.

Suchman (Suchman 1983) fu tra i primi a usare l’etnografia per studiare il lavoro d’ufficio; scoprì che le attività lavorative reali erano molto più ricche, complesse e dinamiche dei semplici modelli ipotizzati dai sistemi di automazione dell’ufficio. La differenza tra il lavoro presunto e quello reale fu la ragione più importante per cui questi sistemi non ebbero alcun effetto significativo sulla produttività. Crabtree (Crabtree 2003) esamina vari studi etnografici e descrive in generale l’uso dell’etnografia nella progettazione dei sistemi. Nelle mie personali ricerche, ho studiato metodi per integrare l’etnografica nel processo di ingegneria del software, collegandola ai metodi di ingegneria dei requisiti (Viller e Sommerville 2000) e documentando schemi di interazione nei sistemi cooperativi (Martin e Sommerville 2004).

L’etnografia è particolarmente efficace per scoprire due tipi di requisiti.

1. Requisiti che derivano dal modo in cui le persone lavorano realmente, anziché dal modo in cui le definizioni di processo dicono che dovrebbero lavorare. In pratica, le persone non seguono mai i processi formali. Per esempio, i controllori di volo possono spegnere un sistema d’allarme di collisione aerea che rileva i velivoli che percorrono rotte di volo che si incrociano, anche se le normali procedure di controllo specificano che il sistema deve essere utilizzato. Il sistema d’allarme è sensibile ed emette segnali udibili anche quando gli aerei sono molto distanti. I controllori potrebbero essere distratti da questi segnali e, quindi, potrebbero preferire altre strategie per evitare che gli aerei si trovino su rotte di volo a rischio di collisione.
2. I requisiti che derivano dalla cooperazione e dalla consapevolezza delle attività di altre persone. Per esempio, i controllori di volo (ATC) possono sfruttare la conoscenza del lavoro di altri controllori per prevedere il numero di velivoli che entreranno nel loro settore di controllo, e quindi modificare le proprie strategie di controllo a seconda del carico di lavoro previsto. Pertanto, un sistema ATC automatizzato dovrebbe permettere ai controllori di un settore di avere una certa visibilità del lavoro dei settori adiacenti.

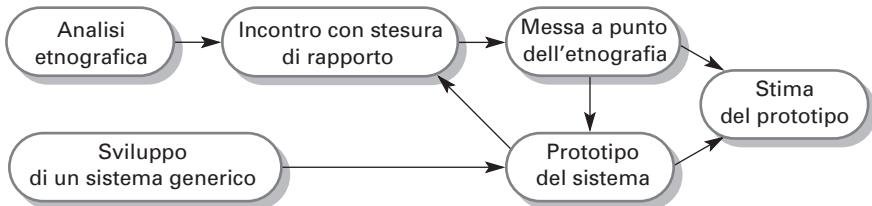


Figura 4.8 Etnografia e prototipazione per l'analisi dei requisiti.

L'etnografia può essere combinata con la prototipazione (Figura 4.8): la prima fornisce informazioni per lo sviluppo del prototipo, in modo da ridurne i cicli di perfezionamento. La prototipizzazione concentra l'etnografia sull'identificazione di problemi e domande da discutere poi con l'etnografo, che dovrebbe quindi cercare le relative risposte nella fase successiva dello studio (Sommerville et al. 1993).

Gli studi etnografici possono rivelare dettagli critici per il processo che sono spesso dimenticati da altre tecniche di deduzione dei requisiti. Tuttavia, poiché questo approccio si focalizza sull'utente finale, non è adatto a scoprire requisiti organizzativi o di dominio più ampi. Pertanto, l'etnografia dovrebbe essere utilizzata come una di tante tecniche di deduzione dei requisiti.

4.3.2 Storie e scenari

Le persone in genere trovano più semplice riferirsi a esempi di vita reale, piuttosto che a descrizioni astratte. Non sono brave nel descrivere i requisiti di un sistema, ma sono in grado di spiegare come gestiscono particolari situazioni o immaginare le cose che potrebbero fare in un nuovo contesto di lavoro. Storie e scenari sono modi di catturare questo tipo di informazioni; potete quindi utilizzarli quando intervistate gruppi di stakeholder per discutere il sistema con altri stakeholder e sviluppare requisiti di sistema più specifici.

Storie e scenari sono essenzialmente la stessa cosa. Sono descrizioni di come il sistema può essere utilizzato per svolgere particolari compiti. Descrivono che cosa fanno le persone, quali informazioni utilizzano e producono, e quali sistemi possono utilizzare in questo processo. La differenza è nel modo in cui le descrizioni sono strutturate e nel livello dei dettagli in cui sono presentate. Le storie sono scritte come testi narrativi e presentano una descrizione di alto livello del modo in cui il sistema è utilizzato; gli scenari di solito sono strutturati con informazioni specifiche raccolte come input e output. Le storie sono più efficaci per preparare una “visione d’insieme”. Parti delle storie possono poi essere sviluppate più dettagliatamente e rappresentate come scenari.

La Figura 4.9 è un esempio di storia che ho sviluppato per capire i requisiti per l’ambiente di apprendimento digitale iLearn, che ho presentato nel Capitolo 1. Questa storia descrive una situazione in una scuola elementare dove il maestro sta

Condivisione di fotografie nella classe

Jack è un maestro elementare a Ullapool (un villaggio nel nord della Scozia). Ha deciso che un progetto per la classe potrebbe riguardare l'industria della pesca in quell'area, descrivendone la storia, lo sviluppo e l'impatto economico. Come parte di questo progetto, agli alunni viene chiesto di acquisire ricordi dai parenti, usare archivi di giornali e raccogliere vecchie fotografie relative alla pesca e alle comunità di pescatori in quell'area. Gli alunni usano un wiki di iLearn per raccogliere le storie sulla pesca e SCAN (un sito di risorse storiche) per accedere agli archivi di giornali e fotografie. Jack ha anche bisogno di un sito di condivisione per le foto, perché vuole che ogni alunno possa guardare e commentare le foto degli altri alunni e possa caricare le scansioni di vecchie foto che potrebbe trovare a casa sua.

Jack invia una e-mail a un gruppo di maestri, di cui è membro, per vedere se qualcuno possa consigliare un sistema appropriato. Due maestri rispondono, ed entrambi suggeriscono che usano KidsTakePics, un sito di condivisione di foto che permette ai maestri di controllarne il contenuto. Poiché KidsTakePics non è integrato con il servizio di autenticazione di iLearn, Jack prepara un account per un maestro e la classe. Usa il servizio di configurazione di iLearn per aggiungere KidsTakePics ai servizi visti dagli alunni nella sua classe, in modo che quando si collegano (login), possono immediatamente utilizzare il sistema per caricare foto dalle loro unità mobili o dai computer della classe.

Figura 4.9 Una storia utente per il sistema iLearn.

utilizzando l'ambiente per supportare i progetti degli studenti sull'industria della pesca. Come potete notare, si tratta di una descrizione di alto livello. Il suo scopo è facilitare la discussione su come il sistema iLearn può essere utilizzato, e agire anche come punto di partenza per dedurre i requisiti per tale sistema.

Il vantaggio delle storie è che chiunque può facilmente mettersi in relazione con esse. Questo approccio è particolarmente utile per ottenere informazioni da un pubblico più vasto di quello che potremmo realisticamente intervistare. Dopo aver messo a disposizione le storie in un wiki, abbiamo invitato maestri e alunni di tutta la nazione a commentarle.

Queste storie di alto livello non entrano nei dettagli di un sistema, ma possono essere sviluppate in scenari più specifici. Gli scenari sono descrizioni di esempi di sessioni di interazione tra utenti. Credo che sia meglio presentare gli scenari in modo strutturato, anziché come testo narrativo. Le storie utente utilizzate nei metodi agili, come la programmazione estrema, in effetti sono scenari narrativi, anziché storie generiche, per facilitare la deduzione dei requisiti.

Uno scenario inizia con la descrizione dell'interazione. Durante il processo di deduzione, si aggiungono dettagli per creare una descrizione più completa dell'interazione. Nella sua forma più generale, uno scenario include:

1. una descrizione di ciò che il sistema e gli utenti si aspettano quando inizia lo scenario;
2. una descrizione del flusso normale di eventi nello scenario;

Caricare le fotografie su KidsTakePics

Ipotesi iniziale: un utente o un gruppo di utenti hanno una o più foto digitali da caricare nel sito di condivisione delle foto. Queste foto sono salvate su un tablet o su computer. Gli utenti si sono collegati con successo al sistema KidsTakePics.

Normale: l'utente sceglie di caricare le foto; il sistema gli chiede di selezionare le foto da caricare e il nome del progetto sotto il quale le foto dovranno essere memorizzate. L'utente dovrà avere anche la possibilità di inserire le parole chiave da associare a ogni foto caricata. A ogni foto caricata viene assegnato il nome formato dal nome dell'utente seguito dal nome del file della foto nel computer.

Dopo aver caricato le foto, il sistema invia automaticamente una e-mail al moderatore del progetto, per chiedergli di controllare il nuovo contenuto; poi visualizza un messaggio sullo schermo dell'utente per avvisarlo che il controllo è stato fatto.

Che cosa può andare male: nessun moderatore è associato al progetto selezionato. Il sistema genera automaticamente una e-mail per il preside della scuola, per chiedergli di nominare un moderatore di quel progetto. Gli utenti devono essere informati su un possibile ritardo nella visualizzazione delle loro foto.

Le foto con lo stesso nome sono state già caricate dallo stesso utente. Il sistema dovrà chiedere all'utente se desidera ricaricare le foto con lo stesso nome, rinominare le foto o annullare l'operazione. Se l'utente sceglie di ricaricare le foto, quelle esistenti vengono sovrascritte; se sceglie di rinominare le foto, viene generato automaticamente un nuovo nome ad ogni foto aggiungendo un numero al nome del file esistente.

Altre attività: il moderatore può essere collegato al sistema e può approvare le foto quando vengono caricate.

Stato del sistema al completamento: un utente è collegato. Le foto caricate ricevono lo stato “in attesa del moderatore”. Le foto sono visibili al moderatore e all'utente che le ha caricate.

Figura 4.10 Scenario per caricare le foto su KidsTakePics.

3. una descrizione di cosa può causare errori e come possono essere gestiti i problemi risultanti;
4. informazioni su altre attività che potrebbero svolgersi contemporaneamente;
5. una descrizione dello stato del sistema al termine dello scenario.

Come esempio di scenario, la Figura 4.10 descrive che cosa accade quando un alunno carica delle foto sul sistema KidsTakePics, descritto nella Figura 4.9. La differenza chiave tra questo e altri sistemi è che un maestro può controllare le foto caricate per verificare che siano appropriate alla condivisione.

Come potete notare, questa è una descrizione molto più dettagliata della storia riportata nella Figura 4.9; quindi, può essere utilizzata per proporre i requisiti per il sistema iLearn. Come le storie, anche gli scenari possono essere utilizzati per agevolare la discussione con gli stakeholder che potrebbero avere modi differenti di raggiungere lo stesso risultato.

4.4 Specifica dei requisiti

La specifica dei requisiti è il processo di scrivere i requisiti dell’utente e del sistema in un documento. Teoricamente, i requisiti dell’utente e del sistema dovrebbero essere chiari, senza ambiguità, facili da capire, completi e coerenti. In pratica, questo è quasi impossibile. Gli stakeholder interpretano i requisiti in modi differenti, e spesso i requisiti includono incoerenze e conflitti intrinseci.

I requisiti dell’utente sono quasi sempre scritti nel linguaggio naturale. Il documento dei requisiti include tabelle e diagrammi appropriati. Anche i requisiti del sistema possono essere scritti nel linguaggio naturale, ma possono essere utilizzate altre notazioni basate su moduli, grafici o modelli matematici. La Figura 4.11 riassume le possibili notazioni per scrivere i requisiti del sistema.

I requisiti dell’utente dovrebbero descrivere i requisiti funzionali e non funzionali del sistema, in modo che siano comprensibili dagli utenti del sistema che non hanno conoscenze tecniche speciali. Teoricamente, questi requisiti dovrebbero specificare soltanto il comportamento esterno del sistema. Il documento dei requisiti non dovrebbe includere dettagli sull’architettura o sulla progettazione del sistema. Di conseguenza, se state scrivendo i requisiti dell’utente, non dovreste usare un gergo tecnico, notazioni strutturate o formali. Dovreste scrivere i requisiti dell’utente nel linguaggio naturale, con semplici tabelle, moduli e diagrammi intuitivi.

Notazione	Descrizione
Frasi nel linguaggio naturale	I requisiti sono scritti utilizzando frasi numerate nel linguaggio naturale. Ogni frase deve esprimere un requisito.
Linguaggio naturale strutturato	I requisiti sono scritti nel linguaggio naturale secondo una forma o modello standard. Ogni campo in un modello fornisce le informazioni su un aspetto del requisito.
Notazioni grafiche	Modelli grafici, corredati da annotazioni di testo, sono usati per definire i requisiti funzionali del sistema. Sono comunemente utilizzati i casi d’uso e i diagrammi di sequenza del linguaggio UML (Unified Modeling Language).
Specifiche matematiche	Sono notazioni basate su concetti matematici, come gli insiemi o le macchine a stati finiti. Sebbene queste specifiche non ambigue possano ridurre l’ambiguità del documento dei requisiti, molti clienti non capiscono una specifica formale; non possono verificare che il documento dei requisiti rappresenta ciò che essi vogliono e, quindi, sono riluttanti ad accettarlo come parte del contratto. (Descriverò questo approccio nel Capitolo 10, che è dedicato alla fidatezza del sistema.)

Figura 4.11 Notazioni per scrivere i requisiti del sistema.

I requisiti del sistema sono versioni espanso dei requisiti dell’utente; sono utilizzati dagli ingegneri del software come base di partenza per la progettazione del sistema; aggiungono dettagli e spiegano come il sistema dovrebbe realizzare i requisiti dell’utente. Possono essere usati come parte del contratto per l’implementazione del sistema e quindi dovrebbero essere una specifica completa e dettagliata dell’intero sistema.

In teoria, i requisiti del sistema dovrebbero descrivere soltanto il comportamento esterno del sistema e i suoi vincoli operativi. Non dovrebbero riguardare la progettazione o l’implementazione del sistema. Tuttavia, al livello di dettaglio richiesto per specificare in modo completo un sistema software complesso, non è possibile né auspicabile escludere tutte le informazioni sulla progettazione per diverse ragioni.

1. Potrebbe essere necessario progettare un’architettura iniziale del sistema per agevolare la definizione della struttura delle specifiche dei requisiti. I requisiti del sistema sono organizzati a seconda dei diversi sottosistemi che lo compongono. Abbiamo fatto questo quando abbiamo definito i requisiti per il sistema iLearn, dove abbiamo proposto l’architettura illustrata nella Figura 1.8.
2. In molti casi, i sistemi devono interagire con altri sistemi esistenti. Questo vincola la progettazione e impone requisiti al nuovo sistema.
3. Potrebbe essere necessario l’uso di una specifica architettura per soddisfare requisiti non funzionali, come la programmazione a N-versioni per raggiungere l’affidabilità (descritta nel Capitolo 11). Un ente certificatore esterno, che deve attestare la sicurezza del sistema, potrebbe richiedere l’uso di una determinata architettura che è già stata certificata.

4.4.1 Specifica nel linguaggio naturale

Il linguaggio naturale è stato utilizzato per scrivere requisiti del software fino agli anni ’50. È espressivo, intuitivo e universale. È anche potenzialmente vago e ambiguo, e la sua interpretazione dipende dal background culturale del lettore. Di conseguenza, ci sono state varie proposte per scrivere in altre maniere i requisiti. Tuttavia, nessuna di queste proposte è stata largamente accettata, e il linguaggio naturale continuerà ad essere il modo più utilizzato di specificare i requisiti dei sistemi e del software.

Per ridurre al minimo le incomprensioni quando si scrivono i requisiti nel linguaggio naturale, vi consiglio di seguire queste semplici linee guida:

1. Inventate un formato standard e assicuratevi che tutte le definizioni dei requisiti siano conformi a questo formato. L’uso di un formato standard riduce il rischio di omissioni e semplifica il controllo dei requisiti. Se possibile, dovreste scrivere i requisiti con una o due frasi nel linguaggio naturale.

2. Utilizzare con coerenza il linguaggio per distinguere un requisito obbligatorio da un requisito desiderabile. I requisiti obbligatori sono quelli che il sistema deve supportare e, di solito, si scrivono utilizzando l'indicativo “deve”. I requisiti desiderabili non sono essenziali e, quindi, si scrivono con il condizionale “dovrebbe”.
3. Utilizzate la formattazione del testo (grassetto, corsivo o i colori) per mettere in evidenza parti di un requisito.
4. Non date per scontato che i lettori capiscano il linguaggio tecnico dell'ingegneria del software. È facile che parole come “architettura” e “modulo” siano fraintese. Se possibile, dovreste evitare termini del gergo, abbreviazioni e acronimi.
5. Se possibile, dovreste tentare di associare una logica al requisito di ciascun utente. La logica dovrebbe spiegare il motivo per cui il requisito è stato incluso e indicare chi lo ha proposto (la fonte del requisito), in modo che sappiate chi consultare se il requisito dovesse essere modificato. La logica dei requisiti è particolarmente utile quando i requisiti vengono cambiati, in quanto può aiutare a decidere quali modifiche potrebbero essere indesiderabili.

La Figura 4.12 illustra come queste linee guida possono essere utilizzate. Include due requisiti per il software integrato per la pompa dell’insulina descritta nel Capitolo 1. Gli altri requisiti sono definiti nel documento dei requisiti della pompa.

4.4.2 Specifiche strutturate

Il linguaggio naturale strutturato è un modo di scrivere i requisiti del sistema secondo una forma standard, anziché come testo libero. Questo approccio mantiene gran parte dell'espressività e della comprensibilità del linguaggio naturale, ma garantisce una certa uniformità delle specifiche. Le notazioni del linguaggio strutturato usano schemi per specificare i requisiti dei sistemi. La specifica può usare costrutti del linguaggio di programmazione per mostrare alternative e itera-

3.2 Il sistema deve misurare il livello degli zuccheri nel sangue e rilasciare l’insulina, ogni 10 minuti (variazioni degli zuccheri nel sangue sono relativamente lente, quindi non sono necessarie misure più frequenti; misure meno frequenti potrebbero portare a livelli di zuccheri inutilmente elevati).

3.6 Il sistema deve eseguire una routine di auto-test ogni minuto con le condizioni da provare e le azioni associate definite nella Tabella 1 (una routine di auto-test può scoprire problemi nel software e nell’hardware e avvisare l’utente che la pompa potrebbe non funzionare correttamente).

Figura 4.12 Esempio di requisiti per il sistema software della pompa per l’insulina.

Problemi del linguaggio naturale

La flessibilità del linguaggio naturale, che è così utile per specificare i requisiti, spesso è causa di problemi. C'è il rischio di scrivere requisiti poco chiari, e i lettori (i progettisti) possono fraintendere i requisiti perché hanno un background culturale differente dagli utenti. È facile includere più requisiti in una singola frase, e questo complica il processo di strutturazione dei requisiti.

<http://software-engineering-book.com/web/natural-language/>

zioni, e può mettere in evidenza elementi chiave utilizzando colori o caratteri differenti.

I Robertson (Robertson e Robertson 2013), nel loro libro sul metodo di ingegneria dei requisiti VOLERE, raccomandano di scrivere inizialmente i requisiti dell'utente su schede, un requisito per scheda. Consigliano un certo numero di campi per ogni scheda, come la logica dei requisiti, le dipendenze da altri requisiti, la fonte dei requisiti e i materiali di supporto. Questo metodo è simile all'approccio utilizzato nell'esempio della specifica strutturata illustrato nella Figura 4.13.

Per utilizzare un approccio strutturato alla specifica dei requisiti del sistema, occorre definire uno o più schemi standard per i requisiti e rappresentare questi schemi come moduli strutturati. La specifica può essere strutturata attorno agli oggetti manipolati dal sistema, alle funzioni svolte dal sistema o agli eventi elaborati dal sistema. Nella Figura 4.13 è illustrato un esempio di specifica basata su schemi (in questo caso, la specifica definisce come calcolare la dose di insulina da rilasciare quando gli zuccheri nel sangue sono all'interno della fascia di sicurezza).

Se si utilizza un modulo standard per specificare i requisiti funzionali, dovrebbero essere incluse le seguenti informazioni:

1. una descrizione della funzione o dell'entità che si sta specificando;
2. una descrizione dei suoi input e delle origini di questi input;
3. una descrizione dei suoi output e delle destinazioni di questi output;
4. informazioni sui dati necessari per i calcoli o su altre entità che sono richieste dal sistema (la parte “Richiede”);
5. la descrizione dell'azione da eseguire;
6. se si utilizza un metodo funzionale, una precondizione che spieghi cosa deve verificarsi prima che la funzione sia chiamata, e una postcondizione che specifichi cosa avviene sicuramente dopo che la funzione è stata chiamata;
7. una descrizione degli effetti collaterali dell'operazione (se ce ne sono).

Pompa di insulina/Software di controllo/SRS/ 3.3.2	
Funzione	Calcolo della dose di insulina: livello sicuro degli zuccheri.
Descrizione	Calcola la dose di insulina che deve essere fornita quando il livello corrente degli zuccheri è nella zona sicura tra 3 e 7 unità.
Input	La misura corrente degli zuccheri (r_2) e le due misure precedenti (r_0 e r_1).
Sorgente	La misura corrente degli zuccheri fatta dal sensore. Altre misure memorizzate.
Output	CompDose: la dose di insulina che deve essere fornita.
Destinazione	Ciclo di controllo principale.
Azione	CompDose è zero se il livello degli zuccheri è stabile o sta scendendo, o se il livello sta crescendo ma il tasso di crescita è in diminuzione. Se il livello sta crescendo e anche il tasso di crescita sta aumentando, allora CompDose è calcolato dividendo per 4 la differenza tra il livello corrente degli zuccheri e il precedente livello, e arrotondando il risultato. Se il risultato è arrotondato a zero, allora CompDose viene impostato alla minima dose che può essere fornita. (Si veda la Figura 4.14.)
Richiede	Due precedenti misure in modo da poter calcolare il tasso di variazione del livello degli zuccheri.
Precondizione	La riserva di insulina deve contenere almeno la quantità massima permessa per una singola dose.
Postcondizione	r_0 viene sostituito da r_1 , poi r_1 viene sostituito da r_2 .
Effetti collaterali	Nessuno.

Figura 4.13 La specifica strutturata di un requisito per la pompa di insulina.

L'uso delle specifiche strutturate risolve alcuni problemi delle specifiche scritte nel linguaggio naturale. La variabilità è ridotta e i requisiti sono organizzati in modo più efficace. Tuttavia, è difficile scrivere i requisiti in modo non ambiguo, specialmente quando bisogna specificare calcoli complessi (per esempio, come calcolare la dose di insulina).

Per risolvere questo problema è possibile aggiungere informazioni supplementari al linguaggio naturale, per esempio tabelle o modelli grafici del sistema, che possono spiegare come devono essere svolti i calcoli, come cambia lo stato del sistema, come gli utenti interagiscono con il sistema e come vengono effettuate le sequenze di operazioni.

Le tabelle sono particolarmente utili quando c'è una serie di possibili situazioni alternative ed è necessario descrivere le azioni da intraprendere per ciascuna di esse. La pompa calcola la richiesta d'insulina in base al tasso di variazione dei

Condizione	Azione
Livello degli zuccheri in discesa ($r2 < r1$)	CompDose = 0
Livello degli zuccheri stabile ($r2 = r1$)	CompDose = 0
Livello degli zuccheri in salita e tasso di crescita in diminuzione $((r2 - r1) < (r1 - r0))$	CompDose = 0
Livello degli zuccheri in salita e tasso di crescita stabile o in aumento $r2 > r1 \& ((r2 - r1) \geq (r1 - r0))$	CompDose = arrotondamento $((r2 - r1) / 4)$ Se il risultato arrotondato = 0, allora CompDose = MinimumDose

Figura 4.14 Specifica tabellare del calcolo della dose di insulina.

livelli di zuccheri nel sangue. I tassi di variazione sono calcolati usando la misura corrente e quella precedente. La Figura 4.14 è una descrizione in forma tabellare di come viene utilizzato il tasso di variazione del livello di zuccheri nel sangue per calcolare la quantità di insulina da rilasciare.

4.4.3 Casi d'uso

I casi d'uso sono un modo di descrivere le interazioni tra utenti e sistema utilizzando un modello grafico e un testo strutturato. Furono introdotti per la prima volta nel metodo Objectory (Jacobsen et al. 1993) e ora sono diventati elementi fondamentali del linguaggio UML (*Unified Modeling Language*). Nella loro forma più semplice, un caso d'uso identifica gli attori coinvolti in un'interazione e assegna un nome al tipo di interazione. Poi si aggiungono altre informazioni che descrivono l'interazione con il sistema; queste informazioni possono essere descrizioni testuali o modelli grafici, come i diagrammi di sequenza o di stato UML (Capitolo 5).

I casi d'uso sono documentati utilizzando un diagramma di casi d'uso di alto livello. L'insieme dei casi d'uso rappresenta tutte le possibili interazioni che saranno descritte nei requisiti del sistema. Gli attori nel processo, che possono essere uomini o altri sistemi, sono rappresentati da figure stilizzate. Ogni classe di interazione è rappresentata da un'ellisse con un nome. Le linee collegano gli attori con l'interazione. Facoltativamente, possono essere aggiunte le frecce alle linee per indicare come inizia l'interazione. Questo è illustrato nella Figura 4.15, che mostra alcuni dei casi d'uso del sistema Mentcare.

I casi d'uso identificano le singole interazioni tra il sistema e i suoi utenti o altri sistemi. Ogni caso d'uso dovrebbe essere documentato con una descrizione testuale. I casi d'uso possono essere collegati ad altri modelli UML che svilupperanno lo scenario con maggiori dettagli. Per esempio, una breve descrizione del caso d'uso “Prepara il consulto” della Figura 4.15 potrebbe essere la seguente:

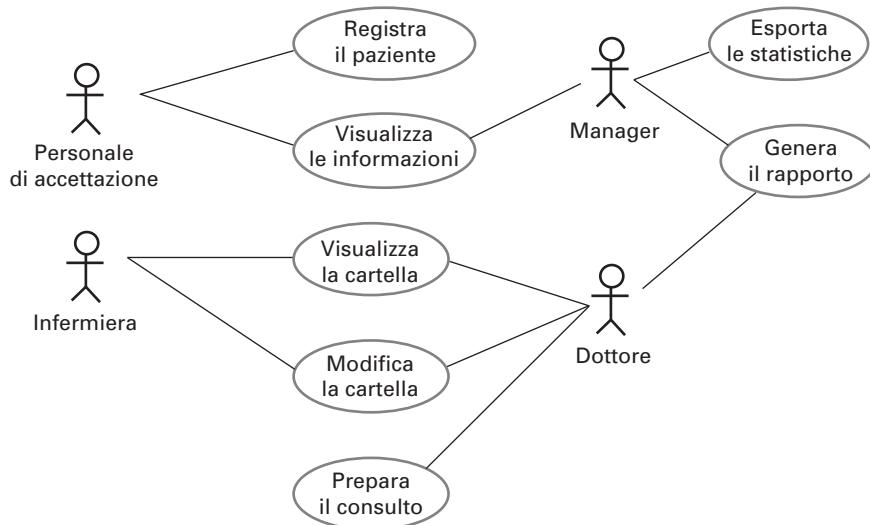


Figura 4.15 Casi d'uso per il sistema Mentcare.

La preparazione del consulto consente a due o più dottori, che lavorano in uffici differenti, di vedere contemporaneamente la cartella di uno stesso paziente. Un dottore inizia il consulto scegliendo le persone coinvolte da un menu di dottori che sono online. La cartella del paziente viene quindi visualizzata sui loro schermi, ma soltanto il dottore che ha avviato il consulto può modificare la cartella. In aggiunta, viene creata una finestra di chat.

UML è uno standard per la modellazione orientata agli oggetti, perciò i casi d'uso e la deduzione dei requisiti basata sui casi d'uso sono utilizzati sempre di più nel processo di ingegneria dei requisiti. Tuttavia, la mia esperienza con i casi d'uso è che essi sono troppo dettagliati per essere utili nella discussione dei requisiti. Gli stakeholder non capiscono il termine *caso d'uso*; non trovano utile il modello grafico e spesso non sono interessati alle descrizioni dettagliate di ciascuna interazione del sistema. Di conseguenza, credo che i casi d'uso siano più utili nella progettazione dei sistemi, anziché nell'ingegneria dei requisiti. Tratterò i casi d'uso ancora nel Capitolo 5 per spiegare come siano utilizzati insieme ad altri modelli per documentare il progetto di un sistema.

Alcune persone credono che ogni caso d'uso sia un singolo scenario di interazioni di basso livello. Altri, come Stevens e Pooley (Stevens e Pooley 2006), ritengono che ogni caso d'uso includa una serie di scenari correlati di basso livello. Ciascuno di questi scenari è un filo della trama del caso d'uso. Pertanto, ci sarà uno scenario per l'interazione normale, più altri scenari per ogni possibile eccezione. In pratica, potete utilizzarli in entrambi i modi.

4.4.4 Documento dei requisiti

Il documento dei requisiti del software (a volte chiamato “specifiche dei requisiti del software” o SRS, *Software Requirements Specification*) è una definizione ufficiale di quello che gli sviluppatori del sistema dovrebbero implementare. Può includere sia i requisiti dell’utente sia una specifica dettagliata dei requisiti del sistema. In alcuni casi, i requisiti dell’utente e del sistema possono essere integrati in un’unica descrizione; in altri casi, i requisiti dell’utente sono definiti in un’introduzione alle specifiche dei requisiti del sistema.

I documenti dei requisiti sono essenziali quando lo sviluppo dei sistemi viene appaltato all’esterno, quando differenti team sviluppano parti diverse del sistema e quando è obbligatoria un’analisi dettagliata dei requisiti. In altre situazioni, per esempio nello sviluppo di un sistema aziendale o di un prodotto software, potrebbe non essere necessario un documento dettagliato dei requisiti.

I sostenitori dei metodi di sviluppo agile ritengono che il documento dei requisiti è obsoleto nel momento stesso in cui viene scritto, in quanto i requisiti cambiano velocemente; quindi tale documento è uno spreco di energie. Anziché un documento formale, gli approcci agili spesso raccolgono i requisiti in modo incrementale e li scrivono su schede o lavagne bianche come storie utente. Gli utenti quindi danno priorità a queste storie da implementare nella successiva versione del sistema.

Questo metodo può essere appropriato per i sistemi aziendali, dove i requisiti sono instabili; tuttavia, è sempre utile scrivere un piccolo documento di supporto che definisce i requisiti aziendali e di fidatezza per il sistema; è facile dimenticare i requisiti che si riferiscono al sistema completo quando ci si concentra sui requisiti funzionali per la successiva release del sistema.

Il documento dei requisiti ha un insieme di utenti molto vario che va dai dirigenti più alti dell’organizzazione che acquista il sistema agli ingegneri responsabili dello sviluppo del software. La Figura 4.16 illustra i possibili utenti di un documento dei requisiti e come viene utilizzato.

La varietà dei possibili utenti implica che il documento dei requisiti deve essere un compromesso; deve descrivere i requisiti per i clienti, definire dettagliatamente i requisiti per gli sviluppatori e i tester, e includere le informazioni che riguardano la futura evoluzione del sistema. Le informazioni sulle modifiche previste aiutano i progettisti del sistema a evitare decisioni restrittive e gli ingegneri della manutenzione del sistema ad adattarlo ai nuovi requisiti.

Il livello di dettaglio del documento dei requisiti dipende dal tipo di sistema da sviluppare e dal processo di sviluppo utilizzato. Per i sistemi critici servono requisiti dettagliati, in quanto la sicurezza e la protezione devono essere analizzate con cura per scovare eventuali errori nei requisiti. Quando il sistema viene sviluppato da una società esterna, le specifiche del sistema devono essere precise e dettagliate. Se si usa uno sviluppo iterativo all’interno della società, il documen-

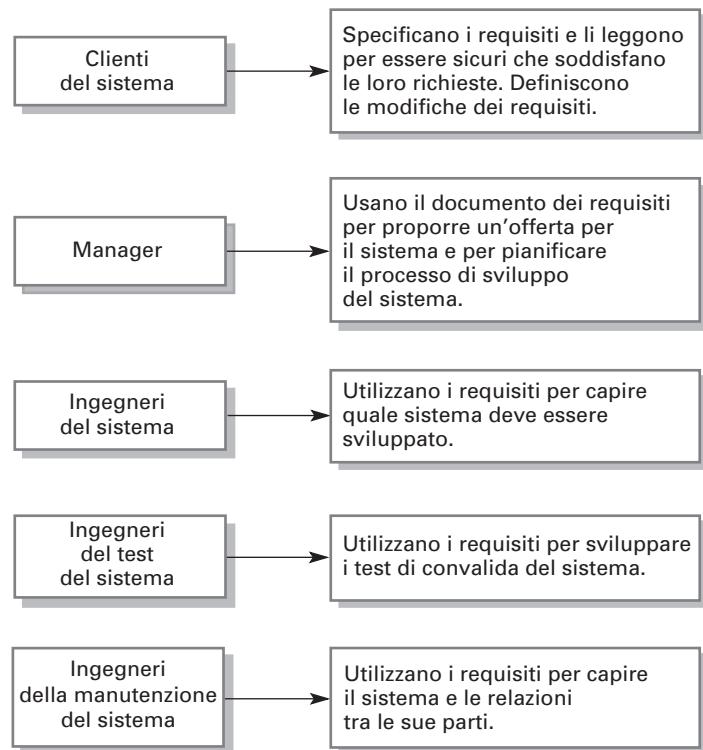


Figura 4.16 Utenti di un documento dei requisiti.

to dei requisiti può essere meno dettagliato. Per eliminare ogni ambiguità durante lo sviluppo del sistema, è possibile aggiungere altri dettagli ai requisiti.

La Figura 4.17 mostra come si può strutturare un documento dei requisiti facendo riferimento allo standard IEEE per i documenti dei requisiti. Questo standard è generico e può essere adattato a specifiche situazioni. In questo caso, lo standard è stato esteso per includere informazioni sulla prevista evoluzione del sistema. Queste informazioni aiutano i manutentori del sistema e permette ai progettisti di includere il supporto per future funzionalità del sistema.

Ovviamente, le informazioni incluse in un documento dei requisiti dipendono dal tipo di software che si sta sviluppando e dalla tecnica di sviluppo utilizzata. Un documento dei requisiti con una struttura simile a quella della Figura 4.17 potrebbe essere prodotto per un sistema complesso che include hardware e software sviluppati da varie società. Il documento dei requisiti molto probabilmente sarà lungo e dettagliato. È quindi importante includere un sommario generale e un indice analitico dei contenuti del documento, in modo che i lettori possano facilmente trovare le informazioni di cui hanno bisogno.

Capitolo	Descrizione
Prefazione	Definisce i potenziali lettori del documento e descrive la storia delle versioni, includendo la logica della creazione di una nuova versione e un riassunto delle modifiche fatte in ciascuna versione.
Introduzione	Describe gli obiettivi del sistema. Dovrebbe presentare brevemente le funzioni del sistema e spiegare come opererà con altri sistemi. Dovrebbe descrivere anche come il sistema si adatta agli obiettivi strategici dell'organizzazione che ha commissionato il software.
Glossario	Definisce i termini tecnici utilizzati nel documento. Non si dovrebbe fare alcuna ipotesi circa l'esperienza e la competenza dei lettori.
Definizione dei requisiti dell'utente	Describe i servizi forniti all'utente. Dovrebbero essere inclusi anche i requisiti funzionali e non funzionali del sistema. Questa descrizione può usare il linguaggio naturale, diagrammi o altre notazioni che sono comprensibili ai clienti. Dovrebbero essere specificati gli standard del prodotto e dei processi che devono essere seguiti.
Architettura del sistema	Questo capitolo presenta una panoramica di alto livello dell'architettura prevista per il sistema, mostrando la distribuzione delle funzioni nei vari moduli del sistema. Si dovrebbero evidenziare i componenti strutturali che vengono riutilizzati.
Specifiche dei requisiti del sistema	Describe in modo dettagliato i requisiti funzionali e non funzionali. Se necessario, ulteriori dettagli possono essere aggiunti ai requisiti non funzionali. Possono essere definite le interfacce con altri sistemi.
Modelli del sistema	Questo capitolo include i modelli grafici del sistema che illustrano le relazioni tra i componenti del sistema, il sistema e il suo ambiente. Possono essere modelli di oggetti, diagrammi di flusso dei dati e modelli semantici.
Evoluzione del sistema	Describe le ipotesi fondamentali su cui si basa il sistema, le modifiche future dovute all'evoluzione dell'hardware, alle mutate necessità degli utenti e così via. Questo capitolo è utile ai progettisti del sistema, in quanto può aiutarli a evitare decisioni che potrebbero limitare futuri cambiamenti del sistema.
Appendici	Forniscono informazioni dettagliate sull'applicazione che si sta sviluppando, per esempio le descrizioni del database e dell'hardware. I requisiti dell'hardware definiscono le configurazioni minime e ottimali per il sistema. I requisiti del database definiscono l'organizzazione logica dei dati usati dal sistema e le loro relazioni.
Indice	Può includere diversi indici del documento; oltre a un indice alfabetico potrebbe esserci un indice dei diagrammi, delle funzioni e così via.

Figura 4.17 Struttura di un documento dei requisiti.

Standard per i documenti dei requisiti

Molte grandi organizzazioni, come il Dipartimento della Difesa statunitense e l'IEEE, hanno definito degli standard per i documenti dei requisiti. Si tratta di standard molto generici, ma utili come base per sviluppare standard organizzativi più dettagliati. L'IEEE (Institute of Electrical and Electronic Engineers) è uno dei più noti creatori di standard; ha sviluppato uno standard per la struttura dei documenti dei requisiti. Questo standard è particolarmente appropriato per i sistemi militari di controllo e di comando che hanno lunga durata e di solito sono sviluppati da un gruppo di organizzazioni.

<http://software-engineering-book.com/web/requirements-standard/>

Al contrario, il documento dei requisiti per un prodotto software sviluppato all'interno della società può escludere gran parte dei capitoli elencati nella Figura 4.17. Si definiranno con attenzione i requisiti dell'utente e i requisiti del sistema non funzionali di alto livello. I progettisti e i programmati utilizzeranno il loro senso critico per decidere come soddisfare i requisiti dell'utente così definiti.

4.5 Convalida dei requisiti

La convalida dei requisiti è il processo che verifica se i requisiti definiscono il sistema realmente voluto dal cliente. Si sovrappone alla deduzione e all'analisi perché cerca di scoprire i problemi nei requisiti. La convalida è importante perché gli errori nel documento dei requisiti possono portare a costi di rilavorazione molto alti se tali problemi vengono scoperti durante lo sviluppo o dopo la messa in servizio del sistema.

Per risolvere un problema nei requisiti, di solito è molto più costoso modificare il sistema che correggere gli errori di progettazione o di codifica. Una modifica dei requisiti di solito implica che anche la progettazione e l'implementazione del sistema devono cambiare; inoltre, occorre eseguire nuovi test sul sistema.

Esistono diverse tipologie di controllo dei requisiti da effettuare durante il processo di convalida.

1. *Controlli di validità:* verificano se i requisiti riflettono le reali esigenze degli utenti del sistema. A causa delle mutevoli circostanze, potrebbe essere necessario modificare i requisiti dell'utente rispetto a come furono originalmente dedotti.
2. *Controlli di consistenza:* tra i requisiti non dovrebbero esserci conflitti, ovvero non dovrebbero esserci vincoli in contraddizione o descrizioni differenti per la stessa funzione del sistema.
3. *Controlli di completezza:* il documento dei requisiti deve includere i requisiti che definiscono tutte le funzioni e tutti i vincoli richiesti dall'utente del sistema.

Revisione dei requisiti

La revisione dei requisiti è un processo in cui un gruppo di persone, da parte del cliente e del team di sviluppo, leggono il documento dei requisiti nei dettagli e controllano che il documento non presenti errori, anomalie o incoerenze. Una volta che tutti questi elementi sono stati evidenziati e registrati, il cliente e lo sviluppatore si accordano sul modo in cui i problemi identificati dovranno essere risolti.

<http://software-engineering-book.com/web/requirements-reviews/>

4. *Controlli di realismo*: sfruttando le conoscenze delle tecnologie esistenti, si dovrebbero controllare i requisiti per assicurarsi che possano essere realmente implementati, rispettando il budget proposto per il sistema. Questi controlli dovrebbero tenere conto anche della tempistica per lo sviluppo del sistema.
5. *Verificabilità*: per ridurre le potenziali discussioni tra cliente e appaltatore, i requisiti di sistema dovrebbero essere scritti sempre in modo da essere verificabili. Questo significa che occorre scrivere un insieme di test per dimostrare che il sistema consegnato soddisfa ogni requisito specificato.

È possibile utilizzare diverse tecniche di validazione, singolarmente o combinandole insieme.

1. *Revisione dei requisiti*: i requisiti vengono analizzati sistematicamente da un team di revisori, che controllano errori e incoerenze.
2. *Prototipazione*: gli utenti finali e i clienti provano un modello eseguibile del sistema per vedere se esso soddisfa le loro necessità e aspettative. Gli stakeholder sperimentano il sistema e richiedono al team di sviluppo eventuali modifiche dei requisiti.
3. *Generazione di test case*: i requisiti dovrebbero essere testabili. Se i test sono concepiti come parte del processo di convalida, spesso svelano i problemi presenti nei requisiti. Se un test è difficile o impossibile da progettare, significa che i requisiti saranno difficili da implementare e dovranno essere riconsiderati. Sviluppare i test dai requisiti dell'utente, prima che sia scritto il codice, è parte integrante dello sviluppo guidato dai test.

Non bisognerebbe sottovalutare i problemi connessi alla convalida dei requisiti. È difficile dimostrare che un insieme di requisiti soddisfa le necessità dell'utente. Gli utenti devono immaginare il sistema in funzione e come questo dovrebbe adattarsi al loro lavoro. È difficile per i professionisti informatici qualificati eseguire questo tipo di analisi astratta, e lo è ancora di più per gli utenti del sistema.

Per questo motivo è raro scoprire tutti i problemi durante il processo di convalida dei requisiti. È inevitabile che saranno necessarie ulteriori modifiche per correggere omissioni e fraintendimenti dopo che il documento dei requisiti è stato accettato dalle parti.

4.6 Modifica dei requisiti

I requisiti per i grandi sistemi software variano costantemente. Una ragione delle frequenti variazioni è che questi sistemi sono spesso sviluppati per risolvere i cosiddetti *wicked problem* – problemi che non possono essere completamente definiti (Rittel e Webber 1973). Poiché un problema non può essere completamente definito, anche i requisiti del software restano indefiniti. Durante il processo di sviluppo del software, la conoscenza di un problema da parte degli stakeholder cambia in continuazione (Figura 4.18). I requisiti del sistema devono dunque evolversi per riflettere la mutata conoscenza del problema.

Una volta che un sistema è stato installato e regolarmente utilizzato, emergono inevitabilmente nuovi requisiti. Questo è dovuto in parte agli errori e alle omissioni nei requisiti originali che devono essere corretti. Tuttavia, molte modifiche dei requisiti del sistema derivano dai cambiamenti dell’ambiente aziendale in cui opera il sistema:

1. L’ambiente tecnico e aziendale del sistema cambia sempre dopo la sua installazione. Potrebbe essere introdotto un nuovo hardware e quello esistente potrebbe essere aggiornato. Potrebbe essere necessario interfacciare il sistema con altri sistemi. Potrebbero cambiare le priorità aziendali, con conseguenti cambiamenti nel supporto del sistema. Potrebbero essere introdotte nuove legislazioni e nuove norme cui il sistema dovrà conformarsi.
2. Le persone che comprano il sistema e gli utenti del sistema raramente sono le stesse. Gli acquirenti del sistema impongono ai requisiti vincoli organizzativi e di budget, che possono essere in conflitto con i requisiti degli utenti finali; quindi, potrebbe essere necessario aggiungere nuove funzioni dopo la consegna, se il sistema non soddisfa le necessità degli utenti.
3. I grandi sistemi di solito hanno diversi gruppi di stakeholder, ciascuno con requisiti differenti. Le loro priorità possono essere in conflitto o in contraddizione. I requisiti finali del sistema sono inevitabilmente un compromesso, e ad alcuni stakeholder occorre dare la precedenza. Con l’esperienza, si scopre spesso che la distribuzione del supporto dato ai vari stakeholder deve essere cambiata e le priorità dei requisiti riformulate.

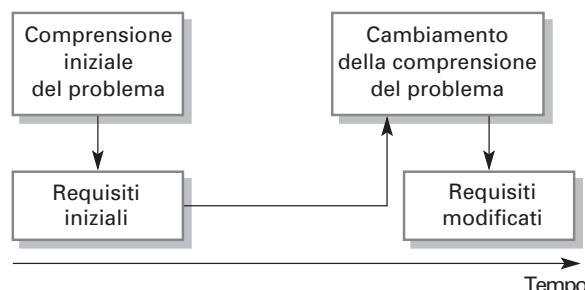


Figura 4.18 Evoluzione dei requisiti.

Requisiti duraturi e volatili

Alcuni requisiti sono più suscettibili di altri a subire modifiche. I requisiti duraturi sono quelli legati alle attività di base di un'organizzazione e sono caratterizzati da lente variazioni. I requisiti volatili cambiano con maggiore frequenza; di solito sono associati ad attività che riflettono il modo in cui l'organizzazione svolge il suo lavoro, anziché il lavoro in sé.

<http://software-engineering-book.com/web/changing-requirements/>

Durante l'evoluzione dei requisiti, occorre tenere traccia dei singoli requisiti e conservare i collegamenti tra i requisiti dipendenti, in modo che si possa stabilire l'impatto delle modifiche dei requisiti. Occorre un processo formale per proporre i cambiamenti e collegarli ai requisiti del sistema. Questo processo di “gestione dei requisiti” dovrebbe iniziare appena è disponibile una bozza del documento dei requisiti.

I processi di sviluppo agile sono stati ideati per far fronte ai requisiti che cambiano durante il processo di sviluppo. In tali processi, quando un utente propone una modifica dei requisiti, non segue un processo di gestione formale; piuttosto, l'utente deve indicare una priorità per la modifica, e se è alta, decidere quali funzionalità del sistema, che erano state pianificate per la successiva iterazione, dovranno essere tralasciate per implementare la modifica.

Il problema di questo approccio è che gli utenti non sono necessariamente le persone migliori per stabilire se le modifiche dei requisiti siano globalmente convenienti. Nei sistemi con più stakeholder, le modifiche beneficeranno alcuni stakeholder e non altri. Spesso è preferibile che ci sia un'autorità esterna che decida quali modifiche accettare, bilanciando le esigenze di tutti gli stakeholder.

4.6.1 Pianificare la gestione dei requisiti

Pianificare la gestione dei requisiti significa stabilire come dovrà essere gestita una serie di cambiamenti dei requisiti. Durante la fase di pianificazione, occorrerà decidere su vari argomenti.

1. *Identificazione dei requisiti*: ogni requisito deve essere univocamente identificato, in modo che possa avere un riferimento incrociato con altri requisiti e possa essere usato nelle definizioni di tracciabilità.
2. *Processo di gestione delle modifiche*: un insieme di attività che stabilisce l'impatto e il costo delle modifiche. Descriverò questo processo più dettagliatamente nel prossimo paragrafo.
3. *Politiche di tracciabilità*: definiscono le relazioni tra ciascun requisito, e tra i requisiti e il progetto del sistema; devono definire anche come registrare e mantenere queste relazioni.



Figura 4.19 Gestione delle modifiche dei requisiti.

4. *Supporto degli strumenti*: la gestione dei requisiti richiede l’elaborazione di una grande quantità di informazioni. Gli strumenti che possono essere usati spaziano dai sistemi specializzati nella gestione dei requisiti ai fogli di calcolo condivisi ai semplici sistemi di database.

La gestione dei requisiti richiede un supporto automatizzato e gli strumenti software che devono essere scelti durante la fase di pianificazione. Occorre un supporto per:

1. *memorizzare i requisiti*: i requisiti dovrebbero essere conservati in una memoria sicura, gestita e accessibile a tutti coloro che sono coinvolti nel processo di ingegneria dei requisiti;
2. *gestire le modifiche*: il processo di modifica dei requisiti (Figura 4.19) viene semplificato se è disponibile un supporto attivo degli strumenti. Gli strumenti possono tenere traccia delle modifiche suggerite e le risposte a tali suggerimenti;
3. *gestire la tracciabilità*: il supporto alla tracciabilità permette di scoprire i requisiti correlati. Alcuni strumenti utilizzano delle tecniche di elaborazione del linguaggio naturale per aiutare a scoprire le possibili relazioni tra i requisiti.

Per i piccoli sistemi potrebbe non essere necessario usare strumenti specialistici di gestione dei requisiti. Il processo di gestione può essere supportato utilizzando documenti web, fogli di calcolo e database condivisi. Per i sistemi più grandi, invece, un supporto specialistico, che usa strumenti come DOORS (IBM 2013), agevola la gestione di un gran numero di requisiti che cambiano.

4.6.2 Gestione delle modifiche dei requisiti

La gestione delle modifiche dei requisiti (Figura 4.19) dovrebbe essere applicata a tutti i cambiamenti proposti per i requisiti di un sistema, dopo che il documento dei requisiti è stato approvato. La gestione dei requisiti è essenziale perché occorre decidere se i benefici derivanti dall’implementazione di nuovi requisiti sono giustificati dai costi dell’implementazione. Il vantaggio di usare un processo formale per la gestione delle modifiche è che tutte le proposte di cambiamento vengono trattate in modo coerente e le modifiche del documento dei requisiti vengono effettuate in modo controllato. Ci sono tre stadi principali in un processo di gestione delle modifiche.

Tracciabilità dei requisiti

Occorre tenere traccia delle relazioni fra i requisiti, le loro fonti e il progetto del sistema, in modo che sia possibile analizzare le ragioni delle modifiche proposte e l'impatto che queste modifiche possono avere su altre parti del sistema. È importante saper tracciare come una modifica cambi direzione attraversando il sistema. Perché?

<http://software-engineering-book.com/web/traceability/>

1. *Analisi dei problemi e specifica delle modifiche:* il processo inizia dalla scoperta di un problema nei requisiti o, a volte, da una specifica proposta di cambiamento. In questa fase, viene analizzata la validità del problema o della proposta. I risultati di questa analisi sono consegnati a chi ha richiesto la modifica, che potrebbe rispondere con una proposta di cambiamento più precisa o decidere di ritirare la proposta.
2. *Analisi e stima dei costi delle modifiche:* viene valutato l'effetto della modifica proposta, usando le informazioni di tracciabilità e le conoscenze generali dei requisiti del sistema. Il costo della modifica viene stimato in base al numero di modifiche che dovrebbero essere apportate al documento dei requisiti e, se opportuno, al progetto del sistema e alla sua implementazione. Una volta completata questa analisi, si deve decidere se procedere o meno con la modifica dei requisiti.
3. *Implementazione delle modifiche:* vengono modificati il documento dei requisiti e, se necessario, il progetto del sistema e la sua implementazione. Si dovrebbe organizzare il documento dei requisiti in modo che sia possibile modificarlo senza ricorrere a grandi riscrittture o riorganizzazioni. Come per i programmi, anche la modificabilità di un documento si ottiene minimizzando i riferimenti esterni e rendendo le sue sezioni più modulari. Così facendo, le singole sezioni possono essere modificate e sostituite senza incidere su altre parti del documento.

Se una nuova modifica deve essere implementata con urgenza, c'è sempre la tentazione di modificare subito il sistema e poi retroattivamente il documento dei requisiti. Questo porta quasi inevitabilmente ad avere una dissonanza tra la specifica dei requisiti e l'implementazione del sistema. Una volta apportate le modifiche al sistema, ci si potrebbe dimenticare di riportarle sul documento dei requisiti. A volte potrebbe essere necessario apportare modifiche urgenti al sistema; in questi casi, è importante aggiornare quanto prima possibile il documento dei requisiti, in modo da includere le ultime versioni dei requisiti.

Punti chiave

- I requisiti di un sistema software definiscono che cosa il sistema dovrebbe fare e quali sono i vincoli alle sue operazioni e alla sua implementazione.
- I requisiti funzionali sono dichiarazioni sui servizi che il sistema deve fornire o descrizioni di come alcuni calcoli devono essere effettuati.
- I requisiti non funzionali spesso vincolano il sistema e il processo di sviluppo che deve essere seguito. Possono essere requisiti del prodotto, requisiti organizzativi o esterni. Sono spesso in relazione con le proprietà emergenti del sistema e, quindi, si applicano al sistema nel suo complesso.
- Il processo di ingegneria dei requisiti include la deduzione, la specifica, la convalida e la gestione dei requisiti.
- La deduzione dei requisiti è un processo iterativo che può essere rappresentato come una spirale di attività: scoperta, classificazione, organizzazione, negoziazione e documentazione dei requisiti.
- La specifica dei requisiti è il processo che definisce formalmente i requisiti dell’utente e del sistema e genera un documento dei requisiti del software.
- Il documento dei requisiti del software è una serie di definizioni concordate sui requisiti del sistema. Dovrebbe essere organizzato in modo tale che possa essere utilizzato sia dai clienti del sistema sia dagli sviluppatori del software.
- La convalida dei requisiti è il processo che verifica se i requisiti sono validi, coerenti, completi, realistici e verificabili.
- I cambiamenti aziendali, organizzativi e tecnici portano inevitabilmente a modifiche nei requisiti di un sistema software. La gestione dei requisiti è il processo che gestisce e controlla queste modifiche.

Esercizi

- 4.1 Identificate e descrivete brevemente quattro tipi di requisiti che possono essere definiti per un sistema informatico.
- * 4.2 Individuate ambiguità e omissioni nella seguente dichiarazione dei requisiti per una parte del sistema di emissione dei biglietti.

Un sistema automatico di emissione dei biglietti vende biglietti ferroviari. Gli utenti selezionano la loro destinazione e inseriscono la carta di credito e il codice personale. Il biglietto viene emesso e addebitato sulla carta di credito. Quando l’utente preme il pulsante di avvio, compare un menu che mostra le varie destinazioni possibili, insieme a un messaggio che chiede di selezionare una destinazione e il tipo di biglietto. Una volta selezionata la destinazione, sullo schermo appare il prezzo del biglietto e all’utente è chiesto di inserire la carta di credito. Viene verificata la sua validità, e all’utente viene chiesto di inserire il codice personale di identificazione (PIN). Quando la transazione economica è completata, il biglietto viene emesso.

- 4.3 Riscrivete la precedente descrizione utilizzando il metodo strutturato descritto in questo capitolo. Risolvete in modo adeguato le ambiguità scoperte.

-
- * 4.4 Scrivete alcuni requisiti non funzionali per il sistema di emissione dei biglietti, specificando l'affidabilità richiesta e il tempo di risposta.
 - 4.5 Utilizzando la tecnica qui suggerita, dove le descrizioni nel linguaggio naturale sono presentate in forma standard, scrivete alcuni requisiti utente plausibili per le seguenti funzioni:
 - un sistema per il rifornimento fai-da-te del carburante che include un lettore di carte di credito. Il cliente inserisce la carta di credito nel lettore; poi specifica la quantità di carburante richiesta. Il carburante viene erogato e l'importo viene addebitato sul conto del cliente;
 - una funzione per il prelievo di contante da uno sportello bancomat;
 - una funzione che consente ai clienti di una banca online di trasferire fondi da un conto a un altro della stessa banca.
 - * 4.6 Suggerite come un ingegnere responsabile di definire le specifiche dei requisiti di un sistema possa tenere traccia delle relazioni tra requisiti funzionali e non funzionali.
 - * 4.7 Utilizzando le vostre conoscenze sull'utilizzo di uno sportello Bancomat, sviluppatе un insieme di casi d'uso che possono servire come base per capire i requisiti di un sistema Bancomat.
 - 4.8 Chi dovrebbe essere coinvolto nella revisione dei requisiti? Disegnate un modello di processo che mostra come dovrebbe essere organizzata una revisione dei requisiti.
 - * 4.9 Supponete che debbano essere apportate delle modifiche di emergenza a un sistema. Ciò potrebbe richiedere la modifica del software prima che siano approvate le modifiche dei requisiti. Suggerite un modello di processo che apporti tali modifiche e assicuri che il documento dei requisiti e l'implementazione del sistema siano coerenti.
 - 4.10 Vi è stato affidato lo sviluppo di un software da un utente che aveva commissionato lo stesso progetto al vostro precedente datore di lavoro. Scoprite che l'interpretazione dei requisiti data dall'attuale società è diversa dalla precedente. Spiegate che cosa occorre fare in questa situazione. Sapete che, se le ambiguità non saranno risolte, i costi per l'attuale datore di lavoro cresceranno, ma avete anche la responsabilità di agire con riservatezza verso il precedente datore di lavoro.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Integrated Requirements Engineering: A Tutorial.” È un articolo tutorial che descrive le attività di ingegneria dei requisiti e come queste possono essere adattate per soddisfare le moderne pratiche di ingegneria del software. (I. Sommerville, *IEEE Software*, 22(1), January–February 2005) <http://dx.doi.org/10.1109/MS.2005.13>.

“Research Directions in Requirements Engineering.” È un’indagine che mette in evidenza le sfide future per risolvere problemi quali la scalabilità e l’agilità delle tecniche di ingegneria dei requisiti. (B. H. C. Cheng e J. M. Atlee, *Proc. Conf. on Future of Software Engineering*, IEEE Computer Society, 2007) <http://dx.doi.org/10.1109/FOSE.2007.17>.

Mastering the Requirements Process, 3rd ed. Un libro ben scritto, semplice da leggere, basato su un particolare metodo (VOLERE), ma che include anche molti buoni consigli sull’ingegneria dei requisiti (S. Robertson e J. Robertson, 2013, Addison-Wesley).

CAPITOLO

5

Modelli di sistema

L'obiettivo di questo capitolo è introdurre i modelli dei sistemi che possono essere sviluppati come parte dell'ingegneria dei requisiti e della progettazione dei sistemi. Dopo aver letto questo capitolo:

- capirete come i modelli grafici possono essere utilizzati per rappresentare i sistemi software e perché sono necessari vari tipi di modelli per rappresentare completamente un sistema;
- capirete i concetti fondamentali di contesto, interazione, struttura e comportamento nella modellazione dei sistemi;
- conoscerete i principali tipi di diagrammi del linguaggio UML (Unified Modeling Language) e capirete come questi diagrammi possono essere utilizzati nella modellazione dei sistemi;
- capirete il concetto di ingegneria guidata da modelli, in cui un sistema eseguibile viene generato automaticamente da modelli strutturali e comportamentali.

- 5.1 Modelli contestuali
- 5.2 Modelli di interazione
- 5.3 Modelli strutturali
- 5.4 Modelli comportamentali
- 5.5 Architettura guidata da modelli

La modellazione dei sistemi è il processo che sviluppa modelli astratti di un sistema, dove ogni modello rappresenta una differente vista o prospettiva del sistema. Oggi per modellazione dei sistemi di solito s'intende la rappresentazione di un sistema utilizzando qualche tipo di notazione grafica basata sui diagrammi UML (Unified Modeling Language). Tuttavia, è anche possibile sviluppare modelli (matematici) formali di un sistema, di solito nella forma di una specifica dettagliata del sistema. Tratterò la modellazione grafica tramite l'UML qui, mentre nel Capitolo 10 descriverò brevemente la modellazione formale.

I modelli sono utilizzati durante il processo di ingegneria dei requisiti per facilitare la deduzione dettagliata dei requisiti per un sistema, durante il processo di progettazione per descrivere il sistema agli ingegneri che lo devono implementare, e dopo l'implementazione per documentare la struttura e il funzionamento del sistema. È possibile sviluppare modelli sia di sistemi esistenti sia di nuovi sistemi.

1. I modelli di un sistema esistente si usano durante l'ingegneria dei requisiti. Servono a chiarire che cosa fa il sistema esistente e possono essere utilizzati per focalizzare la discussione degli stakeholder sui punti di forza e di debolezza del sistema.
2. I modelli di un nuovo sistema si usano durante l'ingegneria dei requisiti per descrivere con maggior chiarezza i requisiti proposti ad altri stakeholder del sistema. Gli ingegneri utilizzano questi modelli per discutere le proposte di progettazione e per documentare l'implementazione del sistema. Se si utilizza un processo di ingegneria guidato da modelli (Brambilla, Cabot e Wimmer 2012), è possibile generare una implementazione completa o parziale del sistema da questi modelli.

È importante capire che il modello di un sistema non è una rappresentazione completa del sistema. Il modello tralascia deliberatamente i dettagli per rendere più comprensibile il sistema. Un modello è un'astrazione del sistema che si sta studiando, non una rappresentazione alternativa del sistema. La rappresentazione di un sistema dovrebbe mantenere tutte le informazioni sull'entità che rappresenta. Un'astrazione deliberatamente semplifica il progetto di un sistema e ne evidenzia le caratteristiche più salienti. Per esempio, le slide di PowerPoint che sono associate a questo libro nella lingua inglese originale sono un'astrazione dei punti chiave del libro; ma il libro tradotto in italiano è una *rappresentazione* alternativa del libro originale. L'intento del traduttore dovrebbe essere quello di mantenere tutte le informazioni che sono presenti nella versione inglese.

È possibile sviluppare vari modelli per rappresentare il sistema da differenti prospettive; per esempio:

1. una prospettiva esterna: viene modellato il contesto o l'ambiente in cui opera il sistema;
2. una prospettiva di interazioni: vengono modellate le interazioni tra il sistema e il suo ambiente, o tra i componenti del sistema;

Unified Modeling Language (UML)

Il linguaggio UML è un insieme di 13 tipi di diagrammi che possono essere utilizzati per modellare i sistemi software. Nacque negli anni '90 dalla modellazione orientata agli oggetti, dove analoghe notazioni orientate agli oggetti furono integrate per creare il linguaggio UML. Una versione principale (UML 2) fu completata nel 2004. L'UML è universalmente accettato come l'approccio standard per sviluppare modelli di sistemi software. Varianti, come SysML, sono state proposte per la modellazione più generale di sistemi.

<http://software-engineering-book.com/web/uml/>

3. una prospettiva strutturale: viene modellata l'organizzazione del sistema o la struttura dei dati elaborati dal sistema;
4. una prospettiva comportamentale: vengono modellati il comportamento dinamico del sistema e le sue risposte agli eventi.

Quando si sviluppano i modelli dei sistemi, occorre una certa flessibilità nel modo di utilizzare la notazione grafica. Non sempre bisogna essere rigidi nell'applicare i dettagli di una notazione. Il dettaglio e il rigore di un modello dipendono dal modo in cui s'intende utilizzarlo. Ci sono tre modi in cui i modelli grafici sono comunemente utilizzati.

1. Per stimolare e focalizzare la discussione su un sistema proposto o già esistente. Scopo di questo modello è stimolare e focalizzare la discussione tra gli ingegneri del software che si occupano dello sviluppo del sistema. I modelli possono essere incompleti (se riguardano i punti chiave della discussione), e possono utilizzare in modo informale la notazione grafica. È questo il modo in cui i modelli sono normalmente utilizzati nella modellazione agile (Ambler e Jeffries 2002).
2. Per documentare un sistema esistente. Se i modelli vengono utilizzati nella documentazione, non devono essere completi, in quanto possono servire per documentare soltanto alcune parti di un sistema. Tuttavia, questi modelli devono essere corretti – devono utilizzare la notazione correttamente e descrivere accuratamente il sistema.
3. Per fornire una descrizione dettagliata del sistema che può essere utilizzata per generare l'implementazione del sistema. Se i modelli del sistema sono utilizzati come parte del processo di sviluppo, devono essere completi e corretti. Poiché questi modelli sono utilizzati come base per generare il codice sorgente del sistema, occorre molta attenzione nel non confondere simboli simili, come le frecce tratteggiate e piene, che possono avere significati differenti.

In questo capitolo utilizzerò i diagrammi definiti nel linguaggio UML (Rumbaugh, Jacobson e Booch 2004; Booch, Rumbaugh e Jacobson 2005), che è diventato un linguaggio standard per la modellazione orientata agli oggetti.

L’UML ha 13 tipi di diagrammi e quindi consente la creazione di vari tipi di modelli di sistemi. Tuttavia, un’indagine (Erickson e Siau 2007) ha indicato che molti utenti del linguaggio UML ritengono che 5 tipi di diagrammi siano sufficienti per rappresentare gli elementi essenziali di un sistema. Io utilizzerò questi cinque diagrammi UML:

1. *diagrammi di attività*: mostrano le attività coinvolte in un processo o nell’elaborazione dei dati;
2. *diagrammi di casi d’uso*: mostrano le interazioni tra un sistema e il suo ambiente;
3. *diagrammi di sequenza*: mostrano le interazioni tra attori e sistema e tra i componenti del sistema;
4. *diagrammi di classe*: mostrano le classi del sistema e le loro associazioni;
5. *diagrammi di stato*: mostrano come il sistema reagisce agli eventi interni ed esterni.

5.1 Modelli contestuali

Nelle prime fasi del processo di specifica di un sistema, si devono fissare i confini del sistema, ovvero cosa fa parte e cosa non fa parte del sistema da sviluppare. Per far questo occorre lavorare con gli stakeholder del sistema e decidere quali funzionalità includere nel sistema e quali processi e operazioni eseguire nell’ambiente del sistema. Potreste decidere che il supporto automatizzato per alcuni processi aziendali sia implementato nel software da sviluppare o sia supportato da sistemi differenti. Dovreste stare attenti a possibili sovrapposizioni delle funzionalità con i sistemi esistenti e decidere se implementare le nuove funzionalità. Queste decisioni devono essere prese all’inizio del processo per limitare i costi e i tempi necessari per capire i requisiti e il progetto del sistema.

In alcuni casi, il confine tra il sistema e il suo ambiente è relativamente chiaro. Per esempio, quando un sistema automatico sta sostituendo un sistema esistente, manuale o computerizzato, di solito l’ambiente del nuovo sistema è lo stesso di quello esistente. In altri casi, dove c’è più flessibilità, occorre decidere qual è il confine tra il sistema e il suo ambiente durante il processo di ingegneria dei requisiti.

Per esempio, supponete di dover sviluppare la specifica per il sistema Mentcare. Questo sistema ha il compito di gestire le informazioni sui pazienti che frequentano le cliniche psichiatriche e sulle cure che vengono prescritte. Per sviluppare la specifica di questo sistema, dovrete decidere se il sistema dovrà focalizzarsi esclusivamente sulla raccolta delle informazioni sui consulti dei dottori (utilizzando altri sistemi per raccogliere informazioni personali sui pazienti) o se dovrà anche raccogliere informazioni personali sui pazienti. Il vantaggio di affidare ad altri sistemi le informazioni sui pazienti è che si evita di duplicare i dati.

Lo svantaggio principale è che, utilizzando altri sistemi, l'accesso alle informazioni è più lento e, se questi sistemi non sono disponibili, diventa impossibile utilizzare il sistema Mentcare.

In alcune situazioni, la base dell'utenza di un sistema è molto diversa, e gli utenti hanno un'ampia gamma di requisiti per il sistema. Potreste decidere di non definire i confini in modo esplicito, ma di sviluppare un sistema configurabile che possa essere adattato alle esigenze di utenti differenti. Questo è stato l'approccio adottato nei sistemi iLearn, presentati nel Capitolo 1, dove gli utenti spaziavano dai bambini che non sapevano leggere agli adulti, ai loro insegnanti e agli amministratori scolastici. Poiché questi gruppi richiedono confini di sistema differenti, abbiamo specificato una configurazione che consente di definire i confini durante lo sviluppo del sistema.

La definizione dei confini di un sistema non è un giudizio senza valore: problemi sociali e organizzativi possono richiedere che la posizione di un confine sia determinata da fattori non tecnici. Per esempio, un confine potrebbe essere posto in modo che il processo di analisi si svolga in un unico luogo, oppure potrebbe essere scelto in modo da non dover consultare un manager particolarmente problematico, oppure ancora potrebbe essere scelto in modo da aumentare i costi del sistema e, di conseguenza, la divisione dello sviluppo del sistema dovrà essere ampliata per progettare e implementare il sistema.

Una volta prese alcune decisioni sui confini del sistema, occorre svolgere una parte dell'attività di analisi che riguarda la definizione del contesto e delle dipendenze che il sistema ha nel suo ambiente. Normalmente il primo passo di questa attività è la produzione di un semplice modello architettonurale.

La Figura 5.1 è un modello contestuale che mostra il sistema Mentcare e gli altri sistemi nel suo ambiente. Come potete notare, il sistema Mentcare è collegato al sistema degli appuntamenti e a un sistema più generale di registrazione dei pazienti con il quale condivide i dati. Il sistema è anche collegato ai sistemi per i report amministrativi e le accettazioni ospedaliere, e a un sistema di statistica che raccoglie le informazioni che servono per effettuare le ricerche. Infine, il sistema utilizza un sistema per generare le prescrizioni dei farmaci per i pazienti.

I modelli contestuali di solito mostrano che l'ambiente include molti altri sistemi automatizzati, ma non indicano le relazioni tra i sistemi nell'ambiente e il sistema che si sta specificando. I sistemi esterni possono produrre dati per il sistema o consumare dati dal sistema; possono condividere dati con il sistema, o possono essere collegati direttamente, attraverso una rete o non essere collegati affatto. Possono trovarsi fisicamente nello stesso luogo o in edifici separati. Tutte queste relazioni possono influire sui requisiti e il progetto del sistema che si sta definendo e, quindi, devono essere prese in considerazione. Per questo, si usano semplici modelli contestuali insieme ad altri modelli, come i modelli dei processi aziendali. Questi modelli descrivono i processi umani e automatizzati in cui si usano particolari sistemi software.

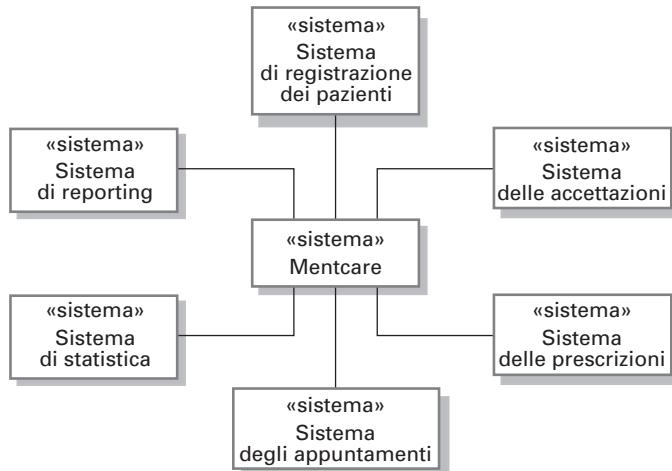


Figura 5.1 Il contesto del sistema Mentcare.

I diagrammi di attività UML possono essere utilizzati per mostrare i processi aziendali in cui si usano i sistemi. La Figura 5.2 è un diagramma di attività UML che mostra dove il sistema Mentcare è utilizzato in un importante processo di salute mentale: la detenzione involontaria.

A volte, i pazienti che soffrono di problemi mentali possono essere un pericolo per se stessi o per altre persone; per questo potrebbero essere detenuti contro la loro volontà in un ospedale dove possono essere curati. Tale detenzione è soggetta a rigorose garanzie giuridiche – per esempio, la decisione di detenere un paziente

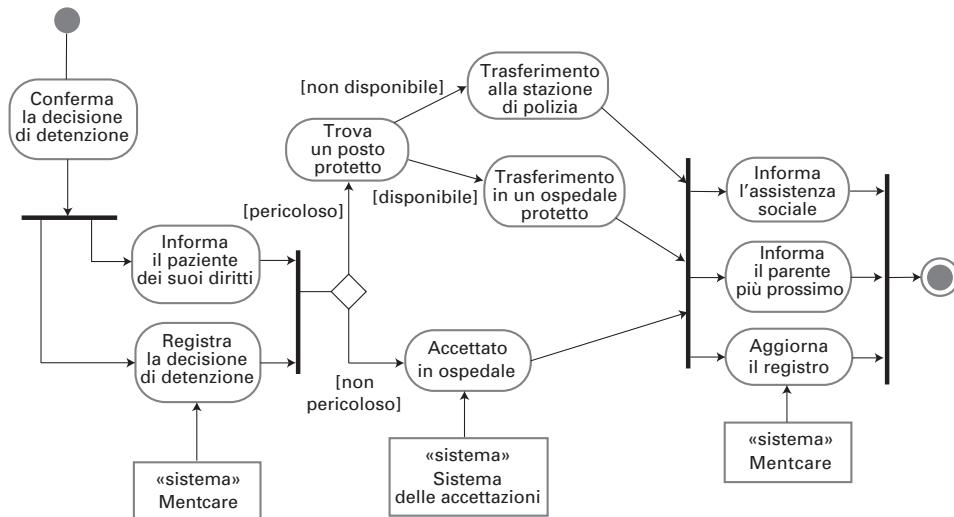


Figura 5.2 Un modello di processo di detenzione involontaria.

deve essere periodicamente riesaminata, in modo che le persone non restino detenute indefinitamente senza una buona ragione. Una funzione critica del sistema Mentcare è assicurare che tali garanzie siano implementate e che i diritti dei pazienti siano rispettati.

I diagrammi di attività UML mostrano le attività in un processo e il flusso di controllo da un’attività all’altra. L’inizio di un processo è indicato da un cerchio pieno, la fine da un cerchio all’interno di un altro cerchio. I rettangoli con gli spigoli arrotondati rappresentano le attività, ovvero specifici sottoprocessi che devono essere eseguiti. È possibile includere degli oggetti nei diagrammi delle attività. La Figura 5.2 mostra i sistemi che sono utilizzati per supportare differenti sottoprocessi all’interno di un processo di detenzione involontaria. Ho mostrato che questi sono sistemi separati utilizzando gli stereotipi UML, dove il tipo di entità è indicato tra parentesi quadre.

Le frecce rappresentano il flusso di lavoro da un’attività all’altra; un barra piena indica il coordinamento delle attività. Quando il flusso da più attività porta a una barra piena, allora tutte queste attività devono essere completate prima che il processo possa avanzare. Quando il flusso da una barra piena porta a più attività, queste possono essere eseguite in parallelo. Per esempio, nella Figura 5.2, le attività di informare l’assistenza sociale e il parente più prossimo del paziente, e l’aggiornamento del registro delle detenzioni possono svolgersi contemporaneamente.

Le frecce possono essere associate ad annotazioni tra parentesi quadre, che specificano quando tale flusso viene seguito. Nella Figura 5.2 potete notare che le parentesi quadre mostrano i flussi per i pazienti che sono socialmente pericolosi oppure no. I pazienti socialmente pericolosi devono essere detenuti in un posto protetto. I pazienti, invece, che sono pericolosi per se stessi possono essere ammessi in un reparto appropriato di un ospedale, dove sono tenuti sotto stretta sorveglianza.

5.2 Modelli di interazione

Tutti i sistemi hanno un’interazione di qualche tipo; per esempio, un’interazione con l’utente, che si realizza tramite input e output; un’interazione tra il software che si sta sviluppando e altri sistemi nel suo ambiente; o un’interazione tra i componenti di un sistema software. La modellazione dell’interazione con l’utente è importante perché aiuta a identificare i requisiti dell’utente. La modellazione dell’interazione tra sistemi mette in evidenza eventuali problemi di comunicazione. La modellazione dell’interazione tra i componenti aiuta a capire se la struttura proposta per un sistema sarà in grado di fornire la fidatezza e le prestazioni richieste.

Questo paragrafo descrive due approcci correlati alla modellazione delle interazioni.

1. Modellazione dei casi d’uso, che è particolarmente utilizzata per modellare le interazioni tra un sistema e gli agenti esterni (utenti o altri sistemi).
2. Diagrammi di sequenza, che sono utilizzati per modellare le interazioni tra i componenti di un sistema, sebbene possano essere aggiunti anche gli agenti esterni.

I modelli dei casi d’uso e i diagrammi di sequenza presentano interazioni a differenti livelli di dettaglio e, quindi, possono essere utilizzati insieme. Per esempio, i dettagli delle iterazioni riguardanti un caso d’uso di alto livello possono essere documentati in un diagramma di sequenza. Il linguaggio UML include anche i diagrammi di comunicazione che possono essere utilizzati per modellare le interazioni. Non descriverò questo tipo di diagrammi perché i diagrammi di comunicazione sono semplicemente una rappresentazione alternativa dei diagrammi di sequenza.

5.2.1 Modellazione dei casi d’uso

La modellazione dei casi d’uso venne originariamente sviluppata da Ivar Jacobson negli anni ’90 (Jacobson et al. 1993), e un tipo di diagramma UML che supporta questa modellazione è parte del linguaggio UML. Un caso d’uso può essere considerato come una semplice descrizione di ciò che un utente si aspetta da un sistema in tale interazione. Nel Capitolo 4 ho trattato i casi d’uso per la deduzione dei requisiti. Come detto nel Capitolo 4, i modelli dei casi d’uso sono più utili nelle prime fasi della progettazione di un sistema, anziché durante l’ingegneria dei requisiti.

Ciascun caso d’uso rappresenta un compito discreto che richiede un’interazione esterna con un sistema. Nella sua forma più semplice, un caso d’uso è rappresentato da un’ellisse, mentre gli attori coinvolti nel caso d’uso sono rappresentati da figure stilizzate. La Figura 5.3 mostra un caso d’uso che rappresenta il compito di trasferire i dati dal sistema Mentcare a un sistema più generale di registrazione dei pazienti. Questo sistema più generale conserva i dati di sintesi su un paziente, anziché i dati su ciascun consulto medico, che sono registrati nel sistema Mentcare.

Notate che ci sono due attori in questo caso d’uso – l’operatore che sta trasferendo i dati e il sistema di registrazione dei pazienti. La notazione con le figure stilizzate venne originariamente sviluppata per includere le interazioni umane, ma



Figura 5.3 Il caso d’uso di trasferimento dei dati.

Sistema Mentcare – Trasferimento dei dati	
Attori	Personale di accettazione, sistema di registrazione dei pazienti (SRP).
Descrizione	Un addetto all'accettazione può trasferire i dati dal sistema Mentcare a un database generale di registrazione dei pazienti che è mantenuto da un'autorità sanitaria. Le informazioni trasferite possono essere informazioni personali aggiornate (indirizzo, numero di telefono ecc.) o una sintesi della diagnosi e della cura dei pazienti.
Dati	Informazioni personali dei pazienti, sintesi della cura.
Stimolo	Comando emesso dal personale di accettazione.
Risposta	Conferma che l'SRP è stato aggiornato.
Commenti	L'addetto all'accettazione deve avere le autorizzazioni appropriate per accedere alle informazioni dei pazienti e all'SRP.

Figura 5.4 Descrizione tabellare del caso d'uso “Trasferimento dei dati”.

è utilizzata anche per rappresentare altri sistemi e hardware esterni. Formalmente, i diagrammi dei casi d'uso dovrebbero utilizzare linee senza frecce, in quanto le frecce nel linguaggio UML indicano la direzione del flusso dei messaggi. Ovviamen-te, in un caso d'uso, i messaggi viaggiano in entrambe le direzioni. Tuttavia, nella Figura 5.3 le frecce sono state utilizzate informalmente per indicare che il personale di accettazione inizia la transazione e i dati vengono trasferiti nel sistema di registrazione dei pazienti.

I diagrammi dei casi d'uso forniscono una semplice panoramica su un'interazione, quindi occorre aggiungere altri dettagli per completare la descrizione dell'interazione. Questi dettagli possono essere una semplice descrizione testuale, una descrizione strutturata in una tabella o un diagramma di sequenza. Scegliete la forma più appropriata in funzione del caso d'uso e del livello di dettagli che credete sia richiesto dal modello. Io ritengo che la forma tabellare standard sia quella più utile. La Figura 5.4 mostra una descrizione tabellare del caso d'uso “Trasferimento dei dati”.

I diagrammi di casi d'uso composti mostrano un certo numero di casi d'uso differenti. A volte è possibile includere tutte le possibili interazioni all'interno di un sistema in un unico diagramma di casi d'uso composto. Altre volte questo potrebbe essere impossibile per l'elevato numero di casi d'uso; in questi casi, è bene sviluppare più diagrammi, ciascuno dei quali mostra i relativi casi d'uso. Per esempio, la Figura 5.5 mostra tutti i casi d'uso del sistema Mentcare in cui è coinvolto l'attore “Personale di accettazione”. Ciascuno di questi casi d'uso dovrebbe essere accompagnato da una descrizione più dettagliata.

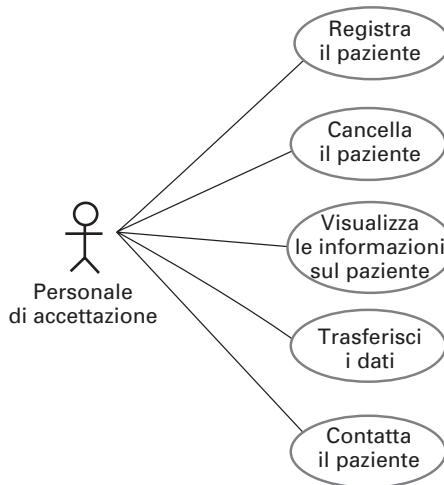


Figura 5.5 I casi d'uso in cui è coinvolto l'attore “Personale di accettazione”.

Il linguaggio UML include un certo numero di costrutti per condividere un intero caso d'uso o una sua parte in altri diagrammi di casi d'uso. Sebbene questi costrutti a volte possano essere utili per i progettisti del sistema, nella mia esperienza ho constatato che molte persone, specialmente gli utenti finali, li trovano difficili da capire. Per questo motivo non li descriverò.

5.2.2 Diagrammi di sequenza

I diagrammi di sequenza nel linguaggio UML sono utilizzati principalmente per modellare le interazioni tra gli attori e gli oggetti di un sistema e le interazioni tra gli stessi oggetti. L'UML ha una ricca sintassi di diagrammi di sequenza, che consentono di modellare vari tipi di interazioni. Poiché non è possibile trattarli tutti, mi limiterò a descrivere gli elementi fondamentali di questo tipo di diagrammi.

Come indica il nome, un diagramma di sequenza mostra la sequenza delle interazioni che si svolgono durante un particolare caso d'uso o una sua istanza. La Figura 5.6 è un esempio di diagramma di sequenza che illustra gli elementi fondamentali della notazione. Questo diagramma è un modello delle interazioni che riguardano il caso d'uso della visualizzazione delle informazioni dei pazienti, dove il personale addetto all'accettazione può vedere alcune informazioni sui pazienti.

Gli oggetti e gli attori coinvolti sono elencati all'inizio del diagramma, con una linea tratteggiata verticale che parte da ciascuno di essi. Le frecce con i commenti indicano le interazioni tra gli oggetti. Il rettangolo sulle linee tratteggiate indica la linea di vita dell'oggetto interessato (ovvero il tempo durante il quale l'istanza dell'oggetto prende parte al calcolo). La sequenza delle interazioni va letta dall'alto verso il basso. I commenti sulle frecce indicano le chiamate degli

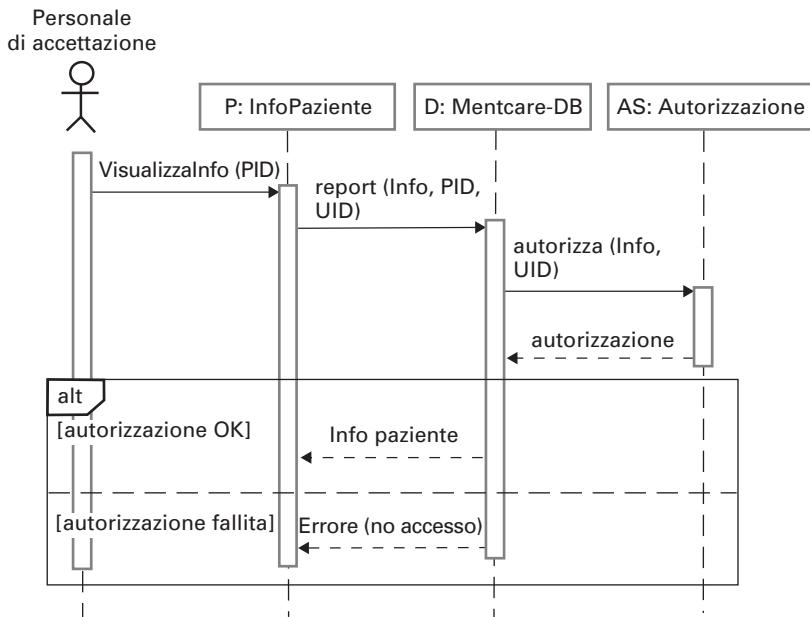


Figura 5.6 Diagramma di sequenza per la visualizzazione delle informazioni dei pazienti.

oggetti, i loro parametri e i valori di ritorno. Questo esempio mostra anche la notazione utilizzata per indicare le alternative. Un box chiamato “alt” viene utilizzato con le condizioni indicate tra parentesi quadre, con le opzioni di interazione alternative separate da una linea tratteggiata.

La Figura 5.6 può essere letta nel seguente modo:

- l’addetto all’accettazione avvia il metodo VisualizzaInfo in un’istanza P della classe di oggetti InfoPaziente, fornendo l’identificatore del paziente, il PID per identificare le informazioni richieste. P è un oggetto dell’interfaccia utente, che è visualizzato come un modulo che mostra le informazioni sul paziente;
- l’istanza P chiama il database per ottenere le informazioni richieste, fornendo l’identificatore dell’addetto all’accettazione (UID) per consentire il controllo dell’accesso ai dati (a questo livello non è importante da dove proviene il codice di identificazione dell’addetto);
- il database controlla tramite un sistema di autorizzazioni che l’addetto all’accettazione sia autorizzato a svolgere questa operazione;
- se l’addetto è autorizzato, le informazioni sul paziente vengono caricate dal database e visualizzate in un modulo sullo schermo dell’utente. Se l’autorizzazione fallisce, viene visualizzato un messaggio di errore. Il box “alt” contiene le opzioni che indicano quale delle possibili interazioni sarà eseguita. La condizione che seleziona l’opzione è indicata tra parentesi quadre.

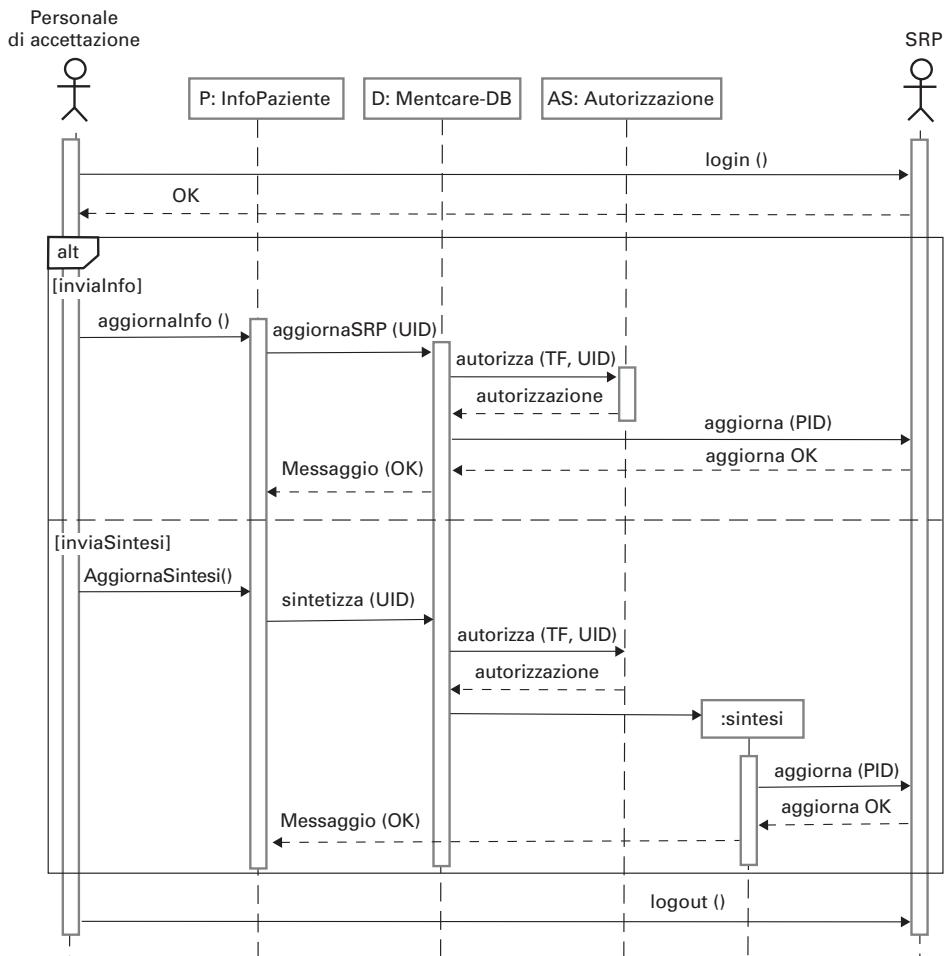


Figura 5.7 Diagramma di sequenza per il trasferimento dei dati.

La Figura 5.7 è un ulteriore esempio di diagramma di sequenza dello stesso sistema che illustra due caratteristiche aggiuntive: la comunicazione diretta tra gli attori nel sistema e la creazione di oggetti come parte di una sequenza di operazioni. In questo esempio, un oggetto di tipo Sintesi viene creato per conservare i dati riassuntivi che vengono caricati in un sistema di registrazione dei pazienti (SRP). Il diagramma può essere letto nel seguente modo:

1. l'addetto all'accettazione si collega (login) al sistema SRP;
2. sono disponibili due opzioni (come indica il box "alt"). Queste opzioni consentono il trasferimento diretto delle informazioni aggiornate sui pazienti dal database Mentcare al sistema SRP e il trasferimento dei dati riassuntivi dal database Mentcare al sistema SRP;

3. in entrambi i casi, vengono controllate le autorizzazioni dell'addetto all'accettazione mediante il sistema di autorizzazioni;
4. le informazioni personali possono essere trasferite direttamente dall'oggetto dell'interfaccia utente al sistema SRP. In alternativa, può essere creato un record di sintesi dal database, e questo record viene trasferito;
5. completato il trasferimento, il sistema SRP emette un messaggio di stato e l'utente chiude il collegamento (logout).

A meno che non stiate utilizzando i diagrammi di sequenza per generare il codice o un documento dettagliato, in questi diagrammi non vanno incluse le interazioni. Se state sviluppando modelli di sistema nelle fasi iniziali del processo di sviluppo a supporto dell'ingegneria dei requisiti e della progettazione di alto livello, ci saranno molte interazioni che dipendono dalle scelte di implementazione. Per esempio, nella Figura 5.7, la decisione su come ottenere l'identificatore dell'utente per controllarne l'autorizzazione può essere rimandata. In una implementazione, questo potrebbe richiedere l'interazione con un oggetto Utente; poiché questo non è importante in questa fase, non occorre includere l'interazione nel diagramma di sequenza.

5.3 Modelli strutturali

I modelli strutturali del software mostrano l'organizzazione di un sistema in funzione dei suoi componenti e le relazioni fra questi componenti. I modelli strutturali possono essere statici, se rappresentano l'organizzazione del progetto del sistema, o dinamici, se rappresentano l'organizzazione del sistema durante la sua esecuzione. Questi modelli non sono la stessa cosa – l'organizzazione dinamica di un sistema come insieme di elementi interagenti può essere molto diversa da un modello statico dei componenti del sistema.

Noi creiamo modelli strutturali di un sistema quando descriviamo e progettiamo l'architettura del sistema. Questi possono essere modelli dell'architettura globale sistema o modelli più dettagliati degli oggetti del sistema e delle loro relazioni.

In questo paragrafo descriverò l'uso dei diagrammi di classe per modellare la struttura statica delle classi degli oggetti in un sistema software. La progettazione architettonica è un aspetto importante dell'ingegneria del software, e nella presentazione dei modelli architettonici possono essere utilizzati tutti i diagrammi UML di componenti, package e consegna. Tratterò la modellazione architettonica nei Capitoli 6 e 17.

5.3.1 Diagrammi di classe

I diagrammi di classe si usano quando si sviluppa un modello di sistema orientato agli oggetti per mostrare le classi di un sistema e le loro associazioni. Generi-



Figura 5.8 Classi e associazioni UML.

camente, una classe di oggetti può essere pensata come una definizione generale di un tipo di oggetti del sistema. Un’associazione è un collegamento tra classi che indica che esiste qualche relazione tra queste classi. Ne consegue che ciascuna classe potrebbe avere qualche conoscenza della sua classe associata.

Quando si sviluppano i modelli durante le prime fasi del processo di ingegneria del software, gli oggetti rappresentano qualcosa del mondo reale, come un paziente, una medicina o un dottore. Durante l’implementazione si definiscono gli oggetti che rappresentano i dati che saranno elaborati dal sistema. Questo paragrafo tratta in particolare la modellazione degli oggetti del mondo reale come parte dei requisiti o dei processi iniziali di progettazione del software. Un approccio simile si usa nella modellazione della struttura dei dati.

I diagrammi di classe del linguaggio UML possono essere rappresentati a vari livelli di dettaglio. Quando si sviluppa un modello, di solito la prima fase è osservare il mondo reale, identificare gli oggetti essenziali e rappresentarli come classi. Il modo più semplice per rappresentare questi diagrammi è scrivere il nome della classe all’interno di un rettangolo. È anche possibile rappresentare l’esistenza di un’associazione tracciando una linea tra le classi. Per esempio, la Figura 5.8 è un semplice diagramma di classe che mostra due classi, Paziente e Record del Paziente, con un’associazione tra di esse. A questo livello, non occorre specificare il tipo di associazione.

La Figura 5.9 sviluppa il semplice diagramma di classe della Figura 5.8 per mostrare che gli oggetti della classe Paziente sono anche in relazione con molte altre classi. Questo esempio mostra come sia possibile dare un nome alle associazioni per dare al lettore un’indicazione del tipo di relazione che esiste.

Le Figure 5.8 e 5.9 mostrano un’importante caratteristica dei diagrammi di classe – la capacità di mostrare come più oggetti sono coinvolti in un’associazione. Nella Figura 5.8 ciascuna estremità dell’associazione è annotata con il numero 1, in quanto c’è una relazione 1:1 tra gli oggetti di questa classe. Questo significa che ciascun paziente ha esattamente un record, e ciascun record contiene le informazioni su un solo paziente.

Come potete notare dalla Figura 5.9, sono possibili altri rapporti (molteplicità) nelle relazioni. È possibile definire il numero esatto di oggetti che sono coinvolti nell’associazione (per esempio, 1..4) oppure, utilizzando un asterisco (*), indicare che nell’associazione c’è un numero indefinito di oggetti. Per esempio, la molteplicità 1..* nella Figura 5.9 sulla relazione tra Paziente e Malattia indica che un paziente può soffrire di diverse malattie e che la stessa malattia può essere associata a diversi pazienti.

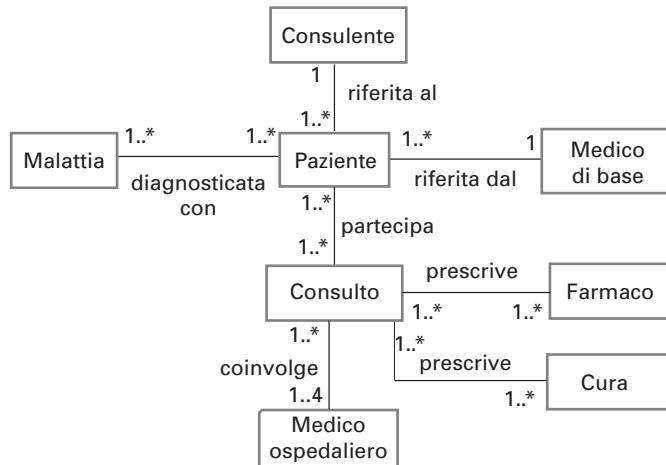


Figura 5.9 Classi e associazioni nel sistema Mentcare.

A questo livello di dettaglio, i diagrammi di classe somigliano ai modelli semantici dei dati. Questi modelli sono utilizzati nella progettazione dei database; mostrano le entità dei dati, gli attributi associati e le relazioni tra queste entità (Hull e King 1987). L'UML non ha un tipo di diagramma per la modellazione dei database, in quanto modella i dati utilizzando gli oggetti e le loro relazioni. Tuttavia, è possibile utilizzare l'UML per rappresentare un modello semantico di dati. Potete immaginare le entità di un modello semantico come le classi di oggetti (non hanno operazioni), gli attributi come gli attributi delle classi di oggetti e le relazioni come le associazioni tra le classi di oggetti.

Quando mostrate le associazioni tra le classi, è meglio rappresentare queste classi nel modo più semplice possibile, senza operazioni o attributi. Per definire gli oggetti più dettagliatamente, aggiungete informazioni sui loro attributi (le caratteristiche di un oggetto) e le operazioni (le funzioni di un oggetto). Per esempio, l'oggetto Paziente ha l'attributo Indirizzo e può includere l'operazione CambiaIndirizzo, che viene chiamata quando un paziente comunica che ha cambiato indirizzo.

Nell'UML, per mostrare le operazioni e gli attributi di una classe, basta ampliare il rettangolo che rappresenta la classe. La Figura 5.10 mostra un oggetto che rappresenta un consulto tra un dottore e un paziente:

1. il nome della classe di oggetti è posto in cima al rettangolo;
2. gli attributi della classe sono nella parte centrale; sono inclusi i nomi degli attributi e, facoltativamente, i loro tipi (nella Figura 5.10 non sono indicati i tipi);

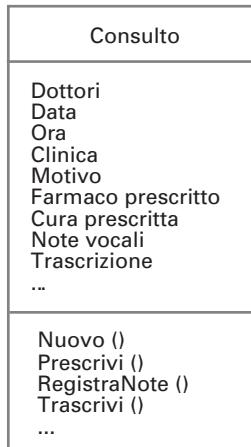


Figura 5.10 La classe Consulto.

3. le operazioni (chiamate metodi in Java o in altri linguaggi di programmazione orientata agli oggetti) associate alla classe di oggetti sono indicate nella parte inferiore del rettangolo. Nella Figura 5.10 sono indicate solo alcune operazioni.

Nell'esempio della Figura 5.10, si suppone che i dottori registrino le note vocali, che saranno trascritte successivamente per memorizzare i dettagli del consulto. Per prescrivere un farmaco, un dottore deve usare il metodo Prescrivi che consente di generare una prescrizione elettronica.

5.3.2 Generalizzazione

La generalizzazione è una tecnica comune che utilizziamo per gestire la complessità. Anziché conoscere le caratteristiche dettagliate di tutto ciò che incontriamo, ci limitiamo a conoscere soltanto le classi generali (animali, automobili, case ecc.) e le loro caratteristiche. Utilizziamo queste conoscenze per classificare le cose e concentrarci sulle differenze tra queste e la loro classe di appartenenza. Per esempio, scoiattoli e ratti sono membri della classe “roditori” e, quindi, hanno in comune le caratteristiche dei roditori. Le definizioni generali si applicano a tutti i membri della classe; per esempio, tutti i roditori hanno denti per rosicchiare.

Quando modelliamo un sistema, spesso è utile esaminare le classi del sistema per vedere se c'è la possibilità di generalizzare e creare delle classi. Questo significa che le informazioni comuni saranno mantenute in un solo posto. Questa è una buona pratica di progettazione in quanto, se vengono proposte delle modifiche, non è necessario esaminare tutte le classi del sistema per verificare se siano influenzate dalle modifiche. Possiamo apportare le modifiche al livello più generale. Nei linguaggi orientati agli oggetti, come Java, la generalizzazione è implementata utilizzando i meccanismi di ereditarietà delle classi.

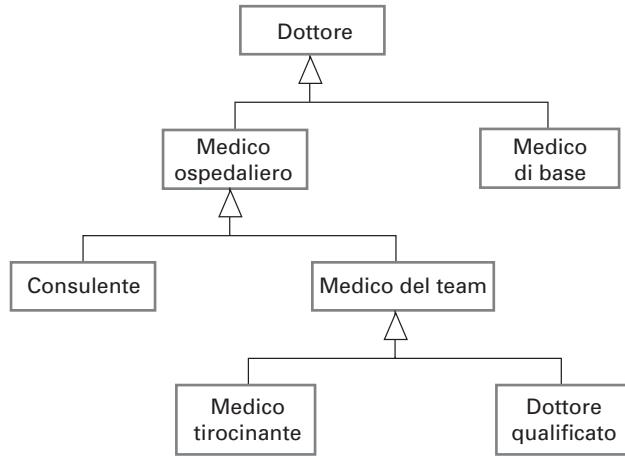


Figura 5.11 Gerarchie di una generalizzazione.

L’UML ha un apposito tipo di associazione per denotare la generalizzazione, come illustra la Figura 5.11. La generalizzazione è rappresentata da una freccia che punta a una classe più generale. Per esempio, i medici di base e i medici ospedalieri possono essere generalizzati come dottori, e ci sono tre tipi di medici ospedalieri: quelli che si sono appena laureati e che hanno bisogno di supervisori (medici tirocinanti); quelli che possono lavorare senza supervisori in una equipe medica (dottori qualificati); medici esperti con piena responsabilità decisionale (consulenti).

In una generalizzazione, le operazioni e gli attributi associati alle classi di livello più alto sono associati anche alle classi di livello più basso. Le classi di livello più basso sono sottoclassi che ereditano operazioni e attributi dalle loro superclassi. Queste classi di livello più basso quindi aggiungono operazioni e attributi più specifici.

Per esempio, tutti i dottori hanno un nome e un numero di telefono, e tutti i medici ospedalieri hanno un codice personale e un cercapersone. I medici di base non hanno questi attributi, in quanto lavorano autonomamente, ma hanno uno studio e un indirizzo. La Figura 5.12 mostra parte della gerarchia della generalizzazione, che ho esteso con gli attributi di classe, per la classe Dottore. Le operazioni associate alla classe Dottore servono a registrare e a cancellare un dottore dal sistema Mentcare.

5.3.3 Aggregazione

Gli oggetti del mondo reale spesso sono composti da diverse parti. Per esempio, un pacchetto di studio per una materia può essere formato da un libro, slide di PowerPoint, quiz e una bibliografia. A volte nel modello di un sistema, occorre illustrare questo. Il linguaggio UML fornisce un tipo speciale di associazione tra

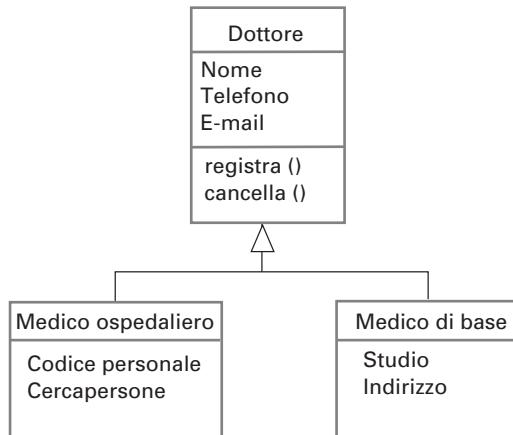


Figura 5.12 Gerarchie di una generalizzazione con più dettagli.

classi chiamato aggregazione, che significa che un oggetto (l'intero) è composto da altri oggetti (le parti). Per definire l'aggregazione, un rombo viene aggiunto al collegamento vicino alla classe che rappresenta l'intero.

La Figura 5.13 mostra che il record di un paziente è un'aggregazione tra un paziente e un numero indefinito di consulti; ovvero il record del paziente contiene le informazioni personali sul paziente e anche un record distinto per ogni consulto con un dottore.

5.4 Modelli comportamentali

I modelli comportamentali sono modelli del comportamento dinamico di un sistema durante la sua esecuzione. Mostrano che cosa accade o che cosa si suppone accada quando un sistema risponde a uno stimolo dal suo ambiente. Gli stimoli possono essere dati o eventi.

1. Si rendono disponibili quei dati che devono essere elaborati dal sistema. La disponibilità dei dati innesca il processo di elaborazione.
2. Un evento innesca l'elaborazione del sistema. Gli eventi possono essere associati a dati, sebbene non sia sempre così.

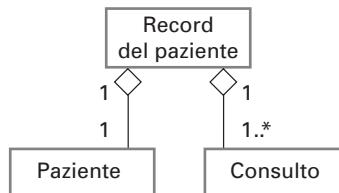


Figura 5.13 Associazione di un'aggregazione.

Diagrammi di flusso dei dati

I diagrammi di flusso dei dati sono modelli di sistemi che mostrano una prospettiva funzionale, dove ciascuna trasformazione rappresenta una singola funzione o processo. Si utilizzano per mostrare come i dati fluiscano attraverso una sequenza di passi di elaborazione. Per esempio, un passo di elaborazione potrebbe essere il filtraggio dei record duplicati in un database di clienti. I dati vengono trasformati in ciascun passo prima di passare al passo successivo. Questi passi di elaborazione o trasformazioni rappresentano le funzioni o i processi software, dove i diagrammi di flusso dei dati si usano per documentare il progetto di un'applicazione software. I diagrammi di attività dell'UML possono essere utilizzati per rappresentare i diagrammi di flusso dei dati..

<http://software-engineering-book.com/web/dfds/>

Molti sistemi aziendali sono sistemi di elaborazione dei dati, che sono controllati principalmente dall'input di dati, con una modesta elaborazione di eventi esterni. L'elaborazione dei dati consiste in una sequenza di azioni sui dati e nella generazione di un output. Per esempio, un sistema di fatturazione telefonica accetta le informazioni sulle telefonate fatte da un cliente, calcola i costi di queste telefonate e genera una fattura per il cliente.

D'altra parte, i sistemi real-time di solito sono guidati da eventi, con minime elaborazioni di dati. Per esempio, un sistema di commutazione per la telefonia fissa risponde agli eventi, generando un segnale a toni quando l'utente alza la cornetta, componendo il numero telefonico formato dai tasti premuti dall'utente e così via.

5.4.1 Modelli guidati dai dati

I modelli guidati dai dati mostrano la sequenza delle azioni coinvolte nell'elaborazione dei dati di input e nella generazione dell'output associato. Possono essere usati durante l'analisi dei requisiti in quanto mostrano l'elaborazione dall'inizio alla fine in un sistema, ovvero l'intera sequenza di azioni che si svolgono dall'input iniziale dei dati da elaborare fino al corrispondente output, che è la risposta del sistema.

I modelli guidati dai dati furono tra i primi modelli grafici del software. Negli anni '70, i metodi di progettazione strutturata utilizzavano i diagrammi di flusso dei dati come un modo per illustrare i passi di elaborazione di un sistema. I modelli di flusso dei dati sono utili perché, tracciando e documentando come i dati associati a un particolare processo si spostano all'interno del sistema, gli analisti e i progettisti possono capire più facilmente che cosa accade nel processo. I diagrammi di flusso dei dati sono semplici e intuitivi e, quindi, più accessibili agli stakeholder rispetto ad altri tipi di modelli. Di solito è possibile spiegarli ai potenziali utenti del sistema che partecipano alla convalida dei modelli.

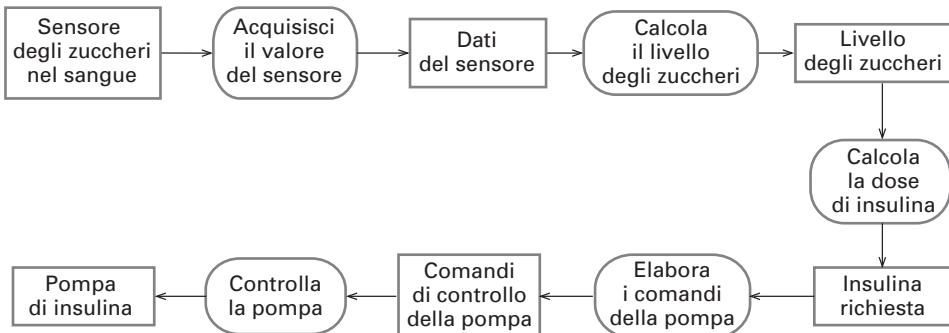


Figura 5.14 Un modello di attività del software della pompa di insulina.

I diagrammi di flusso dei dati possono essere rappresentati nel linguaggio UML utilizzando i diagrammi di attività, descritti nel Paragrafo 5.1. La Figura 5.14 è un semplice diagramma di attività che mostra la catena dei processi di elaborazione che riguardano il software della pompa di insulina. Potete notare i passi di elaborazione, rappresentati come attività (rettangoli a spigoli arrotondati), e i dati che fluiscono tra questi passi, rappresentati come oggetti (rettangoli).

Un modo alternativo di mostrare la sequenza di elaborazione in un sistema consiste nell’usare i diagrammi di sequenza UML. Abbiamo visto come questi diagrammi possono essere utilizzati per modellare le interazioni, ma se li disegniamo in modo che i messaggi siano inviati soltanto da sinistra a destra, allora essi mostrano l’elaborazione sequenziale dei dati nel sistema. La Figura 5.15 mostra questo, utilizzando un modello di sequenza per elaborare un ordine e inviarlo a un fornitore. I modelli di sequenza mettono in evidenza gli oggetti in un sistema, mentre i diagrammi di flusso dei dati mettono in evidenza le operazioni o le attività. Nella pratica, le persone meno esperte trovano più intuitivi i diagrammi di flusso, mentre gli ingegneri preferiscono i diagrammi di sequenza.

5.4.2 Modelli guidati dagli eventi

I modelli guidati dagli eventi mostrano come un sistema risponde agli eventi esterni e interni. Si basa sull’ipotesi che un sistema ha un numero finito di stati e che gli eventi (stimoli) possono causare una transizione da uno stato all’altro. Per esempio, un sistema che controlla una valvola può passare dallo stato “valvola aperta” allo stato “valvola chiusa”, quando riceve il comando (stimolo) di un operatore. Questo modo di vedere il sistema è particolarmente appropriato per i sistemi real-time. I modelli guidati dagli eventi si usano molto per progettare e documentare i sistemi real-time.

Il linguaggio UML supporta la modellazione basata sugli eventi tramite i diagrammi di stato, che si basano sulla notazione Statecharts (Harel 1987). I diagrammi di stato mostrano gli stati e gli eventi che causano le transizioni da uno

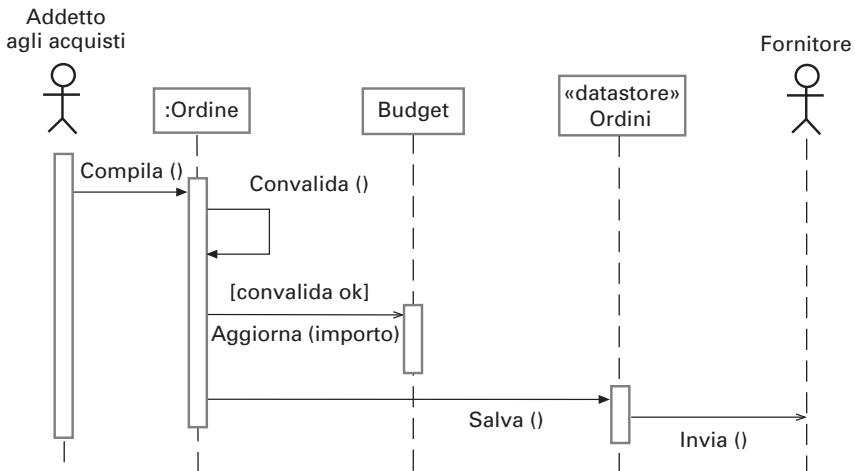


Figura 5.15 Elaborazione di un ordine.

stato all’altro. Non mostrano il flusso dei dati all’interno del sistema, ma possono includere informazioni addizionali sui calcoli da eseguire in ciascuno stato.

Per illustrare la modellazione guidata dagli eventi userò un esempio di controllo del software per un semplice forno a microonde (Figura 5.16). I forni a microonde reali sono molto più complessi di questo sistema, ma il sistema semplificato è più facile da capire. Questo semplice forno ha un interruttore per selezionare tutta la potenza o la metà, una tastiera numerica per immettere il tempo di cottura, un pulsante di avvio/stop e un display alfanumerico.

Per semplicità si suppone che la sequenza delle azioni per usare il forno sia la seguente:

1. selezionare il livello di potenza (potenza massima o metà);
2. impostare il tempo di cottura mediante la tastiera numerica;
3. premere Avvio; il cibo sarà cucinato nel tempo impostato.

Per ragioni di sicurezza, il forno non funziona quando la porta è aperta, e alla fine della cottura un cicalino suona per 5 secondi. Il forno ha un display molto semplice che serve per visualizzare diversi messaggi di avviso o allarme.

Nei diagrammi di stato UML, i rettangoli a spigoli arrotondati rappresentano gli stati del sistema. Possono includere una breve descrizione (dopo la parola “do”, letteralmente “fai”) delle azioni svolte in quello stato. Le frecce con le annotazioni rappresentano gli stimoli che forzano una transizione da uno stato all’altro. Potete indicare gli stati iniziale e finale utilizzando i cerchi pieni, come nei diagrammi di attività.

Come potete notare dalla Figura 5.16, il sistema inizia da uno stato di attesa e poi risponde al pulsante di massima potenza o di metà potenza. Gli utenti possono cambiare idea dopo averne selezionato uno e premere l’altro pulsante. Una volta impostato il tempo, se la porta è chiusa, viene abilitato il pulsante Avvio; premen-

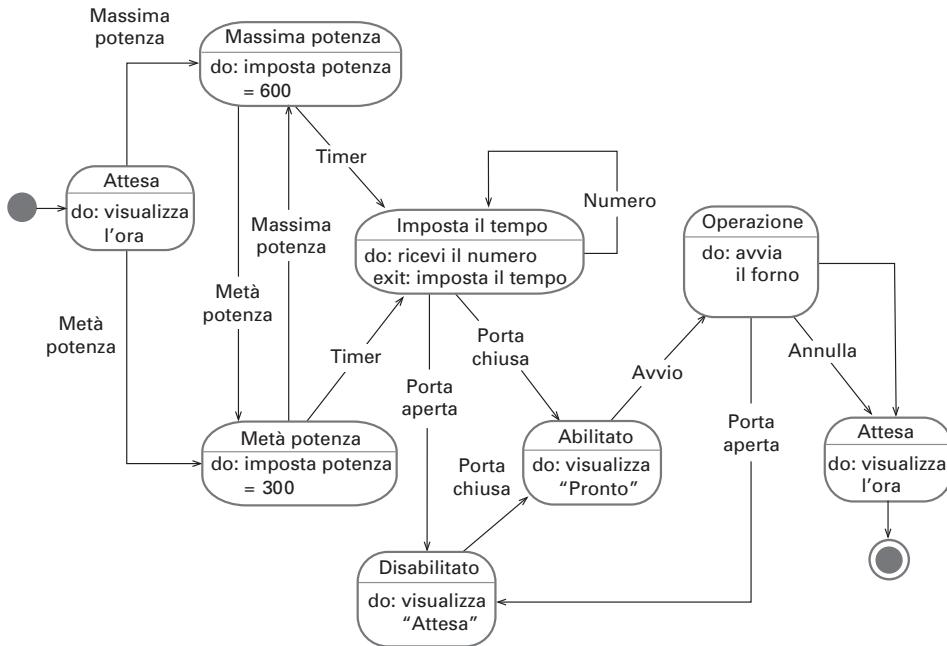


Figura 5.16 Diagramma a stati di un forno a microonde.

do questo pulsante, inizia la cottura che avviene nel tempo specificato. Alla fine della cottura, il sistema ritorna nello stato di attesa.

Il problema della modellazione basata sugli stati è che il numero di possibili stati aumenta rapidamente. Di conseguenza, è necessario nascondere i dettagli nei modelli di grandi sistemi. Un modo per farlo è usare il concetto di “superstato” che comprende una serie di stati separati. Il superstato somiglia a un singolo stato in un modello di alto livello, che poi viene espanso per mostrare maggiori dettagli su un diagramma distinto. Per illustrare questo concetto, si consideri lo stato Operazione nella Figura 5.16; questo è un superstato che può essere espanso come mostra la Figura 5.17.

Lo stato Operazione comprende una serie di sottostati: l'operazione inizia con un controllo di stato e, se si scopre un problema, si attiva un allarme e l'operazione viene disabilitata. La cottura comprende l'avvio del generatore di microonde per il tempo specificato; alla fine, un cicalino suona per alcuni secondi. Se la porta viene aperta durante l'operazione, il sistema passa allo stato disabilitato, come mostra la Figura 5.17.

I modelli degli stati di un sistema forniscono una panoramica dell'elaborazione degli eventi; di solito occorre estendere questi modelli con una descrizione più dettagliata degli stimoli e degli stati del sistema. La Figura 5.18 mostra una descrizione tabellare dei singoli stati e del modo in cui vengono generati gli stimoli che forzano le transizioni di stato.

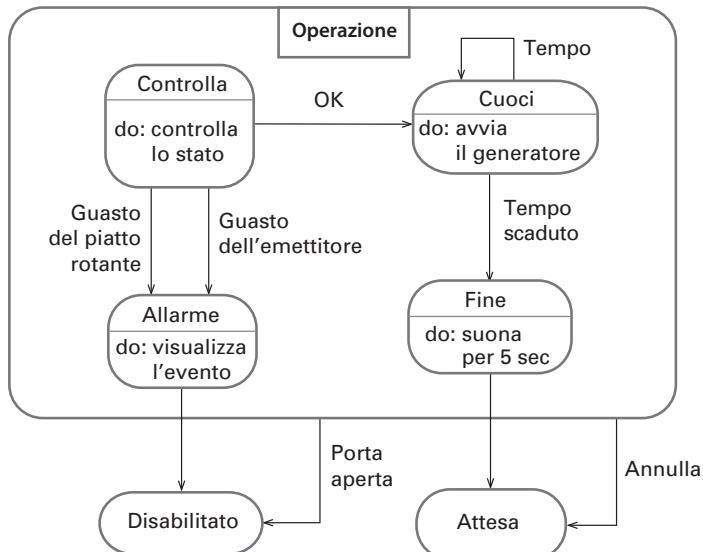


Figura 5.17 Un modello dei sottostati dello stato Operazione.

5.4.3 Ingegneria guidata da modelli

L'ingegneria guidata da modelli (MDE, *Model-Driven Engineering*) è un approccio allo sviluppo del software dove i modelli, anziché i programmi, sono i principali output del processo di sviluppo (Brambilla, Cabot e Wimmer 2012). I programmi che sono eseguiti su piattaforme hardware/software vengono automaticamente generati dai modelli. I sostenitori dell'ingegneria guidata da modelli ritengono che così si alza il livello di astrazione nell'ingegneria del software, in quanto gli ingegneri non devono più preoccuparsi dei dettagli dei linguaggi di programmazione o delle specifiche delle piattaforme di esecuzione.

L'ingegneria guidata da modelli è stata sviluppata dal concetto di architettura guidata da modelli (MDA, *Model-Driven Architecture*), proposto dall'Object Management Group (OMG) come un nuovo paradigma di sviluppo del software (Mellor, Scott e Weise 2004). L'MDA si basa sulle fasi di progettazione e implementazione dello sviluppo del software, mentre l'MDE abbraccia tutti gli aspetti del processo di ingegneria del software. Pertanto, argomenti quali l'ingegneria dei requisiti basata su modelli, processi software per lo sviluppo basato su modelli e test basati su modelli fanno parte dell'MDE, ma non sono considerati nell'MDA.

L'MDA come approccio all'ingegneria dei sistemi è stato sviluppato da alcune grandi compagnie per supportare i loro processi di sviluppo. Il prossimo paragrafo, anziché trattare gli aspetti più generali dell'MDE, descrive l'uso dell'MDA nell'implementazione del software. La diffusione dell'MDE è particolarmente lenta, e poche compagnie hanno adottato questo approccio nel ciclo di vita di

Stato	Descrizione
Attesa	Il forno aspetta l'input. Il display mostra l'ora corrente.
Metà potenza	La potenza del forno è impostata a 300 W. Il display visualizza "Metà potenza".
Massima potenza	La potenza del forno è impostata a 600 W. Il display visualizza "Massima potenza".
Imposta il tempo	Il tempo di cottura è impostato al valore immesso dall'utente. Il display visualizza il tempo di cottura selezionato e viene aggiornato non appena l'utente ha finito di impostarlo.
Disabilitato	Le operazioni del forno sono disattivate per sicurezza; la luce interna è spenta, il display visualizza "Attesa".
Abilitato	Le operazioni del forno sono attive; la luce interna è accesa; il display visualizza "Pronto".
Operazione	Il forno è in funzione; la luce interna è accesa; il display visualizza il conto alla rovescia del timer. Alla fine della cottura, il cicalino suona per 5 secondi. La luce del forno è accesa. Il display mostra "Fine cottura" mentre il cicalino suona.
Stimolo	Descrizione
Metà potenza	L'utente ha premuto il pulsante di metà potenza.
Massima potenza	L'utente ha premuto il pulsante di massima potenza.
Timer	L'utente ha premuto uno dei pulsanti del timer.
Numero	L'utente ha premuto un tasto numerico.
Porta aperta	L'interruttore della porta non è chiuso.
Porta chiusa	L'interruttore della porta è chiuso.
Avvio	L'utente ha premuto il pulsante Avvio.
Annulla	L'utente ha premuto il pulsante Annulla.

Figura 5.18 Stati e stimoli per il forno a microonde.

sviluppo del loro software. Nel suo blog, den Haan espone le possibili ragioni per cui l'MDE non è stato largamente adottato (den Haan 2011).

5.5 Architettura guidata da modelli

L'architettura guidata da modelli (Mellor, Scott e Weise 2004; Stahl e Voelter 2006) è una tecnica basata sui modelli per progettare e implementare il software che usa un sottoinsieme di modelli UML per descrivere un sistema. I

modelli sono creati a diversi livelli di astrazione. Da un modello di alto livello, indipendente dalla piattaforma, teoricamente è possibile generare un programma funzionante senza interventi manuali.

Il metodo dell’architettura guidata da modelli (MDA, *Model-Driven Architecture*) consiglia di produrre tre tipi di modelli astratti per un sistema.

1. I modelli indipendenti dal calcolo (CIM, *Computation Independent Model*) si basano sulle principali astrazioni dei domini utilizzate in un sistema e, per questo, sono anche detti modelli dei domini. È possibile sviluppare vari CIM, che riflettono diversi modi di vedere il sistema. Per esempio, ci sono CIM di protezione in cui vengono identificate importanti astrazioni della protezione, come una risorsa, un ruolo o il record di un paziente, in cui sono descritte astrazioni come i pazienti e i consulti medici.
2. I modelli indipendenti dalle piattaforme (PIM, *Platform Independent Model*) si basano sul funzionamento del sistema, senza riferimento alla sua implementazione. Un PIM di solito è descritto utilizzando i modelli UML che mostrano la struttura statica di un sistema e come questa risponde agli eventi interni ed esterni.
3. I modelli specifici delle piattaforme (PSM, *Platform Specific Model*) sono trasformazioni dei modelli indipendenti dalle piattaforme con un PSM distinto per ciascuna piattaforma di applicazioni. Teoricamente, possono esserci più strati di PSM, dove ogni strato aggiunge qualche dettaglio specifico per una piattaforma. Così, il PSM di primo livello potrebbe essere specifico del middleware (letteralmente “software di mezzo”), ma indipendente dal database. Quando viene scelto un particolare database, può essere generato un PSM specifico per questo database.

L’ingegneria basata sui modelli consente agli ingegneri di considerare i sistemi con un alto livello di astrazione, senza preoccuparsi dei dettagli della loro implementazione. Questo riduce le probabilità di errore, accelera il processo di progettazione e implementazione, e consente la creazione di modelli di applicazioni riutilizzabili e indipendenti dalle piattaforme. Utilizzando strumenti appropriati, le implementazioni dei sistemi possono essere generati dallo stesso modello per piattaforme differenti. Pertanto, per adattare il sistema alla tecnologia di una nuova piattaforma, basta scrivere un traduttore di modelli per tale piattaforma. Fatto questo, tutti i modelli indipendenti dalla piattaforma possono essere rapidamente adattati alla nuova piattaforma.

Il concetto fondamentale dell’MDA è che le trasformazioni fra modelli possono essere definite e applicate automaticamente da strumenti software, come illustra la Figura 5.19. Questo schema mostra anche un livello finale di trasformazioni automatiche, dove una trasformazione si applica al PSM per generare un codice eseguibile che sarà eseguito sulla piattaforma software designata. Pertanto, almeno in teoria, il software eseguibile può essere generato da un modello di sistema di alto livello.

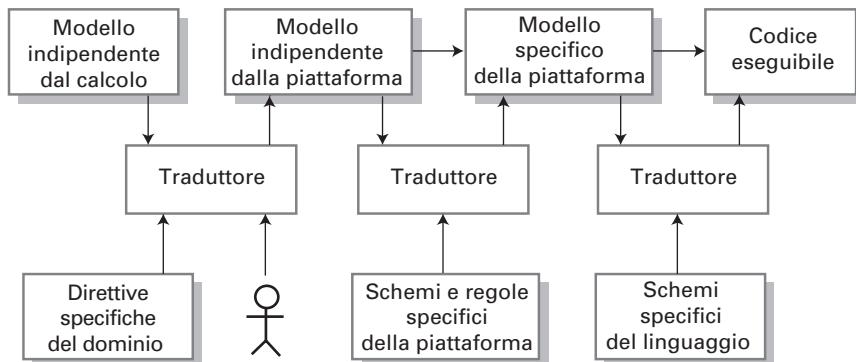


Figura 5.19 Trasformazioni MDA.

In pratica, la traduzione completamente automatica dei modelli in codice eseguibile raramente è possibile. La traduzione di CIM di alto livello in PIM resta un problema della ricerca e, per i sistemi produttivi, di solito è richiesto l'intervento umano, come illustrato dall'uomo stilizzato nella Figura 5.19. Un problema particolarmente difficile delle trasformazioni automatiche dei modelli è l'esigenza di collegare i concetti utilizzati nei differenti CIM. Per esempio, il concetto di ruolo in un CIM di protezione, che include il controllo degli accessi guidato da ruoli, può richiedere di essere mappato nel concetto di membro del personale nel CIM di un ospedale. Soltanto una persona che s'intende di protezione e conosce l'ambiente di un ospedale può effettuare questa mappatura.

La traduzione di un modello indipendente dalla piattaforma a un modello specifico della piattaforma è un problema tecnico più semplice. Sono disponibili strumenti commerciali e open-source (Koegel 2012) per tradurre i PIM in modelli per le comuni piattaforme, come Java e J2EE. Usano una libreria estesa di regole e schemi specifici delle piattaforme per convertire un PIM in un PSM. Ci possono essere più PSM per ciascun PIM nel sistema. Se un sistema software è stato ideato per essere eseguito su piattaforme differenti (per esempio, J2EE e .NET), allora in teoria basta mantenere un solo PIM. I PSM per ciascuna piattaforma vengono generati automaticamente (Figura 5.20).

Sebbene gli strumenti che supportano l'MDA includano traduttori specifici per piattaforme, questi a volte offrono un supporto soltanto parziale per tradurre i PIM in PSM. L'ambiente di esecuzione per un sistema è più della piattaforma standard di esecuzione, come J2EE o Java; esso include anche altri sistemi applicativi, librerie specifiche per applicazioni che possono essere create per società, servizi esterni e librerie di interfaccia utente.

Questi elementi variano da una società all'altra, quindi non è possibile disporre di strumenti pronti all'uso che li tengano in considerazione. Pertanto, quando l'MDA viene introdotta in una organizzazione, potrebbe essere necessario creare

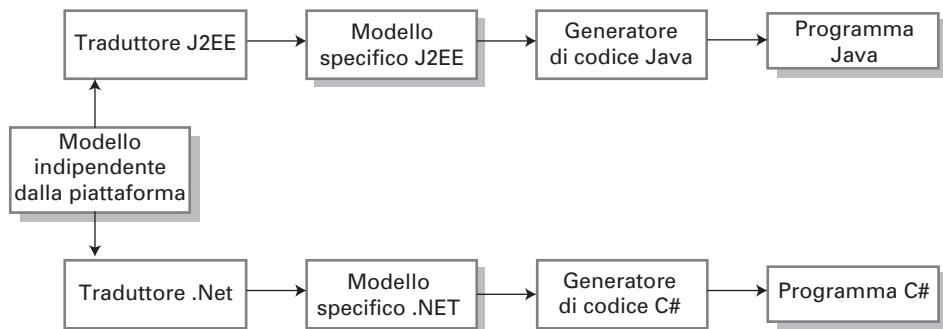


Figura 5.20 Modelli specifici per due piattaforme.

dei traduttori speciali per poter utilizzare le funzionalità disponibili nell’ambiente locale. Questo è uno dei motivi per cui molte società sono state riluttanti ad adottare approcci guidati da modelli nello sviluppo del software. Non vogliono sviluppare o mantenere i loro propri strumenti o affidarsi a piccole società di software, che potrebbero fallire, per sviluppare gli strumenti. Senza questi strumenti specialistici, lo sviluppo basato su modelli richiede una codifica manuale aggiuntiva, che riduce la convenienza economica di questo approccio.

I credo che ci siano molte altre ragioni per cui l’MDA non è diventata la tecnica dominante nello sviluppo del software.

1. I modelli sono un buon modo per facilitare le discussioni sul progetto di un software. Tuttavia, non è sempre vero che le astrazioni utili per le discussioni siano quelle appropriate per l’implementazione. Si potrebbe decidere di utilizzare un approccio all’implementazione completamente differente che si basa sul riutilizzo di sistemi applicativi pronti all’uso.
2. Per molti sistemi complessi, l’implementazione non è il problema principale – l’ingegneria dei requisiti, la protezione e la fidatezza, l’integrazione con i sistemi tradizionali e i test sono tutti più importanti. Di conseguenza, i vantaggi derivanti dall’MDA sono limitati.
3. Gli argomenti a favore dell’indipendenza dalle piattaforme sono validi soltanto per grandi sistemi di lunga durata, dove le piattaforme diventano obsolete durante la vita dei sistemi. Per i prodotti software e i sistemi informativi che sono sviluppati per piattaforme standard, come Windows e Linux, i risparmi derivanti dall’uso dell’MDA probabilmente sono vanificati dai costi della sua introduzione e dei suoi strumenti.
4. L’ampia diffusione dei metodi agili che è avvenuta contemporaneamente all’evoluzione dell’MDA ha distorto l’attenzione dagli approcci guidati dai modelli.

UML esegibile

Il concetto fondamentale che sta alla base dell'ingegneria guidata da modelli è che dovrebbe essere possibile trasformare un modello in codice in modo completamente automatico. Per fare questo, bisogna saper costruire modelli grafici con elementi chiaramente definiti che possono essere compilati in codice eseguibile. Occorre anche un metodo per aggiungere ai modelli grafici informazioni sui modi in cui le operazioni definite nel modello devono essere implementate. Questo è possibile utilizzando un sottoinsieme di UML 2, detto Executable UML o xUML (Mellor e Balcher 2002).

<http://software-engineering-book.com/web/xuml/>

Le storie di successo per l'MDA (OMG 2012) provengono principalmente da società che sviluppano prodotti per sistemi che includono sia il software sia l'hardware. Il software in questi prodotti ha una vita lunga e può essere modificato per rispecchiare i progressi delle tecnologie hardware. Il dominio dell'applicazione (settore auto, controllo del traffico aereo ecc.) spesso è ben conosciuto e quindi può essere formalizzato in un CIM.

Hutchinson e i suoi colleghi (Hutchinson, Rouncefield e Whittle 2012) hanno studiato l'uso industriale dell'MDA; il loro lavoro conferma che i successi nell'uso dello sviluppo guidato da modelli si sono avuti nei prodotti per sistemi. La loro analisi suggerisce che le società hanno avuto risultati contrastanti adottando questo approccio, ma la maggior parte degli utenti hanno riferito che l'uso dell'MDA ha incrementato la produttività e ridotto i costi di manutenzione. Hanno scoperto che l'MDA è stata particolarmente utile nel facilitare il riutilizzo del software, e questo ha permesso di conseguire significativi miglioramenti della produttività.

C'è una relazione difficile tra metodi agili e architettura guidata da modelli. Il concetto di modellazione up-front¹ contraddice le idee fondamentali del manifesto dello sviluppo agile e credo che pochi sviluppatori agili si trovino a loro agio con l'ingegneria guidata da modelli. Ambler, un pioniere nello sviluppo dei metodi agili, ritiene che alcuni aspetti dell'MDA possano essere utilizzati nei processi agili (Ambler 2004), ma considera impraticabile la generazione automatica del codice. Tuttavia, Zhang e Patel riferiscono di un successo di Motorola nell'uso dello sviluppo agile con generazione automatica del codice (Zhang e Patel 2011).

Punti chiave

- Un modello è una visione astratta di un sistema, che ignora deliberatamente alcuni dettagli del sistema. Per mostrare il contesto, le interazioni, la struttura e il comportamento di un sistema, possono essere sviluppati dei modelli complementari del sistema.

¹ Con il termine *up-front* s'intende una metodologia che pone la soluzione davanti al problema, e non viceversa.

- I modelli contestuali mostrano come il sistema che si sta modellando si posiziona in un ambiente con altri sistemi e processi; agevolano la definizione dei vincoli del sistema da sviluppare.
- I diagrammi dei casi d'uso e i diagrammi di sequenza sono utilizzati per descrivere le interazioni tra gli utenti e il sistema che si sta sviluppando. I casi d'uso descrivono le interazioni tra un sistema e gli attori esterni; i diagrammi di sequenza aggiungono altre informazioni, mostrando le interazioni tra gli oggetti del sistema.
- I modelli strutturali mostrano l'organizzazione e l'architettura di un sistema. I diagrammi di classe sono utilizzati per definire la struttura statica delle classi di un sistema e le loro associazioni.
- I modelli comportamentali sono utilizzati per descrivere il comportamento dinamico di un sistema in esecuzione. Questo comportamento può essere modellato dalla prospettiva dei dati elaborati dal sistema o dagli eventi che simulano le risposte di un sistema.
- I diagrammi di attività possono essere utilizzati per modellare l'elaborazione dei dati, dove ogni attività rappresenta un passo del processo.
- I diagrammi di stato sono utilizzati per modellare il comportamento di un sistema in risposta a eventi interni o esterni.
- L'ingegneria guidata da modelli è un approccio allo sviluppo del software in cui un sistema è rappresentato come un insieme di modelli che possono essere automaticamente trasformati in codice eseguibile.

Esercizi

- 5.1 Spiegate perché è importante modellare il contesto del sistema che sta per essere sviluppato. Fate due esempi di possibili errori che potrebbero verificarsi se gli ingegneri del software non capissero il contesto del sistema.
- * 5.2 Come usereste un modello di un sistema che esiste già? Spiegate perché non è sempre necessario che tale modello sia completo e corretto. Ciò sarebbe ancora valido se dovreste sviluppare un modello per un nuovo sistema?
- 5.3 Vi è stato chiesto di sviluppare un sistema che aiuterà a pianificare eventi e ricevimenti su larga scala, come matrimoni, lauree e feste di compleanno. Utilizzando un diagramma di attività, modellate il contesto del processo per tale sistema in modo da mostrare le attività che riguardano la pianificazione di un ricevimento (prenotazione del luogo, preparazione degli inviti ecc.) e gli elementi del sistema che potrebbero essere utilizzati in ciascuna fase.
- 5.4 Con riferimento al sistema Mentcare, proponete una serie di casi d'uso che illustrano le interazioni tra un dottore, che visita i pazienti e prescrive le cure e i farmaci, e il sistema Mentcare.
- * 5.5 Sviluppatte un diagramma di sequenza che mostra le interazioni che si svolgono quando uno studente si registra a un corso di laurea universitaria. I corsi di laurea possono avere un numero limitato di iscrizioni, quindi il processo di iscrizione deve includere il controllo che ci siano ancora posti disponibili. Supponete che lo studente possa accedere a un catalogo elettronico per trovare i corsi di laurea disponibili.

- * 5.6 Osservate attentamente come sono rappresentati i messaggi e le caselle di posta elettronica nel sistema e-mail che utilizzate. Modellate le classi degli oggetti che possono essere utilizzati nell'implementazione del sistema per rappresentare un messaggio e una casella di posta elettronica.
 - * 5.7 Sulla base della vostra esperienza, disegnate un diagramma di attività che modella l'elaborazione dei dati quando un cliente ritira del contante da uno sportello Bancomat.
 - 5.8 Disegnate un diagramma di sequenza per lo stesso sistema. Spiegate perché potrebbe essere necessario sviluppare sia un diagramma di attività sia un diagramma di sequenza per modellare il comportamento di un sistema.
 - 5.9 Disegnate i diagrammi di stato del software di controllo per:
 - una lavatrice automatica con programmi diversi per tipi di vestiti differenti;
 - un lettore DVD;
 - la fotocamera del vostro telefono cellulare; ignorate il flash, se presente nel vostro cellulare.
 - * 5.10 Supponete di essere il responsabile di un team di ingegneria del software; un membro più anziano del team vi propone di applicare l'ingegneria guidata da modelli per sviluppare un nuovo sistema. Quali fattori dovreste prendere in considerazione per decidere se adottare o no questo approccio per lo sviluppo del software?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Qualsiasi libro introduttivo al linguaggio UML fornisce più informazioni sulla notazione UML di quelle che io possa presentare in questo libro. UML ha subito piccole modifiche negli ultimi anni, quindi, sebbene alcuni di questi libri siano vecchi di quasi 10 anni, essi sono ancora validi.

Using UML: Software Engineering with Objects and Components, 2nd ed. Questo libro è una breve e chiara introduzione all'uso dell'UML per specificare e progettare i sistemi. Credo che sia uno strumento eccellente per capire e imparare la notazione UML, sebbene sia meno completo delle descrizioni fornite nel manuale di riferimento dell'UML (P. Stevens con R. Pooley, Addison-Wesley, 2006).

Model-driven Software Engineering in Practice. È un libro completo sui metodi guidati dai modelli, con particolare attenzione alla progettazione e all'implementazione guidate dai modelli. Oltre al linguaggio UML, tratta anche lo sviluppo di linguaggi di modellazione specifica per i domini (M. Brambilla, J. Cabot e M. Wimmer. Morgan Claypool, 2012).

Progettazione architetturale

L'obiettivo di questo capitolo è presentare i concetti di architettura del software e progettazione architetturale. Dopo aver letto questo capitolo:

- capirete perché è importante la progettazione architettonica del software;
- capirete le decisioni che devono essere prese sull'architettura del software durante il processo di progettazione architettonica;
- conoscerete gli schemi architettonici – modi collaudati di organizzare le architetture software che possono essere riutilizzate nella progettazione dei sistemi;
- capirete come gli schemi architettonici specifici di un'applicazione possono essere utilizzati nei sistemi di elaborazione delle transazioni e dei linguaggi.

- 6.1 Decisioni di progettazione architettonica
- 6.2 Viste architettoniche
- 6.3 Schemi architettonici
- 6.4 Architetture applicative

La progettazione architetturale si occupa dell’organizzazione di un sistema software e della progettazione della sua struttura complessiva. Nel modello del processo di sviluppo del software descritto nel Capitolo 2, la progettazione architetturale è il primo stadio del processo di progettazione del software. È il collegamento critico tra progettazione e ingegneria dei requisiti, in quanto identifica i principali componenti strutturali di un sistema e le loro relazioni. L’output del processo di progettazione architetturale è un modello architetturale che descrive come il sistema è organizzato in funzione dei componenti di comunicazione.

Nei processi agili è generalmente accettato che una fase iniziale del processo di sviluppo agile sia dedicata alla progettazione dell’architettura complessiva del sistema. La rifattorizzazione dei componenti in risposta ai cambiamenti è un compito relativamente facile. Tuttavia, la rifattorizzazione dell’architettura del sistema è costosa, in quanto potrebbe richiedere la modifica di molti componenti del sistema per adattarli ai cambiamenti architettonici.

Per aiutarvi a capire che cosa intendo con architettura del sistema, esamineate la Figura 6.1. Il diagramma mostra un modello astratto dell’architettura di un sistema di imballaggio robotizzato. Questo sistema può impacchettare diversi tipi di oggetti. Usa un componente visivo per raccogliere gli oggetti da un nastro trasportatore, identificare il tipo di oggetto e selezionare l’imballaggio appropriato. Il sistema poi sposta gli oggetti dal nastro di consegna per imballarli e pone gli oggetti imballati su un altro nastro. Il modello architettonico mostra questi componenti e i loro collegamenti.

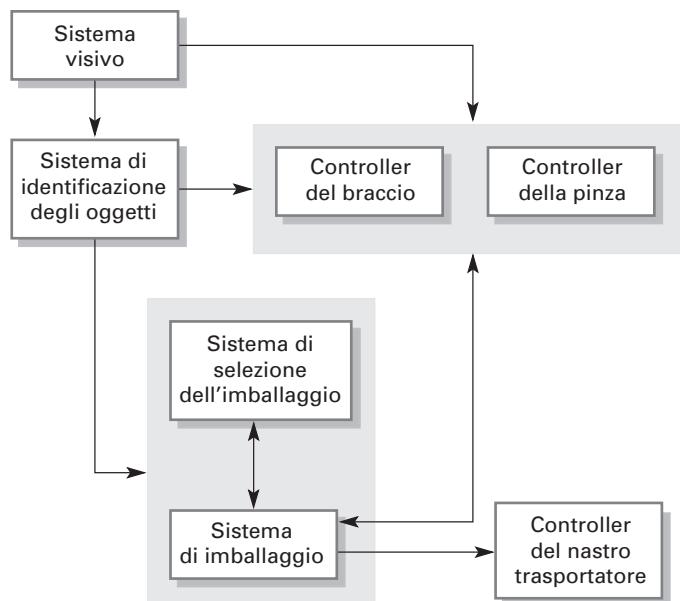


Figura 6.1 Architettura di un sistema di imballaggio robotizzato.

In pratica, c’è una significativa sovrapposizione tra i processi di ingegneria dei requisiti e la progettazione architetturale. In teoria, una specifica del sistema non dovrebbe includere alcuna informazione sulla progettazione; questo è irrealizzabile, tranne per i sistemi molto piccoli. Occorre identificare i componenti architettonici principali, perché questi riflettono le caratteristiche di alto livello del sistema. Pertanto, come parte del processo di ingegneria dei requisiti, si potrebbe proporre un’architettura astratta del sistema, dove vengono associati gruppi di funzioni o caratteristiche del sistema con componenti su larga scala o sottosistemi; poi si potrebbe usare questa scomposizione per discutere con gli stakeholder i requisiti e altre caratteristiche più dettagliate del sistema.

È possibile progettare l’architettura del software a due livelli di astrazione, che io ho chiamato *architettura in piccolo* e *architettura in grande*.

1. *Architettura in piccolo* – riguarda l’architettura di singoli programmi. A questo livello, ci interessa il modo in cui un singolo programma viene scomposto nei suoi componenti. Questo capitolo riguarda principalmente l’architettura di singoli programmi.
2. *Architettura in grande* – riguarda l’architettura di sistemi aziendali complessi che includono altri sistemi, programmi e componenti di programmi. Questi sistemi aziendali possono essere distribuiti su differenti computer, che possono essere di proprietà o gestiti da differenti società. (L’architettura in grande è trattata nei Capitoli 17 e 18.)

L’architettura del software è importante perché influisce sulle prestazioni, robustezza, distribuzione e manutenzione di un sistema (Bosch 2000). Come spiega Bosch, i singoli componenti implementano i requisiti funzionali di un sistema, ma è l’architettura che ha un’influenza predominante sulle caratteristiche non funzionali di un sistema. Chen e altri (Chen, Ali Babar e Nuseibeh 2013) hanno confermato questo in uno studio sui “requisiti architettonicamente significativi”, dal quale è emerso che i requisiti non funzionali hanno l’effetto più rilevante sull’architettura del sistema.

Bass e altri (Bass et al. 2003) indicano tre vantaggi della progettazione e documentazione esplicita dell’architettura del software.

1. *Comunicazione tra gli stakeholder*: l’architettura è una presentazione di alto livello del sistema che può essere utilizzata per concentrare i temi di discussione fra diversi stakeholder.
2. *Analisi del sistema*: rendere esplicita l’architettura di un sistema, in un primo stadio di sviluppo, richiede alcune analisi. Le decisioni di progettazione architettonica influenzano profondamente la conformità del sistema ai requisiti critici come prestazioni, affidabilità e mantenibilità.
3. *Riutilizzo su vasta scala*: un modello architettonico di un sistema descrive, in modo compatto e gestibile, come è organizzato un sistema e come integrano i suoi componenti. L’architettura del sistema è spesso la stessa

per sistemi con requisiti simili e, quindi, può supportare il riutilizzo del software su vasta scala. Come è spiegato nel Capitolo 15, le architetture di linee di prodotti sono un approccio al riutilizzo, dove la stessa architettura viene riutilizzata su un’ampia gamma di sistemi correlati.

Le architetture dei sistemi sono spesso modellate in modo informale utilizzando semplici diagrammi a blocchi, come nella Figura 6.1. Ciascun box nel diagramma rappresenta un componente. I box all’interno di un altro box indicano che il componente è stato scomposto in sottocomponenti. Le frecce indicano che i dati o i segnali di controllo passano da un componente all’altro nella direzione delle frecce. Il manuale di Booch sull’architettura del software (Booch 2014) descrive molti esempi di questo tipo di modello architetturale.

I diagrammi a blocchi offrono una vista di alto livello della struttura del sistema, che possono essere facilmente compresi da persone di varie discipline scolastiche che sono interessate al processo di sviluppo del sistema. Nonostante il loro largo impiego, Bass e altri (Bass, Clements e Kazman 2012) non apprezzano i diagrammi a blocchi informali per descrivere un’architettura, sostenendo che questi diagrammi siano rappresentazioni architettoniche scadenti, in quanto non mostrano né il tipo di relazione tra i componenti del sistema né le proprietà esternamente visibili dei componenti.

Le apparenti contraddizioni tra la teoria architettonica e la pratica industriale derivano dal fatto che ci sono due modi di utilizzare il modello architettonico di un programma.

1. *Come un modo di incoraggiare le discussioni sulla progettazione del sistema.* Una vista architettonica di alto livello di un sistema è utile alla comunicazione con gli stakeholder del sistema e alla pianificazione del progetto, in quanto non è confusa dai dettagli. Gli stakeholder possono farvi riferimento e avere una visione astratta del sistema. Il modello architettonico identifica i componenti chiave che devono essere sviluppati, in modo che i manager possano iniziare ad assegnare alle persone la pianificazione dello sviluppo di questi sistemi.
2. *Come un modo di documentare un’architettura che è stata progettata.* L’obiettivo qui è produrre un modello completo del sistema che mostra i vari componenti del sistema, le loro interfacce e i loro collegamenti. La giustificazione di tale modello è che una descrizione architettonica dettagliata semplifica la conoscenza e lo sviluppo del sistema.

I diagrammi a blocchi sono un buon modo di supportare la comunicazione tra le persone coinvolte nel processo di progettazione del software. Essendo tali diagrammi intuitivi, gli esperti dei domini e gli ingegneri del software possono utilizzarli come riferimenti nelle discussioni sul sistema. I manager li trovano utili per pianificare i progetti. Per molti progetti, i diagrammi a blocchi sono l’unica descrizione architettonica.

Teoricamente, se l’architettura di un sistema deve essere documentata nei dettagli, è meglio utilizzare una notazione più rigorosa per la descrizione architetturale. Per questo sono stati sviluppati vari linguaggi di descrizione architetturale (Bass, Clements e Kazman 2012). Una descrizione più dettagliata e completa significa che si riduce il rischio di fraintendere le relazioni tra i componenti architettonici. Tuttavia, lo sviluppo di una descrizione architetturale dettagliata è un processo lungo e costoso. Questo approccio non è largamente adottato perché è praticamente impossibile sapere se esso sia economicamente conveniente.

6.1 Decisioni di progettazione architetturale

La progettazione architetturale è un processo creativo in cui si progetta un’organizzazione che soddisfa i requisiti funzionali e non funzionali di un sistema. Non esiste un processo di progettazione architetturale sperimentato. Il processo dipende dal tipo di sistema che si sta sviluppando, dal background e dall’esperienza dell’architetto e dai requisiti specifici del sistema. Di conseguenza, io credo che sia più utile considerare il processo di progettazione architetturale come una serie di decisioni da prendere, anziché come una serie di attività.

Durante il processo di progettazione architetturale gli architetti del sistema devono prendere una serie di decisioni fondamentali che influiscono profondamente sul sistema e sul suo processo di sviluppo. Basandosi sulle proprie conoscenze ed esperienze, essi devono rispondere ad alcune domande fondamentali, riportate nella Figura 6.2.

Per quanto ogni sistema software sia unico, i sistemi nello stesso dominio di applicazione hanno spesso architetture simili che ne riflettono i concetti fondamentali. Per esempio, le linee dei prodotti applicativi sono costruite attorno a un’architettura base con varianti che soddisfano specifiche richieste dei clienti. Quando si progetta l’architettura di un sistema si deve decidere cosa hanno in comune il sistema e le classi di applicazioni più generali e stabilire quanto può essere riutilizzato da queste architetture applicative.

Per i sistemi integrati e le applicazioni per PC e unità mobili, non si deve progettare un’architettura distribuita per il sistema. Tuttavia, molti grandi sistemi sono sistemi distribuiti, dove il software è distribuito su molti computer differenti. La scelta di un’architettura distribuita è una decisione chiave che influenza le prestazioni e l’affidabilità del sistema. Questo è un argomento importante che sarà trattato nel Capitolo 17.

L’architettura di un sistema software può basarsi su un particolare *schema* o *stile architettonico* (questi termini hanno assunto lo stesso significato). Uno schema architettonico è una descrizione dell’organizzazione del sistema (Garlan e Shaw 1993), per esempio un’organizzazione client-server o un’architettura a strati. Gli schemi architettonici esprimono l’essenza dell’architettura che è stata utilizzata in diversi sistemi software. Quando si prendono decisioni sull’architet-

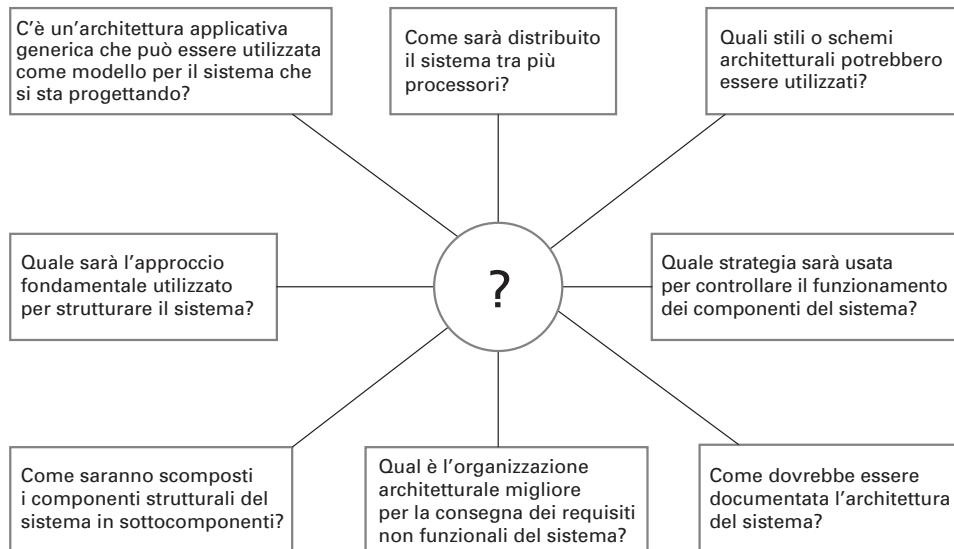


Figura 6.2 Decisioni di progettazione architetturale.

tura di un sistema, è importante conoscere gli schemi architetturali più comuni, le loro applicazioni, i loro punti di forza e le loro debolezze. Il Paragrafo 6.3 descrive alcuni schemi architetturali più utilizzati.

La nozione di schema architettonico di Garlan e Shaw riguarda le domande 4, 5 e 6 elencate nella Figura 6.2. Si deve scegliere la struttura più appropriata, per esempio una struttura client-server o a strati, che permetta di soddisfare i requisiti del sistema; per scomporre le unità strutturali del sistema, bisogna scegliere la strategia di scomposizione dei componenti in sottocomponenti; infine, nel processo di modellazione del controllo, occorre sviluppare un modello generale delle relazioni di controllo tra le varie parti del sistema e decidere come controllare l'esecuzione dei componenti.

A causa della stretta relazione tra le caratteristiche non funzionali del sistema e l'architettura del software, la scelta dello schema architettonico e della struttura dipende dai requisiti non funzionali del sistema.

1. *Prestazioni*: se le prestazioni sono un requisito critico, l'architettura dovrebbe essere progettata per localizzare le operazioni critiche all'interno di un piccolo numero di componenti, con questi componenti installati sullo stesso computer, anziché distribuiti nella rete. Questo potrebbe richiedere l'uso di componenti relativamente grandi, al posto di moduli più piccoli. L'uso di componenti grandi riduce il numero di comunicazioni tra i componenti, in quanto la maggior parte delle interazioni tra le funzionalità correlate del sistema avviene all'interno di un componente. Dovreste anche tenere in considerazione le organizzazioni del sistema a runtime, che consentono al sistema di essere replicato ed eseguito su processori differenti.

2. *Protezione*: se la protezione è un requisito critico, dovrebbe essere utilizzata un’architettura strutturata a strati, con le risorse più critiche protette nello strato più interno, e con un alto livello di convalida della protezione a ogni livello.
3. *Sicurezza*: se la sicurezza è un requisito critico, l’architettura dovrebbe essere progettata in modo che le operazioni relative alla sicurezza siano tutte collocate in un singolo componente o in un piccolo numero di componenti; in questo modo, si riducono i costi e i problemi di convalida della sicurezza ed è possibile fornire i relativi sistemi di protezione che possono bloccare con sicurezza il sistema in caso di guasto.
4. *Disponibilità*: se la disponibilità è un requisito critico, l’architettura dovrebbe essere progettata per includere i componenti ridondanti, in modo che sia possibile sostituirli e aggiornarli senza fermare il sistema. Le architetture resistenti ai guasti, per i sistemi ad alta disponibilità, sono trattate nel Capitolo 11.
5. *Mantenibilità*: se la mantenibilità è un requisito critico, l’architettura del sistema dovrebbe essere progettata utilizzando componenti piccoli e autonomi, che possono essere modificati velocemente. I produttori di dati dovrebbero essere separati dai consumatori, e le strutture dati condivise dovrebbero essere evitate.

Ovviamente ci sono dei potenziali conflitti tra alcune di queste architetture. Per esempio, l’uso di grandi componenti migliora le prestazioni, mentre l’uso di piccoli componenti migliora la mantenibilità. Se questi requisiti del sistema sono entrambi importanti, si deve trovare un compromesso, che a volte può essere raggiunto usando diversi schemi architettonurali per parti differenti del sistema. La sicurezza è adesso quasi sempre un requisito critico, e quindi occorre progettare un’architettura che garantisce la sicurezza, soddisfacendo anche altri requisiti non funzionali.

Valutare un progetto architettonurale è difficile perché il vero test di un’architettura è controllare in che modo il sistema soddisfa i suoi requisiti funzionali e non funzionali una volta che viene utilizzato; tuttavia, in molti casi si può fare una valutazione confrontando il proprio progetto con schemi architettonurali di riferimento o generici. La descrizione di Bosch (Bosch 2000) delle caratteristiche non funzionali di alcuni schemi architettonurali può essere di aiuto nel valutare un progetto architettonurale.

6.2 Viste architettonurali

Nell’introduzione di questo capitolo ho spiegato che i modelli architettonurali di un sistema software possono essere utilizzati per focalizzare la discussione sui requisiti o sulla progettazione del software; in alternativa, possono essere utilizzati per

documentare un progetto in modo che questo possa servire da base per una progettazione e implementazione più dettagliata del sistema. In questo paragrafo tratterò due argomenti importanti per entrambi questi processi.

1. Quali viste o prospettive sono utili quando si progetta e si documenta l’architettura di un sistema?
2. Quali notazioni dovrebbero essere utilizzate per descrivere i modelli architettonici?

È impossibile rappresentare tutte le informazioni relative all’architettura di un sistema in un singolo diagramma, in quanto un modello grafico può mostrare soltanto una vista o prospettiva del sistema. Occorre mostrare come un sistema viene scomposto in moduli, come interagiscono i processi a runtime, o in quali modi i componenti del sistema sono distribuiti in una rete. Poiché tutti questi elementi sono utili in momenti differenti della progettazione e della documentazione, di solito bisogna fornire più viste dell’architettura del software.

Ci sono differenti opinioni su quali viste siano richieste. Krutchen (Krutchen 1995), nel suo famoso modello a 4+1 viste dell’architettura del software, suggerisce quattro viste architettoniche fondamentali, che possono essere collegate tramite scenari o casi d’uso comuni (Figura 6.3).

1. *Vista logica* – mostra le astrazioni chiave nel sistema come oggetti o classi di oggetti. In questa vista dovrebbe essere possibile mettere in relazione i requisiti del sistema con le entità.
2. *Vista dei processi* – mostra come, a runtime, il sistema è composto da processi interattivi. Questa vista è utile per valutare le caratteristiche non funzionali del sistema, come le prestazioni e la disponibilità.
3. *Vista di sviluppo* – mostra come il software viene scomposto per lo sviluppo; ovvero mostra la suddivisione del software nei suoi componenti che sono implementati da un singolo sviluppatore o da un team di sviluppatori. Questa vista è utile per i programmati e i manager del software.
4. *Vista fisica* – mostra l’hardware del sistema e come i componenti del software sono distribuiti tra i processori nel sistema. Questa vista è utile per gli ingegneri che pianificano l’installazione dei sistemi.

Hofmeister e altri (Hofmeister, Nord e Soni 2000) consigliano l’uso di simili viste, con l’aggiunta di una vista concettuale. Si tratta di una vista astratta del sistema che può essere la base per scomporre i requisiti di alto livello in specifiche più dettagliate; aiuta gli ingegneri a prendere decisioni sui componenti che possono essere riutilizzati e consente di rappresentare una linea di prodotti (descritta nel Capitolo 15), anziché un singolo sistema. La Figura 6.1, che descrive l’architettura di un sistema di imballaggio robotizzato, è un esempio di vista concettuale del sistema.

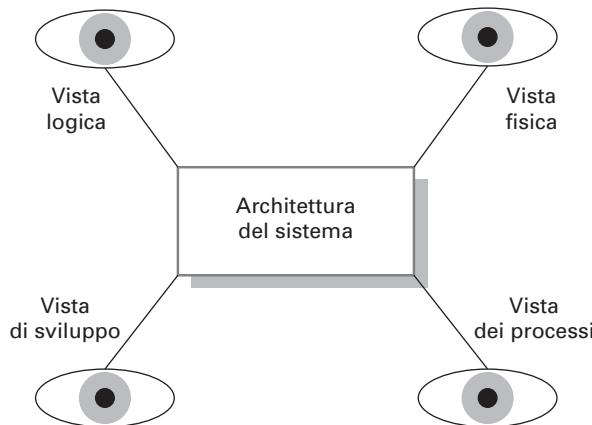


Figura 6.3 Viste architettoniche.

In pratica, le viste concettuali dell’architettura di un sistema sono quasi sempre sviluppate durante la fase di progettazione. Sono utilizzate per spiegare l’architettura del sistema agli stakeholder e per rendere note le decisioni di progettazione architettonica. Durante la progettazione, possono essere sviluppate anche alcune delle altre viste, quando si discutono altri aspetti del sistema, ma raramente è necessario avere una descrizione completa con tutte le altre viste. È anche essere associare gli schemi architettonici, descritti nel prossimo paragrafo, alle differenti viste del sistema.

Ci sono opinioni discordanti sul fatto che gli architetti del software debbano usare o no il linguaggio UML per descrivere e documentare le architetture del software. Uno studio del 2006 (Lange, Chaudron e Muskens 2006) ha dimostrato che, quando si usa l’UML, questo viene applicato principalmente in modo informale. Gli autori di questo studio non hanno apprezzato tale impiego dell’UML.

Io non sono d’accordo. L’UML fu ideato per descrivere i sistemi orientati agli oggetti e, nella fase di progettazione architettonica, spesso occorre descrivere i sistemi a un alto livello di astrazione. Le classi di oggetti sono troppo vicine all’implementazione per essere utili alla descrizione architettonica. Non credo che l’UML sia utile durante la progettazione e preferisco le notazioni informali che sono più rapide da scrivere e possono essere facilmente tracciate su una lavagna bianca. L’UML è utile soprattutto quando si documenta un’architettura dettagliatamente o quando si sceglie un processo di sviluppo guidato da modelli, come descritto nel Capitolo 5.

Diversi ricercatori (Bass, Clements e Kazman 2012) hanno proposto l’uso di linguaggi di descrizione architettonica (ADL, *Architectural Description Languages*) più specializzati per descrivere le architetture dei sistemi. Gli elementi base di un ADL sono i componenti e i connettori, e includono regole e linee guida per

architetture affermate. Tuttavia, poiché gli ADL sono linguaggi specializzati, sono difficili da capire e usare dagli specialisti dei domini e delle applicazioni. Gli ADL potrebbero essere di qualche utilità se diventassero parte di un processo di sviluppo guidato da modelli, ma non credo che faranno parte delle pratiche tradizionali di ingegneria del software. I modelli e le notazioni informali, come l’UML, resteranno i metodi più comuni per documentare le architetture dei sistemi.

Gli utenti dei metodi agili ritengono che una documentazione dettagliata del progetto è ormai quasi in disuso. Di conseguenza, è una perdita di tempo e di denaro sviluppare questi documenti. Io concordo ampiamente con questo e credo che, tranne per i sistemi critici, non vale la pena sviluppare descrizioni architettoniche dettagliate secondo le quattro viste di Krutchen. Si dovrebbero sviluppare le viste che sono utili alla comunicazione, senza preoccuparsi del fatto che la documentazione architettonica sia o no completa.

6.3 Schemi architetturali

L’idea degli schemi come modi di presentare, condividere e riutilizzare le conoscenze dei sistemi software è stata adottata in un certo numero di aree di ingegneria del software. Questo avvenne in seguito alla pubblicazione di un libro sugli schemi di progettazione orientata agli oggetti (Gamma et al. 1995). Ciò indusse a sviluppare altri tipi di schemi, come gli schemi per la progettazione organizzativa (Coplien e Harrison 2004), gli schemi di usabilità (Usability Group 1998), gli schemi di interazione cooperativa (Martin e Sommerville 2004) e gli schemi di gestione delle configurazioni (Berczuk e Appleton 2002).

Gli schemi architetturali furono proposti negli anni ’90 con il nome di “stili architetturali” (Shaw e Garlan 1996). Tra il 1996 e il 2007 furono pubblicati cinque manuali molto dettagliati sull’architettura del software orientata agli schemi (Buschmann et al. 1996; Schmidt et al. 2000; Kircher e Jain 2004; Buschmann, Henney e Schmidt 2007a, 2007b).

In questo paragrafo presenterò gli schemi architetturali e descriverò brevemente alcuni di quelli più utilizzati. Gli schemi possono essere descritti in un modo standard (Figure 6.4 e 6.5) utilizzando un mix di testi e diagrammi. Per informazioni più dettagliate sugli schemi e il loro uso, dovreste fare riferimento ai manuali pubblicati.

Uno schema architettonico può essere immaginato come una descrizione stilizzata di una buona pratica, che è stata provata e verificata in vari sistemi e ambienti. Pertanto, uno schema architettonico dovrebbe descrivere l’organizzazione di un sistema che ha avuto successo in altri sistemi preesistenti; dovrebbe includere le informazioni su quando il suo utilizzo è appropriato e i dettagli sui suoi vantaggi e svantaggi.

Nome	MVC (Model-View-Controller)
Descrizione	Presentazione e interazione separate dai dati del sistema. Il sistema è strutturato in tre componenti logiche che interagiscono tra loro. Il componente Model gestisce i dati del sistema e le operazioni associate. Il componente View definisce e gestisce il modo in cui i dati sono presentati all'utente. Il componente Controller gestisce l'interazione degli utenti (tasti premuti, clic del mouse ecc.) e passa queste interazioni ai componenti Model e View. Si veda la Figura 6.5.
Esempio	La Figura 6.6 mostra l'architettura di un sistema di applicazioni basate sul Web organizzate secondo lo schema MVC.
Quando si usa	Si usa quando ci sono più modi di visualizzare e interagire con i dati; si usa anche quando non sono noti i requisiti futuri di interazione e presentazione dei dati.
Vantaggi	Consente ai dati di cambiare indipendentemente dalla loro rappresentazione e viceversa. Supporta la presentazione degli stessi dati in modi differenti, con le modifiche fatte in una rappresentazione mostrate in tutte le altre.
Svantaggi	Potrebbe richiedere altro codice o un codice complesso quando il modello dei dati e le interazioni sono semplici.

Figura 6.4 Lo schema MVC (Model-View-Controller).

La Figura 6.4 descrive il noto schema Model-View-Controller (MVC), che è la base per la gestione delle interazioni in molti sistemi web ed è supportato da molti linguaggi. La descrizione stilizzata dello schema include il nome dello schema, una breve descrizione, un modello grafico e un esempio del tipo di sistema dove lo schema è usato. Dovrebbero essere incluse anche le informazioni su quando lo schema dovrebbe essere usato e i suoi vantaggi e svantaggi.

I modelli grafici dell'architettura associata allo schema MVC sono illustrati nelle Figure 6.5 e 6.6. Queste figure presentano l'architettura da differenti viste: la Figura 6.5 è una vista concettuale; la Figura 6.6 mostra l'architettura di un sistema a runtime quando questo schema è usato per la gestione delle interazioni in un sistema basato sul Web.

In questo breve spazio è impossibile descrivere tutti gli schemi generici che possono essere utilizzati nello sviluppo del software; pertanto, mi limiterò a presentare alcuni esempi significativi di schemi che sono largamente utilizzati e che incorporano i buoni principi della progettazione architetturale.

6.3.1 Architettura a strati

I concetti di separazione e indipendenza sono fondamentali nella progettazione architetturale, in quanto consentono di localizzare le modifiche. Lo schema MVC, illustrato nella Figura 6.4, separa gli elementi di un sistema, consentendo loro di

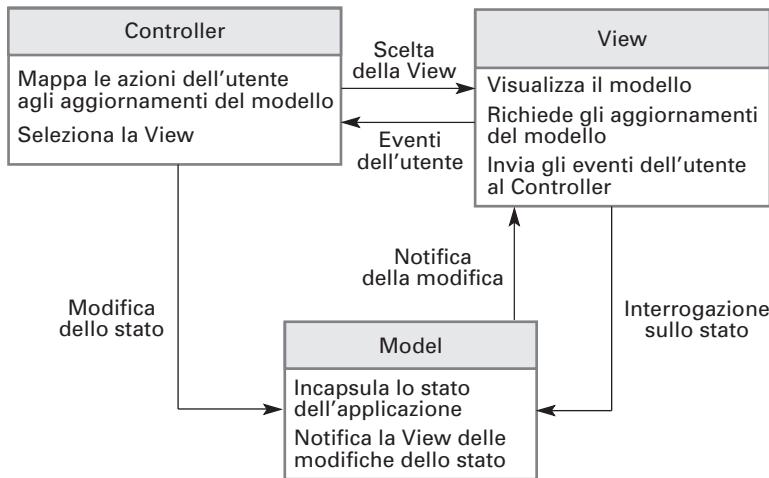


Figura 6.5 Organizzazione dello schema Model-View-Controller.

cambiare in maniera indipendente. Per esempio, aggiungere una nuova view o modificarne una esistente può essere fatto senza cambiare i dati sottostanti del modello. Lo schema di architettura a strati è un altro modo di ottenere la separazione e l'indipendenza. Questo schema è illustrato nella Figura 6.7, dove le funzionalità del sistema sono organizzate in strati separati, ciascuno dei quali si basa sulle funzioni e sui servizi offerti dallo strato immediatamente sottostante.

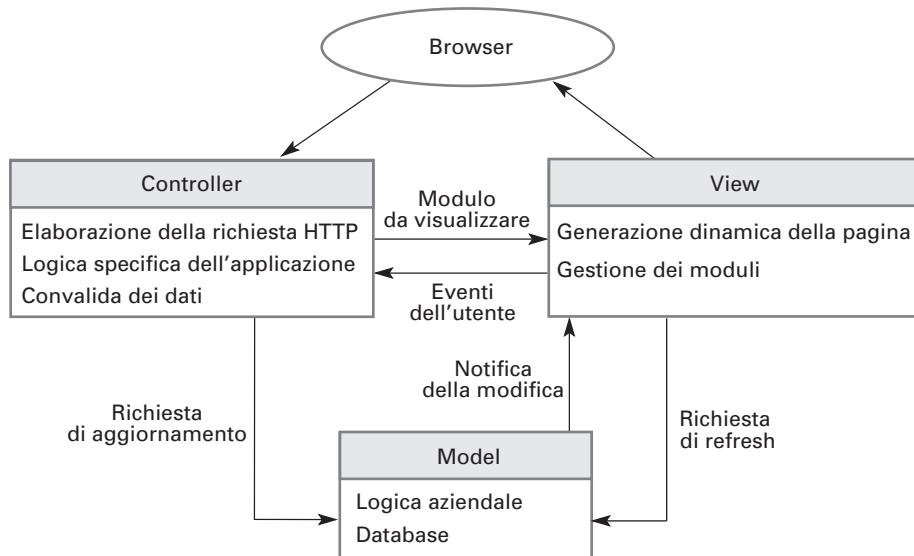


Figura 6.6 Architettura di applicazioni web che usa lo schema MVC.

Nome	Architettura a strati
Descrizione	Organizza il sistema in vari strati, con funzionalità associate a ciascuno strato. Uno strato fornisce i servizi allo strato sopra di esso, quindi gli strati di livello più basso rappresentano i servizi di base che probabilmente saranno utilizzati in tutto il sistema. Si veda la Figura 6.8.
Esempio	Un modello a strati di un sistema di apprendimento digitale che supporta l'apprendimento di tutti i soggetti nelle scuole (Figura 6.9).
Quando si usa	Si usa per costruire nuove funzioni per un sistema esistente; quando lo sviluppo è distribuito fra più team, dove ogni team ha la responsabilità di sviluppare le funzioni di uno strato; quando c'è una richiesta di protezione su più livelli.
Vantaggi	Consente la sostituzione di interi strati, se l'interfaccia non viene modificata. Le funzioni ridondanti (per esempio, l'autenticazione) possono essere fornite in ciascuno strato per aumentare la fidatezza del sistema.
Svantaggi	Nella pratica spesso è difficile ottenere una netta separazione fra gli strati, e uno strato di alto livello potrebbe aver bisogno di interagire direttamente con strati di livelli inferiori, anziché con lo strato immediatamente sotto di esso. Le performance possono essere un problema, a causa dei vari livelli di interpretazione di una richiesta di servizio, essendo questa elaborata in ciascuno strato.

Figura 6.7 Lo schema di architettura a strati.

Questo approccio a strati supporta lo sviluppo incrementale dei sistemi. Quando viene sviluppato uno strato, alcuni dei servizi da esso forniti sono resi accessibili agli utenti. Quest'architettura è anche modificabile e portatile. Se la sua interfaccia non viene modificata, un nuovo strato con funzionalità estese può sostituire uno strato esistente, senza modificare le altre parti del sistema. Inoltre, se si modificano le interfacce di uno strato o vengono aggiunte nuove funzionalità, solo lo strato adiacente ne è influenzato. Poiché i sistemi stratificati localizzano le dipendenze della macchina, è più semplice realizzare implementazioni multipiattaforma di un sistema applicativo. Solo gli strati dipendenti dalla macchina devono essere reimplementati per tenere conto delle funzionalità di un diverso sistema operativo o database.

La Figura 6.8 è un esempio di architettura a strati con quattro strati. Lo strato più basso include il software di supporto del sistema – tipicamente, un supporto per il sistema operativo e il database. Lo strato successivo è quello dell'applicazione, che include i componenti che riguardano le utilità e le funzionalità utilizzate da altri componenti dell'applicazione.

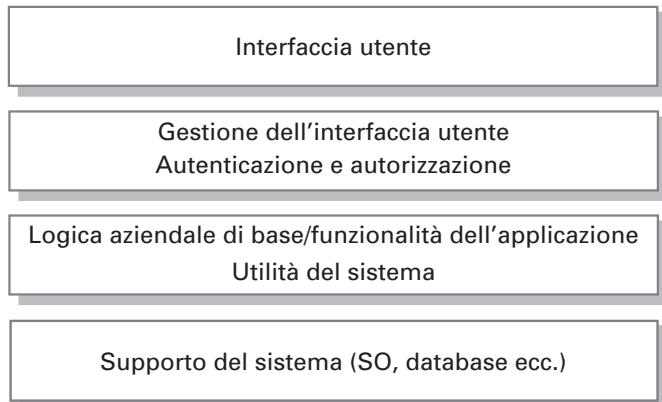


Figura 6.8 Una generica architettura a strati.

Il terzo strato riguarda la gestione dell'interfaccia utente e le funzioni per l'autenticazione e l'autorizzazione degli utenti. Lo strato superiore fornisce le funzioni per l'interfaccia utente. Ovviamente, il numero di strati è arbitrario; qualsiasi strato della Figura 6.6 può essere suddiviso in due o più strati.

La Figura 6.9 mostra che il sistema di apprendimento digitale iLearn, presentato nel Capitolo 1, ha un'architettura a quattro strati. Un altro esempio di schema di architettura a strati è riportato nella Figura 6.19 (Paragrafo 6.4), dove è rappresentata l'organizzazione del sistema Mentcare.

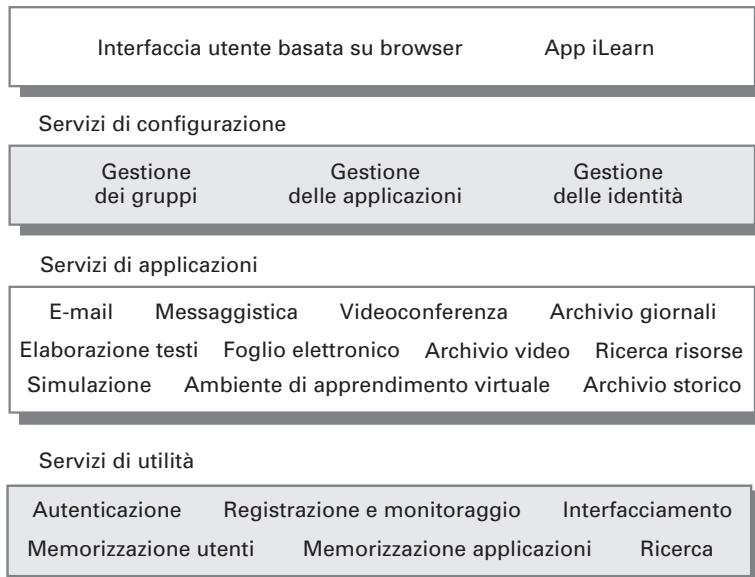


Figura 6.9 L'architettura del sistema di apprendimento digitale iLearn.

6.3.2 Architettura repository

L'architettura a strati e gli schemi MVC sono esempi di schemi dove la vista (view) presentata è l'organizzazione concettuale di un sistema. Il prossimo esempio, lo *schema repository* (Figura 6.10), spiega come una serie di componenti interattivi possano condividere i dati.

La maggior parte dei sistemi che usano grandi quantità di dati sono organizzati attorno a un database condiviso o repository. Questo modello è quindi adatto alle applicazioni nelle quali i dati sono generati da un componente e utilizzati da un altro componente. Esempi di questo tipo di sistemi sono i sistemi di comando e controllo, i sistemi di gestione delle informazioni, i sistemi CAD (Computer Aided Design) e gli ambienti di sviluppo interattivo per il software.

La Figura 6.11 illustra una situazione in cui può essere usato un repository. Lo schema mostra un IDE che include vari strumenti per supportare lo sviluppo guidato da modelli. Il repository in questo caso potrebbe essere un ambiente controllato dalle versioni (come descritto nel Capitolo 22) che tiene traccia delle modifiche apportate al software e consente un passaggio alle versioni precedenti.

Organizzare gli strumenti attorno a un repository è un modo efficiente di condividere grandi quantità di dati. Non è necessario trasmettere i dati esplicitamente da un componente all'altro. Tuttavia, i componenti devono operare attorno a un

Nome	Repository
Descrizione	Tutti i dati di un sistema vengono gestiti in un database centrale (detto repository) che è accessibile da tutti i componenti del sistema. I componenti non interagiscono direttamente, ma soltanto attraverso il repository.
Esempio	La Figura 6.11 è un esempio di IDE dove i componenti usano un repository di informazioni per la progettazione di un sistema. Ogni strumento software genera informazioni, che poi sono messe a disposizione degli altri strumenti.
Quando si usa	Si usa quando nel sistema vengono generati grandi volumi di informazioni che devono essere memorizzate per lungo tempo. Si può usare anche nei sistemi guidati dai dati dove l'inclusione dei dati nel repository innesca un'azione o l'utilizzo di uno strumento.
Vantaggi	I componenti possono essere indipendenti; un componente non deve necessariamente sapere dell'esistenza di un altro componente. Le modifiche fatte da un componente possono essere rese note agli altri componenti. Tutti i dati vengono gestiti in modo coerente (per esempio, i backup vengono effettuati contemporaneamente) in quanto si trovano nello stesso posto.
Svantaggi	Il repository è un punto comune di malfunzionamento, nel senso che i suoi problemi influiscono sull'intero sistema. Potrebbero esserci delle inefficienze nell'organizzazione di tutte le comunicazioni con il repository. Potrebbe essere difficile distribuire il repository su più computer.

Figura 6.10 Lo schema repository.

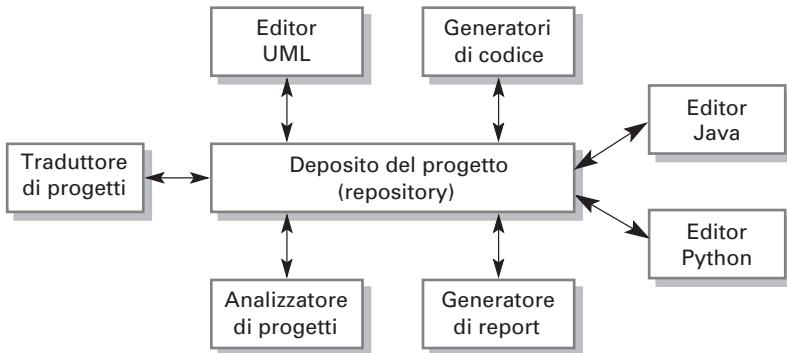


Figura 6.11 L'architettura repository per un IDE.

modello repository concordato. Inevitabilmente, questo è un compromesso tra le esigenze specifiche di ciascuno strumento, e potrebbe essere difficile o impossibile integrare nuovi componenti se i loro modelli di dati non sono conformi allo schema concordato. In pratica, potrebbe essere difficile distribuire il repository su un numero di macchine. Sebbene sia possibile distribuire un repository logicamente centralizzato, questo richiede la manutenzione di più copie di dati. Per mantenere coerenti e aggiornate queste copie, si aggiungono altri overhead al sistema.

Nell'architettura repository, illustrata nella Figura 6.11, il repository è passivo e del suo controllo sono responsabili i componenti che lo utilizzano. Un approccio alternativo, che è stato derivato per i sistemi di intelligenza artificiale, usa un modello “lavagna” (blackboard) che attiva i componenti quando sono disponibili determinati dati. Questo modello è utile quando i dati del repository non sono strutturati. La decisione su quale strumento attivare può essere presa solo quando tutti i dati sono stati analizzati. Questo modello, descritto da Nii (Nii 1986) e Bosch (Bosch 2000), include una discussione su come questo stile sia ben correlato agli attributi di qualità del sistema.

6.3.3 Architettura client-server

Lo schema repository riguarda la struttura statica di un sistema e non mostra la sua organizzazione a runtime. Il prossimo esempio, lo schema client-server (Figura 6.12), illustra una tipica organizzazione a runtime di sistemi distribuiti. Un sistema conforme allo schema client-server è organizzato come un insieme di servizi e server associati e di client che accedono e usano tali servizi. I principali componenti di questo modello sono:

1. un insieme di server che offrono servizi ad altri sottosistemi, per esempio servizi di stampa, servizi di gestione dei file e servizi di compilazione dei linguaggi di programmazione. I server sono componenti software, e molti server possono essere eseguiti sullo stesso computer;

Nome	Repository
Descrizione	In un'architettura client-server, il sistema è presentato come un insieme di servizi, ciascuno dei quali è fornito da un server separato. I client sono utenti di questi servizi e accedono ai server per utilizzare tali servizi.
Esempio	La Figura 6.13 è un esempio di una libreria di film e video/DVD organizzata come un sistema client-server.
Quando si usa	Si usa quando occorre accedere ai dati di un database da più postazioni. Poiché i server possono essere replicati, lo schema può essere utilizzato anche quando il carico su un sistema è variabile.
Vantaggi	Il principale vantaggio di questo modello è che i server possono essere distribuiti in una rete. Le funzionalità più comuni (per esempio, il servizio di stampa) possono essere a disposizione di tutti i client e non devono essere necessariamente implementate da tutti i servizi.
Svantaggi	Ciascun servizio è un punto comune di malfunzionamento, nel senso che è suscettibile di attacchi denial-of-service ed è soggetto a guasti del server. Le prestazioni possono essere imprevedibili in quanto dipendono dalla rete e anche dal sistema. Possono nascere problemi di gestione se i server sono di proprietà di più organizzazioni.

Figura 6.12 Lo schema client-server.

2. un insieme di client che richiedono i servizi offerti dai server; di solito si tratta di varie istanze di programmi client eseguiti contemporaneamente su computer differenti;
3. una rete che permette ai client di accedere a questi servizi. I sistemi client-server di solito sono implementati come sistemi distribuiti, collegati attraverso i protocolli Internet.

Le architetture client-server di solito sono concepite come architetture di sistemi distribuiti, ma il modello logico di servizi indipendenti che sono eseguiti su server separati può essere implementato su un singolo computer. Ancora una volta, la separazione e l'indipendenza sono un importante vantaggio. I servizi e i server possono essere modificati senza influire su altre parti del sistema.

I client potrebbero aver bisogno di conoscere il nome dei server disponibili e i servizi che possono fornire. I server, invece, non hanno bisogno di conoscere l'identità dei client o il numero di client che stanno utilizzando i loro servizi. I client accedono ai servizi offerti da un server attraverso chiamate di procedure remote, usando un protocollo richiesta-risposta (come http), dove un client invia una richiesta a un server e resta in attesa finché non riceve una risposta dal server.

La Figura 6.13 mostra l'esempio di un sistema basato sul modello client-server: si tratta di un sistema multiutente, basato sul Web, che fornisce una libreria

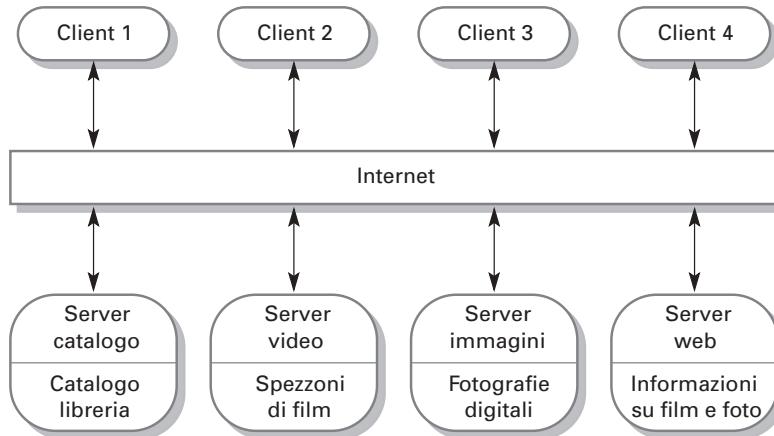


Figura 6.13 Architettura client-server per una libreria di film e immagini.

di film e fotografie. In questo sistema diversi server gestiscono e mostrano i diversi tipi di media. I frame dei video devono essere trasmessi velocemente e in sincronia, ma con una risoluzione relativamente bassa; possono essere compressi in una unità di memoria, in modo che il server dei video possa gestirne la compressione e la decompressione in diversi formati. Le immagini statiche, invece, devono essere memorizzate ad alta risoluzione, quindi è opportuno gestirle in un server separato.

Il catalogo deve essere in grado di gestire una serie di richieste e fornire i collegamenti al sistema informativo del Web, che contiene dati sui film e sui video-clip, e un sistema di commercio elettronico che ne gestisce la vendita. Il programma client è solo un'interfaccia utente integrata per accedere a tali servizi, costruita utilizzando un browser web.

Il vantaggio più importante del modello client-server è che si tratta di un'architettura distribuita. L'uso più efficiente si realizza nei sistemi in rete con molti processori distribuiti. È facile aggiungere un nuovo server e integrarlo con il resto del sistema o aggiornare i server in modo trasparente senza influire su altri parti del sistema. Nel Capitolo 17 tratto le architetture distribuite, dove spiego più dettagliatamente il modello client-server e le sue varianti.

6.3.4 Architettura pipe-and-filter

L'ultimo esempio di schema architetturale è lo schema pipe-and-filter (Figura 6.14), che rappresenta l'organizzazione a runtime di un sistema dove le trasformazioni funzionali elaborano i loro input e generano output. I dati passano da una trasformazione all'altra attraverso una sequenza. Ogni passo di elaborazione viene implementato come una trasformazione: i dati di input passano attraverso

Nome	Pipe-and-filter
Descrizione	L'elaborazione dei dati del sistema è organizzata in modo che ciascun componente di elaborazione (filtro) è discreto e svolge un particolare tipo di trasformazione dei dati. I dati fluiscono, come in un tubo (pipe), da un componente all'altro per essere elaborati.
Esempio	La Figura 6.15 è un esempio di sistema pipe-and-filter per l'elaborazione di fatture.
Quando si usa	Si usa di solito nelle applicazioni per l'elaborazione dei dati (batch o basata su transazioni), dove gli input vengono elaborati in fasi separate per generare i relativi output.
Vantaggi	È facile da capire e supporta il riutilizzo delle trasformazioni. Lo stile del flusso delle operazioni rispecchia la struttura di molti processi aziendali. La sua evoluzione è semplice perché permette di aggiungere agevolmente le nuove trasformazioni. Può essere implementato come sistema sequenziale o parallelo.
Svantaggi	Il formato di trasferimento dei dati deve essere concordato fra le varie trasformazioni. Ciascuna trasformazione deve essere in grado di leggere gli input e fornire gli output nel formato concordato. Questo aumenta gli overhead del sistema e potrebbe rendere impossibile il riutilizzo di quei componenti architetturali che usano strutture di dati non compatibili con il formato concordato.

Figura 6.14 Lo schema pipe-and-filter.

queste trasformazioni finché non vengono convertiti in output. Le trasformazioni possono essere eseguite sequenzialmente o in parallelo. I dati possono essere elaborati da ogni trasformazione elemento per elemento oppure in blocco.

Il nome “pipe-and-filter” (letteralmente “tubo e filtro”) deriva dalla terminologia utilizzata nei sistemi Unix, dove è possibile collegare i processi tramite “pipe”. Le pipe passano un flusso di testo da un processo all’altro. I sistemi conformi a questo schema possono essere implementati combinando comandi Unix, utilizzando le pipe e le funzionalità di controllo della shell di Unix. Il termine “filter” viene utilizzato perché una trasformazione “filtra” i dati che può elaborare dal suo flusso di dati di input.

Varianti di questo schema sono state utilizzate quando i computer erano usati nelle prime elaborazioni automatiche di dati. Quando le trasformazioni sono sequenziali con i dati elaborati in blocco, questo schema architettonico prende il nome di modello batch sequenziale. Questa è una tipica architettura dei sistemi di elaborazione dati, per esempio quelli di fatturazione. Anche l’architettura di un sistema integrato può essere organizzata come una pipeline di processi, dove ciascun processo viene eseguito in parallelo agli altri.

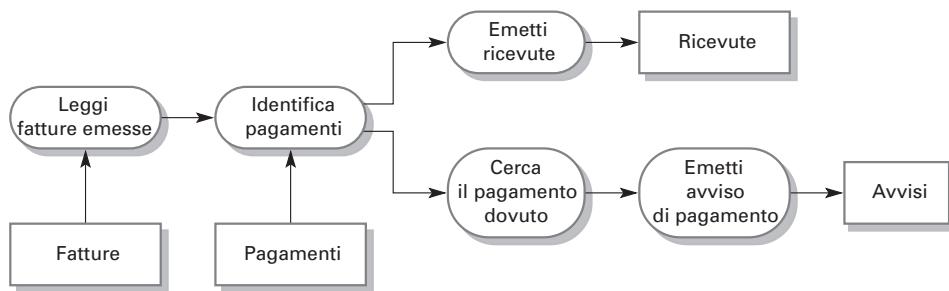


Figura 6.15 Esempio di architettura pipe-and-filter.

Un esempio di questo tipo di architettura di sistema, usato in un'applicazione di elaborazione batch, è illustrato nella Figura 6.15. Un'azienda ha inviato le fatture ai clienti che, dopo una settimana, vengono confrontate con i pagamenti riscossi; per le fatture che sono state pagate viene rilasciata una ricevuta; per quelle non pagate entro la scadenza viene emesso un avviso di sollecito.

Gli schemi pipe-and-filter sono particolarmente adatti ai sistemi di elaborazione batch e ai sistemi integrati, dove l'interazione con gli utenti è limitata. I sistemi interattivi sono difficili da scrivere utilizzando lo schema pipe-and-filter, perché è necessario avere un flusso di dati da elaborare. Mentre gli input e gli output di testo possono essere modellati in questo modo, le interfacce grafiche utente hanno formati di I/O più complessi e una tecnica di controllo che si basa su eventi, come i clic del mouse o le selezioni dei menu. Tutto questo è difficile da implementare come un flusso sequenziale che è conforme al modello pipe-and-filter.

6.4 Architetture applicative

I sistemi applicativi sono ideati per soddisfare necessità aziendali o organizzative. Tutte le aziende hanno molto in comune – devono assumere persone, emettere fatture, tenere la contabilità e così via. Le aziende che operano nello stesso settore usano applicazioni specifiche comuni al loro settore. Per esempio, oltre alle funzioni aziendali generiche, tutte le compagnie telefoniche hanno bisogno di sistemi per collegare e quantificare le telefonate, gestire le reti di comunicazione e inviare le fatture ai clienti. Ne consegue che i sistemi applicativi utilizzati da queste aziende hanno molto in comune.



Modelli architetturali di controllo

Esistono specifici modelli architetturali che rispecchiano i modi più comuni di organizzare il controllo in un sistema. Fra questi figurano il controllo centralizzato, basato su un componente che chiama altri componenti, e il controllo basato su eventi, dove il sistema reagisce agli eventi esterni.

Architettura delle applicazioni

Ci sono molti esempi di architetture delle applicazioni nel sito web del libro, tra cui le descrizioni di sistemi per l'elaborazione batch dei dati, i sistemi di allocazione delle risorse e i sistemi di editing basati sugli eventi.

<http://software-engineering-book.com/web/apparch/>

Questi elementi comuni hanno portato allo sviluppo di architetture software che descrivono la struttura e l'organizzazione di particolari tipi di sistemi software. Le architetture delle applicazioni encapsulano le caratteristiche principali di una classe di sistemi. Per esempio, nei sistemi real-time potrebbero esserci modelli architettonici generici di vari tipi di sistemi, come i sistemi per la raccolta dei dati o i sistemi di monitoraggio. Sebbene le istanze di questi sistemi differiscono nei dettagli, tuttavia la struttura architettonica comune può essere riutilizzata quando si sviluppano nuovi sistemi dello stesso tipo.

L'architettura delle applicazioni può essere reimplementata quando si sviluppano nuovi sistemi. Tuttavia, per molti sistemi aziendali, il riutilizzo dell'architettura delle applicazioni è implicito quando un generico sistema applicativo viene configurato per creare una nuova applicazione. Questo è dimostrato dalla diffusione dei sistemi di pianificazione delle risorse aziendali (ERP, *Enterprise Resource Planning*) e dei pacchetti applicativi configurabili pronti all'uso, come i sistemi per la contabilità e il controllo delle scorte. Questi sistemi hanno architetture e componenti standard. I componenti vengono configurati e adattati per creare un'applicazione per una specifica azienda. Per esempio, un sistema per la gestione di una catena di rifornimenti può essere adattato a diversi tipi di fornitori, merci e specifiche contrattuali.

Un progettista software può utilizzare questi modelli di architetture delle applicazioni in vari modi.

1. *Come punto di partenza per il processo di progettazione architettonica.* Se non avete familiarità con il tipo di applicazione che state sviluppando, potete basare il progetto iniziale su un'architettura applicativa generica. Successivamente, potrete adattarla al sistema specifico che state sviluppando.
2. *Come lista di verifica per il progetto.* Se avete sviluppato il progetto architettonale di un sistema, potete confrontarlo con una generica architettura applicativa per verificare se è coerente con essa.
3. *Come modo di organizzare il lavoro per il team di sviluppo.* Le architetture delle applicazioni identificano le funzionalità strutturali stabili delle architetture dei sistemi, che in molti casi è possibile sviluppare in parallelo. Si può assegnare ai membri del gruppo l'implementazione di differenti componenti all'interno di un'architettura.

4. *Come mezzo per valutare la riusabilità dei componenti.* Se avete componenti che pensate di poter riutilizzare, potete confrontarli con le strutture generiche per vedere se ci sono componenti paragonabili nell'architettura applicativa.
5. *Come vocabolario per discutere delle applicazioni.* Se state discutendo di una specifica applicazione o state tentando di confrontare applicazioni differenti, potete utilizzare i concetti identificati nelle architetture generiche per parlare di queste applicazioni.

Ci sono vari tipi di sistemi applicativi che, in alcuni casi, potrebbero sembrare molto diversi. Tuttavia, applicazioni che superficialmente appaiono diverse possono avere molto in comune e quindi condividere un'astratta architettura applicativa. Per illustrare questo, descriverò le architetture di due tipi di applicazioni.

1. *Applicazioni di elaborazione delle transazioni.* Le applicazioni basate su database che elaborano le richieste effettuate dall'utente e aggiornano le informazioni in un database; sono i tipi più comuni di sistemi aziendali interattivi. Questi sistemi sono organizzati in modo che le azioni dell'utente non possano interferire tra loro e che l'integrità del database sia sempre garantita. Questa classe di sistemi include i sistemi bancari interattivi, i sistemi di commercio elettronico, i sistemi informativi e i sistemi di prenotazione.
2. *Sistemi di elaborazione dei linguaggi.* I sistemi dove le intenzioni dell'utente sono espresse in un linguaggio formale, come un linguaggio di programmazione. Il sistema di elaborazione elabora questo linguaggio in un formato interno e poi interpreta questa rappresentazione interna. I più noti sistemi di elaborazione dei linguaggi sono i compilatori, che traducono i programmi dei linguaggi di alto livello in codice macchina. Questi sistemi sono usati anche per interpretare i linguaggi dei comandi per i database, i sistemi informativi e i linguaggi di markup come XML.

Ho scelto questi particolari tipi di sistemi perché un gran numero di sistemi aziendali basati sul Web sono sistemi di elaborazione delle transazioni e perché tutto lo sviluppo del software si basa sui sistemi di elaborazione dei linguaggi.

6.4.1 Sistemi di elaborazione delle transazioni

I sistemi di elaborazione delle transazioni sono progettati per elaborare le richieste dell'utente di ricevere informazioni da un database o di aggiornarlo (Lewis, Bernstein e Kifer 2003). Tecnicamente, una transazione per un database è parte di una sequenza di operazioni ed è trattata come una singola unità (unità atomica); tutte le operazioni di una transazione devono essere completate prima che i cambiamenti siano resi permanenti. Ciò garantisce che un fallimento nelle operazioni all'interno di una transazione non causi incoerenze nel database.

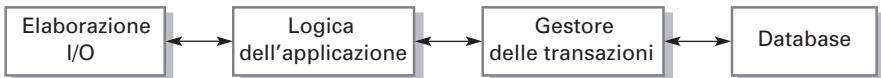


Figura 6.16 Struttura delle applicazioni di elaborazione delle transazioni.

Dal punto di vista dell’utente, una transazione è una sequenza coerente di operazioni che raggiunge un obiettivo, per esempio “trovare gli orari dei voli da Londra a Parigi”. Se la transazione dell’utente non richiede modifiche del database, allora non è necessario incapsularla come transazione tecnica del database.

Un esempio di transazione è la richiesta da parte di un cliente di prelevare soldi da un conto corrente bancario utilizzando uno sportello Bancomat: questa operazione richiede il controllo del saldo del conto corrente per verificare che ci siano fondi sufficienti, modificare il saldo in base alla quantità di soldi prelevati e inviare gli appositi comandi al Bancomat per consegnare il contante. Finché tutti questi passaggi non saranno stati completati, la transazione resterà incompleta e il database non sarà modificato.

I sistemi di elaborazione delle transazioni di solito sono sistemi interattivi in cui gli utenti pongono richieste di servizio asincrone. La Figura 6.16 illustra la struttura architetturale delle applicazioni per l’elaborazione delle transazioni. Innanzitutto, un utente pone una richiesta al sistema attraverso un componente di elaborazione I/O. La richiesta viene elaborata tramite una logica specifica dell’applicazione. Viene creata una transazione; viene passata al gestore delle transazioni, che di solito è integrato nel sistema di gestione del database. Dopo aver accertato che la transazione è stata completata con successo, il gestore segnala all’applicazione che l’elaborazione è terminata.

I sistemi di elaborazione delle transazioni possono essere organizzati come architetture “pipe-and-filter”, con i componenti responsabili dell’input, dell’elaborazione e dell’output. Per esempio, considerate un sistema che permette ai clienti di interrogare il proprio conto corrente bancario e prelevare contante da uno sportello Bancomat. Il sistema è composto da due componenti software che cooperano: il software dello sportello e il software per la gestione della contabilità nel server del database della banca. I componenti di input e output sono implementati nel software dello sportello, anche se il componente per l’elaborazione si trova nel server del database della banca. La Figura 6.17 mostra l’architettura di questo sistema, con le funzioni dei componenti di input, elaborazione e output.

6.4.2 Sistemi informativi

Tutti i sistemi che richiedono l’interazione con un database condiviso possono essere considerati sistemi informativi basati su transazioni. Un sistema informativo permette l’accesso controllato a un’ampia base di informazioni, come il catalogo di una biblioteca, la tabella degli orari dei voli o le cartelle cliniche dei

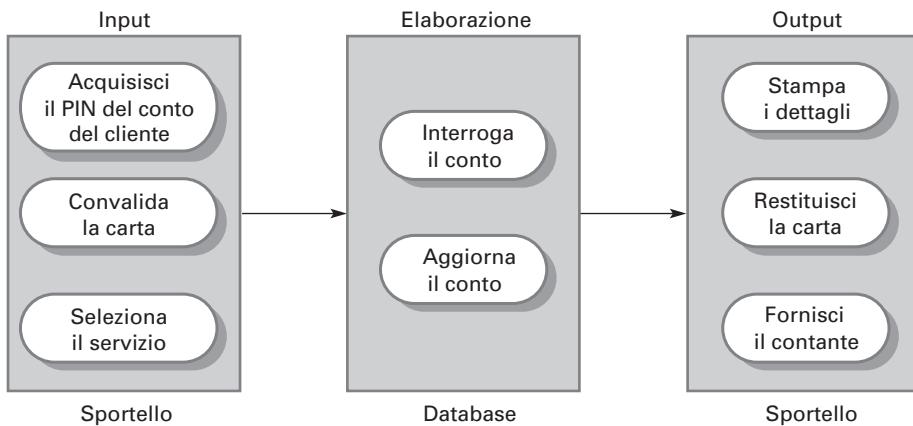


Figura 6.17 Architettura del software per uno sportello Bancomat.

pazienti di un ospedale. Quasi tutti i sistemi basati sul Web sono sistemi informativi, dove l’interfaccia utente è implementata in un browser.

La Figura 6.18 presenta un modello molto generale di un sistema informativo. Il sistema viene modellato utilizzando un’architettura a strati (descritta nel Paragrafo 6.3), dove lo strato superiore supporta l’interfaccia utente e quello inferiore il database del sistema. Lo strato di comunicazione con l’utente gestisce tutti gli input e output dell’interfaccia utente; lo strato di recupero delle informazioni contiene la logica specifica dell’applicazione per accedere e aggiornare il database. Gli strati in questo modello possono essere mappati direttamente nei server di un sistema distribuito basato su Internet.

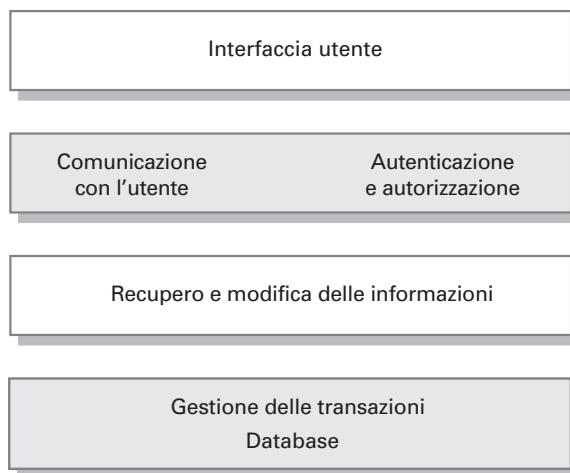


Figura 6.18 Architettura a strati di un sistema informativo.

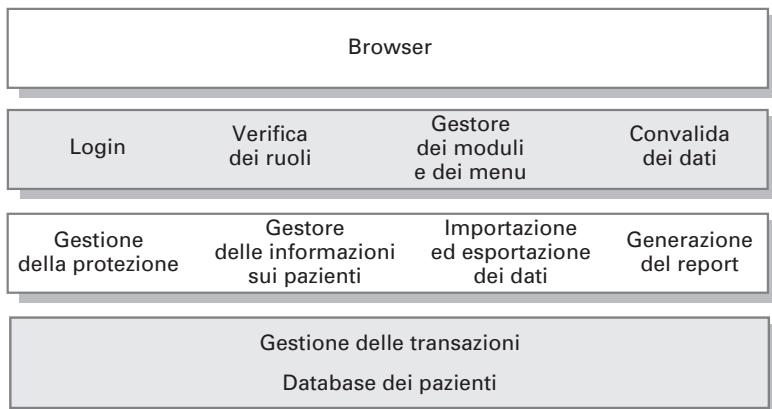


Figura 6.19 L'architettura del sistema Mentcare.

Un esempio di questo modello a strati è illustrato nella Figura 6.19, che mostra l'architettura del sistema Mentcare. Ricordo che questo sistema mantiene e gestisce le informazioni di pazienti che consultano dei medici specializzati in problemi di salute mentale. Ho aggiunto dei dettagli in ogni strato del modello per identificare i componenti che supportano la comunicazione con l'utente e il recupero e l'accesso alle informazioni.

1. Lo strato superiore è l'interfaccia utente basata su un browser.
2. Il secondo strato fornisce le funzionalità dell'interfaccia utente che sono accessibili tramite il browser. Include i componenti che consentono all'utente di collegarsi al sistema (login) e i componenti di controllo che assicurano che le operazioni svolte dall'utente siano quelle consentite dal suo ruolo. Lo strato include anche i componenti per la gestione dei moduli e dei menu che presentano le informazioni agli utenti, e i componenti di convalida dei dati che verificano la coerenza delle informazioni.
3. Il terzo strato implementa le funzionalità del sistema e fornisce i componenti che implementano la sicurezza del sistema, la creazione e l'aggiornamento delle informazioni sui pazienti, l'importazione e l'esportazione dei dati dei pazienti da altri database, e i generatori che creano i report sulla gestione.
4. L'ultimo strato, che è costruito utilizzando un sistema commerciale per la gestione dei database, realizza la gestione delle transazioni e la memorizzazione permanente dei dati.

I sistemi di gestione delle risorse e delle informazioni a volte sono anche sistemi di elaborazione delle transazioni. Per esempio, i sistemi di commercio elettronico sono sistemi di gestione delle risorse basate su Internet che accettano ordini elettronici di merci o servizi e poi dispongono la fornitura di queste merci o servizi

ai clienti. In un sistema di commercio elettronico, lo strato specifico dell'applicazione include alcune funzionalità aggiuntive che supportano un “carrello della spesa” in cui l'utente può collocare un certo numero di elementi in transazioni separate, e poi pagare tutti questi elementi in un'unica transazione.

L'organizzazione dei server in questi sistemi di solito rispecchia il modello generico a quattro strati presentato nella Figura 6.18. Questi sistemi sono spesso implementati come sistemi distribuiti con un'architettura client-server a più strati:

1. il server web è responsabile di tutte le comunicazioni con l'utente, con l'interfaccia utente implementata tramite un browser;
2. il server dell'applicazione è responsabile dell'implementazione della logica specifica dell'applicazione e delle richieste di memorizzazione e recupero delle informazioni;
3. il server del database sposta le informazioni nel database e gestisce le transazioni.

Utilizzando più server è possibile aumentare la quantità di operazioni e gestire migliaia di transazioni al minuto. Al crescere delle richieste, è possibile aggiungere nuovi server in ogni strato per far fronte all'incremento dei processi.

6.4.3 Sistemi di elaborazione dei linguaggi

I sistemi di elaborazione dei linguaggi traducono un linguaggio in una rappresentazione alternativa di quel linguaggio e, per i linguaggi di programmazione, possono anche eseguire il codice risultante. I compilatori traducono un linguaggio di programmazione nel codice macchina. Altri sistemi di elaborazione dei linguaggi possono tradurre una descrizione di dati XML in comandi per interrogare un database o in una rappresentazione XML alternativa. I sistemi di elaborazione dei linguaggi naturali possono tradurre un linguaggio naturale in un altro, per esempio il francese in norvegese.

Una possibile architettura per un sistema di elaborazione dei linguaggi per un linguaggio di programmazione è illustrata nella Figura 6.20. Le istruzioni nel linguaggio sorgente definiscono il programma da eseguire; un traduttore le converte in istruzioni per una macchina astratta. Queste istruzioni sono interpretate da un altro componente che preleva le istruzioni e le esegue usando, se necessario, i dati dell'ambiente. L'output del processo è il risultato dell'interpretazione delle istruzioni sui dati di input.

Per molti compilatori l'interprete è un'unità hardware che elabora le istruzioni macchina, e la macchina astratta è un vero processore. Tuttavia, per i linguaggi digitati dinamicamente, come Ruby o Python, l'interprete è un componente software.

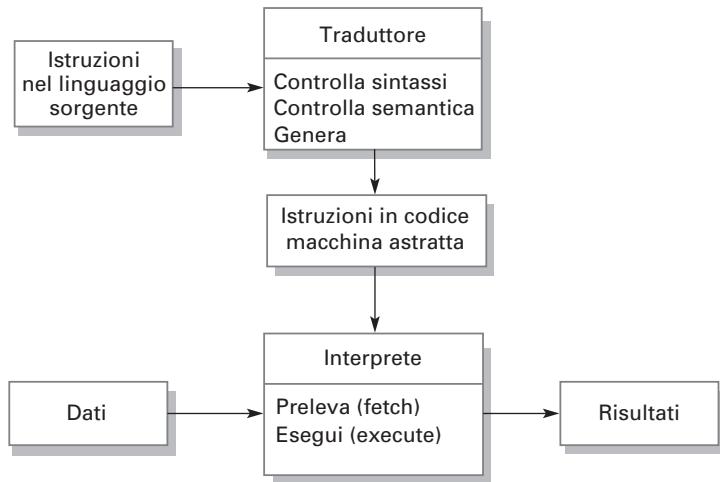


Figura 6.20 Architettura di un sistema di elaborazione dei linguaggi.

I compilatori dei linguaggi di programmazione che fanno parte di un ambiente di programmazione generale hanno un'architettura generica (Figura 6.21) che include i seguenti componenti:

1. un analizzatore lessicale, che prende i *token* (letteralmente “gettoni”) del linguaggio di input e li converte in una forma interna;
2. una tabella di simboli, che contiene le informazioni sui nomi delle entità (variabili, nomi delle classi, nomi degli oggetti ecc.) utilizzate nel testo che si sta traducendo;
3. un analizzatore sintattico, che controlla la sintassi del linguaggio che si sta traducendo; utilizza una grammatica definita del linguaggio e costruisce un albero sintattico;
4. un albero sintattico, ovvero una struttura interna che rappresenta il programma da compilare;
5. un analizzatore semantico, che utilizza le informazioni dell’albero sintattico e la tabella dei simboli per verificare la correttezza semantica del testo in input;
6. un generatore di codice, che “cammina” sull’albero e genera il codice della macchina astratta.

Possono essere inclusi anche altri componenti per analizzare e trasformare l’albero sintattico per migliorare l’efficienza e rimuovere eventuali ridondanze dal codice macchina generato. In altri tipi di sistemi di elaborazione dei linguaggi, come i traduttori dei linguaggi naturali, potrebbero esserci altri componenti, come i dizionari. L’output del sistema è la traduzione del testo di input.

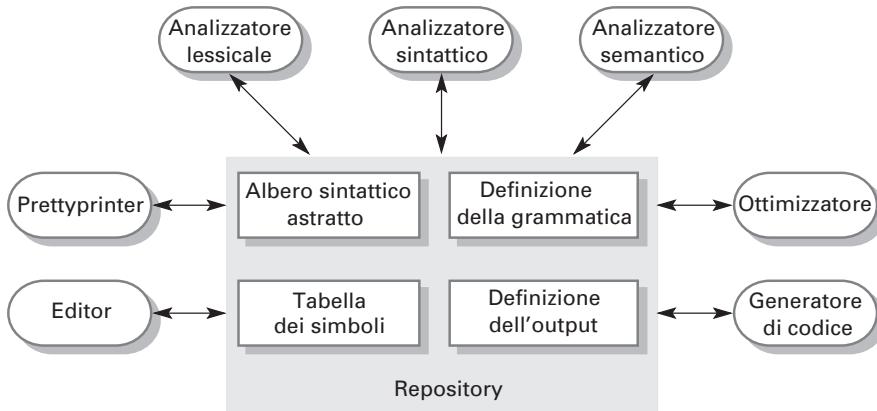


Figura 6.21 Architettura repository per un sistema di elaborazione dei linguaggi.

La Figura 6.21 illustra come un sistema di elaborazione dei linguaggi può essere parte di un set di strumenti integrati di supporto alla programmazione. In questo esempio la tabella dei simboli e l'albero sintattico fungono da repository centrale delle informazioni, tramite il quale gli strumenti comunicano. Altre informazioni che a volte sono integrate negli strumenti, come la definizione della grammatica e la definizione del formato di output per il programma, sono poste fuori dagli strumenti e inserite nel repository. Pertanto, un editor guidato dalla sintassi può verificare che la sintassi del programma sia corretta mentre viene scritto. Un formattore (prettyprinter) può creare listati del programma in un formato semplice da leggere e capire.

Gli schemi architetturali alternativi possono essere utilizzati in un sistema di elaborazione dei linguaggi (Garlan e Shaw 1993). I compilatori possono essere implementati utilizzando un modello repository combinato con un modello pipe-and-filter. Nell'architettura di un compilatore, la tabella dei simboli è un repository per i dati condivisi. Le fasi di analisi lessicale, sintattica e semantica sono organizzate in modo sequenziale, come mostra la Figura 6.22, e la comunicazione avviene tramite la tabella condivisa dei simboli.

Architetture di riferimento

Le architetture di riferimento esprimono le caratteristiche salienti delle architetture dei sistemi in un determinato dominio. Essenzialmente, includono tutto ciò che potrebbe esserci nell'architettura di un'applicazione, sebbene, in effetti, sia assai improbabile che una singola applicazione possa includere tutte le funzionalità espresse da un'architettura di riferimento. Lo scopo principale delle architetture di riferimento è valutare e confrontare le proposte di progettazione e indicare alle persone le caratteristiche architetturali di un particolare dominio.

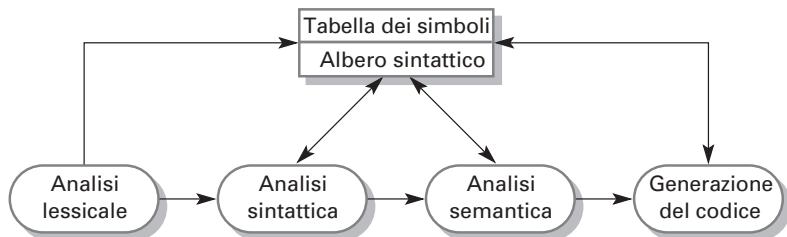


Figura 6.22 Architettura pipe-and-filter per un compilatore.

Il modello pipe-and-filter per la compilazione di un linguaggio è efficace negli ambienti batch, dove i programmi sono compilati ed eseguiti senza interazioni con l'utente, per esempio nella traduzione di un documento XML in un altro. È meno efficace quando il compilatore è integrato in altri strumenti di elaborazione del linguaggio, come un sistema di editing strutturato, un debugger interattivo o un formattatore di programmi. In questi casi, le modifiche di un componente devono essere immediatamente riportate negli altri componenti. È meglio organizzare il sistema attorno a un repository, come mostra la Figura 6.21, se si vuole implementare un generico ambiente di programmazione orientata al linguaggio.

Punti chiave

- Un'architettura software è la descrizione di come è organizzato un sistema software. Le proprietà di un sistema, come le prestazioni, la protezione e la disponibilità, sono influenzate dall'architettura utilizzata.
- Le decisioni di progettazione architettonica includono le scelte sul tipo di applicazione, sulla distribuzione del sistema, sugli schemi architettonici da utilizzare e sui modi in cui l'architettura deve essere documentata e valutata.
- Le architetture possono essere documentate da vari punti di vista o prospettive. Fra le viste possibili figurano la vista concettuale, la vista logica, la vista dei processi, la vista di sviluppo e la vista fisica.
- Gli schemi architettonici sono strumenti per riutilizzare le conoscenze acquisite su generiche architetture di sistemi. Descrivono l'architettura, spiegano quando devono essere utilizzata e ne illustrano vantaggi e svantaggi.
- Gli schemi architettonici più comuni includono gli schemi MVC (Model-View-Controller), a strati, repository, client-server e pipe-and-filter.
- I modelli generici delle architetture applicative aiutano a capire il funzionamento delle applicazioni, a confrontare applicazioni dello stesso tipo, a valutare la progettazione delle applicazioni e a stabilire i componenti da riutilizzare su larga scala.

- I sistemi di elaborazione delle transazioni sono sistemi interattivi che permettono a più utenti di accedere e modificare in modo remoto le informazioni di un database. I sistemi informativi e i sistemi di gestione delle risorse sono esempi di sistemi di elaborazione delle transazioni.
- I sistemi di elaborazione dei linguaggi sono utilizzati per tradurre testi da un linguaggio a un altro e di eseguire le istruzioni specificate nel linguaggio di input. Includono un traduttore e una macchina astratta che esegue il linguaggio generato.

Esercizi

- * 6.1 Spiegate perché, quando si descrive un sistema, potrebbe essere necessario iniziare la progettazione dell’architettura del sistema prima che sia completata la specifica dei requisiti.
- 6.2 Vi è stato chiesto di preparare e consegnare una presentazione destinata a un manager non tecnico per giustificare l’assunzione di un architetto di sistemi per lo sviluppo di un nuovo progetto. Scrivete una lista di punti nella vostra presentazione per mettere in evidenza gli aspetti chiave che spiegano l’importanza dell’architettura del software.
- * 6.3 Spiegate perché potrebbero sorgere conflitti di progettazione quando si progetta un’architettura dove i più importanti requisiti non funzionali sono la disponibilità e la protezione.
- 6.4 Disegnate dei diagrammi che mostrano una vista concettuale e una vista dei processi delle architetture dei seguenti sistemi:
 - un sistema automatico di distribuzione di biglietti per i passeggeri di una stazione ferroviaria;
 - un sistema di videoconferenza controllato da un computer che consente di visualizzare contemporaneamente a diversi partecipanti filmati e dati;
 - un robot per la pulizia di pavimenti in aree relativamente libere, come i corridoi. Il robot dovrà essere in grado di identificare le pareti e altri ostacoli.
- 6.5 Spiegate perché di solito si usano più schemi architetturali per progettare l’architettura di un grande sistema.
- 6.6 Suggerite un’architettura per un sistema (come iTunes) che è utilizzato per vendere e distribuire brani musicali su Internet. Quali schemi architetturali sono alla base dell’architettura che proponete?
- * 6.7 Occorre sviluppare un sistema informativo per mantenere le informazioni sui beni di proprietà di un’azienda di servizi, per esempio edifici, veicoli e attrezzature. Queste informazioni potranno essere modificate dal personale che opera sul campo tramite unità mobili, ogni volta che si rendono disponibili nuove informazioni sui beni. L’azienda ha già vari database sui beni che devono essere integrati in questo sistema. Progettate un’architettura a strati per questo sistema di gestione dei beni facendo riferimento all’architettura di un generico sistema informativo illustrato nella Figura 6.18.

-
- * 6.8 Utilizzando il modello generico del sistema di elaborazione dei linguaggi presentato in questo capitolo, progettate l'architettura di un sistema che accetta i comandi nel linguaggio naturale e li traduce in interrogazioni di un database nel linguaggio SQL.
 - * 6.9 Utilizzando il modello base del sistema informativo illustrato nella Figura 6.18, indicate i possibili componenti di un'applicazione per un dispositivo mobile che visualizza le informazioni sui voli in arrivo e in partenza da un particolare aeroporto.
 - 6.10 Dovrebbe esistere una professione separata di “architetto del software” il cui ruolo è lavorare indipendentemente con un cliente per progettare un’architettura di un sistema software? Il sistema dovrebbe poi essere implementato da qualche società di software. Quali possono essere le difficoltà di far riconoscere una professione simile?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Software Architecture: Perspectives on an Emerging Discipline. È stato il primo libro sull’architettura del software; include una buona analisi dei vari stili architetturali, tuttora valida. (M. Shaw e D. Garlan, 1996, Prentice-Hall).

“The Golden Age of Software Architecture.” Questo articolo fornisce un’indagine sullo sviluppo dell’architettura del software, dalle sue origini negli anni ’80 fino al suo impiego nel XXI secolo. Non c’è molto contenuto tecnico, ma è un’interessante panoramica storica. (M. Shaw e P. Clements, *IEEE Software*, 21 (2), marzo-aprile 2006) <http://dx.doi.org/10.1109/MS.2006.58>.

Software Architecture in Practice (3rd ed.). Un’analisi pratica delle architetture del software che non loda eccessivamente i benefici della progettazione architettonica. Fornisce una logica aziendale chiara, spiegando perché le architetture sono importanti. (L. Bass, P. Clements e R. Kazman, 2012, Addison-Wesley).

Handbook of Software Architecture. Si tratta di un’opera in corso di lavorazione di Grady Booch, uno dei primi evangelisti dell’architettura del software. Ha documentato le architetture di vari tipi di sistemi software in modo che si possa vedere la realtà, anziché l’astrazione accademica. È disponibile sul Web ed è previsto che sarà pubblicato come libro. (G. Booch 2014) <http://www.handbookofsoftwarearchitecture.com/>

7

Progettazione e implementazione

Questo capitolo si propone di presentare la progettazione del software orientata agli oggetti mediante il linguaggio UML e descrivere alcuni importanti problemi di implementazione. Dopo aver letto questo capitolo:

- conoscerete le principali attività di un generico processo di progettazione orientata agli oggetti;
- conoscerete alcuni dei vari modelli che possono essere utilizzati per documentare una progettazione orientata agli oggetti;
- acquisirete il concetto di schema di progettazione e come questo sia un modo per riutilizzare le conoscenze e l'esperienza di progettazione;
- conoscerete i temi chiave che devono essere considerati durante l'implementazione del software, incluso il riutilizzo del software e lo sviluppo open-source.

- 7.1 Progettazione orientata agli oggetti tramite UML
- 7.2 Schemi di progettazione
- 7.3 Problemi di implementazione
- 7.4 Sviluppo open-source

La progettazione e l’implementazione del software sono le fasi nel processo di ingegneria del software in cui viene sviluppato un sistema software eseguibile. Per alcuni sistemi semplici, ingegneria del software significa progettazione e l’implementazione del software, e tutte le altre attività di ingegneria del software si fondono in questo processo. Per i sistemi complessi, invece, la progettazione e l’implementazione del software sono solo uno dei tanti processi di ingegneria del software (ingegneria, verifica e convalida dei requisiti ecc.).

Le attività di progettazione e implementazione del software sono inevitabilmente intrecciate. La progettazione del software è un’attività creativa in cui vengono identificati i componenti e le loro relazioni, in base alle richieste di un cliente. L’implementazione è il processo che realizza il progetto sotto forma di programma. A volte c’è una fase distinta di progettazione, che viene modellata e documentata. Altre volte il progetto è nella testa del programmatore oppure viene abbozzato su un foglio di carta o su una lavagna bianca. La progettazione si occupa delle modalità in cui risolvere un problema, quindi c’è sempre un processo di progettazione. Tuttavia, ciò non sempre è necessario o appropriato per descrivere dettagliatamente un progetto tramite il linguaggio UML o un altro linguaggio di descrizione dei progetti.

Progettazione e implementazione sono strettamente collegate, quindi di solito bisogna prendere in considerazione i problemi di implementazione durante lo sviluppo di un progetto. Per esempio, utilizzare l’UML per documentare un progetto potrebbe essere la cosa giusta da fare se si programma con un linguaggio digitato dinamicamente come Python. Sarebbe inutile utilizzare l’UML se si sta implementando un sistema configurando un’applicazione off-the-shelf. Come ho detto nel Capitolo 3, i metodi agili di solito si basano su bozze informali di un progetto e lasciano ai programmatore le scelte di progettazione.

Una delle decisioni più importanti di implementazione che devono essere prese nella fase iniziale di un progetto software è se conviene creare o acquistare il software dell’applicazione. Per molti tipi di applicazioni adesso è possibile acquistare sistemi off-the-shelf che possono essere adattati alle richieste del cliente. Per esempio, se volete implementare un sistema per la gestione di cartelle cliniche, potete acquistare un package che è già in uso negli ospedali. Di solito, è più economico e veloce adottare questo approccio, anziché sviluppare un nuovo sistema in un linguaggio di programmazione convenzionale.

Quando si sviluppa un sistema applicativo riutilizzando un prodotto off-the-shelf, il processo di progettazione si concentra su come configurare il prodotto per soddisfare i requisiti dell’applicazione. Non occorre sviluppare modelli di progettazione del sistema, quali i modelli degli oggetti del sistema e delle loro interazioni. Il Capitolo 15 descrive questo metodo di sviluppo basato sul riutilizzo.

Suppongo che molti lettori di questo libro abbiano qualche esperienza di progettazione e implementazione di programmi. Si tratta di qualcosa che si acquisisce quando si impara a programmare e a padroneggiare gli elementi di un linguaggio di programmazione come Java o Python. Probabilmente avrete appreso i

concetti della buona programmazione studiando i linguaggi di programmazione, come pure avrete imparato le tecniche di debugging dei programmi che avete sviluppato. Di conseguenza, non tratterò questi argomenti qui. Questo capitolo ha due obiettivi principali:

1. spiegare come la modellazione e la progettazione architetturale (trattate nei Capitoli 5 e 6) sono utilizzate nella pratica per sviluppare un progetto software orientato agli oggetti;
2. trattare alcuni importanti problemi di implementazione che di solito vengono ignorati nei libri di programmazione, come il riutilizzo del software, la gestione della configurazione e lo sviluppo open-source.

Poiché ci sono numerose piattaforme di sviluppo, il capitolo non farà riferimento a un particolare linguaggio di programmazione o a una specifica tecnologia di implementazione. Presenterò tutti gli esempi utilizzando il linguaggio UML, anziché un linguaggio di programmazione come Java o Python.

7.1 Progettazione orientata agli oggetti tramite UML

Un sistema orientato agli oggetti è formato da oggetti interagenti che mantengono il proprio stato locale e forniscono le operazioni su tale stato. La rappresentazione dello stato è privata e non vi si può accedere direttamente dall'esterno dell'oggetto. I processi di progettazione orientata agli oggetti richiedono la progettazione di classi di oggetti e delle loro relazioni. Queste classi definiscono gli oggetti nel sistema e le loro interazioni. Quando la progettazione si realizza in un programma eseguibile, gli oggetti vengono creati dinamicamente dalle definizioni delle classi.

Gli oggetti includono sia i dati sia le operazioni per manipolare tali dati. Essi possono quindi essere concepiti e modificati come entità autonome. La modifica dell'implementazione di un oggetto o l'aggiunta di servizi non dovrebbe influire su altri oggetti del sistema. Poiché gli oggetti sono associati a cose, spesso c'è una chiara corrispondenza tra le entità del mondo reale (come i componenti hardware) e i loro oggetti di controllo nel sistema. Questo migliora la comprensibilità, e quindi la semplicità di manutenzione, del progetto.

Per sviluppare la progettazione di un sistema da un concetto astratto a un progetto dettagliato, orientato agli oggetti, occorre:

1. capire e definire il contesto e le interazioni esterne con il sistema;
2. progettare l'architettura del sistema;
3. identificare i principali oggetti del sistema;
4. sviluppare i modelli di progettazione;
5. specificare le interfacce.

Come tutte le attività creative, la progettazione non è un processo sequenziale netto. Un progetto viene sviluppato formulando idee, proponendo soluzioni e affinando queste soluzioni quando si acquisiscono nuove informazioni. Inevitabilmente, occorre ritornare sui propri passi e riprovare un’altra soluzione quando si presenta un problema. A volte, occorre sperimentare dettagliatamente tutte le opzioni per verificare che esse funzionino correttamente; altre volte, occorre rimandare la verifica dei dettagli a una fase successiva del processo. A volte, si usano le notazioni, come quelle dell’UML, per chiarire con maggiore precisione gli aspetti del progetto; altre volte, le notazioni sono utilizzate in modo informale per stimolare la discussione.

Spiegherò la progettazione del software orientata agli oggetti sviluppando un progetto per la parte del software integrato della stazione meteorologica che ho presentato nel Capitolo 1. Le stazioni meteorologiche vengono poste in alcune aree isolate. Ciascuna stazione registra le informazioni atmosferiche locali e invia periodicamente i dati raccolti al sistema informatico utilizzando un collegamento satellitare.

7.1.1 **Contesto del sistema e interazioni**

Il primo stadio in ogni processo di progettazione del software è sviluppare la comprensione delle relazioni tra il software che si sta progettando e il suo ambiente esterno. Questa comprensione è necessaria per decidere come fornire le funzionalità richieste per il sistema e come strutturare la comunicazione tra il sistema e il suo ambiente. Come detto nel Capitolo 5, la comprensione del contesto consente anche di stabilire i confini del sistema.

Impostare i confini del sistema aiuta a scegliere quali caratteristiche devono essere implementate nel sistema che si sta progettando e quali caratteristiche si trovano in altri sistemi associati. In questo caso, occorre decidere come le funzionalità debbano essere distribuite tra il sistema di controllo di tutte le stazioni meteorologiche e il software integrato nella singola stazione meteorologica.

I modelli del contesto del sistema e i modelli di interazione rappresentano delle viste complementari delle relazioni tra un sistema e il suo ambiente:

1. un modello del contesto di un sistema è un modello strutturale che descrive gli altri sistemi nell’ambiente del sistema che si sta sviluppando;
2. un modello di interazione è un modello dinamico che descrive come il sistema interagisce con il suo ambiente mentre viene utilizzato.

Il modello del contesto di un sistema può essere rappresentato tramite associazioni. Le associazioni mostrano semplicemente che ci sono alcune relazioni tra le entità di ogni associazione. È possibile documentare l’ambiente del sistema utilizzando un semplice diagramma a blocchi che mostra le entità del sistema e le loro associazioni. La Figura 7.1 mostra che i sistemi nell’ambiente di ciascuna stazione meteorologica sono un sistema informatico, un sistema satellitare e un sistema di controllo. Le informazioni di cardinalità sui collegamenti mostrano che

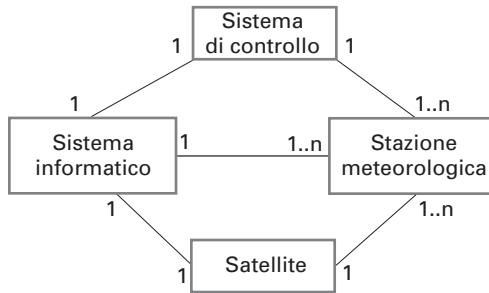


Figura 7.1 Contesto del sistema per la stazione meteorologica.

c’è un solo sistema di controllo, ma più stazioni meteorologiche, un satellite e un sistema informatico generale.

Quando si modellano le interazioni di un sistema con il suo ambiente, si dovrebbe utilizzare un approccio astratto che non include troppi dettagli. Un modo per farlo consiste nell’utilizzare un modello di casi d’uso. Come detto nei Capitoli 4 e 5, ciascun caso d’uso rappresenta un’interazione con il sistema. Ogni possibile interazione viene inserita con un nome all’interno di un’ellisse, e l’entità esterna coinvolta in tale interazione è rappresentata da una figura stilizzata.

Il modello del caso d’uso per la stazione meteorologica è illustrato nella Figura 7.2. Questo mostra che la stazione meteorologica interagisce con il sistema informatico che fornisce un rapporto sui dati meteorologici raccolti e sullo stato dell’hardware della stazione. Altre interazioni avvengono con il sistema di controllo che può emettere specifici comandi di controllo della stazione. La figura

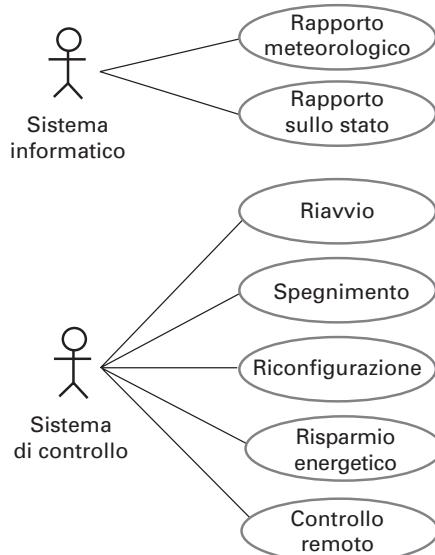


Figura 7.2 Casi d’uso per la stazione meteorologica.

Casi d'uso per la stazione meteorologica

Rapporto meteorologico – invia i dati meteorologici al sistema informatico

Rapporto sullo stato – invia le informazioni sullo stato al sistema informatico

Riavvio – se la stazione meteorologica è ferma, riavvia il sistema

Spegnimento – spegni la stazione meteorologica

Riconfigurazione – riconfigura il software della stazione meteorologica

Risparmio energetico – abilita la modalità di risparmio energetico della stazione meteorologica

Controllo remoto – invia i comandi di controllo al sottosistema di una stazione meteorologica

<http://software-engineering-book.com/web/ws-use-cases/>

stilizzata viene utilizzata nel linguaggio UML per rappresentare altri sistemi e anche gli utenti umani.

Ciascuno di questi casi d'uso dovrebbe essere descritto nel linguaggio naturale strutturato. Questo aiuta i progettisti a identificare gli oggetti del sistema e fornisce loro una comprensione di ciò che il sistema deve fare. Ho utilizzato una forma standard per questa descrizione che identifica chiaramente quali informazioni vengono scambiate, come l'interazione viene avviata e così via. I sistemi integrati sono spesso modellati descrivendo come essi rispondono agli stimoli interni ed esterni. Di conseguenza, gli stimoli e le risposte associate dovrebbero essere elencati nella descrizione. La Figura 7.3 mostra la descrizione del caso d'uso *Rapporto meteorologico* della Figura 7.2 che si basa su questo approccio.

7.1.2 Progettazione architettonicale

Una volta definite le interazioni tra il sistema software e l'ambiente del sistema, si possono utilizzare queste informazioni come base per progettare l'architettura del sistema. Ovviamente occorre combinare queste informazioni con la conoscenza dei principi di progettazione architettonicale e con una più dettagliata conoscenza del dominio di applicazione. Occorre identificare i principali componenti che formano il sistema e le loro interazioni; poi è possibile progettare l'organizzazione del sistema utilizzando uno schema architettonicale, come un modello a strati o client-server.

Il progetto architettonicale di alto livello per il software della stazione meteorologica è illustrato nella Figura 7.4. La stazione meteorologica è composta da sottosistemi indipendenti che comunicano trasmettendo messaggi su una infrastruttura comune, indicata con *Link di comunicazione* nella Figura 7.4. Ciascun sottosistema ascolta i messaggi su quella infrastruttura e raccoglie quelli che lo riguardano. Questo “modello di ascolto” è uno schema architettonicale comunemente utilizzato nei sistemi distribuiti.

Sistema	Stazione meteorologica.
Caso d'uso	Rapporto meteorologico.
Attori	Sistema informatico per la raccolta e l'elaborazione dei dati meteorologici, stazione meteorologica.
Dati	La stazione meteorologica invia una sintesi dei dati meteorologici rilevati dagli strumenti nel periodo di rilevazione al sistema informatico. I dati inviati sono le temperature massime, minime e medie del suolo e dell'aria; le pressioni massime, minime e medie dell'aria; le velocità massime minime e medie del vento; le precipitazioni; la direzione del vento, campionata a intervalli di cinque minuti.
Stimolo	Il sistema informatico stabilisce un collegamento satellitare con la stazione meteorologica e richiede la trasmissione dei dati.
Risposta	La sintesi dei dati è inviata al sistema informatico.
Commenti	Alle stazioni meteorologiche viene solitamente chiesto di fare rapporto una volta ogni ora, ma questa frequenza può cambiare da una stazione all'altra e può essere modificata in futuro.

Figura 7.3 Descrizione del caso d'uso *Rapporto meteorologico*.

Quando il sottosistema di comunicazione riceve un comando di controllo, come lo spegnimento, il comando viene raccolto da ciascuno degli altri sottosistemi, che quindi si spengono nella maniera corretta. Il vantaggio chiave di questa architettura è che è facile supportare diverse configurazioni di sottosistemi, in quanto il trasmettitore di un messaggio non ha bisogno di indirizzare il messaggio a un particolare sottosistema.

La Figura 7.5 mostra l'architettura del sottosistema di raccolta dei dati, che è incluso nella Figura 7.4. Gli oggetti *Trasmettitore* e *Ricevitore* riguardano la gestione delle comunicazioni; l'oggetto *DatiMeteorologici* incapsula le informazioni che vengono raccolte dagli strumenti e trasmesse al sistema informatico della stazione meteorologica.

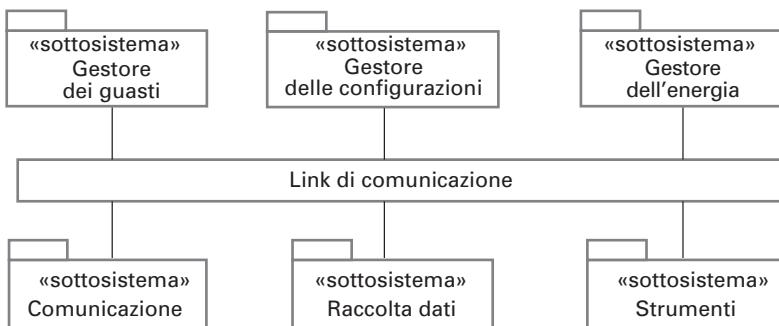


Figura 7.4 Architettura di alto livello della stazione meteorologica.

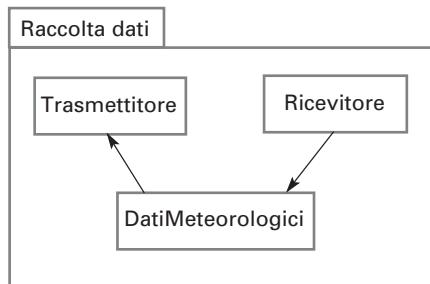


Figura 7.5 Architettura del sistema di raccolta dei dati.

7.1.3 Identificazione delle classi di oggetti

A questo punto del processo di progettazione dovreste avere qualche idea sugli oggetti essenziali del sistema che state progettando. Man mano che le vostre conoscenze del progetto si sviluppano, potete affinare queste idee sugli oggetti del sistema. La descrizione dei casi d'uso aiuta a identificare gli oggetti e le operazioni del sistema. Dalla descrizione del caso d'uso *Rapporto meteorologico* è ovvio che sarà necessario implementare gli oggetti che rappresentano gli strumenti che raccolgono i dati meteo e un oggetto che rappresenta la sintesi di tali dati. Saranno anche necessari alcuni oggetti di alto livello che incapsulano le interazioni del sistema definite nei casi d'uso. Con questi oggetti in mente potete iniziare a identificare le classi di oggetti generali del sistema.

La progettazione orientata agli oggetti si è sviluppata negli anni '80; da allora sono state elaborate diverse tecniche di identificazione delle classi di oggetti per i sistemi orientati agli oggetti.

1. Utilizzare l'analisi grammaticale di una descrizione in linguaggio naturale del sistema da costruire. Gli oggetti e gli attributi sono nomi; le operazioni o i servizi sono verbi (Abbott, 1983).
2. Utilizzare entità tangibili (cose) del dominio di applicazione, per esempio velivoli, ruoli come i manager, eventi come le richieste, interazioni come le riunioni, luoghi come gli uffici, unità organizzative come le società e così via (Wirfs-Brock, Wilkerson e Weiner 1990).
3. Utilizzare un'analisi basata su scenari, in cui i vari scenari di utilizzo del sistema sono identificati e analizzati uno alla volta. Mentre ogni scenario viene analizzato, il team responsabile dell'analisi deve identificare gli oggetti, gli attributi e le operazioni che servono (Back e Cunningham 1989).

In pratica occorre servirsi di varie conoscenze per scoprire le classi di oggetti. Le classi di oggetti, gli attributi e le operazioni definiti inizialmente dalla descrizione informale del sistema, possono essere un punto di partenza per la progettazione. Per affinare ed estendere gli oggetti iniziali, possono essere utilizzate ulteriori informazioni derivate dalla conoscenza del dominio di applicazione o dall'analisi

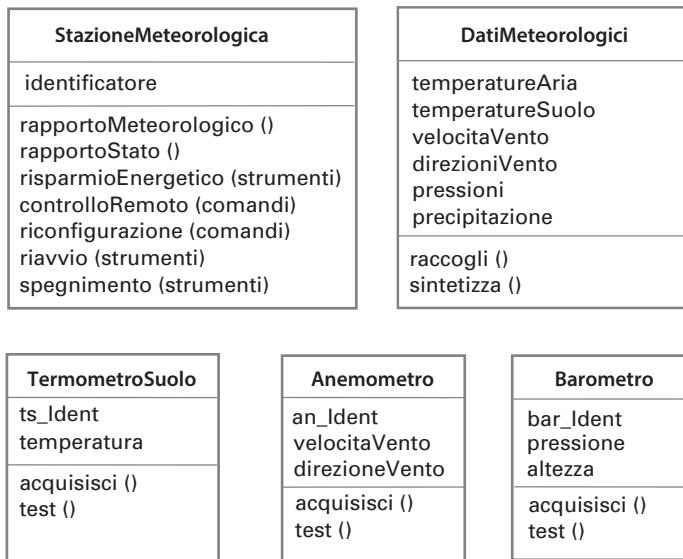


Figura 7.6 Oggetti della stazione meteorologica.

degli scenari. Tali informazioni possono essere raccolte dai documenti dei requisiti, dalle discussioni con gli utenti e da un’analisi dei sistemi esistenti. Oltre agli oggetti che rappresentano entità esterne al sistema, è anche possibile progettare gli “oggetti di implementazione” che sono utilizzati per fornire i servizi generali, come la ricerca e la convalida.

Nel caso della stazione meteorologica, l’identificazione degli oggetti si basa su elementi hardware tangibili del sistema. Non c’è spazio qui per includere tutti gli oggetti del sistema, ma nella Figura 7.6 ho indicato cinque classi di oggetti. *TermometroSuolo*, *Anemometro* e *Barometro* rappresentano gli oggetti del dominio di applicazione; *StazioneMeteorologica* e *DatiMeteorologici* sono stati identificati dalla descrizione del sistema e dello scenario (caso d’uso).

1. La classe di oggetti *StazioneMeteorologica* fornisce l’interfaccia di base della stazione con il suo ambiente. Le sue operazioni si basano sulle interazioni mostrate nella Figura 7.3. In questo caso è stata utilizzata una singola classe di oggetti per incapsulare tutte queste interazioni. In alternativa, si potrebbe progettare l’interfaccia di sistema sotto forma di diverse classi, con una classe per interazione.
2. La classe di oggetti *DatiMeteorologici* ha il compito di elaborare il comando che genera il rapporto meteorologico. Invia al sistema informatico della stazione meteorologica i dati di sintesi che provengono dagli strumenti.
3. Le classi *TermometroSuolo*, *Anemometro* e *Barometro* sono in relazione diretta con gli strumenti del sistema. Riflettono le entità hardware tangibili del sistema e le operazioni si occupano di controllare tale hardware. Questi

oggetti operano in modo autonomo per raccogliere i dati con frequenza specifica e memorizzano localmente tali dati, che vengono poi consegnati all’oggetto *DatiMeteorologici* in seguito a una richiesta.

In base alle vostre conoscenze del dominio di applicazione, potete identificare altri oggetti, attributi e servizi.

1. Le stazioni meteorologiche di solito sono localizzate in posti remoti e includono vari strumenti che a volte forniscono dati errati. I malfunzionamenti degli strumenti dovrebbero essere riportati automaticamente. Questo implica la necessità di avere attributi e operazioni che controllano il corretto funzionamento degli strumenti.
2. Se ci sono molte stazioni meteorologiche remote e bisogna identificarne i dati raccolti, è necessario che ogni stazione abbia un proprio identificatore nelle comunicazioni.
3. Se le stazioni meteorologiche vengono installate in periodi differenti, i tipi di strumenti potrebbero essere diversi. Quindi, ogni strumento dovrebbe essere identificato in modo univoco, e dovrebbe essere creato un database con le informazioni aggiornate sugli strumenti.

In questa fase del processo di progettazione, l’attenzione dovrebbe essere posta sugli oggetti stessi, senza pensare al modo in cui questi dovranno essere implementati. Una volta identificati gli oggetti, è possibile affinare la loro progettazione. Occorre individuare le caratteristiche comuni e poi progettare la gerarchia delle eredità per il sistema. Per esempio, si potrebbe identificare una superclasse *Strumento*, che definisce le caratteristiche comuni di tutti gli strumenti, come un identificatore e le operazioni di acquisizione e test dei dati. Si potrebbero aggiungere nuovi attributi e operazioni alla superclasse, per esempio un attributo che memorizza la frequenza in cui i dati devono essere raccolti.

7.1.4 Modelli di progettazione

I modelli di progettazione o di sistema, come detto nel Capitolo 5, mostrano gli oggetti o le classi di oggetti presenti in un sistema; illustrano anche le associazioni e le relazioni tra queste entità. Questi modelli sono il ponte tra i requisiti di un sistema e la sua implementazione. Devono essere astratti in modo che i dettagli inutili non nascondino le varie relazioni e i requisiti del sistema. Tuttavia, devono anche includere un numero sufficiente di dettagli, affinché i programmatore possano fare le loro scelte di implementazione.

Il livello di dettagli necessari a un modello di progettazione dipende dal processo di progettazione adottato. Dove esistono collegamenti stretti tra ingegneri dei requisiti, progettisti e programmatore, allora i modelli astratti potrebbero essere sufficienti. Specifiche decisioni progettuali possono essere prese durante l’implementazione del sistema, e i problemi possono essere risolti tramite discussioni

informali. Analogamente, se viene adottato uno sviluppo agile, potrebbero bastare i modelli di progettazione tracciati su una lavagna bianca.

Se, invece, viene utilizzato un processo di sviluppo basato su piani, allora potrebbero essere necessari modelli più dettagliati. Quando i collegamenti tra gli ingegneri dei requisiti, i progettisti e i programmatore sono indiretti (per esempio quando un sistema è stato progettato da una parte di un'organizzazione ma implementato altrove), allora sono richieste precise descrizioni del progetto. I modelli dettagliati, derivati da modelli astratti di alto livello, sono utilizzati in modo che tutti i membri di un team abbiano conoscenze comuni del progetto.

Un passo importante nel processo di progettazione è dunque decidere quali modelli di progettazione e quale livello di dettagli sono necessari. Questo dipende dal tipo di sistema che si sta sviluppando. Un sistema di elaborazione dati sequenziale sarà progettato in modo diverso da un sistema integrato real-time, quindi sarà necessario utilizzare modelli di progettazione differenti. Il linguaggio UML supporta 13 tipi diversi di modelli, ma, come detto nel Capitolo 5, molti di questi modelli non sono largamente diffusi. Minimizzando il numero di modelli prodotti, è possibile ridurre i costi di progettazione e i tempi richiesti per completarne il processo.

Quando si usa il linguaggio UML per sviluppare un progetto, occorre sviluppare due tipi di modelli di progettazione:

1. i *modelli strutturali*, che descrivono la struttura statica del sistema tramite le classi di oggetti e le loro relazioni. Le relazioni importanti che possono essere documentate in questo stadio sono relazioni di generalizzazione (ereditarietà), relazioni usa/usato-da e relazioni di composizione;
2. i *modelli dinamici*, che descrivono la struttura dinamica del sistema e mostrano le interazioni tra gli oggetti del sistema. Le interazioni che possono essere documentate comprendono le sequenze delle richieste di servizi fatte dagli oggetti e il modo in cui gli stati del sistema vengono innescati dalle interazioni di questi oggetti.

Ritengo che tre tipi di modelli UML siano particolarmente utili per aggiungere dettagli ai modelli architettonici e dei casi d'uso:

1. i *modelli di sottosistema*, che mostrano il raggruppamento logico degli oggetti in sottosistemi coerenti. Sono rappresentati usando una forma di diagramma delle classi, dove ogni sottosistema è mostrato come package di oggetti. I modelli di sottosistema sono modelli strutturali;
2. i *modelli di sequenza*, che mostrano la sequenza delle interazioni tra gli oggetti. Nel linguaggio UML sono rappresentati tramite un diagramma di sequenza o di collaborazione. I modelli di sequenza sono modelli dinamici;
3. i *modelli di macchine a stati*, che mostrano come i singoli oggetti cambiano il loro stato in risposta agli eventi. Nel linguaggio UML questi modelli sono rappresentati tramite diagrammi di stato. Sono modelli dinamici.

Un modello di sottosistema è un modello statico utile che mostra come un progetto è organizzato in gruppi di oggetti logicamente correlati. Ho già illustrato questo tipo di modello nella Figura 7.4 per presentare i sottosistemi nel sistema di mappatura di una stazione meteorologica. Oltre ai modelli di sottosistema, è anche possibile progettare modelli dettagliati di oggetti, che mostrano gli oggetti nei sistemi e le loro associazioni (ereditarietà, generalizzazione, aggregazione ecc.). Tuttavia c'è un pericolo nel creare troppi modelli: prendere decisioni dettagliate sull'implementazione che sarebbe meglio rimandare nelle fasi finali dell'implementazione.

I modelli di sequenza sono modelli dinamici che descrivono, per ciascuna modalità di interazione, la sequenza delle interazioni che avvengono tra gli oggetti. Quando si documenta un progetto, bisogna produrre un modello di sequenza per ogni interazione significativa. Se avete sviluppato un modello di casi d'uso, allora dovete creare un modello di sequenza per ciascun caso d'uso che avete identificato.

La Figura 7.7 è un esempio di modello di sequenza, rappresentato come diagramma di sequenza UML. Il diagramma mostra la sequenza delle interazioni che avvengono quando un sistema esterno chiede i dati di sintesi di una stazione meteorologica. I diagrammi di sequenza si leggono dall'alto verso il basso.

1. L'oggetto *ComunicSat* riceve una richiesta dal sistema informatico per avere un rapporto da una stazione meteorologica e accetta questa richiesta. La freccia stilizzata sul messaggio inviato indica che il sistema esterno non aspetta una risposta, ma può eseguire altre elaborazioni.

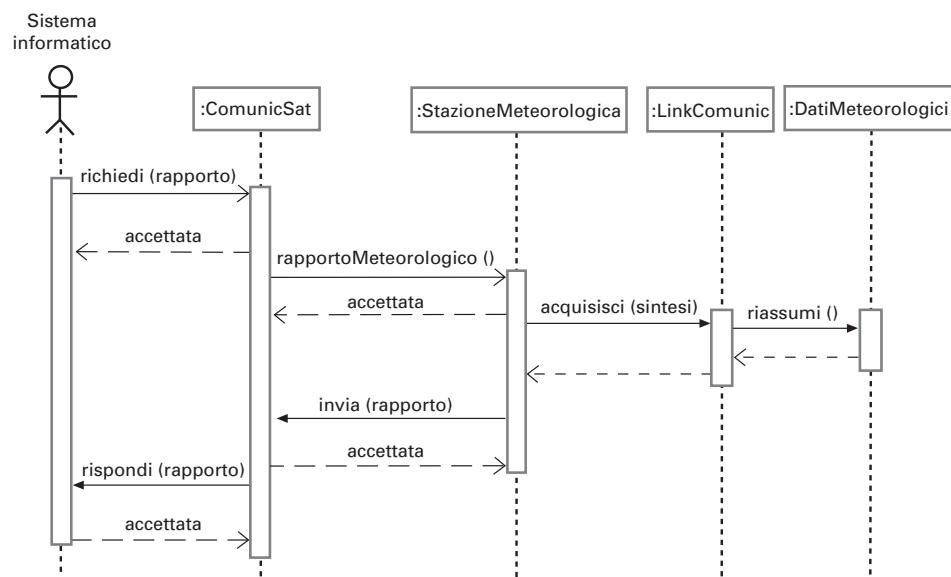


Figura 7.7 Diagramma di sequenza che descrive la raccolta dei dati.

2. *ComunicSat* invia un messaggio alla *StazioneMeteorologica*, tramite un collegamento satellitare, per creare una sintesi dei dati raccolti. Anche qui, la punta stilizzata della freccia sul messaggio inviato indica che *ComunicSat* non resta in attesa di una risposta.
3. *StazioneMeteorologica* invia un messaggio all'oggetto *LinkComunic* per riassumere i dati. In questo caso, la punta piena della freccia indica che l'istanza della classe di oggetti *StazioneMeteorologica* resta in attesa di una risposta.
4. *LinkComunic* chiama il metodo di sintesi dell'oggetto *DatiMeteorologici* e resta in attesa di una risposta.
5. Viene elaborata la sintesi dei dati meteorologici e il risultato viene inviato alla *StazioneMeteorologica* tramite l'oggetto *LinkComunic*.
6. La *StazioneMeteorologica* chiama l'oggetto *ComunicSat* per trasmettere i dati di sintesi al sistema informatico, tramite il sistema di comunicazione satellitare.

Gli oggetti *ComunicSat* e *StazioneMeteorologica* possono essere implementati come processi concorrenti, la cui esecuzione può essere sospesa e ripresa. L'istanza dell'oggetto *ComunicSat* ascolta i messaggi dal sistema esterno, li decodifica e avvia le operazioni della stazione meteorologica.

I diagrammi di sequenza sono utilizzati per modellare il comportamento combinato di un gruppo di oggetti, ma è possibile anche sintetizzare il comportamento di un oggetto o di un sottosistema come risposta a messaggi ed eventi. Per fare questo, si può utilizzare un modello di macchine a stati che mostra come l'istanza dell'oggetto cambia stato in funzione dei messaggi che riceve. Come detto nel Capitolo 5, il linguaggio UML include i diagrammi di stato per descrivere i modelli di macchine a stati.

La Figura 7.8 è un diagramma di stato per il sistema della stazione meteorologica che mostra come questo risponde alle richieste di vari servizi. Il diagramma può essere letto nel modo seguente:

1. se lo stato del sistema è *Spento*, il sistema può rispondere a un messaggio *riavvia()*, *riconfigura()* o *risparmiaEnergia()*. La freccia senza etichetta che parte dal pallino nero indica che *Spento* è lo stato iniziale. Un messaggio *riavvia()* provoca la transizione al normale funzionamento. Entrambi i messaggi *riconfigura()* e *risparmiaEnergia()* provocano la transizione a uno stato in cui il sistema si autoconfigura. Il diagramma di stato mostra che la riconfigurazione è consentita soltanto se il sistema è stato spento;
2. nello stato *Esecuzione*, il sistema attende ulteriori messaggi. Se riceve il messaggio *spegni()*, l'oggetto ritorna allo stato *Spento*;
3. se riceve il messaggio *rappportoMeteorologico()*, il sistema passa allo stato *Sintesi*. Quando la sintesi è completata, il sistema passa allo stato *Trasmis-*

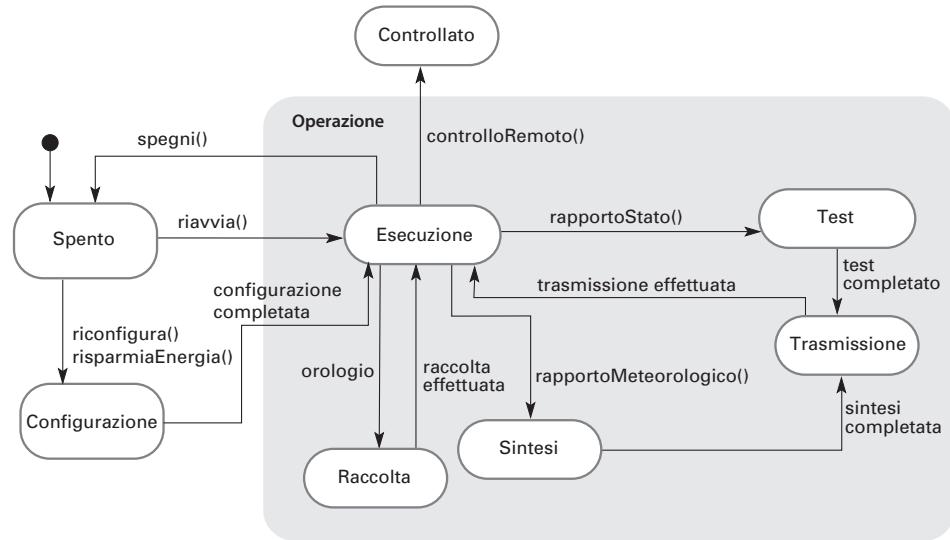


Figura 7.8 Diagramma di stato per la stazione meteorologica.

sione, dove le informazioni vengono trasmesse al sistema remoto; poi ritorna allo stato *Esecuzione*;

4. se riceve un segnale dall’orologio, il sistema passa allo stato *Raccolta*, dove raccoglie i dati dagli strumenti. Ogni strumento, a sua volta, è programmato per raccogliere i dati dai suoi sensori;
5. se riceva un messaggio `controlloRemoto()`, il sistema passa allo stato *Controllato*, in cui risponde a diversi gruppi di messaggi dal sistema di controllo remoto; questi messaggi non sono illustrati nel diagramma di stato.

I diagrammi di stato sono modelli di alto livello molto utili per rappresentare un sistema o le operazioni di un oggetto. Tuttavia, non occorre creare un diagramma di stato per ogni oggetto del sistema. Molti oggetti sono semplici, e le loro operazioni possono essere facilmente descritte senza un modello di stato.

7.1.5 Specifica delle interfacce

Una parte importante di ogni processo di progettazione è la specifica delle interfacce tra i componenti del progetto. Questa specifica è necessaria affinché gli oggetti e i sottosistemi possano essere progettati in parallelo. Una volta specificata un’interfaccia, gli sviluppatori degli altri oggetti possono presumere che tale interfaccia sarà implementata.

La progettazione delle interfacce riguarda la specifica dei suoi dettagli per un oggetto o un gruppo di oggetti. Ciò significa definire le firme e la semantica dei servizi che sono forniti dall’oggetto o dal gruppo di oggetti. Le interfacce possono essere specificate nel linguaggio UML utilizzando la stessa notazione di un dia-



Figura 7.9 Interfacce della stazione meteorologica.

gramma di classi, ma senza la sezione degli attributi, includendo lo stereotipo UML <<interface>> nella parte del nome. La semantica dell’interfaccia può essere definita tramite il linguaggio OCL (Object Constraint Language). L’uso di questo linguaggio è descritto nel Capitolo 16.

Si dovrebbe evitare di includere i dettagli della rappresentazione dei dati nel progetto dell’interfaccia, in quanto gli attributi non sono definiti nella specifica dell’interfaccia, ma dovrebbero essere fornite le operazioni per accedere e aggiornare i dati. Poiché la rappresentazione dei dati è nascosta, essa può essere facilmente cambiata senza influenzare gli oggetti che utilizzano tali dati. Questo porta a un progetto che intrinsecamente è più facile da mantenere; per esempio la rappresentazione di uno stack tramite array può essere modificata in una rappresentazione tramite liste, senza influenzare gli altri oggetti che utilizzano lo stack. D’altra parte, è più sensato esporre gli attributi in un modello di oggetti, poiché è il modo più chiaro di illustrare le caratteristiche essenziali degli oggetti.

Non c’è una semplice relazione 1:1 tra gli oggetti e le interfacce. Lo stesso oggetto può avere diverse interfacce, ognuna delle quali è un punto di vista dei metodi che fornisce. Questo viene supportato direttamente in Java, dove le interfacce sono dichiarate separatamente dagli oggetti e gli oggetti “implementano” le interfacce. Analogamente, si può accedere a un gruppo di oggetti attraverso una sola interfaccia.

La Figura 7.9 mostra due interfacce che possono essere definite per la stazione meteorologica. L’interfaccia di sinistra è un’interfaccia di reporting che definisce i nomi delle operazioni che sono utilizzate per generare i rapporti sulle condizioni meteo e sullo stato. Queste operazioni si mappano direttamente nell’oggetto *StazioneMeteorologica*. L’interfaccia del controllo remoto fornisce quattro operazioni, che si mappano con un singolo metodo nell’oggetto *StazioneMeteorologica*. In questo caso, le singole operazioni sono codificate nella stringa di comando associata al metodo *controlloRemoto*, mostrato nella Figura 7.6.

7.2 Design pattern (schemi di progettazione)

I **design pattern** derivano dalle idee proposte da Christopher Alexander (Alexander 1979), che suggerì l’esistenza di alcuni schemi di progettazione comuni, che

erano intrinsecamente gradevoli ed efficaci. Uno schema è la descrizione di un problema e l'essenza della sua soluzione, in modo che la soluzione possa essere riutilizzata in diverse impostazioni. Il design pattern non è una specifica dettagliata, ma si potrebbe definire come una descrizione delle conoscenze e delle esperienze accumulate, una soluzione sicura a un problema comune.

Una citazione dal sito web Hillside Group (hillside.net/patterns/), dedicato alla gestione di informazioni sugli schemi, incapsula il loro ruolo nel riutilizzo:

Gli schemi e i linguaggi di schematizzazione sono modi di descrivere le migliori procedure e i progetti più validi, e di acquisire esperienza in modo che possa essere riutilizzata da altri.

Gli schemi hanno un forte impatto sulla progettazione orientata agli oggetti. Oltre a essere utilizzati come soluzioni a problemi comuni, gli schemi sono diventati un complesso di termini che consente di discutere un progetto. È così possibile spiegare un progetto descrivendo gli schemi che sono stati utilizzati. Questo è particolarmente vero per gli schemi più noti che furono originariamente descritti dalla “Gang of Four” nel libro sugli schemi (Gamma et al. 1995). Altre importanti descrizioni degli schemi sono quelle pubblicate in una serie di libri da autori della Siemens (Buschmann et al. 1996; Schmidt et al. 2000; Kircher e Jain 2004; Buschmann, Henney e Schmidt 2007a, 2007b).

Gli schemi sono un modo di riutilizzare le conoscenze e le esperienze di altri progettisti; di solito sono associati alla progettazione orientata agli oggetti. Gli schemi pubblicati spesso si basano sulle caratteristiche degli oggetti, quali l'ereditarietà e il polimorfismo per essere generici. Tuttavia, il principio generale di incapsulare l'esperienza in uno schema è applicabile a qualsiasi tipo di progetto software. Per esempio, esistono schemi di configurazione per istanziare sistemi di applicazioni riutilizzabili.

La Gang of Four ha definito i quattro elementi fondamentali degli schemi di progettazione:

1. un nome significativo per riferirsi allo schema;
2. una descrizione dell'area del problema per spiegare quando lo schema può essere applicato;
3. una descrizione delle parti della soluzione del progetto, delle loro relazioni e responsabilità. Non è una descrizione concreta del progetto; è un modello per una soluzione del progetto che può essere istanziata in diversi modi. Viene solitamente espressa graficamente e mostra le relazioni tra gli oggetti e le classi degli oggetti.
4. Una dichiarazione delle conseguenze (i risultati e i compromessi) dell'applicazione dello schema. Questo può aiutare i progettisti a capire se uno schema può essere applicato efficacemente a una particolare situazione.

Nome del pattern: Observer

Descrizione: separa la visualizzazione dello stato di un oggetto dall'oggetto stesso e permette di fornire visualizzazioni alternative. Quando lo stato dell'oggetto cambia, tutte le visualizzazioni sono notificate automaticamente e aggiornate per riflettere il cambiamento.

Descrizione del problema: in molte situazioni è necessario fornire visualizzazioni multiple di alcune informazioni di stato, come le visualizzazioni grafiche e tabulari. Non tutte queste possono essere note quando si specificano le informazioni. Tutte le visualizzazioni alternative possono supportare le interazioni e, quando lo stato cambia, devono essere aggiornate tutte. Questo schema può essere utilizzato in tutti i casi in cui è richiesto più di un formato di visualizzazione delle informazioni di stato e quando non è necessario che l'oggetto che mantiene tali informazioni conosca i formati specifici di visualizzazione utilizzati.

Descrizione della soluzione: definisce due oggetti astratti, Soggetto e Osservatore, e due oggetti concreti, SoggettoConcreto e OsservatoreConcreto, che ereditano gli attributi dei relativi oggetti astratti. Lo stato da visualizzare è mantenuto nel SoggettoConcreto, che eredita anche le operazioni dal Soggetto che permettono di aggiungere e rimuovere gli Osservatori (ciascun osservatore corrisponde a una visualizzazione) e inviare una notifica quando lo stato cambia.

L'OggettoConcreto conserva una copia dello stato del SoggettoConcreto e implementa l'interfaccia `Aggiorna()` dell'Osservatore che permette alle varie copie di essere aggiornate. L'OsservatoreConcreto visualizza automaticamente lo stato e riflette le modifiche quando lo stato viene aggiornato.

Il modello UML dello schema è illustrato nella Figura 7.12.

Conseguenze: il soggetto conosce soltanto l'Osservatore astratto, ma non conosce i dettagli della classe concreta; dunque c'è un accoppiamento minimo tra questi oggetti. A causa di questa mancanza di conoscenza, le ottimizzazioni che migliorano le prestazioni della visualizzazione sono impraticabili. Le modifiche del soggetto possono causare una serie di aggiornamenti degli osservatori, alcuni dei quali potrebbero non essere necessari.

Figura 7.10 Lo schema Observer.

Gamma e i suoi coautori hanno suddiviso la descrizione di un problema in motivazione (una descrizione del perché lo schema è utile) e applicabilità (una descrizione dei casi in cui può essere applicato lo schema). La descrizione della soluzione include la struttura dello schema, i partecipanti, le collaborazioni e l'implementazione.

Per illustrare la descrizione di uno schema, utilizzerò lo schema Observer (letteralmente “Osservatore”), tratto dal libro di Gamma. Questo schema è illustrato nella Figura 7.10. La mia descrizione usa quattro elementi essenziali e una frase che spiega che cosa può fare lo schema. Lo schema può essere utilizzato in casi in cui sono richiesti diverse presentazioni dello stato di un oggetto; separa l'oggetto che deve essere visualizzato dalle diverse forme di presentazione. Questo è illustrato nella Figura 7.11, che mostra due presentazioni grafiche dello stesso insieme di dati.

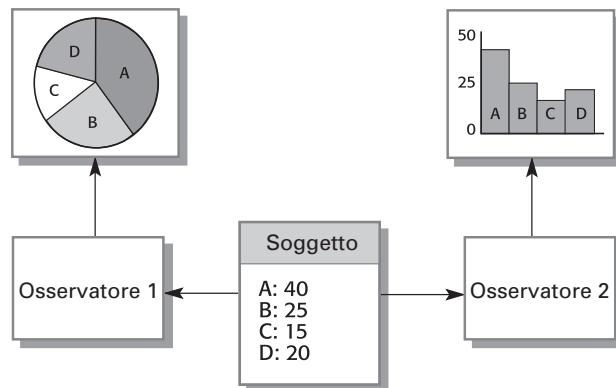


Figura 7.11 Visualizzazioni multiple.

Le rappresentazioni grafiche di solito sono utilizzate per illustrare le classi di oggetti negli schemi e le loro relazioni. Queste rappresentazioni sono un supplemento alla descrizione degli schemi e aggiungono nuovi dettagli alla descrizione della soluzione. La Figura 7.12 è la rappresentazione dello schema Observer nel linguaggio UML.

Per usare gli schemi nei vostri progetti, tenete presente che qualsiasi problema di progettazione potrebbe avere uno schema associato che può essere applicato. Ecco alcuni esempi di tali problemi, documentati nel libro di Gamma:

1. dire a vari oggetti che lo stato di qualche altro oggetto è cambiato (schema Observer);
2. ordinare le interfacce con un numero di oggetti correlati che spesso devono essere sviluppati in modo incrementale (schema Façade);
3. fornire un modo standard di accedere agli elementi di una collezione, indipendentemente dal modo in cui la collezione viene implementata (schema Iterator);

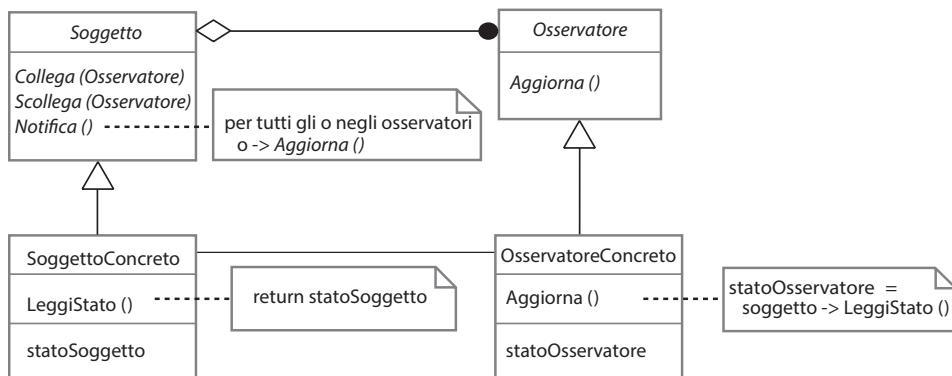


Figura 7.12 Un modello UML dello schema Observer.

4. dare la possibilità di estendere la funzionalità di una classe esistente a runtime (schema Decorator).

Gli schemi di progettazione supportano il riutilizzo dei concetti di alto livello. Quando tentate di riutilizzare componenti eseguibili siete inevitabilmente vincolati dalle decisioni dettagliate di progettazione che sono state prese dagli implementatori di questi componenti, a partire dai particolari algoritmi che sono stati utilizzati per implementare i componenti, per finire agli oggetti e ai tipi di interfacce dei componenti. Se queste decisioni di progettazione entrano in conflitto con le vostre esigenze, il riutilizzo dei componenti diventa impossibile o introduce delle inefficienze nel vostro sistema. Applicare gli schemi significa riutilizzare le idee, ma potete adattare l'implementazione per soddisfare i requisiti del sistema che state sviluppando.

Quando iniziate a progettare un sistema, è difficile sapere in anticipo se avrete bisogno di un particolare schema. Di conseguenza, usare gli schemi è un processo che spesso richiede lo sviluppo di un progetto, la conoscenza di un problema e la capacità di identificare lo schema che può essere applicato. Questo è certamente possibile se vi limitate ai 23 schemi generali documentati nel libro originale degli schemi; ma, se il vostro problema è differente, sarà difficile trovare uno schema appropriato tra le centinaia di schemi differenti che sono stati proposti.

Gli schemi di progettazione sono una grande idea, ma occorre esperienza di progettazione del software per poterli utilizzare efficacemente. Occorre saper riconoscere i casi in cui uno schema può essere applicato. I programmati insperti, anche se hanno studiato questi schemi sui libri, avranno sempre qualche difficoltà nel decidere se sia possibile riutilizzare uno schema o se sia necessario sviluppare una soluzione specifica.

7.3 Problemi di implementazione

L'ingegneria del software include tutte le attività necessarie allo sviluppo del software, dalla definizione dei requisiti iniziali del sistema alla manutenzione e alla gestione del sistema sviluppato. Uno stadio critico di questo processo è, ovviamente, l'implementazione del sistema, dove create una vostra versione eseguibile del software. L'implementazione potrebbe richiedere lo sviluppo di programmi in linguaggi di programmazione di alto o basso livello o l'adattamento di generici sistemi off-the-shelf per soddisfare le specifiche richieste di un'organizzazione.

Suppongo che molti lettori di questo libro conoscano i principi di programmazione e abbiano qualche esperienza di programmazione. Dal momento che questo capitolo è ideato per offrire un approccio indipendente dai linguaggi di programmazione, non farò riferimento ai problemi della buona pratica di programmazione che richiedono l'uso di esempi di un linguaggio specifico. Presenterò invece alcuni aspetti dell'implementazione che sono particolarmente importanti nell'in-

gegneria del software e che spesso non sono trattati nei testi di programmazione; qui di seguito sono elencati i principali aspetti dell’implementazione.

1. *Riutilizzo*. La maggior parte del software moderno viene costruito riutilizzando componenti o sistemi esistenti. Quando sviluppare software, dovreste utilizzare quanto più possibile il codice esistente;
2. *Gestione della configurazione*. Durante il processo di sviluppo, vengono create varie versioni di ciascun componente software. Se non utilizzate un sistema di gestione della configurazione per tenere traccia di queste versioni, correte il rischio di includere le versioni errate di questi componenti nel vostro sistema;
3. *Sviluppo host-target*. Il software prodotto di solito non viene eseguito sullo stesso computer in cui è stato sviluppato; piuttosto, il software viene sviluppato in un computer (sistema host) ed eseguito in un altro computer (sistema target). I sistemi host e target a volte sono dello stesso tipo, ma spesso sono completamente differenti.

7.3.1 Riutilizzo

Dagli anni ’60 agli anni ’90 la maggior parte del nuovo software è stata sviluppata partendo da zero, scrivendo tutto il codice in un linguaggio di programmazione di alto livello. L’unico significativo riutilizzo del software consisteva nel riutilizzare funzioni e oggetti delle librerie dei linguaggi di programmazione. La riduzione dei costi e dei tempi di produzione rese questo approccio sempre più impraticabile, specialmente per i sistemi commerciali e per quelli basati su Internet. Per questo, oggi un approccio allo sviluppo basato sul riutilizzo del software esistente è la norma per molti tipi di sviluppo del software. Un approccio basato sul riutilizzo è adesso largamente adottato per tutti i tipi di sistemi basati sul Web, il software scientifico e, sempre più, nell’ingegneria dei sistemi integrati.

Il riutilizzo del software è possibile a vari livelli, come illustra la Figura 7.13.

1. *Livello di astrazione*. A questo livello, non viene riutilizzato direttamente il software, ma le conoscenze di astrazioni nella progettazione del software. Gli schemi di progettazione e gli schemi architettonici (trattati nel Capitolo 6) sono modi di rappresentare le conoscenze astratte per il riutilizzo del software.
2. *Livello degli oggetti*. A questo livello, riutilizzate direttamente gli oggetti da una libreria, anziché scrivere il codice. Per implementare questo tipo di riutilizzo, dovete trovare librerie appropriate e scoprire se gli oggetti e i metodi offrono le funzionalità che vi servono. Per esempio, se avete bisogno di elaborare messaggi di posta elettronica in un programma Java, potreste usare gli oggetti e i metodi di una libreria JavaMail.
3. *Livello dei componenti*. I componenti sono raccolte di oggetti e classi di oggetti che operano insieme per fornire funzioni e servizi. Molte volte

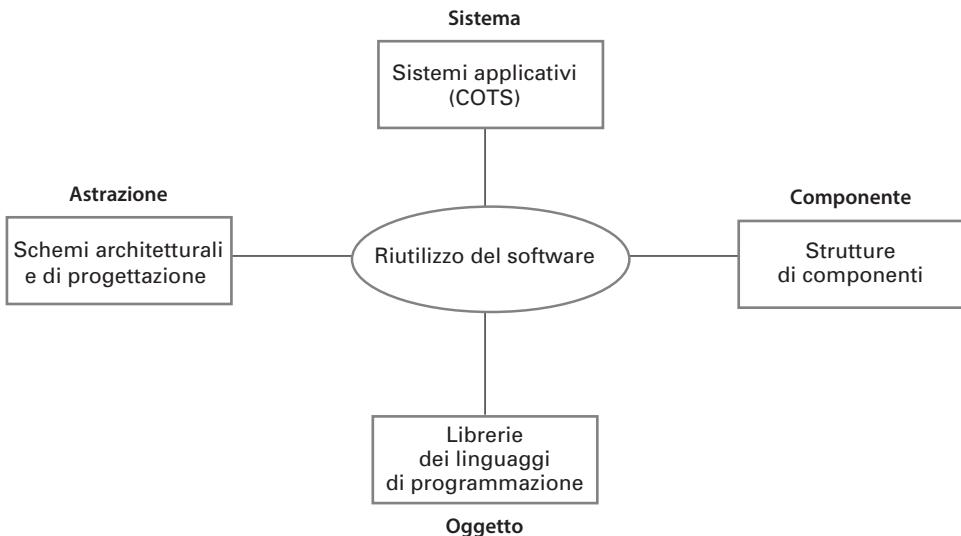


Figura 7.13 Riutilizzo del software.

dovrete adattare ed estendere il componente aggiungendo un po' del vostro codice. Un esempio di riutilizzo a livello dei componenti è quando costruite un'interfaccia utente utilizzando un framework, che è un insieme di classi di oggetti generici che implementano la gestione degli eventi, la gestione delle visualizzazioni ecc. Aggiungete i collegamenti ai dati da visualizzare e scrivete il codice per definire i dettagli di visualizzazione, quali il layout e i colori dello schermo.

4. *Livello del sistema.* A questo livello, riutilizzate interi sistemi applicativi. Questa operazione di solito richiede qualche forma di configurazione del sistema. Ciò può essere fatto aggiungendo e modificando il codice (se state riutilizzando una linea di prodotti software) oppure utilizzando l'interfaccia di configurazione propria del sistema. Molti sistemi commerciali oggi sono costruiti in questo modo, adattando e riutilizzando sistemi di sistemi applicativi generici. A volte questo approccio può richiedere l'integrazione di vari sistemi applicativi per creare un nuovo sistema.

Riutilizzando il software esistente, potete sviluppare nuovi sistemi più rapidamente, con minori rischi di sviluppo e a costi inferiori. Poiché il software riutilizzato è stato provato in altre applicazioni, esso dovrebbe essere più affidabile del nuovo software. Tuttavia, il riutilizzo comporta alcuni costi.

1. Il costo del tempo impiegato per cercare il software da riutilizzare e per stabilire se esso soddisfa le vostre esigenze. Potrebbe essere necessario provare il software per essere sicuri che esso potrà funzionare nel vostro ambiente, specialmente se questo è diverso dall'ambiente in cui il software è stato sviluppato.

2. In alcuni casi, il costo di acquisto del software riutilizzabile. Per grandi sistemi off-the-shelf, questo costo potrebbe essere molto alto.
3. I costi per adattare e configurare i componenti o i sistemi del software riutilizzabile per soddisfare i requisiti del sistema che state sviluppando.
4. I costi per integrare gli elementi del software riutilizzabile tra loro (se state utilizzando software proveniente da sorgenti differenti) e con il nuovo codice che avete sviluppato. Integrare il software riutilizzabile proveniente da vari fornitori può essere difficile e costoso, in quanto i vari fornitori potrebbero fare ipotesi contrastanti sul modo in cui il loro software sarà riutilizzato.

Come riutilizzare le conoscenze e il software esistente dovrebbe essere la prima cosa cui pensare quando si avvia un progetto di sviluppo del software. Bisogna valutare le possibilità di riutilizzo prima di progettare in dettaglio il software, in quanto potrebbe essere conveniente adattare il proprio progetto riutilizzando il software esistente. Come detto nel Capitolo 2, in un processo di sviluppo orientato al riutilizzo, occorre prima cercare gli elementi riutilizzabili, poi modificare i requisiti e il progetto per utilizzare al meglio questi elementi.

Data l'enorme importanza che il riutilizzo del software riveste nella moderna ingegneria del software, ho dedicato a questo argomento tre capitoli di questo libro (Capitoli 15, 16 e 18).

7.3.2 Gestione della configurazione

Durante lo sviluppo del software, le modifiche ci sono sempre, quindi è assolutamente essenziale poterle gestire. Quando più persone sono coinvolte nello sviluppo di un sistema software, è importante che i membri del team non interferiscano fra loro. In altri termini, se due persone stanno lavorando su un componente, le loro modifiche devono essere coordinate, altrimenti un programmatore potrebbe apportare delle modifiche che annullano il lavoro di un altro. Occorre anche accertarsi che chiunque possa accedere alle versioni più aggiornate dei componenti software, altrimenti gli sviluppatori potrebbero rifare un lavoro già fatto. Se qualcuno introduce un errore nella nuova versione di un sistema, deve essere possibile ripristinare una precedente versione corretta del sistema o del componente.

La gestione della configurazione è il nome dato al processo generale di gestione di un sistema software che cambia. Scopo della gestione della configurazione è supportare il processo di integrazione del sistema in modo che tutti gli sviluppatori possano accedere al codice e ai documenti del progetto in modo controllato, trovare le modifiche che sono state apportate, compilare e collegare i componenti per creare un sistema. Come mostra la Figura 7.14, ci sono quattro attività fondamentali nella gestione della configurazione.



Figura 7.14 Gestione della configurazione.

1. *Gestione delle versioni.* Il supporto è fornito per tenere traccia delle differenti versioni dei componenti software. I sistemi di gestione delle versioni includono le funzioni per coordinare lo sviluppo di più programmati; impediscono a uno sviluppatore di sovrascrivere il codice che è stato creato da qualcun altro.
2. *Integrazione del sistema.* Il supporto è fornito per aiutare gli sviluppatori a definire quali versioni di componenti sono utilizzate per creare le singole versioni del sistema. Questa descrizione viene poi utilizzata per costruire un sistema automaticamente, compilando e collegando i componenti richiesti.
3. *Registrazione dei problemi.* Il supporto è fornito per consentire agli utenti di segnalare i bug del codice e altri problemi, e per permettere a tutti gli sviluppatori di vedere chi sta risolvendo questi problemi e quando sono stati risolti.
4. *Gestione delle release.* Le nuove versioni di un sistema software vengono rilasciate ai clienti. La gestione delle release si occupa di pianificare le funzionalità delle nuove release e di organizzare la distribuzione del software.

Ogni strumento di gestione della configurazione del software supporta tutte queste attività. Questi strumenti di solito sono installati in un ambiente di sviluppo integrato, come Eclipse. La gestione delle versioni può essere supportata utilizzando un sistema di gestione delle versioni, come Subversion (Pilato, Collins-Sussman e Fitzpatrick 2008) o Git (Loeliger e McCullough 2012), che sono in grado di supportare lo sviluppo multi-team e multi-site. Il supporto all'integrazione del sistema può essere costruito nel linguaggio o essere affidato a un apposito toolset, come GNU build system. I sistemi di registrazione dei bug e dei problemi, come Bugzilla, sono utilizzati per documentare i bug e altri problemi e per sapere se questi problemi sono stati risolti. Un set completo di strumenti basati sul sistema Git è disponibile nel sito Github (<http://github.com>).

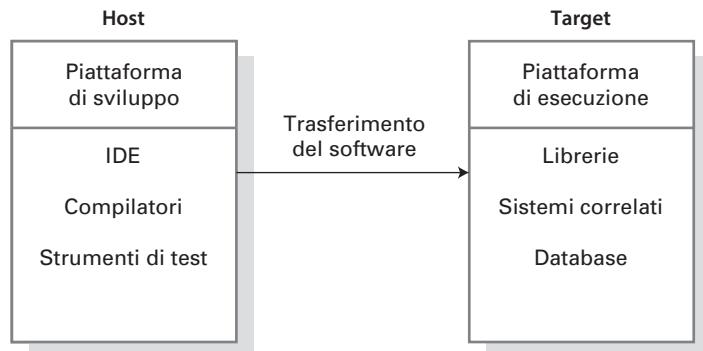


Figura 7.15 Sviluppo host-target.

Data la sua importanza nell’ingegneria del software professionale, nel Capitolo 22 ho trattato più dettagliatamente la gestione della configurazione e delle modifiche.

7.3.3 Sviluppo host-target

Gran parte dello sviluppo del software professionale è basato sul modello host-target (Figura 7.15). Il software viene sviluppato in un computer (host), ma viene eseguito su una macchina separata (target). Più in generale, possiamo parlare di piattaforma di sviluppo (host) e di piattaforma di esecuzione (target). Una piattaforma è molto più del semplice hardware; essa include il sistema operativo più altro software di supporto, come il sistema di gestione dei database o, per le piattaforme di sviluppo, un ambiente di sviluppo interattivo.

A volte, la piattaforma di sviluppo è la stessa di quella di esecuzione, quindi è possibile sviluppare il software e provarlo nella stessa macchina. Per esempio, se state sviluppando il software in Java, l’ambiente target è la macchina virtuale Java. In teoria, questa è la stessa su tutti i computer, quindi i programmi dovrebbero essere portabili da una macchina all’altra. Tuttavia, in particolare per i sistemi integrati e i sistemi mobili, la piattaforma di sviluppo è diversa da quella di esecuzione; in questi casi, occorre trasferire il software sviluppato nella piattaforma di esecuzione per provarlo oppure utilizzare un simulatore nella macchina di sviluppo.

I simulatori vengono spesso utilizzati per sviluppare i sistemi integrati. È possibile simulare i dispositivi hardware, come i sensori, e gli eventi nell’ambiente in cui il sistema viene sviluppato. I simulatori accelerano il processo di sviluppo per i sistemi integrati, in quanto ogni sviluppatore può avere la sua piattaforma di esecuzione, senza bisogno di scaricare il software nell’hardware target. I simulatori purtroppo sono costosi da sviluppare e, quindi, di solito sono disponibili soltanto per le architetture hardware più popolari.

Se nel sistema target è stato installato è un software middleware o di altro tipo che avete bisogno di utilizzare, allora dovete essere in grado di provare il sistema mediante tale software. Potrebbe essere impraticabile installare tale software nella vostra macchina di sviluppo, anche se è uguale alla piattaforma target, a causa dei vincoli della licenza d'uso. In questo caso, dovete trasferire il codice che avete sviluppato sulla piattaforma di esecuzione per provare il sistema.

Una piattaforma di sviluppo del software dovrebbe fornire una serie di strumenti che supportano i processi di ingegneria del software, come i seguenti:

1. un compilatore integrato e un sistema di editing guidato dalla sintassi che consente di creare, modificare e compilare il codice;
2. un sistema di debugging del linguaggio;
3. strumenti di editing grafico, come quelli per modificare i modelli UML;
4. strumenti di test, come JUnit, che possono eseguire automaticamente una serie di test sulla nuova versione di un programma;
5. strumenti che supportano il refactoring e la visualizzazione dei programmi;
6. strumenti per la gestione della configurazione per gestire le versioni del codice e per integrare e compilare i sistemi.

Oltre a questi strumenti standard, il vostro sistema di sviluppo potrebbe includere altri strumenti specializzati, come gli analizzatori statici (descritti nel Capitolo 12). Normalmente, gli ambienti di sviluppo per team includono un server condiviso che dispone di un sistema di gestione della configurazione e delle modifiche e, a volte, anche un sistema di supporto alla gestione dei requisiti.

Gli strumenti di sviluppo del software oggi, di solito, sono installati all'interno di un ambiente di sviluppo integrato (Integrated Development Environment o IDE). Un IDE è un insieme di strumenti software che supportano vari aspetti dello sviluppo del software all'interno di qualche framework e interfaccia utente comuni. In generale, gli IDE sono creati per supportare lo sviluppo secondo uno specifico linguaggio di programmazione, come Java. L'IDE del linguaggio può essere sviluppato in modo speciale oppure può essere l'istanza di un IDE generico, con strumenti di supporto per uno specifico linguaggio.

Un IDE generico è un framework di strumenti host che offre le funzioni di gestione dei dati per il software da sviluppare e i meccanismi di integrazione che consentono agli strumenti di operare insieme. L'IDE generico più noto è l'ambiente Eclipse (<http://www.eclipse.org>); questo ambiente si basa su un'architettura di plug-in che può essere personalizzata per vari linguaggi, come Java, e domini di applicazioni. Pertanto, potete installare Eclipse e adattarlo alle vostre specifiche esigenze aggiungendo i plug-in. Per esempio, potete aggiungere un set di plug-in per supportare lo sviluppo di sistemi in rete in Java (Vogel 2013) o l'ingegneria di sistemi integrati tramite il linguaggio C.

Diagrammi di rilascio UML

I diagrammi di rilascio UML mostrano come i componenti software vengono fisicamente rilasciati nei processori. Un diagramma di rilascio mostra l'hardware e il software di un sistema e il middleware utilizzato per collegare i vari componenti del sistema. In essenza, un diagramma di rilascio può essere immaginato come un modo per definire e documentare l'ambiente target.

<http://software-engineering-book.com/web/deployment/>

Come parte del processo di sviluppo, dovete prendere delle decisioni su come il software sviluppato sarà rilasciato nella piattaforma target. Questo è semplice per i sistemi integrati, dove la piattaforma target di solito è un singolo computer. Per i sistemi distribuiti, invece, dovete scegliere le piattaforme specifiche sulle quali i componenti saranno rilasciati. I temi da affrontare per fare queste scelte sono:

1. *requisiti hardware e software di un componente.* Se un componente è progettato per una specifica architettura hardware o si basa su qualche altro sistema software, ovviamente dovrà essere rilasciato su una piattaforma che fornisce il supporto hardware e software richiesto;
2. *requisiti di disponibilità del sistema.* I sistemi ad alta disponibilità possono richiedere componenti da rilasciare su più piattaforme. Questo significa che, nel caso di malfunzionamento di una piattaforma, è disponibile un'implementazione alternativa del componente;
3. *comunicazione tra i componenti.* Se la comunicazione tra i componenti è intensa, di solito è preferibile rilasciarli sulla stessa piattaforma o su piattaforme che sono fisicamente vicine l'una all'altra. Questo riduce la latenza delle comunicazioni – il tempo che passa da quando un componente invia un messaggio a quando un altro componente riceve il messaggio.

Potete documentare le vostre decisioni sullo sviluppo dell'hardware e del software utilizzando i diagrammi di rilascio UML, che mostrano come i componenti software sono distribuiti tra le piattaforme hardware.

Se state sviluppando un sistema integrato, potrebbe essere necessario valutare le caratteristiche della piattaforma target, come le dimensioni fisiche, la potenza, la necessità di avere risposte in tempo reale agli eventi dei sensori, le caratteristiche fisiche degli attuatori e il sistema operativo in tempo reale.

7.4 Sviluppo open-source

Lo sviluppo open-source è un approccio allo sviluppo del software in cui il codice sorgente di un sistema software è pubblico e i volontari sono invitati a partecipare al processo di sviluppo (Raymond 2001). Le sue radici si trovano nella Free Software Foundation (www.fsf.org), che sostiene che il codice sorgente non deve

essere di proprietà esclusiva di qualcuno, ma a disposizione degli utenti che possono esaminarlo e modificarlo come vogliono. Si supponeva che il codice sarebbe stato controllato e sviluppato da un piccolo gruppo, anziché da tutti gli utenti del codice.

Il software open-source ha esteso questa idea tramite Internet per reclutare un maggior numero di sviluppatori volontari. Molti di questi sono anche utenti del codice. In teoria, chiunque collabori a un progetto open-source può evidenziare e correggere i bug e proporre nuove caratteristiche e funzionalità. In pratica, però, i sistemi open-source più affermati si affidano a un gruppo ristretto di sviluppatori che controllano le modifiche apportate al software.

Il software open-source è la colonna portante di Internet e dell'ingegneria del software. Il sistema operativo Linux è il più diffuso sistema per server, perché è un web server Apache open-source. Altri prodotti open-source importanti e universalmente utilizzati sono Java, l'IDE Eclipse e il sistema di gestione dei database mySQL. Il sistema operativo Android è installato su milioni di dispositivi mobili. I principali protagonisti dell'industria informatica, come IBM e Oracle, supportano il movimento open-source e basano il loro software sui prodotti open-source. Possono essere utilizzati anche moltissimi altri sistemi e componenti meno noti come prodotti open-source.

Di solito è più economico o perfino gratuito procurarsi un software open-source. Normalmente è possibile scaricare senza costi un software open-source. Ma, se volete un supporto tecnico o una documentazione, allora dovete pagare per questo, ma i costi di solito sono molto contenuti. L'altro vantaggio chiave è che i sistemi open-source più diffusi sono molto affidabili. Hanno una vasta popolazione di utenti che desiderano risolvere autonomamente i loro problemi, anziché segnalare i problemi agli sviluppatori e attendere una nuova release del sistema. I bug vengono scoperti e riparati molto più velocemente rispetto al software proprietario.

Per una società che si occupa di sviluppo di software, ci sono due argomenti che devono essere considerati.

1. Il prodotto che si sta sviluppando dovrà utilizzare componenti open-source?
2. Dovrà essere adottato un approccio open-source nello sviluppo dei propri prodotti software?

Le risposte a queste domande dipendono dal tipo di software da sviluppare e dall'esperienza dei membri del team di sviluppo.

Se state sviluppando un prodotto software da vendere, allora i tempi per immetterlo sul mercato e il contenimento dei costi sono fattori critici. Se state sviluppando un software in un dominio in cui sono disponibili sistemi open-source di alta qualità, potete risparmiare tempo e denaro utilizzando questi sistemi. Se, invece, state sviluppando un software per soddisfare una serie specifica di requisiti di un'organizzazione, allora potrebbe non essere appropriato utilizzare componenti open-source. Potreste essere costretti a integrare il vostro software

con sistemi esistenti che non sono compatibili con i sistemi open-source disponibili. Nonostante ciò, tuttavia, potrebbe essere più rapido ed economico modificare il sistema open-source, anziché sviluppare le funzionalità che vi servono.

Molte aziende di prodotti software oggi adottano un approccio open-source allo sviluppo, specialmente per i sistemi specializzati. Il loro modello di business non si basa sulla vendita di un prodotto software, ma sul supporto alla vendita di tale prodotto. Essi credono che coinvolgendo la comunità open-source sia possibile sviluppare il software in modo più rapido ed economico e creare una comunità di utenti per il software.

Alcune aziende ritengono che, adottando un approccio open-source, vengano svelate informazioni riservate ai loro concorrenti e, quindi, sono riluttanti ad adottare tale approccio. Se, invece, state lavorando con una piccola azienda e adottate un approccio open-source, questo potrebbe assicurare i clienti che, anche in caso di fallimento dell’azienda, il software continuerà ad avere un supporto tecnico.

Rendere pubblico il codice sorgente di un sistema non significa che le persone di una vasta comunità forniranno il loro contributo allo sviluppo del sistema. Molti prodotti open-source di successo sono stati prodotti di piattaforme anziché sistemi di applicazioni. Ci sono pochi sviluppatori che potrebbero essere interessati a sistemi di applicazioni specializzate. Rendere un sistema open-source non garantisce la collaborazione di una comunità di sviluppatori. Ci sono migliaia di progetti open-source su Sourceforge e GitHub che hanno solo una manciata di download. Tuttavia, se gli utenti del vostro software sono interessati alla sua futura disponibilità, rendere il software open-source significa che essi potranno scaricarne una copia ed avere così la garanzia che non perderanno l’accesso al software in futuro.

7.4.1 Licenze open-source

Sebbene un principio fondamentale dello sviluppo open-source sia che il codice sorgente deve essere disponibile gratuitamente, questo non significa che chiunque possa fare ciò che vuole con quel codice. Legalmente, lo sviluppatore del codice (un’azienda o un individuo) è proprietario del codice; può imporre delle restrizioni al modo in cui il codice deve essere utilizzato, includendo condizioni legalmente vincolanti in una licenza open-source (St. Laurent 2004). Alcuni sviluppatori open-source suppongono che, se un componente open-source viene utilizzato per sviluppare un nuovo sistema, allora anche questo sistema deve essere open-source. Altri sono disposti a consentire che il loro codice sia utilizzato senza questa restrizione. I sistemi sviluppati possono essere proprietari e venduti come sistemi *closed-source*.

Molte licenze open-source (Chapman 2010) sono varianti di uno dei seguenti schemi generali.

1. GNU General Public License (GPL). Si tratta della cosiddetta licenza reciproca, che semplicemente significa che chi utilizza un software open-source con la licenza GPL deve rendere tale software open-source.
2. GNU Lesser General Public License (LGPL). È una variante della licenza GPL; consente di scrivere componenti che sono collegati a un codice open-source senza l'obbligo di rendere pubblico il codice sorgente di questi componenti. Ma, se viene modificato il componente oggetto della licenza, allora questo deve essere reso pubblico come open-source.
3. Licenza Berkley Standard Distribution (BSD). È una licenza non reciproca, nel senso che non c'è l'obbligo di rendere pubbliche le modifiche apportate al codice open-source. È possibile includere il codice nei sistemi proprietari che vengono venduti. Chi usa componenti open-source deve avvisare il creatore originale del codice. La licenza MIT è una variante della licenza BSD con condizioni analoghe.

Questi argomenti sono importanti perché, se utilizzate un software open-source come parte di un prodotto software, siete obbligati dai termini della licenza a rendere open-source il vostro prodotto. Se volete vendere il vostro software, potreste preferire di mantenere segreto il suo codice; questo significa che per svilupparlo non potete utilizzare un software open-source con licenza GPL.

Se state creando un software che viene eseguito su una piattaforma open-source senza riutilizzare componenti open-source, allora le licenze non sono un problema. Se, invece, integrate un software open-source nel vostro software, allora avete bisogno di processi e database per tenere traccia di ciò che avete utilizzato e delle condizioni previste nelle loro licenze. Bayersdorfer (Bayersdorfer 2007) suggerisce che le aziende che gestiscono progetti che usano componenti open-source debbano:

1. stabilire un sistema per mantenere le informazioni sui componenti open-source che vengono scaricati e utilizzati. Occorre avere una copia della licenza per ogni componente che era valida nel momento in cui un componente è stato utilizzato. Le licenze possono cambiare, quindi è importante conoscere le condizioni che sono state accettate;
2. conoscere i vari tipi di licenze e capire i vincoli della licenza per un componente prima di utilizzarlo. Si può utilizzare un componente in un sistema, ma non in un altro, perché si pensa di utilizzare questi sistemi in modi differenti;
3. conoscere i percorsi evolutivi dei componenti. È necessario conoscere il progetto open-source dove i componenti vengono sviluppati per capire come potrebbero cambiare in futuro;
4. istruire le persone sulle licenze open-source. Non basta avere procedure per garantire il rispetto delle condizioni delle licenze; occorre anche istruire gli sviluppatori sul codice open-source e sulle licenze che lo regolano;

5. avere dei sistemi di auditing. Gli sviluppatori, se sottoposti a scadenze pressanti, potrebbero essere tentati di non rispettare i vincoli di una licenza; per evitare questo, sarebbe opportuno avere un apposito software;
6. far parte della comunità open-source. Se ci si affida ai prodotti open-source, sarebbe opportuno fare parte della comunità e contribuire a sostenere il loro sviluppo.

L'approccio open-source è uno dei vari modelli di business per il software. Secondo questo modello, le aziende rilasciano il codice sorgente del loro software e vendono i relativi servizi e l'assistenza richiesta. Possono anche vendere servizi software basati su cloud – un'opzione attraente per gli utenti che non hanno l'esperienza necessaria per gestire i loro sistemi open-source e le versioni specializzate dei loro sistemi per particolari clienti. L'open-source sembra quindi destinato ad aumentare di importanza come modo di sviluppare e distribuire il software.

Punti chiave

- La progettazione e l'implementazione del software sono attività interacciate. Il livello di dettagli di un progetto dipende dal tipo di sistema da sviluppare e dal metodo di sviluppo (agile o guidato da piani).
- Il processo della progettazione orientata agli oggetti include le attività per progettare l'architettura di un sistema, identificare gli oggetti nel sistema, descrivere il progetto utilizzando vari modelli di oggetti e documentare le interfacce dei componenti.
- Durante un processo di progettazione orientata agli oggetti possono essere prodotti vari modelli, tra cui i modelli statici (modelli di classi, modelli di generalizzazione, modelli di associazione) e dinamici (modelli di sequenza, modelli di macchine a stati).
- Le interfacce dei componenti devono essere definite con precisione in modo che altri oggetti possano utilizzarle. È possibile utilizzare uno stereotipo UML per definire le interfacce.
- Quando si sviluppa un software, occorre sempre considerare la possibilità di riutilizzare il software esistente, come componenti, servizi o sistemi completi.
- La gestione della configurazione è il processo che gestisce le modifiche di un sistema software in evoluzione; è essenziale quando un team di persone collabora allo sviluppo di un software.
- Molti processi di sviluppo del software sono del tipo host-target. Si usa un IDE su una macchina host per sviluppare il software, che poi viene trasferito su una macchina target per essere eseguito.
- Lo sviluppo open-source richiede che il codice sorgente di un sistema sia disponibile a tutti gli utenti. Questo significa che molte persone possono proporre modifiche e miglioramenti del software.

Esercizi

- * 7.1 Utilizzando lo schema tabulare della Figura 7.3, specificate i casi d'uso *Rapporto sullo stato* e *Riconfigurazione* della stazione meteorologica, facendo delle valide ipotesi sulle funzionalità che qui sono richieste.
- 7.2 Supponete di sviluppare il sistema Mentcare utilizzando un approccio orientato agli oggetti. Tracciate un diagramma che mostri almeno sei possibili casi d'uso per questo sistema.
- * 7.3 Utilizzando la notazione grafica UML per le classi di oggetti, progettate le seguenti classi di oggetti, identificando gli attributi e le operazioni. In base alla vostra esperienza, decidete gli attributi e le operazioni da associare a questi oggetti:
 - un sistema di messaggi per un tablet o un telefono cellulare;
 - una stampante per un personal computer;
 - un sistema di musica personale;
 - un conto corrente bancario;
 - il catalogo di una libreria.
- 7.4 Utilizzando gli oggetti della stazione meteorologica identificati nella Figura 7.6 come punto di partenza, identificate altri oggetti che possono essere utilizzati in questo sistema. Progettate una gerarchia di ereditarietà per gli oggetti che avete identificato.
- 7.5 Sviluppate il progetto della stazione meteorologica per mostrare l'interazione tra il sottosistema di raccolta dei dati e gli strumenti che li forniscono. Utilizzate i diagrammi di sequenza per mostrare questa interazione.
- 7.6 Identificate i possibili oggetti nei seguenti sistemi e sviluppate un progetto orientato agli oggetti per ciascuno di essi. Fate delle valide ipotesi sui sistemi quando derivate il progetto.
 - Un sistema di gestione di un'agenda di gruppo ha il compito di supportare un calendario di incontri e appuntamenti per un gruppo di collaboratori. Per fissare un appuntamento che interessa più persone, il sistema deve trovare un buco comune nell'agenda di queste persone. Se questo buco comune non è disponibile, il sistema interagisce con uno o più utenti per riorganizzare le loro agende e trovare l'ora in cui fissare l'appuntamento.
 - Una stazione di servizio (distributore di benzina) deve essere predisposta per eseguire delle operazioni completamente automatizzate. Il cliente striscia la sua carta di credito in un apposito lettore collegato alla pompa di benzina; la carta di credito viene verificata trasmettendo i dati alla banca; il cliente imposta l'importo e preleva il carburante; quando ha finito e aggancia la pistola di erogazione alla colonnina, l'importo impostato viene addebitato sul conto corrente. La carta di credito viene restituita dopo che il cliente ha effettuato il prelievo di carburante. Se la carta non è valida, questa viene restituita immediatamente prima che il cliente possa prelevare il carburante.
- * 7.7 Disegnate un diagramma di sequenza che mostra le interazioni degli oggetti nel sistema di gestione di un'agenda di gruppo, quando un gruppo di persone sta organizzando una riunione.

- 7.8 Disegnate un diagramma di stato UML che mostra le possibili variazioni di stato in un'agenda di gruppo o in una stazione di servizio (distributore di benzina).
 - * 7.9 Tramite alcuni esempi spiegate perché la gestione della configurazione è importante quando un team di persone sta sviluppando un prodotto software.
 - * 7.10 Una piccola azienda ha sviluppato un software specializzato che si configura automaticamente per ciascun cliente. I nuovi clienti di solito hanno requisiti particolari da incorporare nei loro sistemi, e pagano affinché tali requisiti possano essere sviluppati e integrati nei loro prodotti. L'azienda ha l'opportunità di offrire un nuovo contratto, che potrebbe più che raddoppiare la sua clientela. I nuovi clienti chiedono di essere coinvolti nella configurazione dei loro sistemi. Spiegate perché, in queste circostanze, potrebbe essere una buona idea per l'azienda rendere open-source il suo software.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Design Patterns: Elements of Reusable Object-oriented Software. È il manuale originale che ha introdotto gli schemi software per una vasta comunità di utenti. (E. Gamma, R. Helm, R. Johnson e J. Vlissides, Addison-Wesley, 1995).

Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development, 3rd ed. Larman tratta con chiarezza la progettazione orientata agli oggetti e l'uso del linguaggio UML; è un buon testo propedeutico all'uso degli schemi nel processo di progettazione. Sebbene sia vecchio di oltre 10 anni, è ancora il miglior testo in circolazione su questi argomenti. (C. Larman, Prentice-Hall, 2004).

Producing Open Source Software: How to Run a Successful Free Software Project. Questo libro è una guida completa per capire il software open-source, i problemi delle licenze del software e gli aspetti pratici della gestione di un progetto di sviluppo open-source. (K. Fogel, O'Reilly Media Inc., 2008).

Ulteriori letture sul riutilizzo del software sono indicate alla fine dei Capitoli 15 e 22.

8

Test del software

L'obiettivo di questo capitolo è presentare il concetto di test del software e dei suoi processi. Dopo aver letto questo capitolo:

- conoscerete le varie fasi del test del software, dalle prime prove durante lo sviluppo fino alle prove di accettazione da parte dei clienti del software;
- apprenderete le tecniche che aiutano a scegliere i test case (casi di test) prima di scrivere il codice e per eseguire automaticamente questi test;
- conoscerete tre distinti tipi di test: test dei componenti, test del sistema e test della release;
- capirete le differenze tra test di sviluppo e test degli utenti.

- 8.1 Test di sviluppo
- 8.2 Sviluppo guidato da test
- 8.3 Test della release
- 8.4 Test degli utenti

L’obiettivo dei test del software è dimostrare che un programma svolge i compiti per i quali è stato realizzato e identificare eventuali errori prima di metterlo in uso. Per provare il funzionamento di un software, si esegue un programma utilizzando dati artificiali. Si controllano i risultati dei test per verificare che non ci siano errori, anomalie o informazioni su attributi non funzionali del programma.

Il processo di test del software ha due obiettivi distinti.

1. Dimostrare allo sviluppatore e al cliente che il software soddisfa i suoi requisiti. Per il software personalizzato, questo significa che dovrebbe esserci almeno un test per ogni requisito specificato nel documento dei requisiti. Per i prodotti software generici, questo significa che dovrebbero esserci test per tutte le funzionalità del sistema che saranno incorporate nella release del prodotto. È anche possibile provare combinazioni di funzionalità per controllare che non ci siano interazioni indesiderate tra di esse.
2. Scoprire eventuali input o sequenze di input per i quali il comportamento del software è errato, indesiderato o non è conforme alle sue specifiche. Tutto questo è causato da difetti (*bug*) nel software. Quando si prova un software per scoprirne i difetti, si va alla ricerca di tutti i tipi di comportamenti indesiderati del sistema, come crash, interazioni indesiderate con altri sistemi, calcoli errati e danneggiamento dei dati.

Il primo di questi obiettivi è il test di convalida, che controlla che il sistema funzioni correttamente utilizzando una serie di test case che riflettono l’uso previsto del sistema. Il secondo obiettivo è il test dei difetti, dove i test case sono progettati per scoprire i difetti del software. I test case possono essere deliberatamente oscuri e non hanno bisogno di riprodurre esattamente le condizioni in cui il sistema sarà utilizzato normalmente. Ovviamente, non c’è una demarcazione netta tra questi due metodi di test. Durante il test di convalida potrebbero essere identificati alcuni difetti del sistema; durante il test dei difetti potrebbe risultare che il programma soddisfa i suoi requisiti.

La Figura 8.1 mostra le differenze tra il test di convalida e il test dei difetti. Immaginate il sistema come una scatola nera da provare. Il sistema accetta gli input da una sorgente di input I e riversa i risultati in un insieme di output O . Alcuni dei risultati potrebbero essere errati; questi risultati sono registrati nell’insieme O_e e vengono generati dal sistema come risposta agli input dell’insieme I_e . La priorità del test dei difetti è scoprire tali input nell’insieme I_e , in quanto questi svelano i problemi nel sistema. Il test di convalida richiede che il sistema sia provato con gli input corretti, che non appartengono all’insieme I_e . Questi input inducono il sistema a generare output corretti.

I test non possono dimostrare che il software sia privo di difetti o che si comporterà correttamente in qualsiasi circostanza. È sempre possibile che un test che avete trascurato possa svelare ulteriori problemi nel sistema. Come ha affermato eloquentemente Edsger Dijkstra (Dijkstra 1972), uno dei primi pionieri dello sviluppo dell’ingegneria del software:

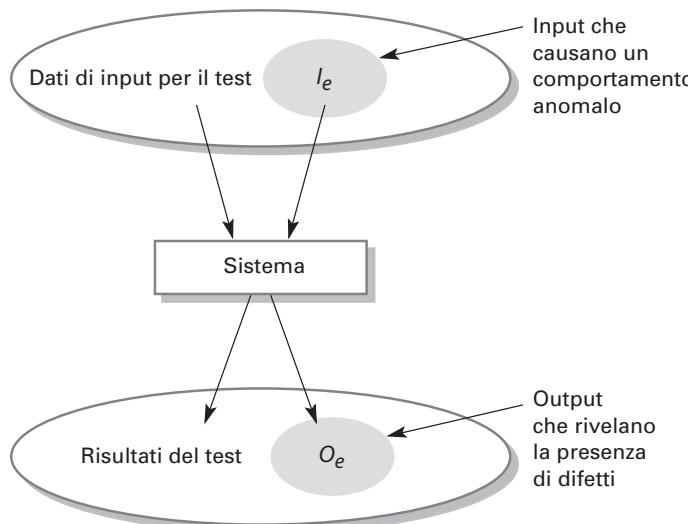


Figura 8.1 Un modello input-output per il test dei programmi.

“I test possono dimostrare soltanto la presenza di errori, non la loro assenza.”¹

I test fanno parte di un processo più ampio della verifica e della convalida del software (V & V, *Verification & Validation*). La verifica e la convalida non sono la stessa cosa, anche se spesso vengono confuse. Barry Boehm, un pioniere dell’ingegneria del software, ha spiegato concisamente la loro differenza (Boehm 1979):

- *Convalida*: stiamo realizzando il prodotto corretto?
- *Verifica*: stiamo realizzando il prodotto correttamente?

Il ruolo dei processi di verifica e convalida è essenzialmente controllare che il software che si sta sviluppando soddisfi le sue specifiche e offra le funzionalità richieste da chi ha acquistato il software. Questi processi di controllo iniziano non appena i requisiti del software si rendono disponibili e proseguono per tutte le successive fasi del processo di sviluppo.

La verifica del software è il processo che controlla se il software soddisfa i suoi requisiti funzionali e non funzionali. La convalida è un processo più generale; il suo compito è garantire che il software rispetti le attese del cliente. Il suo scopo va oltre il semplice controllo di conformità alla specifica per dimostrare che il software fa ciò che il cliente ha richiesto. La convalida è essenziale perché, come detto nel Capitolo 4, le definizioni dei requisiti non sempre riflettono i reali desideri o necessità dei clienti e degli utenti di un sistema software.

¹ Dijkstra, E. W. 1972. “The Humble Programmer.” *Comm. ACM* 15 (10): 859-66. doi:10.1145/355604.361591.

L’obiettivo ultimo dei processi di verifica e convalida è stabilire, con un buon grado di confidenza, che il software è adatto al suo scopo. Ciò significa che il sistema deve essere adatto all’uso cui è destinato. Il livello di confidenza richiesto dipende dallo scopo del software, dalle aspettative degli utenti e dal mercato del software.

1. *Scopo del software.* Quanto più è critico il software, tanto più importante è che esso sia affidabile. Per esempio, il livello di confidenza richiesto per il software che controlla un sistema di sicurezza è molto più alto di quello richiesto per un prototipo di software sviluppato per dimostrare nuove idee.
2. *Aspettative degli utenti.* A volte gli utenti non hanno grandi aspettative sulla qualità del software, a causa di precedenti esperienze con software inaffidabile e pieno di bug. Non si sorprendono quando il software sbaglia. Quando viene installato un nuovo sistema, gli utenti possono tollerare i malfunzionamenti del software, in quanto i benefici del suo utilizzo superano i costi per ripristinare le condizioni preesistenti a un malfunzionamento. Di conseguenza, potrebbe essere richiesta una verifica più approfondita delle successive versioni del sistema.
3. *Mercato del software.* Quando una società di software immette un sistema sul mercato, deve tenere in considerazione i programmi concorrenti, il prezzo che i clienti sono disposti a pagare e la tempistica richiesta per consegnare il sistema. In un mercato molto competitivo, la società può decidere di rilasciare un programma prima che siano stati completati i processi di debugging e i test, perché vuole entrare per prima nel mercato. Se un prodotto software o un’app sono molto economici, gli utenti potrebbero essere disposti a tollerare livelli di affidabilità più bassi.

Analogamente al test del software, anche i processi di verifica e convalida possono richiedere ispezioni e revisioni del software. Le ispezioni e le revisioni analizzano e controllano i requisiti del sistema, gli schemi di progettazione, il codice sorgente dei programmi e perfino i test proposti per il sistema. Si tratta di tecniche “statiche” di V & V, nelle quali non occorre eseguire il software per verificarlo. La Figura 8.2 mostra che le ispezioni e i test del software supportano le tecniche V & V nei vari stadi del processo software. Le frecce indicano gli stadi del processo in cui le tecniche possono essere utilizzate.

Le ispezioni si concentrano soprattutto sul codice sorgente di un sistema, ma può essere ispezionata qualsiasi rappresentazione leggibile del software, come i suoi requisiti o lo schema di progettazione. Quando ispezionate un sistema, sfruttate le conoscenze del sistema, del suo dominio di applicazione e del linguaggio di programmazione o modellazione per scoprire gli errori.

L’ispezione del software ha tre vantaggi rispetto al test:

1. Durante i test, gli errori possono mascherare (nascondere) altri errori. Quando un errore determina output inaspettati, non si può essere sicuri che altri

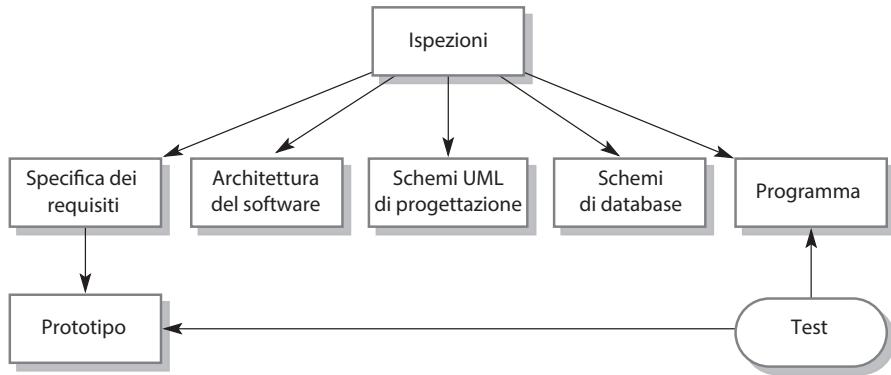


Figura 8.2 Ispezioni e test di un sistema software.

risultati anomali siano dovuti a un nuovo errore o siano un effetto collaterale dell'errore originario. Poiché l'ispezione non richiede l'esecuzione del software, non ci si deve preoccupare delle interazioni tra gli errori. Di conseguenza, una singola sessione di ispezione può scoprire molti errori in un sistema.

2. Le versioni incomplete di un sistema possono essere ispezionate senza costi aggiuntivi. Se un programma è incompleto, occorre sviluppare una serie di test specializzati per testare le parti disponibili; questo ovviamente aggiunge costi allo sviluppo del sistema.
3. Oltre a cercare i difetti di un programma, un'ispezione può anche valutare attributi di qualità più generali, come la conformità agli standard, la portabilità e la facilità di manutenzione. Si possono cercare inefficienze, algoritmi inappropriati e stili di programmazione mediocri che potrebbero rendere il sistema difficile da manutenere e aggiornare.

Le ispezioni sono un'idea vecchia; ci sono stati diversi studi ed esperimenti che hanno dimostrato che le ispezioni sono più efficaci dei test nella scoperta dei difetti. Fagan (Fagan 1986) afferma che più del 60% degli errori di un programma possono essere individuati utilizzando ispezioni informali. Nel processo Cleanroom (Prowell et al. 1999) si ritiene che le ispezioni di un programma possano individuare più del 90% degli errori.

Le ispezioni, tuttavia, non possono rimpiazzare il test del software. Le ispezioni non sono adatte a scoprire i difetti che si presentano a causa di interazioni inaspettate tra varie parti di un programma, i problemi di tempistica o quelli relativi alle prestazioni del sistema. Nelle piccole aziende o nei gruppi di sviluppo, può essere difficile e costoso mettere insieme un apposito team di ispezione, in quanto tutti i potenziali membri del team possono essere anche sviluppatori del software.

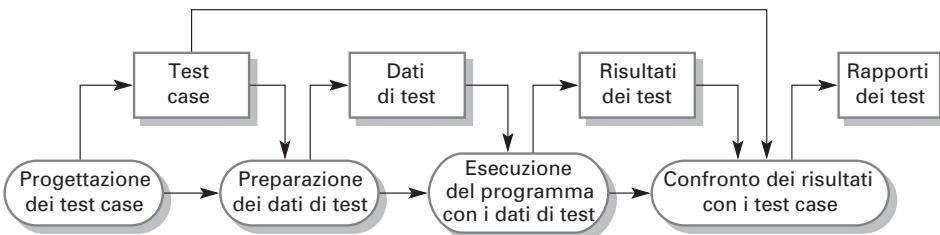


Figura 8.3 Modello si processo di test del software.

Nel Capitolo 21 ho descritto più dettagliatamente le ispezioni e le revisioni del sistema. Nel Capitolo 12 ho trattato l’analisi statica, dove il testo sorgente di un programma viene automaticamente analizzato per scoprire eventuali anomalie.

La Figura 8.3 è un modello astratto del tradizionale processo di prova del software, che viene utilizzato nello sviluppo guidato da piani. I test case sono le specifiche degli input dei test e degli output previsti dal sistema (i risultati dei test), più una descrizione di ciò che si sta provando. I dati di test sono gli input che sono stati previsti per provare il sistema. I dati di test a volte possono essere generati automaticamente, ma non è possibile generare automaticamente un test case. Le persone che sanno che cosa deve fare il sistema devono essere coinvolte nella specifica dei risultati previsti per i test; l’esecuzione dei test può essere automatizzata. I risultati dei test vengono automaticamente confrontati con quelli previsti, in modo che non ci sia bisogno di una persona che controlli errori e anomalie nell’esecuzione dei test.

Tipicamente, un sistema software commerciale è soggetto a tre stadi di test.

1. *Test dello sviluppo.* Il sistema viene provato durante lo sviluppo per scoprire bug e difetti. Sviluppatori e programmatore del sistema potrebbero essere coinvolti nei test.
2. *Test della release.* Un apposito team prova una versione completa del sistema prima che sia rilasciata agli utenti. Scopo del test della release è verificare che il sistema soddisfa i requisiti degli stakeholder del sistema.
3. *Test degli utenti.* Gli utenti o i potenziali utenti di un sistema provano il sistema nel loro ambiente. Per i prodotti software, l’utente può essere un gruppo di marketing interno che decide se il software può essere immesso sul mercato, rilasciato e venduto. Il test di accettazione è un tipo di test degli utenti, dove il cliente prova formalmente il sistema per decidere se esso debba essere accettato o se sia necessario un ulteriore sviluppo.

In pratica, il processo di test di solito consiste in un mix di test manuali e automatici. Nei test manuali, l’esecutore dei test (*tester*) avvia un programma con alcuni dati di prova e confronta i risultati con quelli previsti; annota e commenta le discrepanze per gli sviluppatori del programma. I test automatici sono codifi-

Pianificazione dei test

La pianificazione dei test riguarda la gestione dei tempi e delle risorse di tutte le attività svolte in un processo di prova del software. Richiede la definizione delle procedure dei test, tenendo conto delle persone e del tempo a disposizione. Di solito, un piano di test definisce che cosa deve essere provato, la tempistica dei test e il modo in cui i test devono essere registrati. Per i sistemi critici, il piano dei test può includere anche i dettagli dei test da eseguire sul software.

<http://software-engineering-book.com/web/test-planning/>

cati in un programma che viene eseguito ogni volta che il sistema che si sta sviluppando deve essere provato. Questo procedimento è più veloce di quello manuale, specialmente quando è necessario effettuare dei test di regressione – ripetere i test precedenti per verificare che le modifiche apportate al programma non abbiano introdotto nuovi bug.

I test, purtroppo, non possono essere mai completamente automatici, in quanto i test automatici possono verificare soltanto che un programma svolge i compiti per cui è stato creato. È praticamente impossibile usare test automatici per provare i sistemi che dipendono dall'aspetto visivo delle cose (per esempio, un'interfaccia grafica utente) o per dimostrare che un programma non abbia effetti collaterali imprevisti.

8.1 Test di sviluppo

I test di sviluppo includono tutte le attività di test che sono svolte dal team di sviluppo di un sistema software. Il tester del software di solito è il programmatore che sviluppa il software. Alcuni processi di sviluppo usano la coppia programmatore/tester (Cusamano e Selby 1998), dove ciascun programmatore è associato a un tester che sviluppa ed effettua i test. Per i sistemi critici, si può adottare un processo più formale, con un gruppo di test distinto all'interno del team di sviluppo. Questo gruppo è responsabile dei test di sviluppo e della registrazione dettagliata dei risultati dei test.

Ci sono tre stadi nei test di sviluppo.

1. *Test delle unità.* Vengono provate singole unità di programma o classi di oggetti. Il test delle unità si focalizza sulla verifica delle funzionalità di oggetti e metodi.
2. *Test dei componenti.* Vengono integrati più unità di programma per creare componenti più complessi. Il test dei componenti si focalizza sulla verifica delle interfacce dei componenti che forniscono accesso alle funzioni dei vari componenti.

Debugging

Il debugging è il processo nel quale vengono corretti gli errori e i problemi che sono stati scoperti durante i test. Utilizzando le informazioni ottenute dai test di un programma, i debugger sfruttano le loro conoscenze del linguaggio di programmazione e i risultati che si intendevano ottenere dai test per localizzare e correggere gli errori del programma. Quando si effettua il debugging di un programma, di solito si usano degli strumenti interattivi per ottenere informazioni extra sull'esecuzione del programma.

<http://software-engineering-book.com/web/debugging/>

3. *Test del sistema.* Alcuni o tutti i componenti del sistema vengono integrati e il sistema viene provato con un unico elemento. Il test del sistema si focalizza sulla verifica delle interazioni fra i componenti.

Il test di sviluppo è un processo il cui scopo principale è scoprire i bug del software. Di conseguenza, questo test di solito è interacciato con il debugging – il processo che localizza i problemi nel codice e modifica il programma per risolvere tali problemi.

8.1.1 Test delle unità

Il test delle unità è il processo dove vengono provati i componenti di un programma, come i metodi o le classi di oggetti. I tipi più semplici di componenti sono le singole funzioni o i singoli metodi di un programma. I test dovrebbero eseguire queste routine con differenti parametri di input. Per progettare i test delle funzioni o dei metodi, potete utilizzare le tecniche di progettazione dei test case descritte nel Paragrafo 8.1.2.

Per testare le classi di oggetti, dovreste progettare i vostri test in modo da verificare tutte le funzioni di ciascun oggetto. Questo significa che occorre testare tutte le operazioni associate a un oggetto; impostare e controllare il valore di tutti gli attributi associati all'oggetto, e porre l'oggetto in tutti i possibili stati. Questo implica che occorre simulare tutti gli eventi che provocano un cambiamento di stato dell'oggetto.

Considerate, per esempio, l'oggetto stazione meteorologica dell'esempio descritto nel Capitolo 7. Gli attributi e le operazioni di questo oggetto sono mostrati nella Figura 8.4.

Ha solo un attributo, il suo identificatore, che è una costante impostata quando viene installata la stazione. Quindi è sufficiente un solo un test che controlli se è stato correttamente impostato. Occorre definire dei test case per *rapportoMeteologico* e *rapportoStato*. In teoria si dovrebbero testare i metodi isolatamente, ma in alcuni casi sono necessarie sequenze di test. Per esempio, per poter testare *spegnimento*, il metodo che spegne gli strumenti della stazione meteorologica, è necessario aver eseguito il metodo *riavvio*.

StazioneMeteorologica
identificatore
rapportoMeteorologico ()
rapportoStato ()
risparmioEnergetico (strumenti)
controlloRemoto (comandi)
riconfigurazione (comandi)
riavvio (strumenti)
spegnimento (strumenti)

Figura 8.4 Interfaccia dell'oggetto stazione meteorologica.

La generalizzazione o ereditarietà rende più complicato il test delle classi di oggetti. Non è possibile testare semplicemente un'operazione nella classe in cui essa è definita e supporre che essa funzionerà come previsto in tutte le sottoclassi che ereditano tale operazione. L'operazione che viene ereditata può fare delle ipotesi su altre operazioni e altri attributi. Queste ipotesi potrebbero non essere valide in alcune sottoclassi che ereditano l'operazione; quindi occorre testare l'operazione ereditata in qualsiasi luogo in cui viene utilizzata.

Per testare gli stati della stazione meteorologica, si può usare un modello a stati, come descritto nel Capitolo 7 (Figura 7.8). Tramite questo modello si possono identificare sequenze di transizioni di stato che devono essere testate e definire sequenze di eventi per forzare tali transizioni. In teoria si dovrebbe testare ogni possibile sequenza di transizioni di stato, anche se in pratica ciò potrebbe essere molto costoso. Alcuni esempi di sequenze di stati che dovrebbero essere testate nella stazione meteorologica sono:

Spento → Esecuzione → Spento

Configurazione → Esecuzione → Test → Trasmissione → Esecuzione

Esecuzione → Raccolta → Esecuzione → Sintesi → Trasmissione → Esecuzione

Ove possibile, è consigliabile automatizzare i test delle unità. Per questo potete utilizzare un framework di automazione dei test, come JUnit (Tahchiev et al. 2010), che vi consente di scrivere ed eseguire i test dei vostri programmi. I framework dei test delle unità forniscono generiche classi di test che potete estendere per creare test case specifici. Potrete eseguire tutti i test che avete implementato e documentare il successo o l'insuccesso dei test, utilizzando qualche interfaccia grafica (GUI). Un'intera sequenza di test può essere eseguita in pochi secondi, quindi potete eseguire tutti i test ogni volta che apportate una modifica al programma.

Un test automatico è composto da tre parti.

1. *Parte di impostazione*, dove viene inizializzato il sistema con il test case, ovvero gli input e gli output previsti.
2. *Parte della chiamata*, dove viene chiamato l'oggetto o il metodo da testare.

3. *Parte dell'asserzione*, dove viene confrontato il risultato della chiamata con il risultato previsto. Se l'asserzione è vera, il test ha avuto successo; se è falsa, il test è fallito.

A volte, l'oggetto che si sta testando ha delle dipendenze con altri oggetti che potrebbero non essere stati implementati o il cui uso potrebbe rallentare il processo di test. Per esempio, se un oggetto chiama un database, questo potrebbe richiedere un lungo processo di impostazione prima di poter essere usato. In questi casi, sarebbe opportuno utilizzare oggetti finti.

Gli oggetti finti (*mock*) sono oggetti che hanno la stessa interfaccia degli oggetti esterni realmente utilizzati e che simulano le loro funzionalità. Per esempio, un oggetto finto che simula un database può avere soltanto pochi dati che sono organizzati in un array; l'accesso a questi dati è rapido, in quanto non è necessario chiamare le funzioni del database e accedere ai dischi in cui è memorizzato. Gli oggetti finti sono utilizzati anche per simulare operazioni insolite o eventi rari. Per esempio, se il vostro sistema è ideato per svolgere un'azione in determinate ore del giorno, un oggetto finto potrebbe semplicemente indicare tali ore, indipendentemente dall'ora effettiva del giorno.

8.1.2 Scelta dei test case delle unità

Il test è un processo costoso e lungo, quindi è importante scegliere test case molto efficienti. Efficienza, in questo caso, significa due cose:

1. i test case devono mostrare che, se utilizzato come previsto, il componente che si sta testando svolge correttamente il compito per il quale è stato progettato;
2. se ci sono difetti nel componente, questi devono essere identificati dai test case.

Per questo occorre progettare due tipi di test case. Il primo di questi dovrebbe riflettere il normale funzionamento di un programma e mostrare che il componente funziona. Per esempio, se state testando un componente che crea e inizializza il record di un nuovo paziente, allora il test case dovrebbe mostrare che il record esiste in un database e che i suoi campi sono stati impostati come specificato. L'altro tipo di test dovrebbe essere basato sull'esperienza che suggerisce dove si presentano i problemi più comuni; dovreste utilizzare input anomali per verificare che questi siano correttamente elaborati e non provochino l'arresto del funzionamento del componente in prova.

Due strategie che possono essere efficaci nella scelta dei test case sono le seguenti:

1. *test delle partizioni* – vengono identificati i gruppi di input che hanno caratteristiche comuni e che devono essere elaborati allo stesso modo. Dovreste scegliere i test dall'interno di ciascuno di questi gruppi;

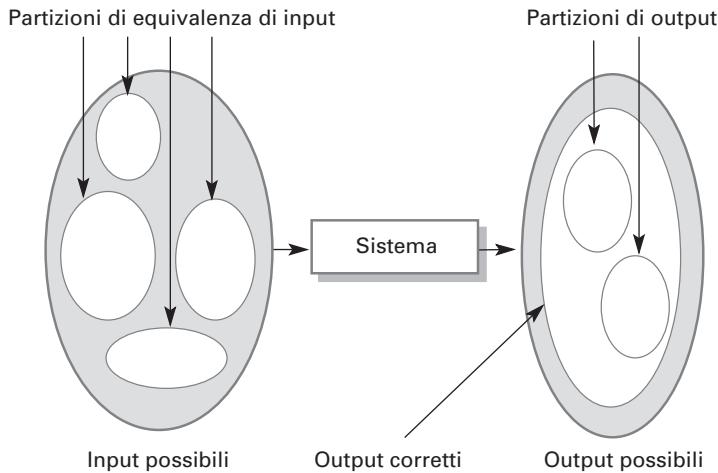


Figura 8.5 Partizioni di equivalenza.

2. *test basati su linee guida* – i test case vengono scelti applicando alcune linee guida di prova. Queste linee guida riflettono le esperienze maturate sui tipi di errori più comuni che i programmatore commettono durante lo sviluppo dei componenti.

I dati di input e i risultati di output di un programma possono essere immaginati come membri di insiemi che hanno caratteristiche comuni. Esempi di questi insiemi sono i numeri positivi, i numeri negativi e le selezioni dai menu. I programmi in genere hanno un comportamento simile con tutti i membri di un insieme; ovvero se si testa un programma che effettua un calcolo e richiede due numeri positivi, allora ci si può aspettare che il programma si comporti allo stesso modo con tutti i numeri positivi.

A causa di questo comportamento equivalente, queste classi a volte sono chiamate partizioni di equivalenza o domini (Bezier 1990): un approccio sistematico alla progettazione di test case si basa sull'identificazione di tutte le partizioni di input e output di un sistema o componente. I test case sono progettati in modo che gli input e gli output ricadano in queste partizioni. I test delle partizioni possono essere utilizzati per progettare test case sia per i sistemi sia per i componenti.

Nella Figura 8.5 la grande ellisse ombreggiata sulla sinistra rappresenta l'insieme di tutti i possibili input del programma che si sta testando. Le ellissi più piccole, non ombreggiate, rappresentano le partizioni di equivalenza. Il programma da testare dovrebbe elaborare tutti i membri di una partizione di equivalenza di input nello stesso modo.

Le partizioni di equivalenza di output sono partizioni all'interno delle quali tutti gli output hanno qualcosa in comune. A volte c'è una corrispondenza 1:1 tra le partizioni di equivalenza di input e quelle di output. Tuttavia, non è sempre

così; potrebbe essere necessario definire una partizione di equivalenza di input separata, dove l'unica caratteristica comune degli input è che questi producano output all'interno della stessa partizione di output. L'area ombreggiata nell'ellisse a sinistra rappresenta gli input che non sono validi. L'area ombreggiata nell'ellisse a destra rappresenta le eccezioni che potrebbero verificarsi, ovvero le risposte agli input non validi.

Una volta identificato un insieme di partizioni, potete scegliere i test case da ciascuna di queste partizioni. Una buona regola per selezionare i test case è scegliere quelli che si trovano sui confini di una partizione e quelli che sono vicini al punto centrale della partizione. Il motivo sta nel fatto che i progettisti e i programmati tendono a considerare i valori tipici degli input quando sviluppano un sistema; voi testate questi valori scegliendo il punto centrale della partizione. I valori di confine spesso sono atipici (per esempio, il valore zero potrebbe comportarsi in modo diverso dagli altri numeri non negativi) e quindi a volte vengono ignorati dagli sviluppatori. I malfunzionamenti dei programmi spesso si verificano proprio quando si elaborano questi valori atipici.

Le partizioni sono identificate usando le specifiche del programma o la documentazione degli utenti e, per esperienza, dalle classi di valori di input che potrebbero innescare errori. Per esempio, secondo la specifica, un programma accetta da 4 a 8 input che sono numeri interi di cinque cifre maggiori di 10 000. Utilizzate queste informazioni per identificare le partizioni di input e tutti i possibili valori degli input, come illustra la Figura 8.6.

Il metodo che utilizza la specifica di un sistema per identificare le partizioni di equivalenza è detto *black-box testing* (test a scatola nera); non è necessario sapere come funziona il sistema. Insieme a questi test, a volte, si utilizzano anche i *white-box test* (test a scatola bianca), dove si esamina il codice del programma per trovare altri possibili test. Per esempio, il codice potrebbe includere alcune ecce-

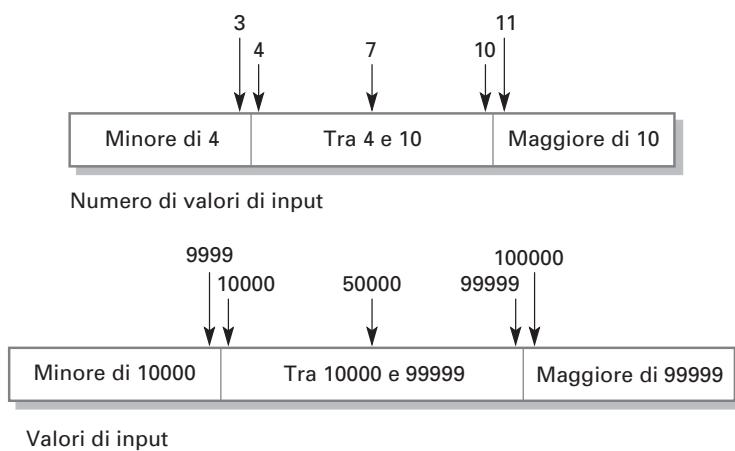


Figura 8.6 Esempi di partizioni di equivalenza.

Test dei percorsi

Il test dei percorsi (*path testing*) è una strategia di test il cui obiettivo è eseguire ogni percorso indipendente di un componente o programma; così facendo, tutte le istruzioni di un componente o programma vengono eseguite almeno una volta; anche tutte le istruzioni condizionali vengono testate sia per i casi veri sia per quelli falsi. In un processo di sviluppo orientato agli oggetti, il test dei percorsi può essere utilizzato per testare i metodi associati agli oggetti.

<http://software-engineering-book.com/web/path-testing/>

zioni per gestire input non validi. Potete sfruttare queste conoscenze per identificare una “partizione delle eccezioni” – dati differenti ai quali dovrebbe essere applicata la stessa gestione delle eccezioni.

Le partizioni di equivalenza sono tecniche di prova efficaci perché aiutano a tenere in considerazione gli errori che i programmatore commettono spesso quando elaborano input ai confini delle partizioni. Per scegliere i test case potete anche seguire le linee guida che si basano sulle conoscenze acquisite sui tipi di test case che sono in grado di scoprire gli errori con maggiore efficacia. Per esempio, quando testate programmi con sequenze, array o liste, le linee guida generali che potrebbero aiutarvi a scoprire i difetti sono le seguenti:

1. testare il software con sequenze che hanno un solo valore. I programmatore di solito credono che le sequenze siano fatte da più valori e, a volte, applicano questa ipotesi ai loro programmi. Di conseguenza, se presentato con una sequenza di un solo valore, un programma potrebbe non operare correttamente;
2. utilizzare varie sequenze di dimensioni differenti in test diversi. Questo riduce le probabilità che un programma con difetti possa produrre accidentalmente un output corretto a causa di qualche caratteristica anomala dell’input;
3. progettare i test in modo da utilizzare il primo elemento della sequenza, quello centrale e l’ultimo; questo metodo identifica i problemi ai confini delle partizioni.

Il libro di Whittaker (Whittaker 2009) include vari esempi di linee guida che possono essere utilizzate nella progettazione di test case; alcune delle linee guida più generali sono le seguenti:

- scegliere input che forzano il sistema a generare tutti i messaggi di errore;
- progettare input che causano un overflow dei buffer di input;
- ripetere lo stesso input o la stessa serie di input numerose volte;
- forzare la produzione di output non validi;
- forzare i calcoli in modo da ottenere valori molto grandi o molto piccoli.

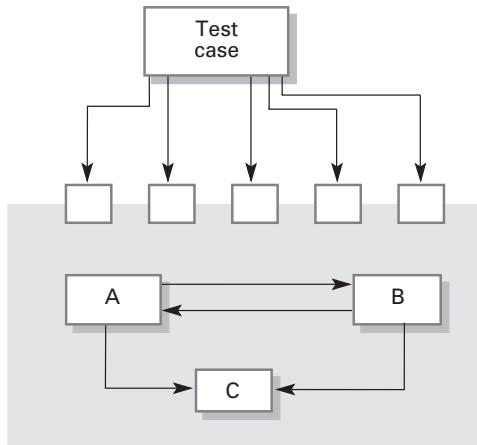


Figura 8.7 Test dell’interfaccia.

Al crescere della vostra familiarità con i test, potrete sviluppare le vostre linee guida su come scegliere i test case più efficienti. Nel prossimo paragrafo presenterò altri esempi di linee guida per i test.

8.1.3 Test dei componenti

I componenti software spesso sono formati da molti oggetti che interagiscono tra di loro. Per esempio, nel sistema delle stazioni meteorologiche, il componente di riconfigurazione include gli oggetti che si occupano di tutti gli aspetti della riconfigurazione. L’accesso alle funzionalità di questi oggetti avviene tramite le interfacce dei componenti (si veda il Capitolo 7). Il test dei componenti composti deve quindi dimostrare che le interfacce dei componenti si comportano conformemente alle loro specifiche. Si può supporre che i test delle unità sui singoli oggetti di un componente siano stati completati.

La Figura 8.7 illustra il processo di test dell’interfaccia dei componenti. Si suppone che i componenti A, B e C siano stati integrati per creare un componente o un sottosistema più grande. I test case non si applicano ai singoli componenti, ma all’interfaccia del componente composto, creato combinando tali componenti. È possibile che gli errori nelle interfacce del componente composto non vengano individuati testando i singoli oggetti, in quanto tali errori risultano dalle interazioni tra gli oggetti del componente.

Ci sono vari tipi di interfacce tra i componenti di un programma e, di conseguenza, diversi tipi di errori che possono verificarsi nelle interfacce.

1. *Interfacce di parametri.* Sono le interfacce in cui i dati o a volte i riferimenti alle funzioni vengono passati da un componente all’altro. I metodi in un oggetto hanno un’interfaccia di parametri.

2. *Interfacce a memoria condivisa.* Sono le interfacce in cui un blocco di memoria viene condiviso tra i componenti. I dati sono posti nella memoria da un sottosistema e da qui caricati da altri sottosistemi. Questo tipo di interfaccia è utilizzato nei sistemi integrati, dove i sensori creano i dati che vengono caricati ed elaborati da altri componenti del sistema.
3. *Interfacce procedurali.* Sono le interfacce in cui un componente incapsula una serie di procedure che possono essere chiamate da altri componenti. Gli oggetti e i componenti riutilizzabili hanno interfacce di questo tipo.
4. *Interfacce a passaggio di messaggi.* Sono le interfacce in cui un componente richiede un servizio a un altro componente passandogli un messaggio. Un messaggio di ritorno include i risultati dell'esecuzione del servizio. Alcuni sistemi orientati agli oggetti hanno questo tipo di interfaccia, come i sistemi client-server.

Gli errori delle interfacce sono una delle più comuni forme di errore nei sistemi complessi (Lutz 1993), e possono essere classificati in tre categorie.

- *Uso errato dell'interfaccia.* Un componente chiama un altro componente commettendo un errore nell'uso della sua interfaccia. Questo tipo di errore è particolarmente comune nelle interfacce dei parametri, dove i parametri possono essere del tipo sbagliato, passati nell'ordine sbagliato o in numero sbagliato.
- *Incomprensione delle interfacce.* Il componente che effettua una chiamata non comprende le specifiche dell'interfaccia del componente chiamato e fa delle ipotesi sul suo comportamento. Il componente chiamato non si comporta come previsto e innesta, a sua volta, un comportamento imprevisto nel componente che lo ha chiamato. Per esempio una routine di ricerca binaria può essere chiamata con un parametro che è un array non ordinato; ovviamente, la ricerca fallirà.
- *Errori di tempistica.* Questi errori si verificano nei sistemi real-time che utilizzano un'interfaccia a memoria condivisa o a passaggio di messaggi. Il produttore e il consumatore dei dati potrebbero operare a velocità diverse. Se non si fa particolare attenzione alla progettazione dell'interfaccia, il consumatore potrebbe accedere a informazioni non aggiornate, in quanto il produttore non ha fatto in tempo ad aggiornare le informazioni nella memoria condivisa.

Il test dei difetti delle interfacce è difficile perché alcuni errori possono manifestarsi soltanto sotto condizioni inusuali. Per esempio, si consideri un oggetto che implementa una coda come una struttura dati a lunghezza fissa. Un oggetto chiamante può presumere che la coda sia implementata sotto forma di struttura dati infinita e, quindi, non controlla un eventuale overflow della coda quando viene inserito un nuovo elemento.

Questa condizione può essere rilevata soltanto durante il test progettando una sequenza di test case che forzano un overflow nella coda. I test dovrebbero controllare come gli oggetti chiamanti gestiscono tale overflow. Tuttavia, poiché questa è una condizione rara, i tester potrebbero pensare che non valga la pena effettuare tale controllo quando scrivono i test per l'oggetto coda.

Un problema ulteriore può sorgere a causa delle interazioni tra gli errori nei vari moduli o oggetti. Gli errori in un oggetto possono essere individuati soltanto quando qualche altro oggetto si comporta in modo inaspettato. Per esempio, un oggetto chiama un altro oggetto per ricevere qualche servizio e suppone che la risposta sia corretta. Se il servizio ricevuto è sbagliato, il valore restituito può essere valido, ma errato. Il problema non sarà immediatamente rilevato, ma si manifestera soltanto quando i successivi calcoli, che usano quel valore restituito, si riveleranno errati.

Ci sono alcune linee guida generali per il test delle interfacce che dovrebbero essere seguite.

1. Esaminare il codice da testare ed elencare esplicitamente ogni chiamata di un componente esterno. Progettare un insieme di test in cui i valori dei parametri passati a componenti esterni sono ai limiti estremi dei loro range. Questi valori estremi sono quelli che, con maggiore probabilità, riveleranno le inconsistenze tra le interfacce.
2. Se vengono passati dei puntatori attraverso un'interfaccia, testare sempre l'interfaccia con parametri di puntatori nulli.
3. Quando un componente viene chiamato attraverso un'interfaccia procedurale, progettare test che causano deliberatamente un fallimento del componente. Diverse ipotesi di fallimento sono una delle cause più comuni di incomprensione delle specifiche.
4. Utilizzare lo stress test nei sistemi a passaggio di messaggi. Questo significa progettare test che generano molti più messaggi di quelli che si verificano realmente. Questa è la tecnica migliore per individuare gli errori di tempistica.
5. Quando diversi componenti interagiscono attraverso la memoria condivisa, progettare test che variano l'ordine in cui questi componenti sono attivati. Tali test possono rivelare ipotesi implicite fatte dal programmatore sull'ordine in cui i dati condivisi sono prodotti e consumati.

In alcuni casi è preferibile utilizzare ispezioni e revisioni, anziché test, per scoprire gli errori nelle interfacce. Le ispezioni possono concentrarsi sulle interfacce dei componenti e controllare il comportamento presunto delle interfacce che è richiesto durante il processo di ispezione.

8.1.4 Test del sistema

Lo sviluppo di un sistema richiede l'integrazione di più componenti nel sistema e poi il test del sistema integrato. Il test del sistema controlla che i componenti siano compatibili, interagiscano correttamente e scambino i dati appropriati al momento giusto tra le loro interfacce. Ovviamente questo processo si sovrappone al test dei componenti, ma ci sono due importanti differenze.

1. Durante il test del sistema, i componenti riutilizzabili che sono stati sviluppati separatamente e i sistemi off-the-shelf possono essere integrati con i nuovi componenti; poi viene testato il sistema completo.
2. I componenti sviluppati da diversi team o sottoteam possono essere integrati in questa fase. Il test del sistema è un processo collettivo, non individuale. In alcune aziende, il test del sistema può richiedere un apposito team di test, senza coinvolgere progettisti e programmati.

Tutti i sistemi hanno un comportamento emergente. Questo significa che alcune funzionalità e caratteristiche del sistema diventano evidenti solo quando tutti i componenti vengono assemblati. Si potrebbe trattare di un comportamento emergente pianificato, che deve essere testato. Per esempio, supponete di integrare un componente di autenticazione con un componente che aggiorna il database del sistema; poi avete una caratteristica del sistema che limita l'aggiornamento delle informazioni agli utenti autorizzati. A volte, il comportamento emergente non è pianificato né voluto. Dovrete sviluppare i test per controllare che il sistema svolga soltanto i compiti previsti.

Il test del sistema dovrebbe focalizzarsi sulle interazioni tra i componenti e gli oggetti che formano il sistema. Si potrebbero testare anche i componenti o i sistemi riutilizzabili per controllare che essi operano nel modo previsto quando vengono integrati con i nuovi componenti. Il test di queste interazioni dovrebbe identificare i bug dei componenti che possono essere scoperti solo quando un componente viene utilizzato da altri componenti nel sistema. Il test delle interazioni serve anche a portare alla luce le ipotesi sbagliate, fatte dagli sviluppatori, sul comportamento di altri componenti del sistema.

A causa della loro attenzione alle interazioni, i test basati sui casi d'uso sono metodi efficaci per testare un sistema. Molti oggetti o componenti di solito implementano i casi d'uso di un sistema. Il test del caso d'uso fa sì che queste interazioni si verifichino. Se avete sviluppato un diagramma di sequenza per modellare l'implementazione di un caso d'uso, potrete vedere gli oggetti o i componenti che sono coinvolti nell'interazione.

Nell'esempio della stazione meteorologica, il software del sistema trasmette una sintesi dei dati meteorologici a un computer remoto, come descritto nella Figura 7.3. La Figura 8.8 mostra la sequenza delle operazioni quando il sistema risponde a una richiesta di raccolta dati per il sistema di mappatura. Si può utiliz-

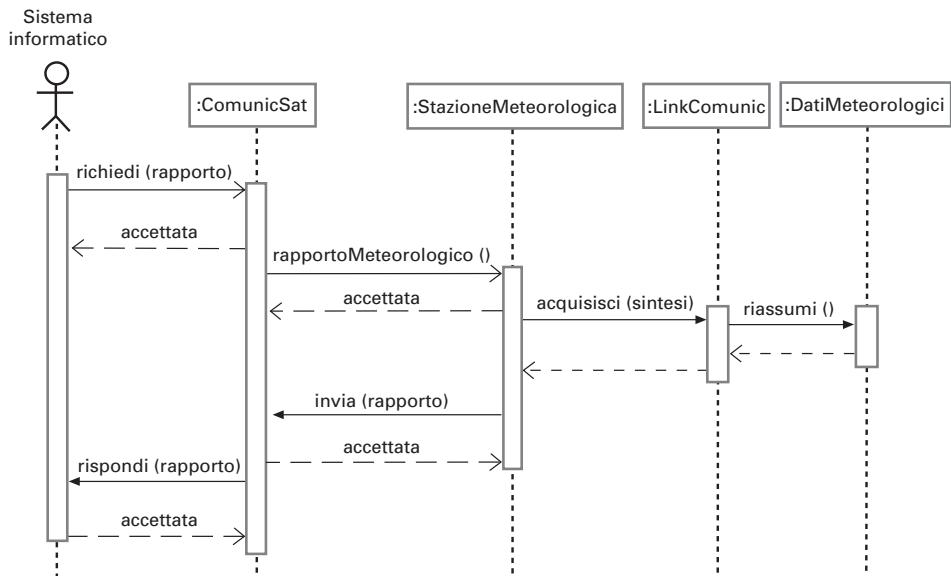


Figura 8.8 Diagramma di sequenza della raccolta dei dati meteorologici.

zare questo diagramma per identificare le operazioni che saranno testate e per aiutare la progettazione dei test case. Quindi, formulare la richiesta di un report causerà l'esecuzione del seguente thread di metodi:

ComunicSat:richiedi → StazioneMeteorologica:rapportoMeteorologico → LinkComunic:acquisisci(sintesi) → DatiMeteorologici:riassumi

Il diagramma di sequenza aiuta a progettare specifici test case, in quanto mostra quali input sono richiesti e quali output sono creati.

1. L'input di una richiesta di report dovrebbe avere una ricevuta di accettazione; dopo dovrebbe essere restituito un rapporto. Durante il test si dovrebbero creare dati riassuntivi che possono essere utilizzati per verificare che il report sia organizzato correttamente.
2. La richiesta di un report alla *StazioneMeteorologica* genera un rapporto riassuntivo. Questo si può testare separatamente creando dati grezzi corrispondenti alla sintesi preparata per il test di *ComunicSat* e verificando che l'oggetto *StazioneMeteorologica* produca correttamente tale sintesi. Questi dati grezzi sono utilizzati anche per testare l'oggetto *DatiMeteorologici*.

Ovviamente, il diagramma di sequenza nella Figura 8.8 è stato semplificato in modo che non mostri le eccezioni. Un test di scenari/casi d'uso completo deve considerare queste eccezioni e garantire che siano gestite correttamente.

Per molti sistemi è difficile sapere quanti test occorre fare e quando interrompere i test. È impossibile realizzare una serie di test esaustivi, dove viene testata

Integrazione e test incrementali

Il test di un sistema richiede prima l'integrazione dei vari componenti e poi l'esecuzione dei test sul sistema integrato risultante. Dovreste adottare sempre un approccio incrementale al test di un sistema: quando integrate un componente nel sistema, testate il sistema; se poi integrate un altro componente, testate di nuovo il sistema e così via. Se si verificano dei problemi, essi probabilmente sono dovuti alle interazioni con l'ultimo componente che avete integrato nel sistema.

L'integrazione e i test incrementali sono fondamentali nei metodi agili, dove i test di regressione vengono eseguiti ogni volta che viene integrato un nuovo incremento.

<http://software-engineering-book.com/web/integration/>

ogni possibile sequenza di esecuzione del programma. I test, quindi, devono basarsi su un sottoinsieme di possibili test case. Le società di software dovrebbero adottare delle apposite strategie per scegliere questo sottoinsieme di test case. Queste strategie potrebbero basarsi su tecniche di test generiche, come quella di eseguire almeno una volta tutte le istruzioni di un programma; in alternativa, ci si può basare sull'esperienza acquisita utilizzando il sistema e concentrarsi sui test delle funzionalità del sistema operativo, per esempio:

1. testare tutte le funzioni a cui si accede attraverso menu;
2. testare le combinazioni di funzioni (per esempio la formattazione del testo) a cui si accede attraverso lo stesso menu;
3. testare tutte le funzioni in cui l'utente fornisce input, con input corretti ed errati.

È chiaro che, grazie all'esperienza acquisita con i principali prodotti software, come i word processor o i fogli di calcolo, durante il test di un prodotto vengono seguite alcune linee guida simili. Quando le funzioni di un software vengono utilizzate isolatamente, di solito tutto procede bene. I problemi sorgono quando si utilizzano combinazioni di funzioni che non sono state testate insieme, come spiega Whittaker (Whittaker 2009) con questo esempio: in un word processor molto noto, quando inserite una nota a piè di pagina in una pagina a colonne multiple, il testo non si dispone correttamente nelle colonne.

Il test automatico di un sistema di solito è più difficile da realizzare del test automatico delle unità o dei singoli componenti. Il test automatico delle unità si basa sulla previsione degli output e poi sulla codifica di queste previsioni in un programma. La previsione viene poi confrontata con il risultato ottenuto. Tuttavia, implementare un sistema di solito significa generare output che sono molto complessi o difficili da prevedere. Potreste essere in grado di esaminare un output e verificarne la credibilità, senza essere necessariamente in grado di crearlo in anticipo.

8.2 Sviluppo guidato da test

Lo sviluppo guidato da test è un approccio allo sviluppo dei programmi in cui si interlacciano sviluppo e test del codice (Beck 2002, Jeffries e Melnik 2007). Il codice viene sviluppato in modo incrementale, insieme a una serie di test per ogni incremento del codice. Non si potrà procedere allo sviluppo del successivo incremento finché il codice sviluppato non avrà superato tutti i suoi test. Lo sviluppo guidato da test venne introdotto come parte del metodo di sviluppo agile XP (*eXtreme Programming*); oggi ha ottenuto un grande successo e può essere utilizzato sia nei processi agili sia in quelli guidati da piani.

Il processo fondamentale dello sviluppo guidato da test è illustrato nella Figura 8.9. Le fasi di questo processo sono elencate qui di seguito.

1. Si inizia identificando l'incremento della funzionalità richiesta. Questo incremento deve essere piccolo e implementabile in poche linee di codice.
2. Si scrive un test per questa funzionalità e lo si implementa come test automatico. Questo significa che il test potrà essere eseguito e dovrà risultare se esso è stato superato oppure no.
3. Si esegue il test, insieme con tutti gli altri test che sono stati implementati. Inizialmente la funzionalità non è stata implementata, quindi il nuovo test non sarà superato. Questo è intenzionale, in quanto dimostra che il test aggiunge qualcosa di nuovo alla serie esistente di test.
4. Si implementa la funzionalità e si ripete il test. Questo potrebbe richiedere la rifattorizzazione del codice esistente per migliorarlo e per aggiungere nuovo codice.
5. Una volta che tutti i test sono stati superati, si passa all'implementazione della successiva funzionalità.

Un ambiente di test automatizzati, come JUnit che supporta il test dei programmi Java (Tahchiev et al. 2010), è essenziale per lo sviluppo guidato da test. Poiché il codice viene sviluppato in piccoli incrementi, è possibile eseguire i test ogni volta che si aggiunge una funzionalità o si rifattorizza il programma. Per questo, i test

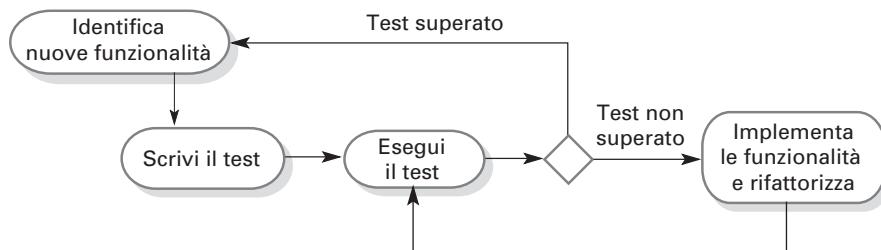


Figura 8.9 Sviluppo guidato da test.

vengono integrati in un programma distinto che li esegue sul sistema da testare. Con questo approccio è possibile eseguire centinaia di test distinti in pochi secondi.

Lo sviluppo guidato da test aiuta i programmatore a fare chiarezza su cosa un segmento di codice deve effettivamente fare. Per scrivere un test, occorre capire a cosa serve, e questa conoscenza aiuta a scrivere il codice richiesto. Ovviamente, se si hanno conoscenze incomplete o errate, lo sviluppo guidato da test non è di grande aiuto.

Se non si hanno sufficienti conoscenze per scrivere i test, non è possibile sviluppare il codice richiesto. Per esempio, se un calcolo prevede la divisione tra due numeri, occorre controllare che un numero non venga mai diviso per zero. Se dimenticate di scrivere un test per questo, allora il codice che effettua questo controllo non sarà mai incluso nel programma.

Oltre a una migliore comprensione dei problemi, lo sviluppo guidato da test offre altri vantaggi.

1. *Copertura del codice.* In teoria, ogni segmento di codice che viene scritto dovrebbe avere almeno un test associato. Quindi, si può essere sicuri che tutto il codice nel sistema è stato effettivamente eseguito. Il codice viene testato mentre viene scritto, quindi i difetti vengono scoperti nelle prime fasi del processo di sviluppo.
2. *Test di regressione.* Una serie di test viene sviluppata in modo incrementale mentre il programma viene sviluppato. È sempre possibile eseguire i test di regressione per controllare che le modifiche del programma non abbiano introdotto nuovi bug.
3. *Debugging semplificato.* Quando un test fallisce, dovrebbe essere ovvio sapere dove si trova il problema. Il codice appena scritto deve essere controllato e modificato. Non occorre utilizzare gli strumenti di debugging per localizzare il problema. I sondaggi sull'uso dello sviluppo guidato da test indicano che quasi mai è necessario utilizzare un debugger automatico in questo tipo di sviluppo (Martin 2007).
4. *Documentazione del sistema.* I test stessi sono una forma di documentazione che descrive che cosa dovrebbe fare il codice. La lettura dei test può agevolare la comprensione del codice.

Uno dei più importanti vantaggi dello sviluppo guidato da test è la riduzione dei costi del test di regressione. Il test di regressione richiede l'esecuzione di una serie di test che sono stati eseguiti con successo dopo le modifiche apportate a un sistema. Il test di regressione controlla che queste modifiche non abbiano introdotto nuovi bug nel sistema e che il nuovo codice interagisca secondo le previsioni con il codice esistente. Il test di regressione è costoso e a volte impraticabile quando un sistema viene testato manualmente, in quanto i costi in termini di tempo e impegno sono molto elevati. Dovrete cercare di scegliere i test più significativi da ripetere ed è alto il rischio di escludere i test più importanti.

I test automatici riducono drasticamente i costi del test di regressione. I test esistenti possono essere ripetuti rapidamente e a buon prezzo. Dopo avere apportato una modifica a un sistema nello sviluppo con test iniziali, tutti i test esistenti dovranno essere eseguiti con successo prima di aggiungere una nuova funzionalità. Un programmatore avrà la certezza che la nuova funzionalità che ha aggiunto non ha rivelato o causato problemi nel codice esistente.

Lo sviluppo guidato da test è particolarmente prezioso nello sviluppo di nuovo software, dove una funzionalità viene implementata nel nuovo codice o utilizzando componenti di librerie standard. Se riutilizzate grossi componenti di codice o sistemi ereditati, allora dovete scrivere test per questi sistemi nel loro insieme. Non potete facilmente scomporli in elementi che possono essere singolarmente testati. Lo sviluppo incrementale guidato da test è impraticabile. Lo sviluppo guidato da test può anche essere inefficiente con i sistemi multithread. I vari thread potrebbero essere interacciati in istanti diversi durante l'esecuzione di test differenti e, quindi, potrebbero produrre risultati differenti.

Se adottate lo sviluppo guidato da test, avrete sempre bisogno di un processo di test per convalidare il sistema, ovvero per verificare che esso soddisfi i requisiti di tutti gli stakeholder del sistema. Il test del sistema testa anche le prestazioni e l'affidabilità, e controlla che il sistema non faccia cose che non dovrebbe fare, come produrre output indesiderati. Andrea (Andrea 2007) indica come estendere gli strumenti di test per integrare alcuni aspetti del test del sistema con lo sviluppo guidato da test.

Lo sviluppo guidato da test oggi è un importante approccio largamente utilizzato nel test del software. Molti programmatore che hanno adottato questo approccio sono soddisfatti e ritengono che sia un metodo più produttivo per sviluppare software. Alcuni credono che l'uso dello sviluppo guidato da test migliori la strutturazione dei programmi e la qualità del codice; tuttavia, alcuni esperimenti non hanno confermato questa credenza.

8.3 Test della release

Il test della release è il processo che testa una particolare release di un sistema che dovrà essere utilizzata all'esterno del team di sviluppo. Di norma, la release di un sistema è destinata ai clienti e agli utenti. In un progetto complesso, tuttavia, la release potrebbe essere utilizzata da membri di altri team che stanno sviluppando sistemi correlati. Per i prodotti software, la release potrebbe essere destinata al management dei prodotti che ne preparano la vendita.

Ci sono due importanti distinzioni tra il test della release e il test del sistema durante il processo di sviluppo.

1. Il team di sviluppo del sistema non è responsabile del test della release.

2. Il test della release è un processo di convalida che controlla se un sistema soddisfa i suoi requisiti ed è sufficientemente buono per essere utilizzato dei clienti del sistema. Il test del sistema eseguito dal team di sviluppo deve focalizzarsi sulla scoperta dei bug nel sistema (test dei difetti).

L'obiettivo principale del test della release è convincere il fornitore del sistema che questo è sufficientemente buono per essere utilizzato; se ciò è vero, il sistema può essere rilasciato come prodotto o consegnato al cliente. Pertanto, il test della release deve dimostrare che il sistema offre le funzionalità, le prestazioni e la fidatezza specificate, e che non fallisce durante il suo normale utilizzo.

Il test della release di solito è un processo di test a scatola nera, dove i test sono derivati dalla specifica del sistema. Il sistema viene trattato come una scatola nera il cui comportamento può essere determinato soltanto studiando i suoi input e i relativi output. Un altro nome per questo processo è *test di funzionalità*, poiché i tester si occupano soltanto delle funzionalità e non dell'implementazione del software.

8.3.1 Test basato sui requisiti

Un principio generale della buona pratica di ingegneria del software è che i requisiti devono essere testabili. Questo significa che un requisito deve essere scritto in modo tale che possa essere progettato un apposito test per verificare che tale requisito sia soddisfatto. Il test basato sui requisiti, quindi, è un approccio sistematico alla progettazione dei test case, dove si prendono in esame i singoli requisiti e si sviluppa una serie di test per ciascun requisito. Il test basato sui requisiti è una convalida, anziché un test dei difetti – si tenta di dimostrare che il sistema implementa correttamente i suoi requisiti.

Per esempio, considerate i seguenti requisiti per il sistema Mentcare che riguardano il controllo delle allergie ai farmaci.

Se un paziente è allergico a un particolare farmaco, allora la prescrizione di quel farmaco dovrà essere associata a un messaggio di avvertimento che dovrà essere inviato all'utente del sistema.

Se un utente del sistema ignora il messaggio di avvertimento, dovrà fornire una spiegazione del perché ha ignorato il messaggio.

Per verificare che questi requisiti sono stati soddisfatti, potrebbe essere necessario sviluppare alcuni test correlati.

1. Predisporre una cartella clinica per un paziente senza allergie. Prescrivere un farmaco per le allergie esistenti. Controllare che non venga emesso un messaggio di avvertimento dal sistema.
2. Predisporre una cartella clinica per un paziente con un'allergia. Prescrivere un farmaco per l'allergia del paziente e controllare che il sistema emetta il messaggio di avvertimento.

3. Predisporre una cartella clinica per un paziente che è allergico a due o più farmaci. Prescrivere separatamente entrambi questi farmaci e controllare che il sistema emetta il corretto messaggio di avvertimento per ciascun farmaco.
4. Prescrivere due farmaci cui è allergico il paziente. Controllare che vengano emessi due messaggi di avvertimento.
5. Prescrivere un farmaco che genera un messaggio di avvertimento e ignorare questo messaggio. Controllare che il sistema richieda all'utente di spiegare perché ha ignorato il messaggio di avvertimento.

Come potete notare da questa lista, il test di un requisito non significa scrivere un solo test. Di norma, occorre scrivere una serie di test per garantire che il requisito sia completamente soddisfatto. Dovreste anche tenere un registro di tracciabilità dei test basati sui requisiti, per mettere in relazione i test ai requisiti specifici che avete testato.

8.3.2 Test degli scenari

Il test degli scenari è un approccio al test della release dove vengono concepiti alcuni tipici scenari d'uso per sviluppare dei test case per un sistema. Uno scenario è una storia che descrive il modo in cui un sistema potrebbe essere utilizzato. Gli scenari devono essere realistici, e gli utenti reali del sistema devono essere in grado di identificarsi in tali scenari. Se utilizzate scenari o storie di utenti come parte del processo di ingegneria dei requisiti (descritto nel Capitolo 4), allora sarete in grado di riutilizzarli come scenari di test.

In un breve articolo sul test degli scenari, Kaner (Kaner 2003) suggerisce che il test degli scenari deve essere una narrazione credibile e abbastanza complessa. Deve essere in grado di motivare gli stakeholder, nel senso che questi devono potersi identificare nello scenario e credere che è importante che il sistema passi il test. Secondo Kaner un test di scenari deve essere anche facile da valutare. Se ci sono problemi con il sistema, allora il team che esegue il test della release deve essere in grado di identifierli agevolmente.

Come esempio di un possibile scenario del sistema Mentcare, la Figura 8.10 descrive un modo in cui il sistema può essere utilizzato per una visita medica a domicilio. Questo scenario testa alcune caratteristiche del sistema Mentcare:

1. autenticazione tramite login con il sistema;
2. download e upload delle cartelle cliniche di un paziente su un computer portatile;
3. fissare l'appuntamento per la visita a domicilio;
4. criptare e decriptare le cartelle cliniche di un paziente su un'unità mobile;
5. caricare e modificare le informazioni di una cartella clinica;

Giorgio è un infermiere specializzato nella cura di malattie mentali. Uno dei suoi compiti è visitare i pazienti a casa per controllare che i farmaci somministrati siano efficaci e non procurino effetti collaterali.

Nel giorno delle visite a domicilio, Giorgio si collega al sistema Mentcare e stampa gli appuntamenti delle visite a domicilio fissate per quel giorno, insieme alle informazioni sintetiche sui pazienti da visitare. Giorgio chiede che le cartelle cliniche di questi pazienti siano scaricate sul suo computer portatile. Il sistema gli chiede la frase chiave per criptare le informazioni delle cartelle cliniche sul suo computer.

Uno dei pazienti da visitare si chiama Giovanni, che sta seguendo una cura di farmaci anti-depressivi. Giovanni sente che i farmaci lo stanno aiutando, ma crede che abbiano l'effetto collaterale di tenerlo sveglio di notte. Giorgio cerca nel portatile la cartella clinica di Giovanni; il sistema gli chiede la frase chiave per decriptare le informazioni della cartella; controlla i farmaci prescritti ed esamina i loro effetti collaterali. L'insonnia è uno degli effetti collaterali. Giorgio annota questo problema nella cartella clinica e suggerisce a Giovanni di recarsi presso la clinica per cambiare farmaci. Giovanni è d'accordo e, quindi, Giorgio digita una nota che gli ricorderà di effettuare una chiamata, quando rientrerà in clinica, per fissare un appuntamento col medico curante. Giorgio conclude la visita e il sistema critpa le registrazioni effettuate nella cartella di Giovanni.

Una volta concluse le sue visite, Giorgio ritorna in clinica e aggiorna il database con le cartelle dei pazienti visitati. Il sistema genera la lista delle chiamate che Giorgio dovrà effettuare per quei pazienti che richiedono controlli aggiuntivi e per fissare gli appuntamenti con i medici curanti.

Figura 8.10 Una storia utente per il sistema Mentcare.

6. accedere al database dei farmaci con le informazioni sugli effetti collaterali;
7. generare la lista delle chiamate da effettuare.

Se siete un tester di release, esamineste questo scenario, svolgete il ruolo di Giorgio e osservate come risponde il sistema ai vari input. Come Giorgio, anche voi potrete commettere degli errori, per esempio digitando la frase chiave sbagliata per decodificare le cartelle cliniche. Questo significa controllare la risposta del sistema agli errori. Dovrete annotare attentamente qualsiasi problema che si verifica, inclusi i problemi relativi alle prestazioni del sistema. Se il sistema è troppo lento, questo potrebbe cambiare il modo in cui esso sarà utilizzato. Per esempio, se il sistema impiega troppo tempo per criptare una cartella, un utente che ha fretta potrebbe saltare questo passaggio. Se poi questo utente perdesse i portatili, una persona non autorizzata potrebbe accedere alle informazioni dei pazienti.

Quando si adotta un approccio basato sugli scenari, di solito si testano numerosi requisiti all'interno dello stesso scenario. Pertanto, oltre ai singoli requisiti, occorre verificare anche che le combinazioni dei requisiti non causino problemi.

8.3.3 Test delle prestazioni

Una volta che il sistema è stato completamente integrato, è possibile testare le proprietà più evidenti, come le prestazioni e l'affidabilità. I test delle prestazioni devono essere progettati per garantire che il sistema possa elaborare il carico di lavoro previsto. Di solito, questo richiede l'esecuzione di una serie di test, dove il carico viene aumentato progressivamente finché le prestazioni del sistema non diventino inaccettabili.

Analogamente ad altri tipi di test, anche i test delle prestazioni devono dimostrare che il sistema soddisfa i suoi requisiti e scoprire i problemi e i difetti del sistema. Per verificare che i requisiti delle prestazioni siano soddisfatti, occorre costruire un profilo operativo (Capitolo 11), ovvero una serie di test che riflettono il mix reale di lavoro che sarà svolto dal sistema. Per esempio, se il 90% delle transazioni di un sistema è di tipo A, il 5% di tipo B e il resto di tipo C, D ed E, allora dovete progettare un profilo operativo in modo che la maggior parte dei test sia di tipo A; altrimenti, non otterrete test accurati delle prestazioni del sistema.

Ovviamente, questo approccio non è necessariamente il migliore per scoprire i difetti di un sistema. L'esperienza ha dimostrato che un modo efficace di identificare i difetti consiste nel progettare test prossimi ai limiti del sistema. Nei test delle prestazioni, questo significa stressare il test con richieste che sono oltre i limiti richiesti dalla progettazione del software; questo approccio è detto *stress test*.

Supponete di voler testare un sistema di elaborazione delle transazioni che è progettato per elaborare fino a 300 transazioni al secondo. Iniziate a testare il sistema con meno di 300 transazioni al secondo; poi, gradualmente aumentate il carico di lavoro a oltre 300 transazioni al secondo, finché non sarà ben oltre il carico massimo previsto dal progetto, con conseguente malfunzionamento del sistema.

Lo stress test aiuta a scoprire due cose.

1. Testare il comportamento in caso di malfunzionamento del sistema. Ci sono casi in cui, per una combinazione imprevista di eventi, il carico di lavoro del sistema supera il massimo previsto. In questi casi, il malfunzionamento del sistema non deve provocare danni ai dati o perdite inaspettate dei servizi per gli utenti. Lo stress test verifica che il sovraccarico provochi un “fallimento graduale” del sistema, non un collasso improvviso sotto il suo carico.
2. Scoprire quei difetti che emergono soltanto quando il sistema opera a pieno carico. Sebbene si possa obiettare che questi difetti difficilmente possano provocare malfunzionamenti del sistema durante l'uso normale, tuttavia ci sono combinazioni insolite di circostanze che possono portare il sistema a operare oltre i suoi limiti.

Lo stress test è particolarmente importante per i sistemi distribuiti che operano in una rete di processori. Questi sistemi spesso subiscono un drastico peggioramento delle prestazioni quando sono pesantemente caricati. La rete viene inondata di dati di coordinazione che devono essere scambiati tra i vari processori. I processi diventano sempre più lenti, in quanto restano in attesa di dati da parte di altri processi. Lo stress test aiuta a scoprire quando inizia questo peggioramento delle prestazioni, in modo da poter aggiungere opportuni controlli che impediscano al sistema di accettare transazioni oltre questo limite.

8.4 Test degli utenti

Il test degli utenti è una fase del processo di test di un sistema in cui gli utenti o clienti danno i loro input e suggerimenti sui test del sistema. Questo potrebbe richiedere un test formale del sistema che è stato commissionato da un fornitore esterno; in alternativa, potrebbe essere un processo informale dove gli utenti sperimentano un nuovo prodotto software per vedere se è di loro gradimento e se soddisfa le loro esigenze. Il test degli utenti è essenziale, anche dopo che sono stati completati i test del sistema e della release. I condizionamenti derivanti dall'ambiente di lavoro degli utenti possono avere effetti rilevanti sulle prestazioni, affidabilità, utilizzabilità e robustezza di un sistema.

È praticamente impossibile per lo sviluppatore di un sistema replicare l'ambiente reale di lavoro del sistema, in quanto i test eseguiti nell'ambiente dello sviluppatore sono inevitabilmente artificiali. Per esempio, un sistema che è stato progettato per essere utilizzato in un ospedale viene utilizzato in una clinica dove avvengono altre cose, come emergenze di pazienti e conversazioni con i parenti. Tutte queste cose influiscono sull'uso del sistema, ma gli sviluppatori non possono includerle nei loro ambienti di sviluppo.

Ci sono tre tipi di test degli utenti.

1. *Alpha test.* Alcuni utenti selezionati operano a fianco del team di sviluppo per testare le prime release del software.
2. *Beta test.* Una release del software viene messa a disposizione di un folto gruppo di utenti per consentire loro di sperimentare il software e segnalare i problemi che scoprano agli sviluppatori del sistema.
3. *Test di accettazione.* I clienti testano un sistema per decidere se è pronto per essere accettato e distribuito nell'ambiente d'uso dei clienti.

Nell'alpha test, gli utenti e gli sviluppatori lavorano insieme per testare un sistema durante il suo sviluppo. Questo significa che gli utenti possono scoprire problemi che non sono facilmente identificabili dai membri del team di test. Gli sviluppatori possono basarsi soltanto sui requisiti del sistema, ma questi spesso

non rispecchiano altri fattori che influiscono sull'uso reale del sistema. Gli utenti possono quindi fornire utili informazioni sull'uso reale del software che possono aiutare la progettazione a sviluppare test più realistici.

L'alpha test viene spesso utilizzato per sviluppare app o prodotti software. Utenti esperti di questi prodotti potrebbero chiedere di essere coinvolti nel processo di alpha test, in quanto questo fornisce loro le prime informazioni sulle nuove caratteristiche del sistema. Si riduce anche il rischio che le modifiche del software non segnalate agli utenti possano avere effetti nocivi sull'uso reale del software. Tuttavia, l'alpha test può essere utilizzato anche durante lo sviluppo di software per particolari clienti. I metodi di sviluppo agile raccomandano il coinvolgimento degli utenti nel processo di sviluppo, e gli utenti dovrebbero svolgere un ruolo chiave nella progettazione dei test del sistema.

Il beta test si effettua quando una prima, a volte incompleta, release del sistema software viene messa a disposizione di un numeroso gruppo di clienti per essere valutata. I beta tester possono essere clienti selezionati che hanno già utilizzato il sistema. In alternativa, il software può essere messo a disposizione di tutti per essere utilizzato da chiunque sia interessato a sperimentarlo.

Il beta test è indicato principalmente per i prodotti software che sono utilizzati in differenti configurazioni d'impiego. Questo è importante in quanto, diversamente dagli sviluppatori di prodotti personalizzati, lo sviluppatore del prodotto non è in grado di limitare l'ambiente operativo del software. È impossibile che gli sviluppatori di un prodotto conoscano e replichino tutte le configurazioni in cui il prodotto sarà utilizzato. Il beta test viene quindi utilizzato per scoprire i problemi di interazione tra il software e le caratteristiche del suo ambiente operativo. Il beta test è anche una forma di commercializzazione del software. I clienti imparano le caratteristiche del sistema e cosa questo può fare per loro.

I test di accettazione sono una parte intrinseca dello sviluppo dei sistemi personalizzati. I clienti testano un sistema, utilizzando i loro dati, e decidono se deve essere accettato dallo sviluppatore del sistema. L'accettazione implica che il pagamento finale del software può essere fatto.

La Figura 8.11 mostra le sei fasi del processo dei test di accettazione.

1. *Definire i criteri di accettazione.* Questa fase in teoria dovrebbe avvenire all'inizio del processo, prima che venga firmato il contratto per il sistema. I criteri di accettazione dovrebbero essere una parte del contratto ed essere approvati dal cliente e dallo sviluppatore. In pratica, però, è difficile definire i criteri così presto nel processo. I requisiti dettagliati del sistema potrebbero non essere disponibili e, di solito, essi cambiano quasi sicuramente durante il processo di sviluppo.
2. *Pianificare i test di accettazione.* In questa fase si decidono risorse, tempi e budget per i test di accettazione. Dovranno essere discussi anche i requisiti dei test e l'ordine in cui le caratteristiche del sistema dovranno essere testa-

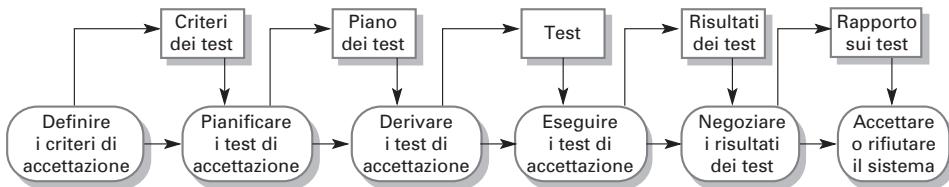


Figura 8.11 Il processo dei test di accettazione.

te. Dovranno essere definiti i rischi del processo di test nei casi di malfunzionamento del sistema e di prestazioni insoddisfacenti, e le tecniche per mitigare tali rischi.

3. *Derivare i test di accettazione.* Una volta stabiliti i criteri di accettazione, dovranno essere progettati i test per controllare se un sistema può essere accettato oppure no. I test di accettazione dovranno riguardare le caratteristiche funzionali e non funzionali (per esempio, le prestazioni) del sistema. In teoria, i test dovrebbero fornire una copertura completa dei requisiti del sistema. In pratica, è difficile stabilire criteri di accettazione assolutamente oggettivi. Spesso un test non può dimostrare con assoluta certezza che un criterio di accettazione è stato soddisfatto.
4. *Eseguire i test di accettazione.* I test concordati vengono eseguiti sul sistema. In teoria, questi test dovrebbero svolgersi nell'ambiente reale in cui il sistema sarà utilizzato, ma ciò potrebbe risultare impraticabile e dannoso. Pertanto, sarebbe opportuno configurare un apposito ambiente in cui eseguire questi test. È difficile automatizzare questo processo, in quanto alcuni test di accettazione potrebbero interessare le interazioni tra gli utenti finali e il sistema. Potrebbe essere necessario addestrare gli utenti finali.
5. *Negoziare i risultati dei test.* È molto improbabile che tutti i test di accettazione concordati saranno superati dal sistema e che non ci saranno problemi con il sistema. Se così fosse, i test di accettazione si ritengono completati e il sistema può essere consegnato al cliente. Di solito, invece, emerge qualche problema durante i test. In questi casi, lo sviluppatore e il cliente devono negoziare per decidere se il sistema è sufficientemente buono per essere utilizzato; devono anche mettersi d'accordo sul modo in cui lo sviluppatore dovrà risolvere i problemi che sono stati identificati.
6. *Accettare o rifiutare il sistema.* Questa fase prevede un incontro tra gli sviluppatori e il cliente per decidere se il sistema può essere accettato oppure no. Se il sistema non è sufficientemente buono per essere utilizzato, allora è necessario un ulteriore sviluppo per correggere i problemi che sono emersi. Fatto questo, deve essere ripetuta la fase dei test di accettazione.

Si potrebbe pensare che i test di accettazione siano una mera questione contrattuale. Se un sistema non supera i test di accettazione, allora deve essere rifiutato e i pagamenti sospesi. In effetti, la realtà è più complessa. I clienti vogliono utilizzare il software quanto prima possibile, per trarre i massimi benefici dalla sua immediata disponibilità. Potrebbero aver bisogno di acquistare nuovo hardware, addestrare il personale e modificare le loro procedure. Potrebbero essere disposti ad accettare il software, indipendentemente dai problemi emersi, in quanto i costi del mancato utilizzo del software potrebbero essere maggiori di quelli di aggirare i problemi.

Il risultato delle negoziazioni potrebbe essere un'accettazione condizionata del sistema. Il cliente potrebbe accettare il sistema in modo che possa iniziare la sua consegna. Il fornitore del sistema si impegna a riparare i problemi urgenti e a consegnare al cliente una nuova versione al più presto possibile.

Nei metodi agili, come la programmazione estrema, potrebbe mancare un'attività distinta per i test di accettazione. L'utente finale è parte del team di sviluppo (ovvero è un alpha tester) e specifica i requisiti del sistema in termini di storie utente; è anche responsabile della definizione dei test, che decidono se il software sviluppato supporta le storie utente. Questi test sono quindi equivalenti ai test di accettazione. I test sono automatizzati, e lo sviluppo non potrà procedere finché i test di accettazione delle storie non saranno stati eseguiti con successo.

Quando gli utenti sono integrati in un team di sviluppo del software, in teoria, dovrebbero essere “tipici” utenti che hanno una conoscenza generale sul modo in cui il sistema sarà utilizzato. In pratica, è difficile trovare tali utenti e, quindi, i test di accettazione potrebbero non essere una vera replica di come il sistema sarà utilizzato nella pratica. Inoltre, il requisito di avere test automatici limita la flessibilità di testare i sistemi interattivi. Per questi sistemi, il test di accettazione potrebbe richiedere gruppi di utenti finali che usano il sistema come se fosse parte del loro lavoro quotidiano. Pertanto, sebbene un “utente integrato” sia in teoria un concetto attraente, tuttavia esso non implica necessariamente test di alta qualità per il sistema.

Il problema del coinvolgimento degli utenti nei team di sviluppo agile è uno dei motivi per cui molte aziende usano un mix di test agili e test più tradizionali. Il sistema può essere sviluppato utilizzando tecniche agili, ma, dopo il completamento di una release principale, si usano appositi test di accettazione per decidere se il sistema può essere accettato.

Punti chiave

- I test possono dimostrare soltanto la presenza di errori in un programma; non possono dimostrare che non ci siano altri difetti.

- Il test di sviluppo è svolto dal team di sviluppo del software. Un team separato ha la responsabilità di testare il sistema prima di rilasciarlo ai clienti. Nel processo di test degli utenti, i clienti o utenti del sistema forniscono i dati di prova e controllano che i test siano superati.
- Il test di sviluppo include i test delle unità, in cui vengono testati singoli oggetti e metodi, i test dei componenti, in cui vengono testati gruppi di oggetti correlati, e i test di sistemi, in cui vengono testati sistemi parziali o completi.
- Durante il test del software, dovreste tentare di “rompere” il software utilizzando l’esperienza e le linee guida per scegliere i tipi di test case che sono risultati efficaci nell’identificare i difetti in altri sistemi.
- Quando possibile, dovreste scrivere test automatici. I test integrati in un programma possono essere eseguiti ogni volta che viene apportata una modifica al sistema.
- Lo sviluppo con test iniziali è un approccio allo sviluppo nel quale i test sono scritti prima che il codice sia testato. Vengono apportate piccole modifiche al codice, e il codice viene rifattorizzato finché tutti i test non saranno superati.
- Il test degli scenari è utile perché replica l’uso reale di un sistema. Occorre immaginare un tipico scenario di impiego del sistema e, poi, utilizzare questo scenario per derivare i test case.
- Il test di accettazione è un processo di test degli utenti il cui scopo è stabilire se il software è sufficientemente buono da essere consegnato al cliente ed essere utilizzato nel suo ambiente operativo.

Esercizi

- 8.1 Spiegate perché non è necessario che un programma sia completamente privo di difetti prima di essere consegnato al suo cliente.
- * 8.2 Spiegate perché i test possono rivelare soltanto la presenza di errori in un programma, non la loro assenza.
- 8.3 Alcuni ritengono che gli sviluppatori non dovrebbero essere coinvolti nei test del loro codice, ma che tutti i test dovrebbero essere eseguiti da un team separato. Descrivete i vantaggi e gli svantaggi dei test eseguiti dagli stessi sviluppatori.
- * 8.4 Vi è stato chiesto di testare un metodo, chiamato *sostitisciSpazi*, che all’interno di un oggetto “Paragrafo” sostituisce le sequenze di caratteri vuoti con un solo spazio. Identificate le partizioni di test per questo esempio e derivate una serie di test per il metodo *sostitisciSpazi*.
- * 8.5 Che cos’è il test di regressione? Spiegate come l’uso di test automatici e di un framework di test, come JUnit, possa semplificare il test di regressione.
- 8.6 Il sistema MentreCare è costruito adattando un sistema informatico off-the-shelf. Quali sono le differenze tra testare un sistema simile e testare un software che è stato sviluppato utilizzando un linguaggio orientato agli oggetti come Java?
- * 8.7 Scrivete uno scenario che potrebbe essere utilizzato per agevolare i test di progettazione per il sistema delle stazioni meteorologiche.

- * 8.8 Che cosa s'intende con il termine *stress test*? Come applichereste lo stress test al sistema Mentcare?
 - 8.9 Quali vantaggi offre il coinvolgimento degli utenti nel test della release in una fase iniziale del processo di test? Ci sono svantaggi in questo coinvolgimento?
 - 8.10 Un tipico approccio al test del sistema consiste nel testare il sistema fino a esaurire il budget disponibile e poi consegnare il sistema al cliente. Spiegate l'etica di questo approccio per i sistemi che vengono consegnati ai clienti esterni.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“How to design practical test cases.” Un articolo sulla progettazione dei test case scritto da un dipendente di un’azienda giapponese che ha una buona reputazione nel rilascio di software con pochi difetti. (T. Yamaura, *IEEE Software*, 15(6), November 1998) <http://dx.doi.org/10.1109/52.730835>.

“Test-driven development.” Questo numero speciale include una buona panoramica sullo sviluppo guidato da test e alcuni articoli su come questo sviluppo è stato adottato per vari tipi di software. (*IEEE Software*, 24 (3) May/June 2007).

Exploratory Software Testing. È un libro pratico, non teorico, sul test del software; sviluppa i concetti del precedente libro di Whittaker, *How to Break Software*. L'autore presenta una serie di linee guida basate sull'esperienza per la progettazione del software. (J. A. Whittaker, 2009, Addison-Wesley).

How Google Tests Software. È un libro sui test di sistemi cloud su larga scala che espone tutta una serie di nuove sfide rispetto alle applicazioni software personalizzate. Sebbene io non creda che l'approccio Google possa essere utilizzato direttamente, tuttavia questo libro include alcune lezioni interessanti per eseguire test di sistemi su larga scala (J. Whittaker, J. Arbon e J. Carollo 2012, Addison-Wesley).

9

Evoluzione del software

Questo capitolo ha due obiettivi: spiegare perché l'evoluzione del software è una parte importante dell'ingegneria del software e descrivere le sfide connesse alla manutenzione di una larga base di sistemi software che sono sviluppati da molti anni. Dopo aver letto questo capitolo:

- capirete che i sistemi software devono adattarsi ed evolversi per poter restare utili e che le modifiche e l'evoluzione del software devono essere considerate come parte integrale dell'ingegneria del software;
- capirete che cosa s'intende per sistemi ereditati e perché questi sistemi sono importanti per le aziende;
- capirete come i sistemi ereditati possono essere valutati per decidere se devono essere scartati, mantenuti, reingegnerizzati o rimpiazzati;
- conoscerete i vari tipi di manutenzione del software e i fattori che influiscono sui costi di aggiornamento dei sistemi software ereditati.

- 9.1 Processi evolutivi
- 9.2 Sistemi ereditati
- 9.3 Manutenzione del software

I grandi sistemi software di solito hanno una lunga vita. Per esempio, i sistemi militari o delle infrastrutture, come i sistemi per il controllo del traffico aereo, possono raggiungere e superare 30 anni di vita. I sistemi software aziendali spesso hanno più di 10 anni di vita. Il costo di questo software è molto elevato, quindi le aziende utilizzano un sistema software per molti anni in modo da avere un ritorno sugli investimenti. I prodotti software e le app di successo sono stati introdotti alcuni anni fa, ma quasi tutti gli anni vengono aggiornati con nuove release. Per esempio, la prima versione di Microsoft Word è stata introdotta nel 1983, quindi più di 30 anni fa.

Durante la loro vita, i sistemi software devono cambiare se vogliono restare utili. I cambiamenti delle aziende e delle aspettative degli utenti generano nuovi requisiti per il software. Parti del software potrebbero richiedere delle modifiche per correggere gli errori che si sono scoperti durante il loro utilizzo, per essere adattate a nuove unità hardware e piattaforme software, e per migliorare le loro prestazioni o altre caratteristiche non funzionali. I prodotti software e le app devono evolversi per far fronte ai cambiamenti delle piattaforme e alle nuove caratteristiche introdotte dai loro concorrenti. I sistemi software, quindi, si adattano e si evolvono durante il loro ciclo di vita, dal rilascio della prima versione fino all'ultima versione.

Le aziende devono cambiare il loro software per continuare ad avere un ritorno dagli investimenti iniziali. I sistemi software sono risorse critiche per le aziende; per questo devono investire nell'aggiornamento del software per mantenere il valore di tali risorse. Molte grandi aziende spendono più nella manutenzione dei sistemi software esistenti che nello sviluppo di nuovi sistemi. Alcuni dati statistici indicano che i costi di evoluzione sono compresi tra il 60% e il 90% dei costi totali del software (Lientz e Swanson 1980; Erlich 2000). Jones (Jones 2006) ha scoperto che, nel 2006, circa il 75% del personale di sviluppo negli USA era impegnato nell'evoluzione del software, e ha indicato che questa percentuale difficilmente sarebbe diminuita nel prossimo futuro.

L'evoluzione del software è particolarmente costosa nei sistemi aziendali quando singoli sistemi software fanno parte di un più ampio “sistema di sistemi”. In questi casi, non si possono considerare semplicemente le modifiche di un sistema, ma occorre esaminare anche come queste modifiche influenzano il più ampio sistema di sistemi. La modifica di un sistema potrebbe significare che anche altri sistemi nel suo ambiente devono evolversi per far fronte a tale modifica.

Ne consegue che, oltre a capire e analizzare l'impatto di una modifica su un sistema, occorre anche valutare gli effetti di tale modifica su altri sistemi dello stesso ambiente operativo. Hopkins e Jenkins (Hopkins e Jenkins 2008) hanno coniato il termine *brownfield software development* per descrivere i casi in cui un sistema software deve essere sviluppato e gestito in un ambiente dove dipende da altri sistemi software.

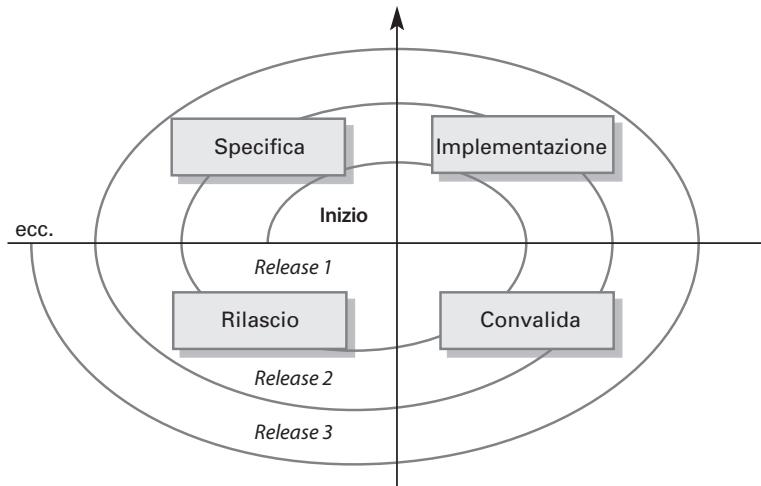


Figura 9.1 Modello a spirale di sviluppo ed evoluzione del software.

I requisiti dei sistemi software installati cambiano al variare delle esigenze aziendali e del loro ambiente operativo; per questo, vengono create periodicamente nuove release dei sistemi che incorporano le modifiche e gli aggiornamenti necessari. L'ingegneria del software si può così immaginare come un processo a spirale, in cui la specifica dei requisiti, la progettazione, l'implementazione e i test continuano durante tutta la vita del sistema (Figura 9.1). Si inizia creando la Release 1 del sistema; una volta consegnata, si propongono nuove modifiche e quasi immediatamente inizia lo sviluppo della Release 2. In effetti, la necessità di aggiornare il software può manifestarsi ancora prima della consegna del sistema, cosicché la successiva release del software può iniziare il suo sviluppo prima ancora che la versione iniziale sia stata rilasciata.

Negli ultimi 10 anni, il tempo fra le iterazioni della spirale si è ridotto drasticamente. Prima della vasta diffusione di Internet, le nuove versioni di un sistema software venivano rilasciate ogni 2 o 3 anni. Oggi, a causa della pressante concorrenza e dell'esigenza di rispondere rapidamente al feedback degli utenti, l'intervallo di tempo tra due successive release delle stesse app o degli stessi sistemi basati sul Web si è ridotto a poche settimane.

Questo modello di evoluzione del software si può applicare quando la stessa azienda è responsabile dello sviluppo del software per tutto il suo ciclo di vita. Dallo sviluppo all'evoluzione c'è una transizione senza soluzione di continuità; gli stessi processi e metodi di sviluppo vengono applicati per tutto il ciclo di vita del software. I prodotti software e le app vengono sviluppati secondo questo approccio.

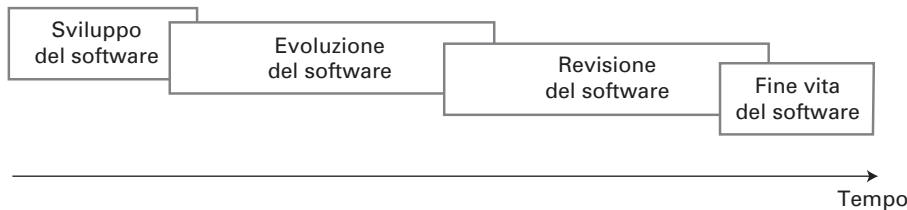


Figura 9.2 Evoluzione e revisione.

L’evoluzione del software personalizzato, invece, di solito segue un modello differente. Il cliente del sistema potrebbe pagare una società per sviluppare il software e poi assumersi la responsabilità del supporto tecnico e dell’evoluzione utilizzando il proprio personale. In alternativa, il cliente del software potrebbe stipulare un contratto distinto con un’altra società di software per il supporto e l’evoluzione del sistema.

In questo caso spesso ci sono discontinuità nel processo evolutivo. I documenti dei requisiti e di progettazione potrebbero non essere trasmessi da una società all’altra; le società potrebbero fondersi o riorganizzarsi, ereditare il software da altre società, e scoprire poi che questo software deve essere modificato. Quando la transizione dallo sviluppo all’evoluzione non è continua, il processo di modifica del software dopo la sua consegna si chiama *manutenzione del software*. Come dirò più avanti in questo capitolo, la manutenzione richiede altre attività, come la comprensione del programma, in aggiunta alle normali attività di sviluppo del software.

Rajlich e Bennett (Rajlich e Bennett 2000) propongono una visione alternativa del ciclo di vita dell’evoluzione del software per i sistemi aziendali. Secondo questo modello c’è una distinzione tra evoluzione e revisione. L’evoluzione è la fase in cui vengono apportate significative modifiche all’architettura e alle funzionalità del software. Durante la revisione, le uniche modifiche che vengono apportate sono relativamente modeste, ma essenziali. Queste fasi si sovrappongono, come mostra la Figura 9.2.

Secondo Rajlich e Bennett, quando il software viene utilizzato con successo per la prima volta, vengono proposte e implementate molte modifiche dei requisiti richiesti dagli stakeholder. Questa è la fase dell’evoluzione. Quando invece viene modificato il software, la sua struttura tende a degradarsi, e le modifiche del sistema diventano sempre più costose. Questo accade spesso dopo pochi anni di utilizzo quando sono richieste anche altre modifiche dell’ambiente operativo, come le unità hardware e i sistemi operativi. In un certo stadio del ciclo di vita, il software raggiunge un punto di transizione dove le modifiche significative e l’implementazione di nuovi requisiti diventano sempre meno convenienti. In questo stadio, il software passa dall’evoluzione alla revisione.

Durante la fase di revisione, il software è ancora utile, ma subisce soltanto piccole modifiche tattiche. In questa fase, l'azienda inizia a considerare il modo in cui può essere sostituito il software. Nella fase finale, il software può essere ancora utilizzato, ma vengono apportate soltanto le modifiche più essenziali. Gli utenti devono aggirare i problemi che incontrano. Infine, il software viene ritirato e messo fuori uso. Questo spesso richiede ulteriori costi, in quanto i dati devono essere trasferiti dal vecchio sistema a quello nuovo.

9.1 Processi evolutivi

Come in tutti i processi software, non esiste un processo evolutivo standard o un processo di modifica standard per il software. Il processo evolutivo più appropriato per un sistema software dipende dal tipo di software che si sta mantenendo, dai processi di sviluppo utilizzati in un'azienda e dall'esperienza delle persone coinvolte nel processo. Per alcuni tipi di sistemi, come le app per dispositivi mobili, l'evoluzione può essere un processo informale in cui le richieste di modifica provengono principalmente dalle conversazioni tra gli utenti del sistema e gli sviluppatori. Per altri tipi di sistemi, come i sistemi critici integrati, l'evoluzione del software può essere formalizzata, tramite documenti strutturati che vengono prodotti in ogni stadio del processo.

Le proposte di modifiche del sistema, formali o informali, sono la guida per l'evoluzione dei sistemi in tutte le aziende. In una proposta di modifica, uno o più individui possono suggerire modifiche e aggiornamenti per un sistema software esistente. Queste proposte possono includere requisiti esistenti che non sono stati implementati nella precedente release del sistema, richieste di nuovi requisiti e correzioni di bug da parte degli stakeholder del sistema, e nuove idee per perfezionare il software da parte del team di sviluppo. Come illustrato nella Figura 9.3, i processi di identificazione delle modifiche e di evoluzione del sistema sono ciclici e continuano per tutta la vita del sistema.

Prima che una proposta di modifica sia accettata, occorre un'analisi del software per stabilire quali componenti devono essere modificati. Questa analisi consente di valutare i costi e l'impatto della modifica. Questo fa parte del processo generale di gestione delle modifiche, che dovrebbe garantire anche che in ogni release del sistema siano incluse le versioni corrette dei componenti. Il Capitolo 22 tratta la gestione delle modifiche e della configurazione.

La Figura 9.4 mostra alcune delle attività coinvolte nell'evoluzione del software. Il processo include le attività fondamentali di analisi delle modifiche, pianificazione delle release, implementazione del sistema e rilascio di un sistema ai clienti. Vengono stimati il costo e l'impatto di una modifica per sapere quanto il sistema viene influenzato dalla modifica e quanto potrebbe costare l'implementazione della modifica.

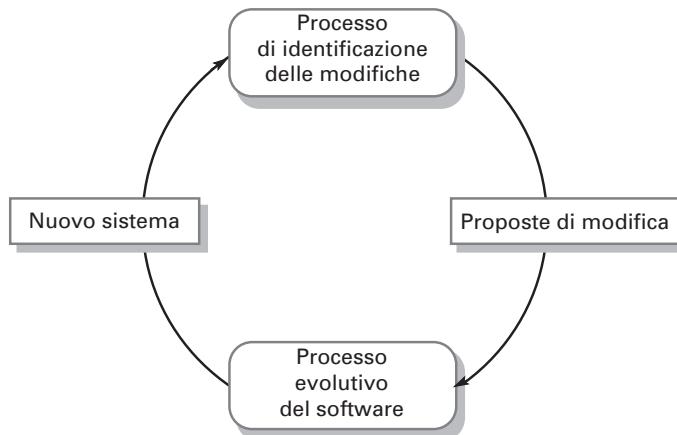


Figura 9.3 Identificazione delle modifiche e processi evolutivi.

Se le modifiche proposte vengono accettate, viene pianificata una nuova release del sistema. Durante questa pianificazione, vengono esaminate tutte le modifiche proposte (correzione degli errori, adattamenti e nuove funzionalità); poi, si decide quali modifiche implementare nella successiva versione del sistema. Le modifiche vengono implementate e convalidate, e viene rilasciata una nuova versione del sistema. Il processo si ripete con una nuova serie di modifiche proposte per la successiva release.

Nei casi in cui sviluppo ed evoluzione sono processi integrati, l'implementazione delle modifiche è semplicemente un'iterazione del processo di sviluppo. Le revisioni del sistema vengono progettate, implementate e testate. L'unica differenza tra lo sviluppo iniziale e l'evoluzione è che deve essere esaminato il feedback del cliente quando si pianificano le nuove release di un'applicazione.

Quando sono coinvolti più team di sviluppo, una differenza importante tra sviluppo ed evoluzione è che il primo stadio dell'implementazione delle modifiche richiede la comprensione del programma. Durante questa fase di comprensione, i nuovi sviluppatori devono capire come è strutturato il programma, come

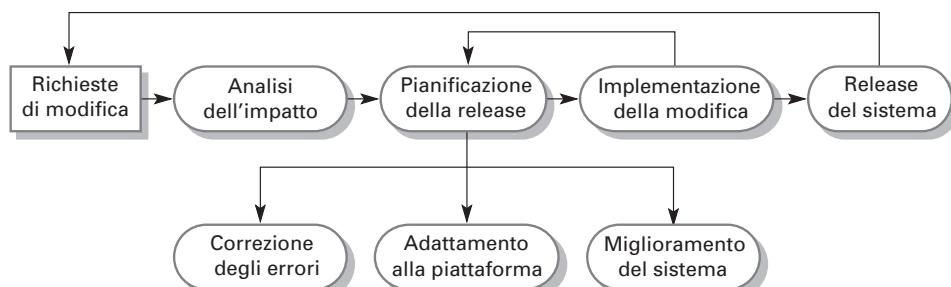


Figura 9.4 Un modello generale del processo evolutivo del software.



Figura 9.5 Implementazione delle modifiche.

vengono fornite le funzionalità e come la modifica proposta potrebbe influire sul programma. Questa comprensione è necessaria per garantire che la modifica implementata non causi nuovi problemi quando viene introdotta nel sistema esistente.

Se la specifica dei requisiti e i documenti di progettazione sono disponibili, questi dovrebbero essere aggiornati durante il processo evolutivo per riflettere le modifiche che sono state richieste (Figura 9.5). I nuovi requisiti del software dovrebbero essere scritti e, successivamente, analizzati e convalidati. Se il progetto è stato documentato utilizzando i modelli UML, questi modelli devono essere aggiornati. Le modifiche proposte possono essere prototipate come parte del processo di analisi delle modifiche, quando vengono definite le implicazioni e i costi per apportare le modifiche.

Le richieste di modifiche a volte possono riguardare problemi del sistema operativo che devono essere risolti in fretta; queste modifiche urgenti possono sorgere per tre motivi:

1. se si verifica un grave errore di sistema che deve essere riparato per permettere la continuazione delle normali operazioni o per rimediare a una grave vulnerabilità della sicurezza;
2. se le modifiche all'ambiente operativo del sistema hanno effetti inattesi che disturbano le normali operazioni;
3. se ci sono cambiamenti imprevisti nell'azienda che sta utilizzando il sistema, come la comparsa di nuovi concorrenti o l'introduzione di nuove normative che riguardano il sistema.

In questi casi, la necessità di apportare velocemente le modifiche potrebbe impedire l'aggiornamento di tutta la documentazione del software. Anziché modificare i requisiti e il progetto, si effettua una correzione d'emergenza del programma per risolvere il problema immediato (Figura 9.6). Il pericolo qui è che i requisiti, il progetto e il codice del software perdano la loro coerenza. Mentre si sta documentando la modifica dei requisiti e del progetto, potrebbero diventare urgenti altre correzioni del software. Queste correzioni hanno priorità sulla documentazione. Alla fine, la modifica originale viene dimenticata, e la documentazione del sistema e il codice non vengono più riallineati. Il problema di mantenere più rappresentazioni di uno stesso sistema è uno degli motivi per minimizzare la documentazione, che è fondamentale nei processi di sviluppo agile.



Figura 9.6 Il processo di correzione di emergenza.

Le correzioni di emergenza devono essere completate il più velocemente possibile. Si sceglie una soluzione veloce e fattibile, anziché la soluzione migliore, se si tratta di un problema che riguarda la struttura del sistema. Questo tende ad accelerare il processo di invecchiamento del software, in quanto i cambiamenti futuri diventano progressivamente più difficili e i costi di manutenzione aumentano. In teoria, quando viene effettuata una correzione d'emergenza del codice, il nuovo codice dovrebbe essere rifattorizzato e migliorato per evitare l'invecchiamento del programma. Ovviamente è possibile riutilizzare il codice della correzione. Una migliore soluzione del problema potrebbe essere trovata se fosse disponibile più tempo per l'analisi.

I metodi e i processi agili, descritti nel Capitolo 3, possono essere utilizzati sia per l'evoluzione sia per lo sviluppo dei programmi. Poiché questi metodi si basano sullo sviluppo incrementale, la transizione dallo sviluppo agile all'evoluzione postconsegna deve essere graduale.

Tuttavia, potrebbero sorgere dei problemi durante il passaggio dal team di sviluppo al team responsabile dell'evoluzione del sistema. Ci sono due casi potenzialmente problematici.

1. Il team di sviluppo ha adottato un approccio agile, mentre il team di evoluzione preferisce un approccio basato su piani. Il team di evoluzione potrebbe aspettarsi una documentazione dettagliata a supporto dell'evoluzione, ma tale documentazione raramente viene prodotta nei processi agili. Potrebbe non esserci una dichiarazione definitiva sui requisiti del sistema che possono essere cambiati mentre le modifiche vengono apportate al sistema.
2. È stato adottato un approccio basato su piani per lo sviluppo, ma il team di evoluzione preferisce utilizzare metodi agili. In questo caso, il team di evoluzione potrebbe essere costretto a iniziare da zero per sviluppare i test automatizzati. Il codice nel sistema potrebbe non essere stato rifattorizzato e semplificato, come prevede lo sviluppo agile. In questo caso, potrebbe essere richiesta qualche opera di reingegnerizzazione del programma per migliorare il codice prima che possa essere utilizzato in un processo di sviluppo agile.

Le tecniche agili, come lo sviluppo guidato da test e i test di regressione automatici, sono utili quando vengono apportate le modifiche al sistema. Queste modifiche possono essere espresse come storie utente, e il coinvolgimento degli utenti può aiutare a definire le priorità delle modifiche che sono richieste in un sistema

operativo. L'approccio Scrum, che si focalizza sul backlog di lavoro da svolgere, può aiutare a definire le priorità delle modifiche più importanti per il sistema. In sintesi, l'evoluzione richiede semplicemente di continuare il processo di sviluppo agile.

Potrebbe essere necessario modificare i metodi agili utilizzati nello sviluppo quando questi metodi vengono utilizzati nella manutenzione e nell'evoluzione dei programmi. Potrebbe essere praticamente impossibile coinvolgere gli utenti nel team di sviluppo quando le proposte di modifica provengono da un folto gruppo di stakeholder. Potrebbe essere necessario interrompere i cicli di sviluppo brevi per gestire le correzioni di emergenza, e il gap tra le release potrebbe allungarsi per evitare di disturbare i processi operativi.

9.2 Sistemi ereditati

Le grandi aziende hanno iniziato a informatizzare le loro operazioni negli anni '60. Da oltre 50 anni, quindi, vengono introdotti nuovi sistemi software. Molti di questi sono stati rimpiazzati (più di una volta, in alcuni casi), perché le aziende cambiano e si evolvono. Tuttavia, molti dei vecchi sistemi sono ancora in uso e giocano un ruolo critico nella gestione delle aziende. Questi vecchi sistemi sono detti *sistemi ereditati* (*legacy system*).

I sistemi ereditati si basano su tecnologie e linguaggi che non sono più utilizzati nello sviluppo di nuovi sistemi. Sono stati mantenuti per un lungo periodo, e la loro struttura potrebbe essere compromessa a causa delle numerose modifiche che hanno subito. Il software ereditato potrebbe essere dipendente dal vecchio hardware, come i mainframe, e potrebbe essere associato a procedure e a processi ereditati. Di solito, è impossibile passare a processi gestionali più efficienti, in quanto il software ereditato non può essere modificato per supportare nuovi processi.

I sistemi ereditati non sono semplicemente sistemi software, ma sistemi socio-tecnici più complessi che includono hardware, software, librerie e altri processi di supporto al software e alla gestione aziendale. La Figura 9.7 mostra la parti logiche di un sistema ereditato e le loro relazioni.

1. *Hardware del sistema.* I sistemi ereditati sono stati scritti per unità hardware che non sono più disponibili, sono costose da mantenere e potrebbero essere incompatibili con le attuali politiche aziendali sugli acquisti di tecnologie informatiche.
2. *Software di supporto.* Il sistema ereditato può basarsi su alcuni software di supporto del sistema operativo e utilità fornite dal costruttore dell'hardware direttamente ai compilatori usati per lo sviluppo del sistema. Tutto questo potrebbe essere obsoleto e non più supportato dai fornitori iniziali.

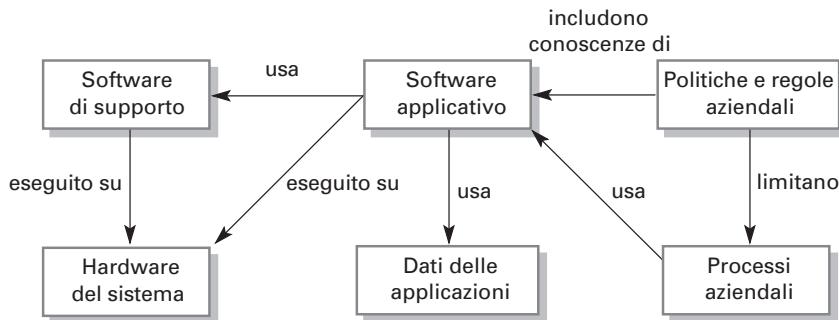


Figura 9.7 Gli elementi di un sistema ereditato.

3. *Software applicativo*. Il sistema applicativo che fornisce i servizi aziendali di solito è composto da alcuni programmi applicativi che sono stati sviluppati in periodi differenti. Alcuni di questi programmi faranno parte anche di altri sistemi applicativi.
4. *Dati delle applicazioni*. Questi dati sono elaborati dal sistema applicativo. In molti sistemi ereditati si sono accumulati immensi volumi di dati durante la vita del sistema. Questi dati potrebbero essere incoerenti fra loro, potrebbero essere duplicati su molti file o distribuiti su un certo numero di database differenti.
5. *Processi aziendali*. Questi processi sono utilizzati nelle aziende per raggiungere determinati obiettivi; per esempio, l'apertura di una pratica assicurativa in un'agenzia di assicurazioni oppure l'accettazione di un ordine di prodotti e la relativa lavorazione in un'azienda di produzione. I processi aziendali possono essere progettati attorno a un sistema ereditato e vincolati dalle funzionalità che esso offre.
6. *Politiche e regole aziendali*. Sono le definizioni di come devono essere gestite le aziende e i relativi vincoli. L'uso di sistemi applicativi ereditati può essere integrato in queste politiche e regole.

Un modo alternativo di vedere i componenti di un sistema ereditato è di immaginarli come una serie di strati (illustrati nella Figura 9.8).

Ogni strato dipende dallo strato immediatamente inferiore e si interfaccia con esso. Se le interfacce fossero mantenute, si potrebbe modificare uno strato senza influenzare gli strati adiacenti. In pratica questo encapsulamento è una semplificazione eccessiva, e le modifiche di uno strato del sistema possono richiedere cambiamenti degli strati superiori e inferiori a quello modificato, per i seguenti motivi.

1. Modificare uno strato del sistema può introdurre nuove funzionalità, e gli strati più alti potrebbero essere modificati per trarne vantaggio; per esempio, introducendo un nuovo database nello strato del software di supporto,



Figura 9.8 Strati di un sistema ereditato.

si potrebbe inserire una funzione che permetta l'accesso ai dati attraverso un browser web, e i processi aziendali potrebbero essere modificati per sfruttare questa funzione.

2. Modificare il software può rallentare il sistema al punto da rendere necessario il rinnovo dell'hardware per migliorarne le prestazioni. L'incremento delle prestazioni derivante dal nuovo hardware potrebbe permettere nuove modifiche al software che prima erano inattuabili.
3. Spesso è impossibile mantenere le interfacce hardware, specialmente se viene introdotto un nuovo hardware. Questo è un problema particolare nei sistemi integrati, dove c'è un collegamento stretto tra software e hardware. Potrebbe essere necessario apportare modifiche significative al software applicativo per rendere efficiente l'uso del nuovo hardware.

È difficile sapere con esattezza quanto codice ereditato è ancora in uso; l'industria informatica ha stimato che ci sono oltre 200 miliardi di linee di codice COBOL negli attuali sistemi aziendali. Il COBOL è un linguaggio di programmazione progettato per scrivere applicazioni software aziendali; è stato il principale linguaggio di sviluppo negli anni '60-'90, in particolare nel settore amministrativo e finanziario (Mitchell 2012). Questi programmi funzionano ancora oggi in modo efficiente, e le aziende che li usano non sentono la necessità di cambiarli. Tuttavia, un problema importante che devono affrontare è la mancanza di programmatore COBOL, in quanto gli sviluppatori originali delle applicazioni sono andati in pensione. Le università non insegnano più il COBOL, e i giovani ingegneri informatici sono più interessati alla programmazione con i moderni linguaggi.

La carenza di competenze è soltanto uno dei problemi di mantenere i sistemi ereditati aziendali. Fra gli altri problemi figurano la vulnerabilità delle protezioni, in quanto questi sistemi sono stati sviluppati prima della vasta diffusione di Internet, e la difficoltà di interfacciarsi con i sistemi scritti con i moderni linguaggi di programmazione. Il fornitore iniziale degli strumenti software potrebbe aver chiuso la sua attività o aver smesso di fornire il supporto agli strumenti utilizzati

per sviluppare il sistema. L'hardware del sistema potrebbe essere obsoleto e, quindi, sempre più costoso da mantenere.

Allora perché le aziende non sostituiscono questi sistemi con sistemi più moderni equivalenti? La risposta a questa domanda è semplice: tale sostituzione sarebbe troppo costosa e troppo rischiosa. Se un sistema ereditato funziona correttamente, i costi di sostituzione potrebbero superare i risparmi che si otterrebbero dalla riduzione dei costi di supporto del nuovo sistema. Scartare un sistema ereditato sostituendolo con un software più moderno metterebbe a rischio il corretto funzionamento delle operazioni e il nuovo sistema potrebbe non soddisfare le esigenze di un'azienda. I manager tentano di ridurre al minimo tali rischi e, quindi, non vogliono affrontare le incertezze dei nuovi sistemi software.

Ho sperimentato di persona alcuni di questi problemi quando sono stato coinvolto nell'analisi di un progetto di sostituzione di sistemi ereditati in una grande azienda. C'erano oltre 150 sistemi ereditati che svolgevano le attività aziendali. Si decise di sostituire tutti questi sistemi con un unico sistema ERP, mantenuto centralmente. Per vari motivi aziendali e tecnici, lo sviluppo del nuovo sistema fu un fallimento, e non si ottennero i miglioramenti previsti. Dopo avere speso più di 10 milioni di sterline, soltanto una parte del nuovo sistema era operativa e funzionava con minore efficienza dei sistemi che aveva sostituito. Gli utenti continuavano a utilizzare i vecchi sistemi, ma non riuscivano a integrarli con la parte del nuovo sistema che era stata implementata; quindi furono richiesti ulteriori interventi manuali.

Molte sono le ragioni che rendono costosa e rischiosa la sostituzione dei sistemi ereditati con nuovi sistemi.

1. Raramente esiste una specifica completa per un sistema ereditato. La specifica originale potrebbe essere stata perduta. Se esiste una specifica, molto probabilmente non è aggiornata con tutte le modifiche che sono state apportate al sistema. Pertanto, non è semplice specificare un nuovo sistema che sia funzionalmente identico a quello in uso.
2. I processi aziendali e i modi in cui operano i sistemi ereditati spesso sono inestricabilmente intrecciati. È probabile che questi processi abbiano subito un'evoluzione per sfruttare i servizi del software e per sopperire alle carenze del software. Se il sistema viene sostituito, questi processi devono essere modificati con costi e conseguenze potenzialmente imprevedibili.
3. Importanti regole aziendali potrebbero essere state integrate nel software senza essere documentate da qualche altra parte. Una regola aziendale è un vincolo che si applica ad alcune funzioni aziendali; rompere tale vincolo può avere conseguenze imprevedibili per l'azienda. Per esempio, una compagnia di assicurazioni potrebbe avere incorporato nel suo software le regole per valutare il rischio delle polizze. Se queste regole non sono rispettate, la compagnia potrebbe accettare polizze ad alto rischio, che potrebbero comportare risarcimenti futuri molto costosi.

4. Lo sviluppo di nuovo software è intrinsecamente rischioso, quindi potrebbero sorgere problemi inaspettati con il nuovo sistema, con conseguente ritardo nella consegna del nuovo sistema e a costi più elevati.

Mantenendo i sistemi ereditati in uso, si evitano i rischi connessi alla loro sostituzione; tuttavia, modificare il software esistente diventa sempre più costoso in quanto i sistemi diventano sempre più vecchi. I sistemi ereditati che sono vecchi da più di pochi anni sono particolarmente costosi da modificare per le seguenti ragioni.

1. Lo stile del programma e le convenzioni d'uso non sono coerenti, in quanto più persone hanno avuto la responsabilità di modificare il sistema. Questo problema si aggiunge alla difficoltà di capire il codice del sistema.
2. Una parte o tutto il sistema potrebbero essere stati implementati utilizzando linguaggi di programmazione obsoleti. Potrebbe essere difficile trovare persone che conoscono questi linguaggi; in questo caso, sarà necessario affidare all'esterno la manutenzione del sistema, con conseguente incremento dei costi.
3. La documentazione del sistema spesso è inadeguata e non è aggiornata. In alcuni casi, l'unica documentazione è il codice sorgente del sistema.
4. Dopo molti anni di manutenzione, di solito, la struttura del sistema si degrada, e il software diventa sempre più difficile da capire. Nuovi programmi potrebbero essere stati aggiunti e interfacciati con altre parti del sistema per specifiche esigenze.
5. Il sistema potrebbe essere stato ottimizzato per utilizzare meglio lo spazio o per aumentare la velocità di esecuzione, in modo da potere essere eseguito con efficienza su un vecchio hardware lento. Questo di solito richiede speciali ottimizzazioni del linguaggio e della macchina, e queste ottimizzazioni molto spesso generano un software difficile da capire. Ciò è causa di problemi per i programmatore che hanno imparato le moderne tecniche di ingegneria del software e che non capiscono i trucchi della programmazione che venivano utilizzati per ottimizzare il software.
6. I dati elaborati dal sistema potrebbero essere registrati in file differenti che hanno strutture incompatibili. Molti dati potrebbero essere duplicati, altri potrebbero essere incompleti, obsoleti e imprecisi. Il sistema potrebbe utilizzare più database realizzati da fornitori differenti.

Nella stessa fase, i costi di gestione e manutenzione del sistema ereditato diventano così alti che occorre necessariamente sostituirlo con un nuovo sistema. Nel prossimo paragrafo descriverò un approccio sistematico da adottare nella sostituzione di un sistema ereditato.

9.2.1 Gestione dei sistemi ereditati

Per i nuovi sistemi software sviluppati utilizzando i moderni processi di ingegneria del software, come lo sviluppo agile e le linee di prodotti software, è possibile pianificare l'integrazione dello sviluppo e dell'evoluzione del sistema. Sempre più società hanno capito che il processo di sviluppo del sistema è un processo unico che dura per tutta la vita. Separare lo sviluppo del software dalla sua evoluzione non è conveniente, perché si avrebbe un incremento dei costi. Tuttavia, come detto in precedenza, ci sono ancora molti sistemi ereditati che sono sistemi aziendali critici. Questi sistemi devono essere ampliati e adattati ai continui cambiamenti delle pratiche di e-business.

Molte organizzazioni, che hanno un budget limitato per la manutenzione e l'aggiornamento dei sistemi ereditati, devono decidere come ottenere il miglior profitto dai loro investimenti. Questo significa che occorre fare una valutazione realistica dei sistemi ereditati e poi scegliere, tra le quattro opzioni più comuni, la strategia più adatta per far evolvere questi sistemi.

1. *Scartare completamente il sistema.* Questa opzione dovrebbe essere scelta quando il sistema non dà un contributo efficace ai processi aziendali. Questo si verifica di solito quando i processi aziendali vengono modificati dopo la prima installazione del sistema e non dipendono più dal sistema ereditato.
2. *Lasciare il sistema inalterato e continuare con la manutenzione regolare.* Questa opzione dovrebbe essere scelta quando il sistema è ancora richiesto, ma non è molto stabile e gli utenti richiedono un numero relativamente modesto di modifiche.
3. *Reingegnerizzare il sistema per migliorarne la manutenibilità.* Questa opzione dovrebbe essere scelta quando la qualità del sistema si è deteriorata a causa delle modifiche e quando gli utenti richiedono ancora modifiche. Questo processo può includere lo sviluppo di nuovi componenti di interfaccia in modo che il sistema originale possa lavorare con altri sistemi più moderni.
4. *Sostituire una o tutte le parti del sistema con un sistema nuovo.* Questa opzione dovrebbe essere scelta quando alcuni fattori, come l'installazione di un nuovo hardware, rendono impossibile il funzionamento del vecchio sistema oppure quando alcuni sistemi off-the-shelf permetterebbero lo sviluppo del nuovo sistema a costi ragionevoli. In molti casi, si potrebbe adottare una strategia di sostituzione evoluzionistica, dove i componenti principali del sistema vengono sostituiti da sistemi off-the-shelf e altri componenti riutilizzabili.

Quando si valuta un sistema ereditato, lo si deve considerare da una prospettiva sia economica sia tecnica (Warren 1998). Da una prospettiva economica, si deve decidere se l'azienda ha realmente bisogno del sistema. Da una prospettiva tecni-

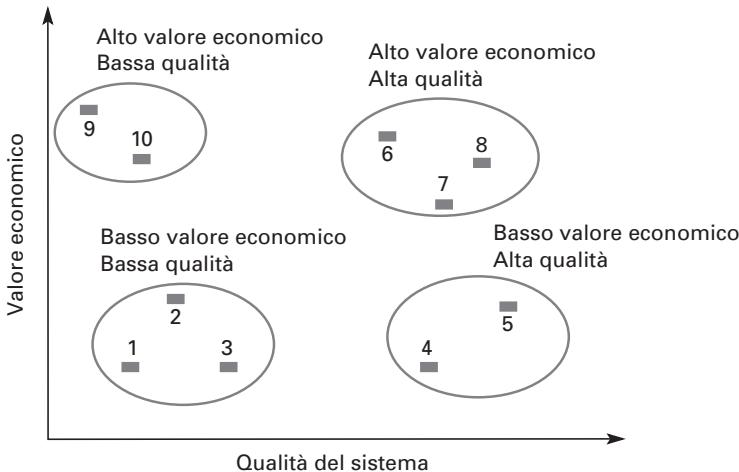


Figura 9.9 Esempio di valutazione dei sistemi ereditati.

ca, si deve valutare la qualità del software applicativo e l'hardware e il software di supporto al sistema. Ci si basa poi su una combinazione del valore economico e della qualità del sistema per decidere cosa fare con il sistema ereditato.

Per esempio, supponiamo che un'organizzazione abbia dieci sistemi ereditati. Dobbiamo stabilire la qualità e il valore economico di ciascuno di questi sistemi; poi possiamo creare un grafico che mostra il valore aziendale relativo e la qualità del sistema. Un esempio è illustrato nella Figura 9.9. Come si può notare da questo diagramma, ci sono quattro cluster di sistemi.

1. *Bassa qualità, basso valore economico.* Lasciare in funzione questi sistemi sarà costoso, e il tasso di ritorno economico sarà abbastanza ridotto. Questi sistemi dovrebbero essere scartati.
2. *Bassa qualità, alto valore economico.* Questi sistemi forniscono un importante contributo alle attività aziendali, quindi non possono essere scartati; tuttavia, la loro bassa qualità rende costosa la manutenzione. Questi sistemi dovrebbero essere reingegnerizzati per migliorarne la qualità oppure potrebbero essere sostituiti se fosse disponibile un appropriato sistema off-the-shelf.
3. *Alta qualità, basso valore economico.* Sono sistemi che non contribuiscono molto alle attività aziendali, ma non sono molto costosi da mantenere. Non è conveniente sostituire questi sistemi, quindi si può continuare la normale manutenzione finché non sono richieste modifiche costose e l'hardware è funzionante. Se occorre apportare modifiche costose, allora questi sistemi dovrebbero essere scartati.

4. *Alta qualità, alto valore economico.* Questi sistemi devono essere tenuti in funzione. Tuttavia, data l'alta qualità, non è conveniente investire nella loro trasformazione o sostituzione. La normale manutenzione deve continuare.

Il valore economico di un sistema è una misura del tempo e dell'impegno che il sistema permette di risparmiare rispetto ai processi manuali o all'uso di altri sistemi. Per determinare il valore economico di un sistema si devono prima identificare gli stakeholder del sistema (utenti finali e loro manager) e, poi, porre loro una serie di domande sul sistema per discutere e definire quattro aspetti fondamentali.

1. *Uso del sistema.* Se un sistema è utilizzato solo occasionalmente o da un ristretto numero di persone, questo può significare che il sistema ha un basso valore economico. Un sistema ereditato potrebbe essere stato sviluppato per soddisfare una necessità aziendale che è cambiata o che adesso può essere soddisfatta più efficacemente in altri modi. Occorre prestare attenzione, comunque, sull'uso occasionale, ma importante, di un sistema. Per esempio, un sistema universitario per la registrazione degli studenti viene utilizzato soltanto all'inizio dell'anno accademico; sebbene sia utilizzato poco frequentemente, tuttavia esso è un sistema essenziale con un alto valore economico.
2. *Processi aziendali supportati.* Quando si introduce un sistema, di solito vengono progettati dei processi aziendali per sfruttare le sue funzionalità. Se il sistema non è flessibile, cambiare questi processi potrebbe essere impossibile. Tuttavia, se l'ambiente operativo cambia, i processi aziendali originali possono diventare obsoleti; quindi, un sistema può avere un basso valore economico perché forza l'uso di processi aziendali inefficienti.
3. *Fidatezza del sistema.* La fidatezza del sistema non è solo un problema tecnico, ma anche economico. Se un sistema non è fidato e i problemi influenzano direttamente i clienti dell'azienda e, di conseguenza, il personale viene distolto da altre attività per risolvere questi problemi, allora il sistema ha un basso valore economico.
4. *Output del sistema.* Il problema chiave è l'importanza che gli output del sistema hanno sul successo delle attività aziendali. Se le attività dipendono da questi output, allora il sistema ha un alto valore economico; viceversa, se questi output possono essere generati facilmente in altri modi oppure se il sistema produce output raramente utilizzati, allora il sistema ha un basso valore economico.

Per esempio, supponiamo che una società fornisca un sistema per la prenotazione di viaggi, che viene utilizzato dal personale responsabile dell'organizzazione dei viaggi. Il personale può effettuare gli ordini presso un'agenzia di viaggi abilitata; i biglietti vengono poi consegnati e alla società viene inviata la fattura. Una valutazione del valore economico rivela però che questo sistema viene utilizzato

solo per una piccola percentuale di ordini effettuati, poiché le persone che organizzano i viaggi trovano più economico e conveniente trattare direttamente con i fornitori attraverso i loro siti web. Questo sistema può essere ancora utilizzato, ma non c'è un motivo reale per mantenerlo – lo stesso servizio è reso da sistemi esterni.

D'altra parte, supponiamo che una società abbia sviluppato un sistema che registra tutti gli ordini precedenti dei clienti e genera automaticamente un pomeriggio da inviare loro per fare nuove ordinazioni. Questo genera un gran numero di ordini ripetitivi, e rende i clienti soddisfatti perché sentono che il loro fornitore è consapevole delle loro necessità. Gli output di tale sistema sono molto importanti, e il sistema ha dunque un grande valore economico.

Per valutare un sistema software da una prospettiva tecnica occorre considerare sia il sistema applicativo sia l'ambiente in cui opera il sistema. L'ambiente include l'hardware e tutto il software di supporto associato, come compilatori, debugger e ambienti di sviluppo che sono richiesti per mantenere il sistema. L'ambiente è importante perché molte modifiche del sistema, quali l'aggiornamento dell'hardware o del sistema operativo, sono dovute ai cambiamenti dell'ambiente.

I fattori da considerare nella valutazione dell'ambiente sono elencati nella Figura 9.10. Si noti che queste non sono tutte caratteristiche tecniche dell'ambiente. Occorre anche considerare l'affidabilità dei fornitori dell'hardware e del software di supporto. Se questi fornitori hanno chiuso le loro attività, i loro sistemi non hanno più un supporto, e quindi potrebbe essere necessario sostituire questi sistemi.

Quando si effettua la valutazione dell'ambiente, se possibile, si dovrebbero raccogliere i dati sul sistema e sulle sue modifiche. Esempi di dati che possono essere utili sono i costi di manutenzione dell'hardware del sistema e del software di supporto, il numero di guasti dell'hardware che si verificano in un dato periodo e la frequenza delle riparazioni del software di supporto del sistema.

Per valutare la qualità tecnica di un sistema applicativo, occorre considerare una serie di fattori (Figura 9.11) che sono collegati principalmente alla fidatezza del sistema, alle difficoltà di manutenzione e alla documentazione del sistema. Si possono anche raccogliere dati che possono aiutare a giudicare la qualità del sistema.

1. *Numero di richieste di modifiche del sistema.* Le modifiche del sistema di solito tendono a corromperne la struttura e a rendere più difficili le successive modifiche. Più è alto questo valore, più è bassa la qualità del sistema.
2. *Numero di interfacce utente.* Questo è un fattore importante per i sistemi basati su moduli, dove ogni modulo può essere considerato un'interfaccia utente separata. Più è alto il numero di interfacce, più è probabile che in queste ci siano incoerenze e ridondanze.

Fattore	Domande
Stabilità del fornitore	Il fornitore del sistema è ancora esistente? È finanziariamente stabile? Continuerà a esistere? Se il fornitore originario non è più in affari, c'è qualcun altro che mantiene il suo sistema?
Tasso di malfunzionamenti	L'hardware ha un alto tasso di malfunzionamenti? Il software di supporto va in blocco e obbliga a riavviare il sistema?
Età	Quanto sono vecchi l'hardware e il software? Più sono vecchi l'hardware e il software di supporto, più il sistema sarà obsoleto. Il sistema potrà funzionare ancora correttamente, ma importanti benefici economici e aziendali potrebbero rendere più conveniente passare a un sistema più moderno.
Prestazioni	Le prestazioni del sistema sono adeguate? Le prestazioni hanno un effetto significativo sugli utenti del sistema?
Requisiti di supporto	Quale supporto locale è richiesto dall'hardware e dal software? Se i costi associati a questo supporto sono alti, potrebbe essere conveniente sostituire il sistema.
Costi di manutenzione	Quali sono i costi di manutenzione dell'hardware e delle licenze del software di supporto? Il vecchio hardware potrebbe avere costi di manutenzione più alti dei sistemi moderni. Il software di supporto potrebbe avere alti costi annuali per le licenze.
Interoperabilità	Ci sono problemi a interfacciare il sistema con altri sistemi? Per esempio, i compilatori possono essere utilizzati con le attuali versioni del sistema operativo?

Figura 9.10 Fattori utilizzati per valutare un ambiente operativo.

3. *Volume dei dati utilizzati dal sistema.* Più è alto il volume dei dati (numero dei file, dimensione del database ecc.) elaborati dal sistema, più è alto il numero di incoerenze e di errori nei dati. Quando i dati vengono raccolti per un lungo periodo di tempo, gli errori e le incoerenze sono inevitabili. Pulire i vecchi dati è un processo lungo e molto costoso.

Teoricamente, occorrerebbe una valutazione obiettiva per decidere che cosa fare di un sistema ereditato; in molti casi, però, queste decisioni non sono realmente oggettive, perché si basano su considerazioni di ordine organizzativo o politico. Per esempio, se due aziende si fondono, in genere si mantengono i sistemi della società politicamente più influente, e si scartano i sistemi dell'altra società. Se il management di una società decide di passare a una nuova piattaforma hardware, questa decisione potrebbe richiedere la sostituzione di alcune applicazioni. Se il budget disponibile non è sufficiente per la trasformazione del sistema in un particolare anno, allora la manutenzione del sistema potrebbe continuare, anche se questo si tradurrà in un aumento dei costi a lungo termine.

Fattore	Domande
Comprensibilità	Quanto è difficile comprendere il codice sorgente del sistema attuale? Quanto complesse sono le strutture di controllo utilizzate? Le variabili hanno nomi significativi che riflettono la loro funzione?
Documentazione	Quale documentazione del sistema è disponibile? La documentazione è completa, coerente e aggiornata?
Dati	C'è un esplicito modello di dati per il sistema? Qual è il livello di duplicazione dei dati nei file? I dati utilizzati dal sistema sono aggiornati e coerenti?
Prestazioni	Le prestazioni del sistema sono adeguate? Le prestazioni hanno un effetto significativo sugli utenti del sistema?
Linguaggio di programmazione	Sono disponibili compilatori moderni per il linguaggio di programmazione utilizzato per sviluppare il sistema? Il linguaggio di programmazione è ancora utilizzato per sviluppare nuovi sistemi?
Gestione della configurazione	Tutte le versioni di tutte le parti del sistema sono gestite da un sistema di gestione della configurazione? C'è una descrizione esplicita delle versioni dei componenti che sono utilizzati nel sistema attuale?
Dati di test	Esistono dati di test del sistema? C'è una registrazione dei test di regressione eseguiti quando sono state aggiunte nuove funzionalità al sistema?
Competenze del personale	Sono disponibili persone capaci di mantenere l'applicazione? Ci sono persone che hanno esperienza con il sistema?

Figura 9.11 Fattori utilizzati per valutare le applicazioni.

9.3 Manutenzione del software

La manutenzione del software è il processo generale di modifica di un sistema dopo che è stato consegnato. Il termine viene solitamente applicato al software personalizzato, dove operano gruppi di sviluppo separati prima e dopo la consegna. Le modifiche apportate al software possono essere semplici correzioni di errori nel codice, correzioni più consistenti di errori di progettazione o miglioramenti significativi per correggere errori della specifica o per adattare nuovi requisiti. Le modifiche sono implementate modificando i componenti esistenti del sistema e, se necessario, aggiungendone di nuovi.

Dinamica evolutiva dei programmi

La dinamica evolutiva dei programmi è lo studio dell'evoluzione dei sistemi software, i cui pionieri sono stati Manny Lehman e Les Belady negli anni '70. Lo studio ha prodotto le cosiddette leggi di Lehman, che si applicano a tutti i sistemi software su larga scala. Le più importanti di queste leggi sono:

1. un programma deve cambiare continuamente per rimanere utile;
2. quando un programma in fase di evoluzione viene modificato, la sua struttura si degrada;
3. durante la vita di un programma, la frequenza delle modifiche è approssimativamente costante e indipendente dalle risorse disponibili;
4. le modifiche incrementali in ogni release del sistema sono approssimativamente costanti;
5. occorre aggiungere nuove funzionalità ai sistemi per aumentare il grado di soddisfazione degli utenti.

<http://software-engineering-book.com/web/program-evolution-dynamics/>

Ci sono tre tipi di manutenzione del software.

1. *Eliminare errori e punti vulnerabili del software.* Gli errori di codifica di solito sono relativamente economici da correggere, mentre quelli di progettazione sono più costosi perché possono richiedere la riscrittura di diversi componenti del programma. Gli errori dei requisiti sono i più costosi da riparare, perché potrebbero richiedere una significativa riprogettazione del sistema.
2. *Adattare il software a nuovi ambienti operativi e piattaforme.* Questo tipo di manutenzione è richiesto quando alcuni elementi dell'ambiente del sistema, come l'hardware, il sistema operativo o altro software di supporto, cambiano. I sistemi applicativi devono essere modificati per adattarsi a queste modifiche dell'ambiente operativo.
3. *Aggiungere nuove funzionalità o supportare nuovi requisiti.* Questo tipo di manutenzione è necessaria quando cambiano i requisiti del sistema a causa di modifiche organizzative o aziendali. L'entità delle modifiche richieste dal software spesso è molto più grande di quella richiesta per gli altri tipi di manutenzione.

In pratica, non c'è una netta distinzione tra questi tipi di manutenzione. Quando si adatta il software a un nuovo ambiente, si possono aggiungere nuove funzionalità per sfruttare le caratteristiche del nuovo ambiente. Gli errori del software spesso vengono scoperti perché gli utenti utilizzano il sistema in modo imprevedibile. Modificare il sistema per adattarlo ai loro metodi di lavoro è il modo migliore per correggere questo tipo di errori.

Questi sono i tipi di manutenzione generalmente riconosciuti, anche se a volte sono chiamati con termini differenti. Il termine "manutenzione correttiva" viene universalmente utilizzato per fare riferimento alla correzione degli errori. Il termine "manutenzione adattiva" a volte è utilizzato per indicare le attività per adat-

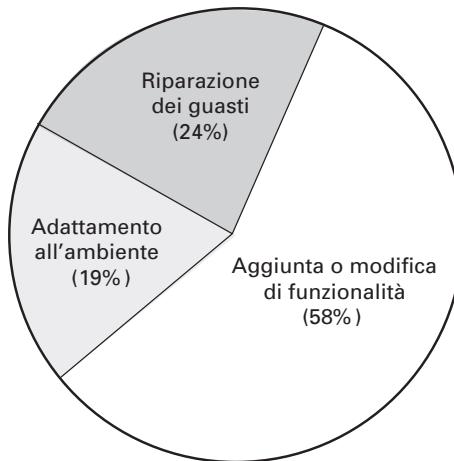


Figura 9.12 Distribuzione dell'attività di manutenzione.

tare un software a un nuovo ambiente; altre volte indica le attività per adattare un software a nuovi requisiti. Il termine “manutenzione perfettiva” a volte significa perfezionare il software implementando nuovi requisiti; altre volte significa mantenere le funzionalità del sistema migliorando la sua struttura e le sue prestazioni. A causa dell’ambiguità di questi termini, eviterò di utilizzarli in questo libro.

La Figura 9.12 mostra una distribuzione approssimata dei costi di manutenzione, basata sui dati della più recente indagine disponibile (Davidsen e Krogstie 2010). Questo studio confronta la distribuzione dei costi di manutenzione con un certo numero di studi precedenti dal 1980 al 2005. Gli autori hanno scoperto che la distribuzione dei costi di manutenzione è cambiata molto poco in 30 anni. Sebbene non siano disponibili dati più recenti, questo suggerisce che tale distribuzione è ancora ampiamente corretta. Riparare i guasti del sistema non è l’attività di manutenzione più costosa. Far evolvere il sistema per far fronte a nuovi ambienti e a requisiti nuovi o modificati è l’attività che richiede il maggior impegno di manutenzione.

L’esperienza ha dimostrato che di solito è più costoso aggiungere una nuova funzionalità a un sistema durante la manutenzione, mentre è più conveniente implementare la stessa funzionalità durante la fase iniziale dello sviluppo, per le seguenti ragioni.

1. *Un nuovo team deve capire il programma da manutenere.* Dopo che un sistema è stato consegnato, è normale sciogliere il team di sviluppo per permettere alle persone di lavorare su nuovi progetti. Il nuovo team o i singoli responsabili della manutenzione del sistema potrebbero non capire il sistema o tutte le decisioni prese durante la progettazione del sistema; dovranno impiegare molto tempo per capire il sistema esistente prima di poter implementare qualsiasi modifica.

Documentazione

La documentazione del sistema può agevolare il processo di manutenzione fornendo ai responsabili della manutenzione le informazioni sulla struttura e l'organizzazione del sistema e sulle funzionalità che il sistema offre ai suoi utenti. Sebbene i sostenitori dei metodi agili suggeriscano che il codice debba essere la principale documentazione di un sistema, tuttavia i modelli di progettazione di livello più elevato e le informazioni sulle dipendenze e i vincoli del sistema possono agevolare significativamente la comprensione e la modifica del codice.

<http://software-engineering-book.com/web/documentation/>

2. *Separare la manutenzione dallo sviluppo significa che il team di sviluppo non è incentivato a scrivere un software facile da manutenere.* Il contratto per la manutenzione del sistema di solito è separato dal contratto di sviluppo, perché può essere stipulato con una società diversa da quella che ha sviluppato inizialmente il sistema. In questi casi, un team di sviluppo non ha alcun beneficio nell'impegnarsi a scrivere un software sia facile da manutenere. Se il team di sviluppo può prendere una scorciatoia per limitare gli sforzi durante lo sviluppo, lo fa perché è conveniente farlo, anche se questo significa rendere più difficile la manutenzione futura del software.
3. *La manutenzione dei programmi è un compito impopolare.* La manutenzione ha una scarsa considerazione tra gli ingegneri del software; è vista come un processo meno qualificato rispetto allo sviluppo del sistema e spesso viene assegnata al personale meno esperto. I vecchi sistemi di solito sono scritti con linguaggi di programmazione obsoleti. Il personale responsabile della manutenzione potrebbe non avere alcuna conoscenza di questi linguaggi e, quindi, è costretto a impararli per manutenere il sistema.
4. *Quando un programma invecchia, la sua struttura si deteriora e diventa più difficile modificarlo.* Quando si modifica un programma, la sua struttura tende a deteriorarsi; di conseguenza, il programma diventa più difficile da capire e modificare. Alcuni sistemi sono stati sviluppati senza le moderne tecniche di ingegneria del software; non sono mai stati ben strutturati e probabilmente sono stati ottimizzati per essere efficienti, non per essere facili da capire. La documentazione del sistema può essere andata persa o è diventata incoerente. I vecchi sistemi potrebbero non essere stati soggetti alla rigorosa gestione della configurazione, quindi gli sviluppatori devono impiegare molto tempo per trovare le versioni corrette dei componenti da modificare.

I primi tre problemi derivano dal fatto che molte organizzazioni considerano ancora lo sviluppo e la manutenzione come attività separate. La manutenzione è vista come attività di seconda classe, e non c'è incentivo a spendere soldi durante lo sviluppo per ridurre i costi delle future modifiche del sistema. L'unica soluzione a lungo termine di questo problema è pensare che i sistemi si evolvono

durante tutta la loro vita attraverso un processo di sviluppo continuo. La manutenzione dovrebbe essere elevata allo stesso status dello sviluppo di un nuovo software.

Il quarto problema – la struttura del sistema che si deteriora – è in qualche modo più semplice da risolvere. Per migliorare la struttura e la comprensibilità del sistema, si potrebbero applicare le tecniche di reingegnerizzazione del software (descritte più avanti in questo capitolo). Le trasformazioni architetturali possono adattare il sistema al nuovo hardware. La rifattorizzazione può migliorare la qualità del codice del sistema e renderlo più facile da modificare.

In teoria, è quasi sempre economicamente vantaggioso impegnarsi nella progettazione e nell'implementazione del sistema per ridurre i costi delle future modifiche. Aggiungere nuove funzionalità dopo la consegna di un sistema è un'operazione costosa, perché occorre tempo per capire il sistema e analizzare l'impatto delle modifiche proposte. Il lavoro svolto durante lo sviluppo per strutturare il software e renderlo più comprensibile e più facile da modificare ridurrà i costi di evoluzione. Le buone tecniche di ingegneria del software, quali le specifiche accurate, lo sviluppo con test iniziali, lo sviluppo orientato agli oggetti e la gestione della configurazione, contribuiscono a ridurre i costi di manutenzione.

Questi sani principi di riduzione dei costi attraverso lo sviluppo di sistemi più facili da mantenere, purtroppo, sono impossibili da suffragare con i dati reali. Raccogliere i dati è costoso, e il valore di tali dati è difficile da stimare; quindi, la maggior parte delle società non credono che sia conveniente accumulare e analizzare i dati di ingegneria del software.

In realtà, molte aziende sono riluttanti a investire nello sviluppo del software per ridurre i costi di manutenzione di lungo termine. Due sono le ragioni che giustificano questa riluttanza.

1. Le società definiscono piani di spesa trimestrali o annuali, e i manager sono incentivati a ridurre le spese di breve termine. Investire nella manutenibilità dei sistemi software implica un aumento dei costi di breve termine, che sono subito quantificabili. I risparmi di lungo termine, invece, non sono quantificabili immediatamente, quindi le società sono riluttanti a investire soldi in qualcosa che ha un ritorno futuro indefinito.
2. Gli sviluppatori di solito non sono responsabili della manutenzione del sistema che hanno sviluppato; di conseguenza, non sono incentivati a svolgere un lavoro aggiuntivo che potrebbe ridurre i costi futuri di manutenzione, in quanto non ne trarrebbero alcun vantaggio.

L'unico modo per risolvere questo problema consiste nell'integrare lo sviluppo e la manutenzione, in modo che il team di sviluppo originale resti responsabile per tutta la vita del software. Questo è possibile per i prodotti software e per società, come Amazon, che sviluppano e mantengono il software di loro proprietà (O'Hanlon 2006). Questo è difficile che accada per il software personalizzato sviluppato da una società per un determinato cliente.

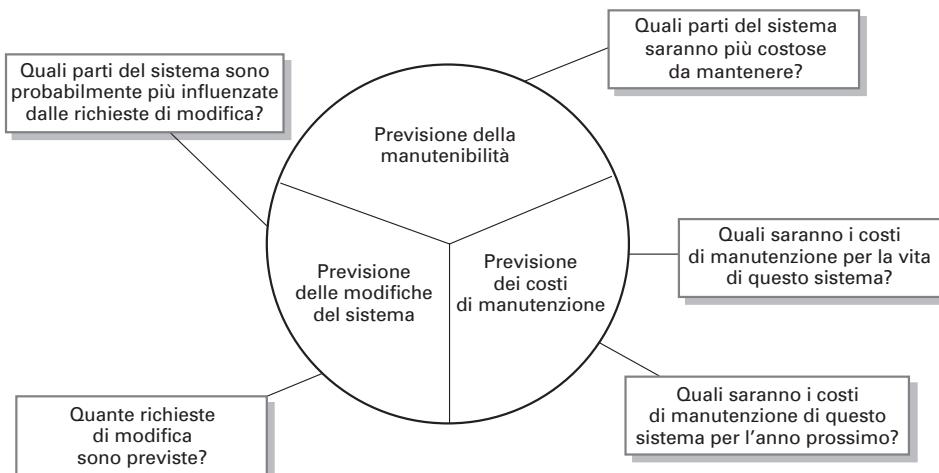


Figura 9.13 Previsione della manutenzione.

9.3.1 Previsione della manutenzione

La previsione della manutenzione si occupa di tentare di stabilire le probabili modifiche che potrebbero essere richieste da un sistema software e identificare le parti del sistema che probabilmente saranno le più costose da modificare. Se si capisce questo, è possibile progettare le componenti software che più probabilmente dovranno essere modificate per renderle più adattabili. È anche opportuno investire del tempo per migliorare tali componenti in modo da ridurre i costi di manutenzione durante la loro vita. Prevedendo le modifiche, inoltre, è possibile stabilire i costi di manutenzione complessivi di un sistema in un dato periodo di tempo e, quindi, definire il budget per la manutenzione del software. La Figura 9.13 mostra le possibili previsioni e le domande da porre per fare queste previsioni.

Per prevedere il numero di modifiche richieste per un sistema occorre conoscere le relazioni tra il sistema e il suo ambiente esterno. Alcuni sistemi hanno una relazione molto complessa con il loro ambiente esterno, e le modifiche dell'ambiente inevitabilmente comportano modifiche del sistema. Per valutare le relazioni tra un sistema e il suo ambiente, si dovrebbero esaminare i seguenti aspetti.

1. *Il numero e la complessità delle interfacce del sistema.* Quanto più è grande il numero di interfacce e quanto più queste sono complesse, tanto più è probabile che sarà necessario modificarle quando saranno proposti nuovi requisiti.
2. *Il numero dei requisiti di sistema intrinsecamente volatili.* Come detto nel Capitolo 4, i requisiti che riflettono le politiche e le procedure organizzative sono probabilmente più volatili dei requisiti che si basano su stabili caratteristiche del dominio.

3. *I processi aziendali in cui il sistema viene utilizzato.* L’evoluzione dei processi aziendali genera richieste di modifica del sistema. Se un sistema viene integrato con un numero sempre crescente di processi aziendali, le richieste di modifiche sono destinate ad aumentare.

In alcuni studi sulla manutenzione del software, i ricercatori hanno analizzato le relazioni tra complessità e manutenibilità dei programmi (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). Da questi studi è emerso che quanto più è complesso un sistema o un componente, tanto più è costosa la loro manutenzione. Le misure della complessità sono particolarmente utili nell’identificare i componenti dei programmi che molto probabilmente saranno più costosi da mantenere. Quindi, per ridurre i costi di manutenzione, si dovrebbe tentare di sostituire i componenti complessi del sistema con elementi più semplici.

Dopo che un sistema è stato messo in servizio, si possono utilizzare i dati dei processi per agevolare le previsioni di manutenibilità. Ecco alcuni esempi di misurazioni che possono essere utilizzate per valutare la manutenibilità.

1. *Il numero di richieste di manutenzione correttiva.* Un aumento del numero di informazioni sui fallimenti e i bug può indicare che si stanno introducendo nel programma più errori di quelli che sono stati corretti durante il processo di manutenzione. Questo potrebbe indicare un peggioramento della manutenibilità.
2. *Tempo medio richiesto per l’analisi dell’impatto.* Questo tempo è correlato al numero di componenti del programma che sono influenzati dalla richiesta di modifica. Se il tempo richiesto per l’analisi dell’impatto aumenta, significa che sempre più componenti vengono influenzati, con conseguente peggioramento della manutenibilità.
3. *Tempo medio richiesto per implementare una richiesta di modifica.* È diverso dal tempo richiesto per l’analisi dell’impatto, anche se è correlato con esso. È la quantità di tempo che occorre per modificare il sistema e la sua documentazione, dopo aver stabilito quali componenti sono influenzati dalle modifiche. Un aumento di questo tempo può indicare un deterioramento della manutenibilità del sistema.
4. *Numeri di richieste di modifiche significative.* Un aumento di questo numero nel tempo può indicare un peggioramento della manutenibilità.

Le informazioni sulle richieste di modifica e sulle previsioni di manutenibilità di un sistema servono a prevedere i costi di manutenzione. La maggior parte dei manager combina queste informazioni con l’intuito e l’esperienza per stimare tali costi. Il modello di stima dei costi COCOMO 2, descritto nel Capitolo 20, suggerisce che una stima degli sforzi di manutenzione del sistema può basarsi sullo sforzo per comprendere il codice esistente e per sviluppare il nuovo codice.

9.3.2 Reingegnerizzazione del software

La manutenzione del software richiede la comprensione del programma da modificare e l'implementazione delle modifiche richieste. Molti sistemi, tuttavia, specialmente i sistemi ereditati più vecchi, sono difficili da capire e modificare. I programmi potrebbero essere stati ottimizzati, originariamente, per migliorare le prestazioni del codice o l'utilizzo dello spazio, a scapito della comprensibilità; oppure, nel corso del tempo, la struttura iniziale del programma potrebbe essersi deteriorata in seguito a una serie di modifiche.

Per rendere più agevole la manutenzione dei sistemi software ereditati, si può decidere di reingegnerizzare questi sistemi per migliorarne la struttura e la comprensibilità. La reingegnerizzazione del software potrebbe richiedere una nuova documentazione del sistema, la rifattorizzazione dell'architettura del sistema, la traduzione dei programmi in un linguaggio di programmazione moderno, la modifica e l'aggiornamento della struttura e dei dati del sistema. Le funzionalità del software non vengono modificate e, di solito, si dovrebbe evitare di apportare modifiche significative all'architettura del sistema.

La reingegnerizzazione ha due vantaggi importanti rispetto alla sostituzione del software.

1. *Rischio ridotto.* Sviluppare di nuovo un software critico per un'azienda è un rischio molto alto. Si potrebbero commettere errori nelle specifiche del sistema o potrebbero sorgere problemi durante lo sviluppo. I ritardi nell'introduzione del nuovo software potrebbero causare perdite e costi extra nell'attività aziendale.
2. *Costi ridotti.* I costi di reingegnerizzazione potrebbero essere significativamente inferiori ai costi di sviluppo di un nuovo software. Ulrich (Ulrich 1990) cita l'esempio di un sistema commerciale per il quale i costi di reimplementazione erano stimati in 50 milioni di dollari. Il sistema fu successivamente reingegnerizzato con una spesa di 12 milioni di dollari. Si potrebbe obiettare che, con le moderne tecnologie software, i costi relativi di reimplementazione potrebbero essere minori di quelli riportati da Ulrich, ma resterebbero comunque più elevati dei costi di reingegnerizzazione.

La Figura 9.14 illustra un modello generale del processo di reingegnerizzazione. L'input del processo è un programma ereditato; l'output è una versione ristrutturata e perfezionata dello stesso programma. Le attività di questo processo di reingegnerizzazione sono qui elencate.

1. *Traduzione del codice sorgente.* Utilizzando uno strumento di traduzione, si converte il codice del programma dal vecchio linguaggio di programmazione in una versione più moderna dello stesso linguaggio o in un linguaggio diverso.

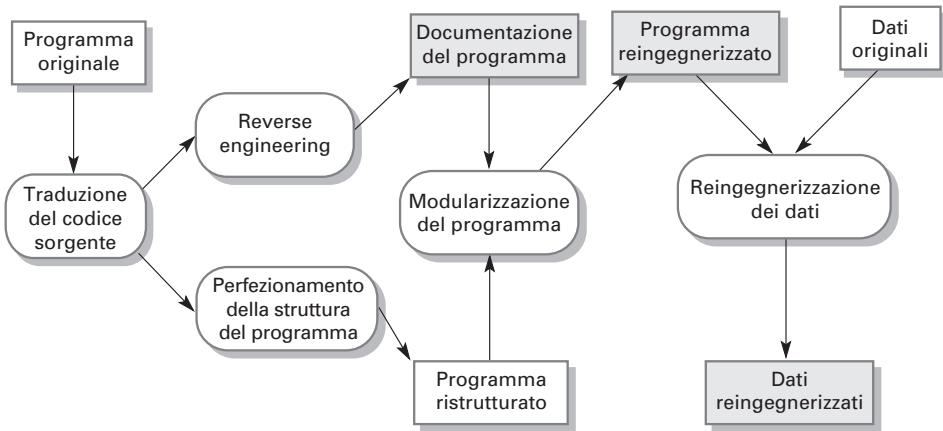


Figura 9.14 Il processo di reingegnerizzazione.

2. *Reverse engineering.* Il programma viene analizzato e vengono estratte le informazioni utili a documentare la sua organizzazione e le sue funzionalità. Di solito questo processo è completamente automatizzato.
3. *Perfezionamento della struttura del programma.* La struttura di controllo viene analizzata e modificata per renderla più facile da leggere e capire. Questo processo può essere parzialmente automatizzato, ma di solito è richiesto qualche intervento manuale.
4. *Modularizzazione del programma.* Si raggruppano le parti correlate del programma e, dove appropriato, vengono rimosse le parti ridondanti. In alcuni casi, questo stadio può richiedere la rifattorizzazione dell'architettura (per esempio, un sistema che usa vari data store può essere rifattorizzato per utilizzare un solo repository). Questo è un processo manuale.
5. *Reingegnerizzazione dei dati.* Si modificano i dati elaborati dal programma perché riflettano le modifiche apportate al programma. Questo potrebbe implicare la ridefinizione degli schemi dei database e la conversione dei database esistenti nella nuova struttura. Di solito è anche richiesta una pulizia dei dati (trovare e correggere gli errori, eliminare i record doppioni e così via). Si tratta di un processo lungo e molto costoso.

Non tutte le fasi illustrate nella Figura 9.14 sono sempre richieste dal processo di reingegnerizzazione. La traduzione del codice sorgente potrebbe non essere necessaria se viene mantenuto il linguaggio di programmazione utilizzato per sviluppare l'applicazione. Se la reingegnerizzazione può essere effettuata in modo automatico, potrebbe non essere necessario recuperare la documentazione attraverso il reverse engineering. La reingegnerizzazione dei dati è richiesta soltanto se le strutture dei dati del programma vengono modificate durante la reingegnerizzazione del programma.

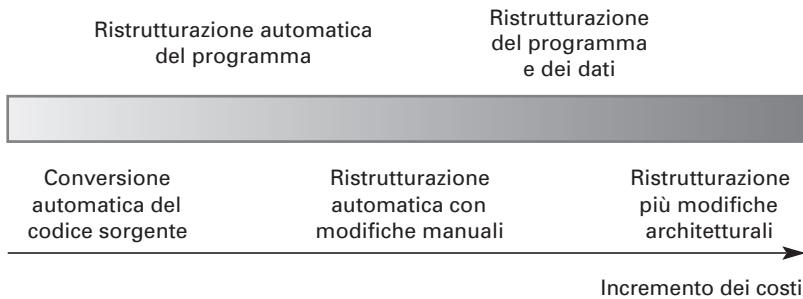


Figura 9.15 Opzioni di reingegnerizzazione.

Per far sì che il sistema reingegnerizzato interagisca con il nuovo software, occorre sviluppare degli appositi servizi adattatori, che nascondono le interfacce originali del sistema software, presentando nuove interfacce, meglio strutturate, che possono essere utilizzate da altri componenti. Questo processo di mascheramento dei sistemi ereditati è una tecnica importante nello sviluppo di servizi riutilizzabili su larga scala.

I costi della reingegnerizzazione dipendono ovviamente dall'entità del lavoro che viene eseguito. C'è una gamma di possibili approcci alla reingegnerizzazione, come illustra la Figura 9.15. I costi aumentano da sinistra a destra, quindi la traduzione del codice sorgente è l'opzione più economica, mentre la reingegnerizzazione, come parte di una migrazione architetturale, è la più costosa.

Il problema principale della reingegnerizzazione del software è che ci sono dei limiti pratici al grado di perfezionamento che può essere raggiunto con la reingegnerizzazione. Non è possibile, per esempio, convertire un sistema scritto applicando un approccio funzionale a un sistema orientato agli oggetti. Le principali modifiche architettoniche o la riorganizzazione radicale della gestione dati del sistema non possono essere eseguite in modo automatico, per questo sono molto costose. Sebbene la reingegnerizzazione migliori la manutenibilità, il sistema reingegnerizzato probabilmente non sarà manutenibile come un nuovo sistema sviluppato utilizzando metodi moderni di ingegneria del software.

9.3.3 Rifattorizzazione

La rifattorizzazione è il processo di perfezionamento di un programma volto a rallentare il suo deterioramento. Questo significa che occorre modificare un programma per migliorarne la struttura, ridurne la complessità o renderlo più semplice da leggere. Qualcuno crede che la rifattorizzazione sia limitata allo sviluppo orientato agli oggetti, tuttavia i suoi principi possono essere applicati a qualsiasi metodo di sviluppo. Quando si rifattorizza un programma, non si dovrebbero aggiungere nuove funzionalità, ma piuttosto concentrarsi sul miglioramento del programma. La rifattorizzazione può essere considerata come una “manutenzione preventiva”, che riduce i problemi delle future modifiche.

La rifattorizzazione è parte intrinseca dei metodi agili, in quanto questi metodi si basano sulle modifiche. La qualità dei programmi è esposta a un rapido deterioramento, quindi gli sviluppatori agili rifattorizzano frequentemente i loro programmi per evitare questo deterioramento. L'importanza che hanno i test di regressione nei metodi agili riduce il rischio di introdurre nuovi errori durante il processo di rifattorizzazione. Qualsiasi errore introdotto dovrebbe essere facilmente identificato, in quanto i test precedentemente superati dovrebbero successivamente fallire. La rifattorizzazione non dipende da altre “attività agili”.

Sebbene la reingegnerizzazione e la rifattorizzazione siano entrambe ideate per semplificare la comprensione e la modifica del software, tuttavia esse non sono la stessa cosa. La reingegnerizzazione si effettua dopo che un sistema è stato manutenuto per qualche periodo di tempo, con costi di manutenzione in crescita. Si usano strumenti automatizzati per elaborare e reingegnerizzare un sistema ereditato in modo da creare un nuovo sistema che è più facile da manutenere. La rifattorizzazione è un processo continuo di miglioramento che si svolge durante lo sviluppo e l’evoluzione di un sistema. Il suo obiettivo è evitare il deterioramento della struttura e del codice che causa un aumento dei costi e dei problemi di manutenzione del sistema.

Fowler e altri (Fowler et al. 1999) parlano di casi stereotipici (che Fowler chiama “cattivi odori”), nei quali il codice di un programma può essere migliorato. Esempi di questi casi che possono essere migliorati tramite la rifattorizzazione sono elencati qui di seguito.

1. *Codice duplicato*. Lo stesso codice, o un codice molto simile, può essere ripetuto in vari punti di un programma. Il codice duplicato può essere rimosso e implementato come un singolo metodo (o funzione) che viene chiamato quando serve.
2. *Metodi lunghi*. Se un metodo è troppo lungo, dovrebbe essere riprogettato come una serie di metodi più brevi.
3. *Istruzioni switch (case)*. Queste istruzioni spesso implicano duplicazioni, quando l’istruzione *switch* dipende dal tipo di un valore. Questa istruzione può essere ripetuta più volte all’interno di un programma. Nei linguaggi orientati agli oggetti, spesso si usa il polimorfismo per ottenere la stessa cosa.
4. *Aggregazione dei dati*. Si verifica quando lo stesso gruppo di elementi di dati (campi nelle classi, parametri nei metodi) si ripetono in più posti all’interno di un programma. Queste aggregazioni spesso possono essere sostituite con un oggetto che incapsula tutti i dati.
5. *Generalità speculativa*. Si verifica quando gli sviluppatori includono la generalità in un programma nel caso sia richiesta in futuro. Tale generalità spesso può essere semplicemente rimossa.

Fowler, nel libro e nel suo sito web, suggerisce anche alcune trasformazioni primitive di rifattorizzazione che possono essere utilizzate singolarmente o congiuntamente per risolvere i casi dei cattivi odori. Esempi di queste trasformazioni includono il metodo *Extract*, dove si eliminano le duplicazioni per creare un nuovo metodo, l'espressione condizionale *Consolidate*, dove si sostituisce una sequenza di test con un singolo test, e il metodo *Pull up*, dove si sostituiscono metodi simili nelle sottoclassi con un singolo metodo in una superclasse. Gli ambienti di sviluppo interattivo, come Eclipse, di solito includono un supporto alla rifattorizzazione nei loro editor. Ciò rende più semplice l'identificazione delle parti dipendenti di un programma che devono essere modificate per implementare la rifattorizzazione.

La rifattorizzazione, eseguita durante lo sviluppo di un programma, è una tecnica efficace per ridurre i costi di manutenzione a lungo termine del programma. Tuttavia, se vi viene affidato un programma da manutenere la cui struttura si è notevolmente deteriorata, allora potrebbe essere impossibile rifattorizzare soltanto il codice; potrebbe essere necessario rifattorizzare anche il progetto, e questo sarebbe un problema molto più difficile e costoso. La rifattorizzazione del progetto richiede l'identificazione degli schemi di progettazione (descritti nel Capitolo 7) e la sostituzione del codice esistente con il codice che implementa questi schemi (Kerievsky 2004).

Punti chiave

- Lo sviluppo e l'evoluzione del software possono essere immaginati come un processo iterativo e integrato che può essere rappresentato mediante un modello a spirale.
- Per i sistemi personalizzati, i costi di manutenzione del software di solito superano i costi di sviluppo.
- Il processo dell'evoluzione del software è guidato dalle richieste di modifiche e include l'analisi dell'impatto delle modifiche, la pianificazione delle release e l'implementazione delle modifiche.
- I sistemi ereditati sono vecchi sistemi software, sviluppati mediante tecnologie software e hardware obsolete, che sono ancora utili per le aziende.
- Spesso è più economico e meno rischioso mantenere un sistema ereditato che sviluppare un nuovo sistema utilizzando le moderne tecnologie.
- Il valore economico di un sistema ereditato e la qualità del software applicativo e del suo ambiente devono essere valutati per determinare se un sistema deve essere sostituito, trasformato o manutenuto.
- Ci sono tre tipi di manutenzione del software: correzione dei bug, modifica del software per operare in un nuovo ambiente e implementazione di requisiti nuovi o modificati.

- La reingegnerizzazione riguarda la ristrutturazione del software e la creazione di nuova documentazione per agevolare la comprensione e la modifica del software.
- La rifattorizzazione – apportare piccole modifiche a un programma per preservarne le funzionalità – può essere immaginata come una manutenzione preventiva.

Esercizi

- * 9.1 Spiegate perché un sistema software che è utilizzato in un ambiente reale deve cambiare o diventa progressivamente meno utile.
 - 9.2 Dalla Figura 9.4 si può notare che l'analisi dell'impatto delle modifiche è un importante sottoprocesso nell'evoluzione del software. Utilizzando un diagramma, suggerite quali attività potrebbero essere coinvolte nell'analisi dell'impatto delle modifiche.
 - 9.3 Spiegate perché i sistemi ereditati dovrebbero essere considerati sistemi sociotecnici, anziché semplici sistemi software che sono stati sviluppati tramite vecchie tecnologie.
 - * 9.4 In quali circostanze un'organizzazione potrebbe decidere di scartare un sistema, anche se la valutazione del sistema indica che la sua qualità e il suo valore economico sono elevati?
 - * 9.5 Quali sono le opzioni strategiche per l'evoluzione dei sistemi ereditati? Quando è opportuno sostituire tutto il sistema o una sua parte, anziché continuare la manutenzione del software?
 - 9.6 Spiegate perché i problemi con il software di supporto potrebbero obbligare un'organizzazione a sostituire i suoi sistemi ereditati.
 - * 9.7 In qualità di manager della progettazione del software di una società, che è specializzata nello sviluppo di software per l'industria petrolifera offshore, vi è stato assegnato il compito di scoprire i fattori che influiscono sulla manutenibilità dei sistemi sviluppati dalla vostra società. Suggerite come si potrebbe impostare un programma per analizzare il processo di manutenzione e determinate le metriche appropriate della manutenibilità per la società.
 - * 9.8 Descrivete brevemente i tre tipi principali di manutenzione del software. Perché a volte è difficile distinguerli?
 - 9.9 Spiegate le differenze tra reingegnerizzazione e rifattorizzazione del software.
 - 9.10 Gli ingegneri del software hanno la responsabilità professionale di sviluppare un codice che possa essere facilmente manutenuto, anche se ciò non è stato esplicitamente richiesto dai loro datori di lavoro?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Working Effectively with Legacy Code. Consigli pratici per superare problemi e difficoltà con i sistemi ereditati. (M. Feathers, 2004, John Wiley & Sons).

“The Economics of Software Maintenance in the 21st Century.” Dopo un’introduzione generale alla manutenzione, questo articolo tratta in modo esauriente i costi di manutenzione. Jones analizza i fattori che influiscono sui costi di manutenzione e stima che il 75% circa della forza lavoro viene impiegata nelle attività di manutenzione del software. (C. Jones 2006) <http://www.compaid.com/cainternet/ezine/capersjones-maintenance.pdf>

“You Can’t Be Agile in Maintenance?” Nonostante il titolo, questo blog sostiene che le tecniche agili sono appropriate alla manutenzione e descrive le tecniche che possono essere efficaci secondo la programmazione estrema. (J. Bird 2011) <http://swreflections.blogspot.co.uk/2011/10/you-can-t-be-agile-in-maintenance.html>

“Software Reengineering and Testing Considerations.” È un’eccellente libro bianco che riassume i problemi di manutenzione di un’importante società di software indiana (Y. Kumar e Dipti 2012) http://www.qaglobalservices.com/minisites/stc2012/Paper_%20Best_Practice/11_Software%20Re-Engineering%20And%20Testing%20Considerations.pdf

Fidatezza e protezione

Poiché i sistemi software oggi fanno parte di tutti gli aspetti della nostra vita, la sfida più significativa che deve affrontare l'ingegneria del software è garantire che questi sistemi siano affidabili. Affinché un sistema sia affidabile, dobbiamo credere che esso sia disponibile quando ne abbiamo bisogno e che si comporti nel modo previsto. Il sistema deve essere protetto in modo che i nostri computer e dati non siano a rischio e che, in caso di guasto o di attacco cibernetico, esso possa riprendersi rapidamente. Ecco perché questa parte del libro tratta i temi importanti della fidatezza e della protezione dei sistemi software.

Il Capitolo 10 presenta le caratteristiche fondamentali della fidatezza e della protezione dei sistemi software, tra i quali l'affidabilità, la disponibilità, la sicurezza e la resilienza. Spiegherò perché realizzare un sistema fidato e protetto non è un problema semplicemente tecnico. Introdurò i concetti di ridondanza e diversità come i meccanismi di base utilizzati per creare sistemi fidati e protetti. I singoli attributi di fidatezza sono trattati più dettagliatamente nei capitoli successivi.

Il Capitolo 11 è dedicato all'affidabilità e alla disponibilità dei sistemi software; spiega come questi attributi possono essere specificati come probabilità di guasti o inattività dei sistemi. Sono descritti alcuni schemi architetturali fault-tolerant e alcune tecniche di sviluppo che possono essere utilizzate per ridurre il numero di guasti in un sistema. Il paragrafo finale spiega come l'affidabilità di un sistema possa essere testata e misurata.

Molti sistemi sono a sicurezza critica, nel senso che un guasto può mettere a repentaglio la sicurezza delle persone. Il Capitolo 12 tratta le tecniche di sicurezza che possono essere adottate per sviluppare questi sistemi a sicurezza critica. Spiega perché la sicurezza è un concetto più ampio dell'affidabilità e descrive i metodi per derivare i requisiti di sicurezza dei sistemi. Spiega inoltre perché sono importanti i processi definiti e documentati per l'ingegneria dei sistemi a sicurezza critica e illustra i casi di sicurezza del software – documenti strutturati che sono utilizzati per giustificare perché un sistema è sicuro.

Le minacce alla protezione dei nostri sistemi sono uno dei principali problemi che devono affrontare le società di oggi; ho dedicato due capitoli a questo argomento. Il Capitolo 13 tratta l'ingegneria della protezione delle applicazioni – i metodi usati per realizzare sistemi software protetti. Descrive le

relazioni fra protezione e altri attributi di fidatezza; tratta inoltre l'ingegneria dei requisiti di protezione, la progettazione di sistemi protetti e i test della protezione.

Il Capitolo 14 è un nuovo capitolo che è dedicato al tema più vasto della resilienza. Un sistema resiliente può continuare a rilasciare i suoi servizi essenziali, anche quando le singole parti del sistema sono guaste o subiscono un attacco cibernetico. Il capitolo illustra i concetti fondamentali della protezione informatica e le modalità in cui la resilienza viene ottenuta utilizzando la ridondanza e la diversità, estendendo le autorizzazioni delle persone e adottando opportuni meccanismi tecnici. Infine, sono descritti i sistemi e i problemi di progettazione del software che possono contribuire a migliorare la resilienza di un sistema.

Sistemi fidati

Questo capitolo ha due obiettivi: presentare il concetto di fidatezza del software e spiegare che cosa significa sviluppare sistemi software fidati. Dopo aver letto questo capitolo:

- capirete perché la fidatezza e la protezione sono attributi importanti per tutti i sistemi software;
- conoscerete i cinque importanti aspetti della fidatezza, ovvero la disponibilità, l'affidabilità, la sicurezza, la protezione e la resilienza;
- apprenderete il concetto di sistemi sociotecnici e capirete perché questi sistemi devono essere considerati nel loro complesso, e non semplicemente come sistemi software;
- capirete perché la ridondanza e la diversità sono i concetti fondamentali utilizzati nello sviluppo di sistemi e processi fidati;
- conoscerete le potenzialità dei metodi formali applicati all'ingegneria dei sistemi fidati.

10.1 Proprietà della fidatezza

10.2 Sistemi sociotecnici

10.3 Ridondanza e diversità

10.4 Processi fidati

10.5 Metodi formali e fidatezza

Da quando i computer sono diventati strumenti indispensabili nei nostri affari e nella nostra vita quotidiana, i problemi che derivano dai guasti di sistema software diventano sempre più importanti. Un errore del software in un server di una società di commercio elettronico potrebbe provocare perdite significative in termini di ricavi e di clienti per la società. Un errore del software nel sistema di controllo di un’automobile potrebbe richiedere il richiamo dal mercato di tutte le automobili che adottano tale sistema e, nel caso peggiore, potrebbe causare gravi incidenti. L’infezione di malware dei PC di un’azienda richiede costose operazioni di pulizia per eliminare il problema e potrebbe causare la perdita o il danneggiamento di informazioni sensibili.

Poiché i sistemi che fanno uso intensivo del software sono molto importanti per le aziende pubbliche e private e per i singoli utenti, è necessario che questi sistemi siano affidabili. Il software dovrebbe essere disponibile quando è richiesto e dovrebbe funzionare correttamente senza effetti collaterali indesiderati, come la divulgazione non autorizzata di informazioni. In poche parole, dovremmo essere in grado di fidarci dei nostri sistemi software.

Il termine *dependability* (*fidatezza*) fu proposto da Jean-Claude Laprie nel 1995 per fare riferimento alle caratteristiche di disponibilità, affidabilità, sicurezza e protezione dei sistemi software. Le sue idee furono riviste negli anni successivi e descritte in un documento pubblicato nel 2004 (Avizienis et al. 2004). Come dirò nel Paragrafo 10.1, queste proprietà sono intrinsecamente collegate, quindi ha senso utilizzare un solo termine per indicarle tutte.

La fidatezza di solito è la funzionalità più importante di un sistema software per varie ragioni.

1. *I guasti di un sistema riguardano un gran numero di persone.* Molti sistemi includono una funzionalità che è raramente utilizzata. Se questa funzionalità viene esclusa dal sistema, soltanto un piccolo numero di utenti ne risentirebbe. Gli errori del sistema che influiscono sulla disponibilità di un sistema influiscono potenzialmente su tutti gli utenti del sistema. L’indisponibilità di un sistema potrebbe significare l’impossibilità di svolgere l’attività normale.
2. *Gli utenti spesso rifiutano i sistemi non affidabili, non sicuri e non protetti.* Se gli utenti si accorgono che un sistema è inaffidabile o non è protetto, si rifiutano di utilizzarlo; possono anche rifiutarsi di acquistare o usare altri prodotti della stessa società che ha prodotto un sistema inaffidabile, perché non vogliono ripetere la stessa brutta esperienza.
3. *I costi per un guasto di un sistema software potrebbero essere enormi.* Per alcune applicazioni, come il sistema di controllo di un reattore nucleare o di un’unità di navigazione aerea, il costo di un guasto è di gran lunga maggiore del costo dell’intero sistema di controllo. I guasti nei sistemi che controllano infrastrutture critiche, come la rete di alimentazione elettrica, hanno conseguenze economiche vastissime.

Sistemi critici

Alcune classi di sistemi software sono “sistemi critici”, in quanto un guasto del sistema potrebbe causare danni alle persone o all’ambiente, o ingenti perdite economiche. Esempi di sistemi critici sono i sistemi integrati nelle apparecchiature medicali, come una pompa di insulina (sicurezza critica), nelle unità di navigazione aerea (missione critica) e nei dispositivi di trasferimento online del denaro (commercio critico).

I sistemi critici sono molto costosi da sviluppare. Non soltanto devono essere sviluppati in modo che i guasti siano molto rari, ma devono anche includere i meccanismi di ripristino da utilizzare quando si verifica un guasto.

<http://software-engineering-book.com/web/critical-systems/>

4. *I sistemi inaffidabili possono causare perdite di informazioni.* I dati sono molto costosi da raccogliere e conservare; potrebbero valere più del sistema software che li elabora. I costi per recuperare i dati persi o danneggiati di solito sono molto alti.

Un sistema, tuttavia, può essere utile senza essere molto affidabile. Non credo che il word processor che ho utilizzato per scrivere questo libro sia un sistema molto affidabile; a volte si è bloccato e ho dovuto riavviarlo. Ciononostante, poiché è molto utile, sono disposto a tollerare qualche malfunzionamento occasionale. Data la mia mancanza di fiducia nel sistema, salvo frequentemente il mio lavoro e tengo più copie di backup dei file. In altre parole, compenso la mia mancanza di fiducia nel sistema con azioni che limitano i danni che potrebbe causare un suo malfunzionamento.

Creare un software fidato fa parte del processo più generale dell’ingegneria dei sistemi fidati. Come detto nel Paragrafo 10.2, il software è sempre parte di un sistema più ampio. Viene eseguito in un ambiente operativo che include l’hardware nel quale il software è eseguito, gli utenti del software e i processi aziendali e organizzativi dove il software è utilizzato. Quando si progetta un sistema fidato, occorre quindi considerare i seguenti punti.

1. *Guasti dell’hardware.* L’hardware di un sistema può fallire a causa di errori di progettazione; i suoi componenti possono fallire a causa di difetti di fabbricazione o di fattori ambientali, come umidità e temperature elevate, o perché hanno raggiunto la fine del loro ciclo di vita.
2. *Guasti del software.* Un sistema software può fallire a causa di errori di specifica, progettazione o implementazione.
3. *Errori di utilizzo.* Gli utenti possono sbagliare a utilizzare il sistema software. Da quando l’hardware e il software sono diventati più affidabili, gli errori di utilizzo del software sono la causa principale dei malfunzionamenti di un sistema software.

Questi guasti sono tra loro correlati. Un componente hardware guasto può richiedere agli operatori del sistema di affrontare situazioni inaspettate con carico di

lavoro aggiuntivo; ciò aumenta il loro stress, e le persone sotto stress spesso commettono errori, che possono causare errori del software che, a loro volta, generano ulteriore lavoro e altro stress per gli operatori, e così via.

Di conseguenza, è particolarmente importante che i progettisti di sistemi fidati che fanno uso intensivo di software abbiano una prospettiva olistica dei sistemi sociotecnici, anziché concentrarsi su un singolo aspetto di un sistema, come le sue unità hardware e software. Se si progettano separatamente hardware, software e processi operativi, senza tener conto delle potenziali debolezze delle altre parti del sistema, è più probabile che si verifichino degli errori nelle interfacce tra le varie parti del sistema.

10.1 Proprietà della fidatezza

Tutti noi abbiamo sperimentato qualche problema provocato dal malfunzionamento dei computer. I nostri computer a volte si bloccano o non operano correttamente senza una ragione apparente. I programmi eseguiti su questi computer non operano come previsto e, occasionalmente, danneggiano i dati gestiti dal sistema. Abbiamo imparato a convivere con questi guasti, e pochi di noi si fidano ciecamente dei computer che usano.

La fidatezza di un sistema software è una proprietà che rispecchia il livello di fiducia che l'utente ripone nel sistema. In questo contesto, l'affidabilità indica essenzialmente il grado di fiducia dell'utente sul corretto funzionamento del sistema e sul fatto che il sistema non “fallirà” durante il normale utilizzo. Non ha senso esprimere numericamente la fidatezza di un sistema; tuttavia, termini come “non affidabile”, “molto affidabile” e “superaffidabile” possono riflettere il grado di fiducia che si ha verso un sistema.

La fidatezza ha cinque proprietà principali, come illustra la Figura 10.1.

1. *Disponibilità*. Informalmente, è la probabilità che un sistema sia attivo e in grado di fornire servizi utili in qualsiasi momento.
2. *Affidabilità*. Informalmente, è la probabilità che, in un determinato periodo di tempo, un sistema sia in grado di fornire correttamente i servizi secondo le aspettative dell'utente.
3. *Sicurezza*. Informalmente, è la stima delle probabilità che il sistema possa causare danni alle persone o all'ambiente.
4. *Protezione*. Informalmente, è la stima delle probabilità che il sistema possa resistere a intrusioni accidentali o deliberate.
5. *Resilienza*. Informalmente, è la stima delle probabilità che il sistema possa garantire la continuità dei servizi critici al verificarsi di eventi dannosi, come il guasto di un componente o un attacco cibernetico. La resilienza è una proprietà che è stata aggiunta a quelle originariamente suggerite da Laprie.

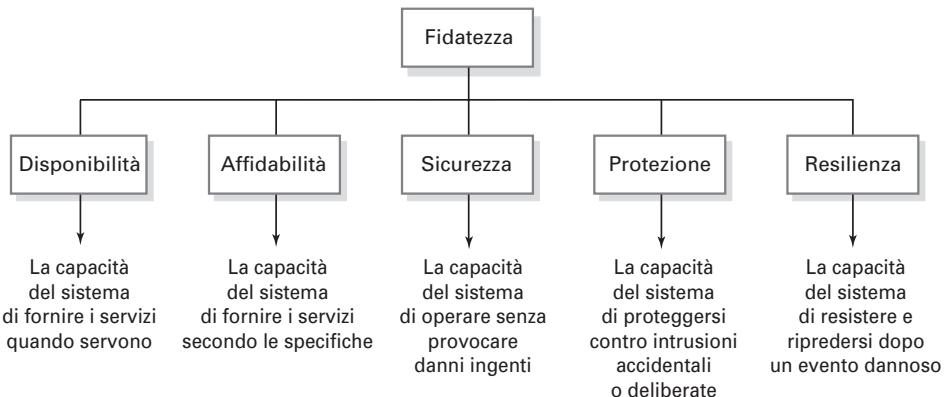


Figura 10.1 Proprietà principali della fidatezza.

Le proprietà della fidatezza mostrate nella Figura 10.1 sono caratteristiche complesse che possono essere suddivise in caratteristiche più semplici. Per esempio, la protezione include l'*integrità* (la garanzia che i programmi e i dati del sistema non vengano danneggiati) e la *riservatezza* (la garanzia che le informazioni siano accessibili soltanto alle persone autorizzate). L'affidabilità include la *correttezza* (la garanzia che i servizi di sistema siano quelli specificati), la *precisione* (la garanzia che le informazioni siano fornite con un livello appropriato di dettagli) e la *puntualità* (la garanzia che le informazioni siano fornite quando richieste).

Ovviamente, non tutte le proprietà della fidatezza sono critiche per tutti i sistemi. Per il sistema di controllo della pompa di insulina, descritto nel Capitolo 1, le proprietà più importanti sono l'affidabilità (il sistema deve fornire la dose corretta di insulina) e la sicurezza (il sistema non dovrà fornire mai una dose nociva di insulina). La protezione non è un problema, in quanto la pompa non contiene informazioni riservate; inoltre, non essendo connessa in rete, non può subire attacchi cibernetici. Per il sistema di controllo delle stazioni meteorologiche, la disponibilità e l'affidabilità sono le proprietà più importanti, in quanto i costi di riparazione possono essere molto alti. Per il sistema Mentcare, la protezione e la resilienza sono particolarmente importanti per la presenza di dati sensibili nel sistema e perché il sistema deve essere disponibile per le consultazioni dei pazienti.

Altre proprietà sono strettamente correlate a queste cinque caratteristiche della fidatezza di un sistema software.

1. *Riparabilità*. I guasti di un sistema sono inevitabili, ma i danni provocati possono essere ridotti al minimo se il sistema può essere riparato rapidamente. Deve essere possibile diagnosticare il problema, accedere al componente che causato il guasto e apportare le modifiche necessarie alla sua riparazione. La riparabilità del software migliora quando l'azienda che utilizza il sistema ha accesso al codice sorgente e ha la possibilità di modifi-

carlo. Il software open-source semplifica questa operazione, mentre il riutilizzo dei componenti potrebbe renderla più difficoltosa.

2. *Mantenibilità*. Quando i sistemi vengono usati, sorgono nuovi requisiti; è importante mantenere il valore di un sistema modificandolo per includere questi nuovi requisiti. Il software mantenibile è quello che può essere adattato in modo economico per soddisfare i nuovi requisiti, con poche probabilità che le modifiche apportate possano introdurre nuovi errori nel sistema.
3. *Tolleranza degli errori*. Questa proprietà può essere considerata parte dell'utilizzabilità; indica fino a che punto il sistema è stato progettato in modo da evitare e tollerare gli errori di input da parte degli utenti. Quando l'utente commette un errore, il sistema dovrebbe essere in grado di identificarlo e correggerlo automaticamente oppure chiedere all'utente di reinserire i dati.

Il concetto di fidatezza di un sistema come proprietà globale è stato introdotto perché le proprietà di disponibilità, protezione, affidabilità, sicurezza e resilienza sono strettamente correlate. Un sistema può diventare inaffidabile perché un hacker ha danneggiato i suoi dati. Gli attacchi denial-of-service hanno lo scopo di compromettere la disponibilità di un sistema. Se un sistema è affetto da un virus, non possiamo essere sicuri della sua affidabilità o protezione, in quanto il virus può aver cambiato il suo comportamento.

Per sviluppare un software fidato occorre quindi:

1. evitare di introdurre errori accidentali nel sistema durante la specifica e lo sviluppo del software;
2. progettare processi di verifica e convalida in grado di identificare errori residui che influiscono sulla fidatezza del sistema;
3. progettare un sistema tollerante agli errori in modo che possa continuare a funzionare anche quando qualcosa va male;
4. progettare meccanismi di protezione contro attacchi esterni che possono compromettere la disponibilità o la protezione del sistema;
5. configurare il sistema e il software di supporto correttamente per il loro ambiente operativo;
6. includere alcune funzionalità in grado di riconoscere e resistere a eventuali attacchi cibernetici esterni;
7. progettare il sistema in modo che possa riprendere il suo corretto funzionamento dopo un guasto o un attacco cibernetico senza perdere dati critici.

Tolleranza ai guasti significa che i sistemi fidati devono includere un codice ridondante per aiutare i sistemi a monitorare se stessi, scoprire stati di errore e riprendersi prima che si verifichi un guasto. Questo influisce sulle prestazioni dei sistemi, in quanto servono controlli addizionali ogni volta che il sistema viene eseguito. Di conseguenza, i progettisti devono trovare un giusto compromesso tra

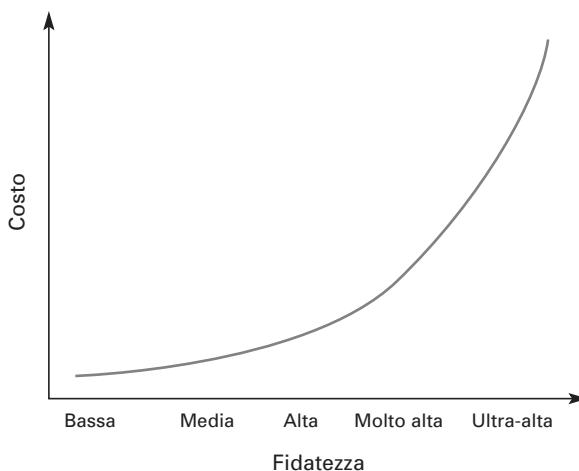


Figura 10.2 Relazione costi/fidatezza.

prestazioni e fidatezza. Qualcuno potrebbe decidere di escludere questi controlli aggiuntivi perché rallentano il sistema; il rischio che ne deriva è che il sistema potrebbe fallire se non viene rilevato un guasto.

Realizzare sistemi fidati è costoso. Aumentare la fidatezza di un sistema significa che occorrono costi extra per la progettazione, l'implementazione e la convalida del sistema. I costi di verifica e convalida sono particolarmente elevati per i sistemi che devono essere ultrafidati, come i sistemi di controllo a sicurezza critica. Analogamente, per convalidare che un sistema soddisfa i suoi requisiti, potrebbe essere richiesta un'autorità esterna. Per esempio, i sistemi di navigazione aerea devono dimostrare all'autorità esterna, come la FAA (Federal Aviation Authority) negli USA, che la probabilità che si verifichi un guasto con esiti catastrofici sulla sicurezza di un aereo è estremamente bassa.

La Figura 10.2 illustra la relazione tra costi e miglioramenti incrementali della fidatezza. Se il vostro software non deve essere molto fidato, potrete ottenere miglioramenti significativi della fidatezza con costi relativamente ridotti adottando tecniche di ingegneria del software più avanzate. Se invece state già adottando delle buone tecniche, i costi di miglioramento sono molto più alti, e i vantaggi derivanti sono meno significativi.

C'è anche il problema del test del software per dimostrare la sua fidatezza. Per risolvere questo problema, occorre eseguire molti test e controllare il numero di guasti che si verificano. Quando il software diventa più fidato, il numero di guasti durante i test si riduce. Di conseguenza, occorre eseguire un numero sempre crescente di test per determinare quanti problemi permangono nel software. Il test del software è un processo molto costoso, quindi può aumentare significativamente il costo dei sistemi ultrafidati.

10.2 Sistemi sociotecnici

In un sistema informatico, il software e l'hardware sono interdipendenti. Senza l'hardware, un sistema informatico è un'astrazione, che è semplicemente una rappresentazione delle conoscenze e delle idee dell'uomo. Senza il software, l'hardware è un insieme di inerti dispositivi elettronici. Tuttavia, se poniamo insieme il software e l'hardware per formare un sistema, possiamo creare una macchina in grado di svolgere calcoli complessi e fornire i risultati di questi calcoli.

Questo illustra una delle caratteristiche fondamentali di un sistema informatico – è molto più che la somma delle sue parti. I sistemi hanno proprietà che diventano apparenti soltanto quando i loro componenti vengono integrati e operano insieme. I sistemi software non sono sistemi isolati, ma parti di sistemi più estesi che hanno finalità umane, sociali e organizzative. Quindi, l'ingegneria del software non è un'attività isolata, ma anche una parte intrinseca dell'ingegneria dei sistemi.

Per esempio, il software del sistema delle stazioni meteorologiche controlla gli strumenti delle varie stazioni meteorologiche, comunica con altri sistemi software ed è parte di un sistema più vasto di previsioni meteorologiche nazionali e internazionali. Oltre al software e all'hardware, questi sistemi includono processi per la previsione del tempo e persone che utilizzano i sistemi e ne analizzano i dati di output. I sistemi includono anche le organizzazioni che forniscono i dati sulle previsioni del tempo a singoli individui e alle aziende pubbliche e private.

Questi sistemi più estesi sono detti *sistemi sociotecnici*. Essi includono elementi non tecnici, come le persone, i processi e i regolamenti, come pure componenti tecnici, come i computer, il software e altre apparecchiature. La fidatezza di un sistema è influenzata da tutti gli elementi in un sistema sociotecnico – hardware, software, persone e organizzazioni.

I sistemi sociotecnici sono così complessi che è impossibile comprenderli come un blocco unico. È meglio considerarli come un insieme di strati, come illustra la Figura 10.3. Questi strati formano lo stack di un sistema sociotecnico.

1. *Strato delle apparecchiature.* È composto da unità hardware, alcune delle quali possono essere computer.
2. *Strato del sistema operativo.* Interagisce con l'hardware e fornisce una serie di funzioni comuni agli strati superiori nel sistema.
3. *Strato delle comunicazioni e gestione dei dati.* Amplia le funzioni offerte dallo strato del sistema operativo e fornisce un'interfaccia che consente le interazioni con funzionalità più estese, quali l'accesso ai sistemi remoti e ai database del sistema. Questo strato è anche detto middleware, in quanto si trova tra il sistema operativo e le applicazioni.
4. *Strato delle applicazioni.* Fornisce funzionalità specifiche per le applicazioni. In questo strato possono trovarsi vari programmi applicativi.



Figura 10.3 Lo stack di un sistema sociotecnico.

5. *Strato dei processi aziendali.* Include i processi aziendali che utilizzano il sistema software.
6. *Strato organizzativo.* Include i processi strategici di alto livello, per esempio le norme, le direttive e le politiche aziendali che devono essere applicate quando si usa il sistema.
7. *Strato sociale.* Contiene le leggi e i regolamenti della società che governa il funzionamento del sistema.

Si noti che non c'è uno "strato del software" separato. Il software di qualsiasi tipo è una parte importante di tutti gli strati del sistema sociotecnico. Le apparecchiature sono controllate dal software incorporato; il sistema operativo e le applicazioni sono software. I processi aziendali, le organizzazioni e le società si affidano a Internet (software) e ad altri sistemi software globali.

In teoria, la maggior parte delle interazioni dovrebbero avvenire tra due strati adiacenti nello stack, dove ciascuno strato nasconde i dettagli dello strato precedente a quello successivo. In pratica, invece, possono verificarsi delle interazioni impreviste tra gli strati, che possono generare problemi nell'intero sistema. Per esempio, supponiamo che sia modificata la legge che regola l'accesso alle informazioni personali. Questo avviene nello strato sociale. La modifica richiede nuove procedure organizzative e alcuni cambiamenti nei processi aziendali. Il sistema applicativo stesso potrebbe non essere più in grado di fornire il livello richiesto di riservatezza, quindi potrebbe essere necessario implementare le modifiche nello strato delle comunicazioni e gestione dei dati.

Esaminando la protezione e l'affidabilità di un sistema software, è essenziale osservare il sistema da un punto di vista olistico, anziché considerare isolatamente i singoli componenti software. Il software stesso è intangibile e, anche se danneggiato, può essere ripristinato in modo semplice ed economico. Tuttavia, se un

malfunzionamento del software si propaga su altre parti del sistema, esso influenza anche sull’ambiente fisico e umano del software; qui le conseguenze del malfunzionamento sono più significative. Dati importanti potrebbero danneggiarsi o perdersi del tutto. Le persone potrebbero essere costrette a svolgere un lavoro addizionale per limitare i danni o per ripristinare il sistema; per esempio, le apparecchiature potrebbero danneggiarsi, i dati potrebbero perdersi o rovinarsi, o le procedure di riservatezza potrebbe essere state violate, con conseguenze imprevedibili.

Occorre avere, quindi, una visione globale del sistema quando si progetta un software che deve essere affidabile e protetto. Occorre prendere in considerazione le conseguenze dei malfunzionamenti del software su altri elementi del sistema. È necessario capire anche come questi elementi possono innescare a loro volta altri malfunzionamenti del software e come proteggersi dalle loro conseguenze.

È importante impedire che, ove possibile, un malfunzionamento del software provochi un guasto dell’intero sistema. Occorre esaminare come il software interagisce con il suo ambiente più immediato per garantire che:

1. i malfunzionamenti del software siano, per quanto possibile, contenuti nello strato in cui si verificano, evitando che influiscano sul funzionamento di altri strati dello stack del sistema;
2. i guasti e i malfunzionamenti di altri strati dello stack del sistema non influiscano sul software che si sta sviluppando. Occorre anche realizzare i controlli appropriati nel software per identificare questi malfunzionamenti e le procedure che consentono di ripristinare il corretto funzionamento del software dopo un guasto.

Poiché il software è intrinsecamente flessibile, la soluzione dei problemi imprevisti del sistema di solito viene lasciata agli ingegneri del software. Supponiamo che un’installazione radar sia stata collocata in un punto in cui si verifica un fenomeno di ghosting (ombre sul monitor). Poiché non è possibile spostare il radar per ridurre le interferenze, gli ingegneri del sistema devono trovare una soluzione per eliminare il problema del ghosting. La loro soluzione potrebbe essere quella di migliorare le capacità di elaborazione delle immagini del software in modo da rimuovere le ombre sul monitor. Questa soluzione potrebbe rallentare il software, rendendo le sue prestazioni inaccettabili. Il problema potrebbe essere classificato come “malfunzionamento del software”, mentre in realtà si tratta di un errore nella progettazione dell’intero sistema.

Questi casi in cui gli ingegneri del software devono migliorare le capacità del software senza aumentare i costi dell’hardware sono molto comuni. Molti dei cosiddetti malfunzionamenti del software non sono una conseguenza di problemi intrinseci del software, ma piuttosto il risultato di un tentativo di modificare il software per adattarlo ai nuovi requisiti di ingegneria del sistema. Un tipico esempio è stato il fallimento del sistema di gestione dei bagagli nell’aeroporto di Denver (Swartz 1996), dove ci si aspettava che il software di controllo riuscisse a

funzionare correttamente nonostante le varie limitazioni delle attrezzature utilizzate.

10.2.1 Norme e conformità

Il modello generale di un'organizzazione economica, che oggi è quasi universale, è che le società di capitale privato offrono beni e servizi ricavandone dei profitti. Operando in un ambiente concorrenziale, queste società possono competere sui costi, sulla qualità, sui tempi di consegna e così via. Tuttavia, per garantire la sicurezza dei loro cittadini, molti governi limitano la libertà delle società private in modo che queste adottino alcuni standard per assicurare che i loro prodotti siano sicuri e protetti. Una società quindi non può vendere prodotti troppo economici, in quanto per abbassare i costi di produzione dovrebbe ridurre la sicurezza dei prodotti.

I governi hanno promulgato una serie di norme e regolamenti in vari settori che definiscono gli standard di sicurezza e protezione dei prodotti. Essi hanno anche istituito gli enti o organismi di regolamentazione il cui compito è assicurare che le società che offrono prodotti in un determinato settore rispettino tali norme. Questi organismi di regolamentazione hanno ampi poteri; possono multare le aziende e perfino fare arrestare i responsabili aziendali se vengono violate le norme. Possono svolgere un ruolo attivo nella concessione delle licenze (per esempio, nell'industria nucleare o aerospaziale), che sono indispensabili per poter utilizzare un nuovo sistema o prodotto. Per esempio, i costruttori di aerei devono ottenere una certificazione di idoneità al volo dagli enti di regolamentazione in ogni stato dove gli aerei dovranno essere utilizzati.

Per ottenere la certificazione, le aziende che sviluppano sistemi a sicurezza critica devono produrre un caso di sicurezza (descritto nel Capitolo 12) che dimostra che le norme e i regolamenti sono stati seguiti. Il caso deve convincere il responsabile dell'ente di regolamentazione che il sistema è in grado di operare in modo sicuro. Sviluppare un caso di sicurezza è molto costoso. Può essere costoso sviluppare sia la documentazione per la certificazione sia il sistema stesso.

Le norme e i regolamenti si applicano a un sistema sociotecnico nel suo complesso, non al singolo componente software del sistema. Per esempio, nell'industria nucleare è importante verificare che, nel caso di surriscaldamento, un reattore nucleare non disperda radioattività nell'ambiente. Gli argomenti per convincere i responsabili di un ente di regolamentazione in questo caso potrebbero basarsi sui sistemi software di protezione, sui processi utilizzati per monitorare il nucleo del reattore e l'integrità delle strutture che contengono le sorgenti di radioattività.

Ciascuno di questi elementi deve avere il suo caso di sicurezza. Pertanto, il sistema di protezione deve avere un caso di sicurezza che dimostra che il software potrà operare correttamente e spegnere il reattore in caso di emergenza. Il caso di sicurezza globale dovrà dimostrare che, se il sistema software di protezione fallisce, ci sono meccanismi di sicurezza alternativi, che non si basano sul software, che intervengono automaticamente.

10.3 Ridondanza e diversità

I guasti dei componenti in qualsiasi sistema sono inevitabili. Le persone commettono errori; bug nascosti nel software provocano comportamenti indesiderati del software; e l'hardware a volte fonde. Utilizziamo una serie di strategie per ridurre al minimo il numero di errori umani, come sostituire i componenti hardware prima della fine del loro ciclo di vita e controllare il software utilizzando strumenti statici di analisi. Tuttavia, non possiamo avere la certezza che queste strategie eliminaranno completamente i guasti dei componenti. Per questo occorre progettare i sistemi in modo che il malfunzionamento di un singolo componente non provochi il blocco di tutto il sistema.

Le strategie per ottenere e migliorare la fidatezza si basano sia sulla ridondanza sia sulla diversità. Ridondanza significa che occorre includere delle capacità di riserva in un sistema che può essere utilizzato quando una sua parte fallisce. Diversità significa che i componenti ridondanti del sistema sono di tipo differente, aumentando così le probabilità che essi non falliranno esattamente nello stesso modo.

Utilizziamo la ridondanza e la diversità per migliorare la fidatezza nella nostra vita quotidiana. Di solito, per proteggere le nostre case, utilizziamo più di una serratura nelle porte (ridondanza) e, spesso, queste serrature sono di tipo differente (diversità). Questo significa che, se un ladro riesce ad aprire una serratura, dovrà trovare un modo diverso per aprire l'altra serratura. In generale, tutti noi dovremmo fare periodicamente il backup dei nostri computer, in modo da conservare delle copie ridondanti dei dati. Per evitare il rischio di un guasto del disco fisso, il backup dovrebbe essere effettuato su dispositivi esterni, diversi e separati.

I sistemi software che sono progettati per la fidatezza possono includere componenti ridondanti che forniscono le stesse funzionalità di altri componenti. I componenti ridondanti possono essere attivati nel sistema se quelli principali falliscono. Se i componenti ridondanti sono diversi dagli altri componenti, un guasto comune nei componenti replicati non provocherà il blocco del sistema. Un'altra forma di ridondanza consiste nell'includere un codice di controllo, che non è strettamente necessario per il funzionamento del sistema. Il codice può rilevare alcuni tipi di problemi, come il danneggiamento dei dati, prima che essi provochino un guasto; può avviare dei meccanismi di recovery per correggere i problemi, consentendo così al sistema di continuare a funzionare.

Nei sistemi per i quali la disponibilità è un requisito critico, di solito, si usano dei server ridondanti, che entrano automaticamente in funzione se il server principale fallisce. A volte, per fare in modo che gli hacker che attaccano il sistema non siano in grado di sfruttare un punto vulnerabile comune, questi server possono essere di tipo differente ed essere eseguiti su sistemi operativi differenti. Usare sistemi operativi differenti è un esempio di diversità e ridondanza del software, dove le stesse funzionalità sono fornite in modi differenti (la diversità del software è descritta dettagliatamente nel Capitolo 12).

L'esplosione di Ariane 5

Nel 1996, il razzo Ariane 5 dell'Agenzia Spaziale Europea esplose 37 secondi dopo il suo primo lancio. Il guasto fu causato da un malfunzionamento dei sistemi software. Era disponibile un sistema di backup, ma non era di tipo differente dal sistema principale; quindi il software nel computer di backup ebbe esattamente lo stesso tipo di guasto. Il razzo e il suo carico furono distrutti.

<http://software-engineering-book.com/web/ariane/>

La diversità e la ridondanza possono essere utilizzate anche nella progettazione di processi fidati per lo sviluppo del software. Questi processi evitano l'introduzione di guasti nel sistema. In un processo fidato le attività, quali la convalida del software, non si affidano a un singolo strumento o tecnica. Questo migliora la fidatezza del software perché riduce le probabilità di guasti nei processi, dove l'errore umano commesso durante lo sviluppo del software porta a errori nel software.

Per esempio, le attività di convalida possono includere i test dei programmi, le ispezioni manuali dei programmi e l'analisi statica, come le tecniche di ricerca dei guasti. Una qualsiasi di queste tecniche potrebbe individuare un guasto che sfugge alle altre tecniche. Inoltre, più membri del team di sviluppo potrebbero essere responsabili della stessa attività di processo (per esempio, l'ispezione di un programma). Le persone svolgono i loro compiti in modi differenti, a seconda delle loro personalità, esperienza e istruzione, quindi questo tipo di ridondanza offre una prospettiva diversa sul sistema.

Tuttavia, come dirò nel Capitolo 11, l'uso stesso della ridondanza e della diversità può introdurre bug nel software. La diversità e la ridondanza rendono i sistemi più complessi e, di solito, anche più difficili da capire. Non solo c'è più codice da scrivere e controllare, ma potrebbe essere necessario aggiungere nuove funzionalità al sistema per identificare un componente guasto e commutare le attività su un componente alternativo. Questa complessità aggiuntiva significa che è più probabile che i programmatori commettano qualche errore ed è meno probabile che le persone che controllano il sistema trovino questi errori.

Alcuni ingegneri ritengono che è meglio evitare la ridondanza e la diversità, in quanto l'approccio migliore è progettare il software nel modo più semplice possibile, seguendo procedure di verifica e convalida molto rigorose (Parnas, van Schouwen e Shu 1990). Non dovendo sviluppare componenti software ridondanti, si ha più tempo da dedicare al perfezionamento delle procedure di verifica e convalida.

Entrambi gli approcci vengono adottati nello sviluppo di sistemi software a sicurezza critica. Per esempio, il software e l'hardware di controllo dell'Airbus 340 includono sia la ridondanza sia la diversità. Il software di controllo di un Boeing 777 viene eseguito su un hardware ridondante, ma ciascun computer esegue lo stesso software, che è stato convalidato tramite procedure molto complesse.

Processi operativi fidati

Questo capitolo tratta i processi di sviluppo fidati, ma anche i processi operativi sono importanti per la fidatezza del sistema. Nel progettare questi processi operativi occorre prendere in considerazione i fattori umani e ricordarsi che le persone possono sbagliare quando usano un sistema software. Un processo fidato dovrebbe essere progettato in modo da evitare gli errori umani e, se viene commesso un errore, il software dovrebbe identificarlo e consentire all'utente di correggerlo.

<http://software-engineering-book.com/web/human-error/>

Gli ingegneri del sistema di controllo dei Boeing 777 hanno preferito la semplicità, alla ridondanza del software. Entrambi questi aerei sono molto affidabili, quindi sia il metodo della diversità sia quello della semplicità sono indubbiamente efficaci.

10.4 Processi fidati

I processi software fidati sono processi che sono progettati per produrre un software fidato. La ragione di investire nei processi fidati è che un buon processo software molto probabilmente genera un software che contiene pochi errori e, quindi, con minori probabilità di malfunzionamento durante l'esecuzione. Una società che usa un processo fidato può essere sicura che il processo è stato appropriatamente collaudato e documentato e che sono state adottate tecniche di sviluppo idonee per sviluppare i sistemi critici. La Figura 10.4 mostra alcuni degli attributi dei processi software fidati.

La prova che è stato utilizzato un processo fidato spesso è importante per convincere un ente di regolamentazione che sono state adottate le più efficienti tecniche di ingegneria del software nello sviluppo del software. Gli sviluppatori di sistemi di solito presentano un modello del processo all'ente di regolamentazione, insieme alla prova che tale modello è stato applicato. Oltre a questo, occorre convincere l'ente di regolamentazione che il modello è stato applicato con coerenza da tutti i partecipanti al processo e che esso può essere adottato in progetti di sviluppo differenti. Questo significa che il processo deve essere esplicitamente definito e ripetibile.

1. Un processo esplicitamente definito è un processo che ha un modello definito, che è stato utilizzato come guida nella produzione del software. Devono essere raccolti dei dati durante il processo per dimostrare che il team di sviluppo ha seguito il processo secondo le direttive del modello.
2. Un processo ripetibile è un processo che non si basa su singole interpretazioni e giudizi, ma può essere applicato a vari progetti con team di sviluppo differenti, indipendentemente dalle persone che si occupano dello sviluppo. Questo è particolarmente importante per i sistemi critici, che spesso hanno

Caratteristica	Descrizione
Verificabile	Il processo dovrebbe essere comprensibile a persone diverse da quelle che prendono parte al processo, che possono verificare che gli standard di progettazione sono applicati e che possono dare suggerimenti per il suo miglioramento.
Differente	Il processo dovrebbe includere attività di verifica e convalida ridondanti e diverse.
Documentabile	Il processo dovrebbe avere un modello di processo definito che stabilisce le attività del processo e la documentazione da produrre durante queste attività.
Robusto	Il processo dovrebbe essere in grado di correggere gli errori derivanti dalle singole attività del processo.
Standardizzato	Dovrebbe essere disponibile una serie completa di standard di sviluppo che definisce come il software deve essere prodotto e documentato.

Figura 10.4 Attributi dei processi software fidati.

un lungo ciclo di sviluppo, durante il quale si verificano cambiamenti significativi nel team di sviluppo.

I processi fidati adottano la ridondanza e la diversità per ottenere la fidatezza. Spesso includono attività differenti che svolgono lo stesso compito. Per esempio, le ispezioni e i test di un programma hanno lo scopo di identificare gli errori del programma. Le due tecniche possono essere utilizzate insieme, in modo che una tecnica possa identificare gli errori che l'altra tecnica non sarebbe in grado di rilevare.

Le attività che sono utilizzate nei processi fidati ovviamente dipendono dal tipo di software che si sta sviluppando. In generale, però, queste attività dovrebbero essere guidate in modo da evitare l'introduzione di errori nel sistema, individuare ed eliminare gli errori, e mantenere le informazioni sul processo stesso.

Il seguente elenco riporta alcuni esempi di attività che potrebbero essere incluse in un processo fidato.

1. *Ispezioni dei requisiti.* Lo scopo è verificare che i requisiti siano, per quanto possibile, completi e coerenti.
2. *Gestione dei requisiti.* Assicura che le modifiche dei requisiti siano controllate e che l'impatto delle modifiche proposte sia compreso da tutti gli sviluppatori interessati da tali modifiche.
3. *Specifica formale.* Viene creato e analizzato un modello matematico del software (i vantaggi della specifica formale sono descritti nel Paragrafo 10.5). Probabilmente il vantaggio più significativo della specifica formale è che viene imposta un'analisi dettagliata dei requisiti del sistema.

È molto probabile che durante questa analisi emergano quei problemi dei requisiti che potrebbero essere ignorati dalle ispezioni dei requisiti.

4. *Modellazione del sistema.* Viene documentato in modo esplicito il progetto del software come una serie di modelli grafici; sono documentati anche i collegamenti tra i requisiti e questi modelli. Se viene adottato un approccio ingegneristico guidato da modelli (si veda il Capitolo 5), è possibile generare automaticamente il codice da questi modelli.
5. *Ispezioni dei progetti e dei programmi.* Le varie descrizioni del sistema vengono ispezionate e controllate da diverse persone. Durante il processo di ispezione può essere utilizzata una lista di controllo degli errori più comuni di progettazione e programmazione.
6. *Analisi statica.* Vengono effettuati alcuni controlli automatici sul codice sorgente dei programmi, con l'obiettivo di scoprire eventuali anomalie che potrebbero indicare errori od omissioni di programmazione (l'analisi statica è descritta nel Capitolo 12).
7. *Pianificazione e gestione dei test.* Viene progettata una serie completa di test per il sistema. Il processo di test deve essere accuratamente gestito per dimostrare che i test offrono una copertura completa dei requisiti del sistema e sono stati correttamente applicati.

Oltre alle attività dei processi che si focalizzano sullo sviluppo e i test dei sistemi, occorrono anche i processi di gestione della qualità e delle modifiche dei requisiti. Mentre le attività specifiche in un processo fidato variano da una società all'altra, l'esigenza di una gestione efficiente della qualità e delle modifiche è universale.

I processi di gestione della qualità (si veda il Capitolo 21) stabiliscono una serie di standard di processo e di prodotto. Essi includono anche alcune attività che catturano le informazioni dei processi per dimostrare che questi standard vengono rispettati. Per esempio, ci potrebbe essere uno standard che definisce come svolgere le ispezioni dei programmi. Il responsabile del team di ispezione ha il compito di documentare il processo per dimostrare che lo standard delle ispezioni è stato applicato.

La gestione delle modifiche, descritta nel Capitolo 22, si occupa delle modifiche da apportare a un sistema, garantendo che le modifiche accettate vengano effettivamente implementate e che siano incluse nelle release pianificate del software. Un problema comune è che vengano inclusi in un sistema software i componenti sbagliati. Ciò potrebbe determinare una situazione nella quale un sistema in esecuzione includa componenti che non sono stati controllati durante il processo di sviluppo. Le procedure di gestione della configurazione devono essere definite come parte del processo di gestione delle modifiche per garantire che ciò non accada.

A causa del crescente utilizzo dei metodi agili, ricercatori e professionisti hanno riflettuto attentamente sul modo in cui impiegare gli approcci agili nello sviluppo di software fidato (Trimble 2012). Molte società che realizzano sistemi software critici affidano il loro sviluppo a processi basati su piani e sono riluttanti ad apportare modifiche radicali alle loro procedure. Esse riconoscono comunque il valore degli approcci agili e stanno studiando il modo in cui rendere più agili i loro processi di sviluppo di software fidato.

Poiché il software fidato spesso richiede una certificazione, occorre preparare sia la documentazione di prodotto sia quella di processo. È essenziale anche l'analisi dei requisiti up-front per scoprire eventuali conflitti tra requisiti che potrebbero compromettere la sicurezza e la protezione del sistema. È importante l'analisi formale delle modifiche per stabilire i loro effetti sulla sicurezza e l'integrità del sistema. Queste esigenze sono in conflitto nell'approccio agile che prevede lo sviluppo agile dei requisiti e del sistema e la minimizzazione della documentazione.

Sebbene la maggior parte dei metodi agili utilizzino processi informali, privi di documentazione, questo non è un requisito fondamentale dell'agilità. Un processo agile può includere tecniche che incorporano lo sviluppo iterativo, lo sviluppo con test iniziali e il coinvolgimento degli utenti nel team di sviluppo. Se il team di sviluppo adotta tali tecniche e documenta le sue azioni, è ancora possibile utilizzare le tecniche agili. Per supportare questo concetto, sono state fatte varie proposte di modifica dei metodi agili che includono i requisiti dell'ingegneria dei sistemi fidati (Douglass 2013). Questi metodi agili modificati combinano le tecniche più appropriate dello sviluppo agile con quelle dello sviluppo basato su piani.

10.5 Metodi formali e fidatezza

Per oltre 30 anni i ricercatori hanno consigliato l'uso di metodi formali nello sviluppo del software. Questi metodi sono approcci matematici allo sviluppo del software, in cui viene definito un modello formale del software. Poi è possibile analizzare questo modello per cercare errori e incongruenze, dimostrare che un programma è coerente con il modello, o applicare una serie di trasformazioni del modello per generare un programma. Abrial (Abrial 2009) sostiene che l'uso dei metodi formali potrebbe portare a realizzare “sistemi privi di errori”, sebbene raccomandi prudenza nell'interpretare questo suo concetto.

In un eccellente sondaggio, Woodcock (Woodcock et al. 2009) ha esaminato alcune applicazioni industriali dove sono stati applicati con successo i metodi formali. Queste applicazioni includono i sistemi di controllo dei treni (Badeau e Amelot 2005), i sistemi di gestione delle carte di credito (Hall e Chapman 2002) e i sistemi di controllo dei voli (Miller et al. 2005). I metodi formali sono la base degli strumenti utilizzati nelle verifiche statiche, come il sistema di verifica dei driver utilizzato da Microsoft (Ball et al. 2006).

L'uso di un approccio matematico formale allo sviluppo del software fu proposto nelle fasi iniziali dello sviluppo dell'informatica. L'idea era che una specifica formale e un programma potevano essere sviluppati in modo indipendente. Successivamente era possibile fornire una prova matematica per dimostrare che il progetto e la sua specifica erano coerenti. Fu subito chiaro che era possibile sviluppare delle prove manuali soltanto per sistemi molto piccoli. La prova dei programmi adesso è supportata da software speciali che eseguono dimostrazioni di teoremi su larga scala, che possono essere applicati a sistemi più grandi. Tuttavia, sviluppare le prove per questo tipo di software è un compito difficile e specializzato; per questo la verifica formale non si è molto diffusa.

Un approccio alternativo, che evita un'attività di prova separata, è lo sviluppo basato sull'affinamento della specifica. Una specifica formale di un sistema viene affinata attraverso una serie di trasformazioni per generare il software. Poiché le trasformazioni sono affidabili in quanto preservano la correttezza del codice, si può avere la certezza che il programma generato sia coerente con la sua specifica formale. Questo approccio venne utilizzato nello sviluppo del software del sistema Paris Metro (Badeau e Amelot 2005). Venne utilizzato un linguaggio detto B (Abrial 2010), che era stato ideato per supportare l'affinamento della specifica.

I metodi formali basati sulla verifica dei modelli (Jhala e Majumdar 2009) sono stati utilizzati in vari sistemi (Bachot et al. 2009; Calinescu e Kwiatkowska 2009). Questi sistemi si basano sulla costruzione o generazione di un modello formale degli stati di un sistema e sull'utilizzo di un verificatore di modelli che controlla che le proprietà del modello, per esempio la sicurezza, siano sempre assicurate. Il programma di verifica dei modelli analizza tutta la specifica e indica se le proprietà del sistema sono rispettate dal modello oppure presenta un esempio che dimostra il contrario. Se un modello può essere automaticamente o sistematicamente generato da un programma, questo significa che i bug nel programma possono essere scoperti (la verifica dei modelli nei sistemi critici è trattata nel Capitolo 12).

I metodi formali per l'ingegneria del software sono efficaci per scoprire e analizzare due classi di errore nelle rappresentazioni del software.

1. *Errori e omissioni.* Il processo di sviluppo e analisi di un modello formale del software può svelare errori e omissioni nei requisiti del software. Se il modello viene generato automaticamente o sistematicamente dal codice sorgente, l'analisi basata sulla verifica dei modelli può scoprire stati indesiderabili che potrebbero verificarsi, come lo stallo (deadlock) di un sistema concorrente.
2. *Incongruenze tra specifica e programma.* Se si usa un metodo di affinamento, si evitano gli errori fatti dagli sviluppatori che rendono incongruente il software con la specifica. La verifica del programma rivela le incongruenze tra un programma e la sua specifica.

Tecniche di specifiche formali

Le specifiche formali di un sistema possono essere espresse utilizzando due approcci fondamentali, come modelli delle interfacce del sistema (specifiche algebriche) o come modelli dello stato del sistema. Un capitolo online su questo argomento descrive alcuni esempi di questi approcci. Il capitolo include una specifica formale di una parte del sistema della pompa di insulina.

<http://software-engineering-book.com/web/formal-methods/> (Capitolo 27)

Il punto di partenza di tutti i metodi formali è un modello matematico del sistema, che funge da specifica del sistema. Per creare questo modello, occorre tradurre i requisiti del sistema, espressi dall'utente nel linguaggio naturale, in un linguaggio matematico che ha una semantica formale. La specifica formale è una descrizione non ambigua di ciò che il sistema dovrebbe fare.

Le specifiche formali sono il modo più preciso di specificare un sistema e, quindi, di ridurre al minimo le cattive interpretazioni dei requisiti. Molti sostenitori dei metodi formali ritengono che è sempre utile creare una specifica formale, anche senza affinamento o verifica del programma. La creazione di una specifica formale richiede un'analisi dettagliata dei requisiti, e questo è un modo efficace per scoprire eventuali problemi dei requisiti. In una specifica scritta nel linguaggio naturale, gli errori potrebbero nascondersi a causa dell'imprecisione del linguaggio. Ciò non può accadere se il sistema viene specificato in modo formale.

I vantaggi di sviluppare una specifica formale dettagliata e di utilizzarla nei processi di sviluppo del software sono elencati qui di seguito.

1. Per sviluppare una specifica formale dettagliata occorre capire perfettamente i requisiti del sistema. Costa meno correggere i problemi dei requisiti che vengono scoperti subito, anziché nelle fasi successive del processo di sviluppo.
2. Poiché la specifica è espressa in un linguaggio che ha una semantica formalmente definita, è possibile analizzarla automaticamente per scoprire incongruenze e incompletezze.
3. Se si usa un metodo, per esempio il metodo B, è possibile trasformare la specifica formale in un programma attraverso una sequenza di trasformazioni che assicurano la correttezza del codice. Si ha così la garanzia che il programma risultante è conforme alla sua specifica.
4. I costi dei test di un programma possono essere ridotti, in quanto avete verificato che il programma soddisfa la specifica. Per esempio, nello sviluppo del software per il sistema Paris Metro, l'uso dell'affinamento richiedeva che fossero effettuati i test soltanto per il sistema, non per i componenti software.

Dal sondaggio di Woodcock (Woodcock et al. 2009) è emerso che gli utenti dei metodi formali hanno riscontrato meno errori nel software consegnato. Il tempo e i costi richiesti per sviluppare il software non erano maggiori di quelli di analoghi progetti di sviluppo. C'erano significativi vantaggi nell'utilizzare gli approcci formali nei sistemi a sicurezza critica che richiedevano una certificazione da parte degli enti di regolamentazione. La documentazione prodotta era una parte importante del caso di sicurezza (si veda il Capitolo 12) per il sistema.

Nonostante questi vantaggi, i metodi formali hanno avuto un impatto limitato sullo sviluppo pratico del software, anche per i sistemi critici. Woodcock riferisce che in 25 anni solo 62 progetti hanno utilizzato metodi formali. Anche se esistono progetti che hanno utilizzato queste tecniche senza averne pubblicizzato l'uso, tuttavia si tratta di una piccola frazione del numero totale di sistemi critici sviluppati in quel periodo. L'industria è stata riluttante ad adottare i metodi formali per vari motivi.

1. I problem owner e gli esperti dei domini non sono in grado di capire una specifica formale, quindi non possono controllare se essa rappresenta i loro requisiti. Gli ingegneri del software, che capiscono la specifica formale, potrebbero non essere in grado di capire il dominio delle applicazioni, quindi neanche loro possono essere sicuri che la specifica formale rispecchi accuratamente i requisiti del sistema.
2. È abbastanza facile quantificare i costi di creazione di una specifica formale, ma è molto più difficile stimare i possibili risparmi che derivano dal suo utilizzo. Di conseguenza, i manager sono riluttanti a correre il rischio di adottare i metodi formali. Essi non si lasciano convincere dalle notizie dei successi di questi metodi in quanto, in generale, provengono da progetti atipici i cui sviluppatori sono sostenitori appassionati dell'approccio formale.
3. Molti ingegneri del software non sono addestrati a utilizzare i linguaggi per le specifiche formali; quindi sono riluttanti a proporne l'uso nei loro processi di sviluppo.
4. È difficile scalare i metodi formali attuali nei sistemi molto grandi. I metodi formali si usano principalmente per specificare il software critico del kernel, non per il sistema completo.
5. Gli strumenti di supporto ai metodi formali sono limitati, e quelli disponibili sono spesso open-source e difficili da utilizzare. Il mercato è troppo piccolo per giustificare l'esistenza di fornitori di tali strumenti.
6. I metodi formali non sono compatibili con lo sviluppo agile, dove i programmi sono sviluppati in modo incrementale. Tuttavia, questo non è un grosso problema, in quanto molti sistemi critici vengono ancora sviluppati utilizzando un approccio basato su piani.

Parnas, uno dei primi sostenitori dello sviluppo formale, ha criticato i metodi formali attuali, sostenendo che essi si basano su un'ipotesi fondamentalmente sbagliata (Parnas 2010). Secondo Parnas, questi metodi non potranno avere una vasta diffusione finché non saranno semplificati radicalmente, e questo richiede una base matematica di tipo diverso. Secondo il mio punto di vista, questo non significa che i metodi formali potranno essere adottati regolarmente nell'ingegneria dei sistemi critici, a meno che non si dimostri chiaramente che il loro utilizzo comporti una riduzione dei costi rispetto ad altri metodi di ingegneria del software.

Punti chiave

- La fidatezza dei sistemi è importante perché il guasto di un sistema critico può causare ingenti perdite economiche, gravi perdite di informazioni e danni fisici alle persone.
- La fidatezza di un sistema è una proprietà che rispecchia il grado di fiducia dell'utente nel sistema. Le caratteristiche più importanti della fidatezza sono la disponibilità, l'affidabilità, la sicurezza, la protezione e la resilienza.
- I sistemi sociotecnici includono l'hardware, il software e le persone e si trovano all'interno di una società. Sono progettati per consentire il raggiungimento degli obiettivi economici e produttivi di una società.
- L'uso di un processo ripetibile e fidato è essenziale per ridurre al minimo i guasti di un sistema. Il processo deve includere attività di verifica e convalida in tutte le fasi, dalla definizione dei requisiti all'implementazione del sistema.
- L'uso della diversità e della ridondanza nell'hardware, nei processi software e nei sistemi software è essenziale per sviluppare sistemi fidati.
- I metodi formali, nei quali si utilizza un modello formale come base di sviluppo di un sistema, aiutano a ridurre il numero di errori di specifica e implementazione di un sistema. I metodi formali hanno un avuto una diffusione limitata nell'industria del software a causa dei dubbi sulla loro effettiva capacità di riduzione dei costi.

Esercizi

- * 10.1 Indicate sei ragioni che spiegano perché è importante la fidatezza del software in molti sistemi sociotecnici.
- * 10.2 Quali sono le caratteristiche più importanti della fidatezza di un sistema?
- 10.3 Mediante un esempio spiegate perché quando si sviluppa un sistema fidato è importante considerarlo come un sistema sociotecnico e non semplicemente un sistema tecnico di hardware e software.
- * 10.4 Indicate due esempi di funzioni governative che sono supportate da sistemi sociotecnici complessi e spiegate perché, nel prossimo futuro, queste funzioni non potranno essere completamente automatizzate.

- 10.5 Spiegate la differenza tra ridondanza e diversità.
- * 10.6 Spiegate perché è ragionevole supporre che l'uso di processi fidati porti alla creazione di software fidato.
- 10.7 Indicate due esempi di attività diverse e ridondanti che potrebbero essere incorporate nei processi fidati.
- * 10.8 Indicate due ragioni che spiegano perché versioni differenti di un sistema basato sulla diversità del software potrebbero fallire in un modo identico.
- 10.9 In qualità di ingegneri siete stati incaricati di sviluppare un piccolo sistema di controllo di treni a sicurezza critica, che deve dimostrarsi sicuro e protetto. Voi suggerite i metodi formali da utilizzare per sviluppare questo sistema, ma il vostro manager si dimostra scettico verso questo approccio. Scrivete un rapporto che mette in evidenza i benefici dei metodi formali, illustrando un caso di un loro utilizzo in questo progetto.
- 10.10 Qualcuno sostiene che la necessità di regolamentare il software ne inibisce l'innovazione e che gli enti di regolamentazione debbano forzare l'uso di metodi di sviluppo che sono stati precedentemente utilizzati in altri sistemi. Spiegate se siete d'accordo con questa opinione e se sia giusto che gli enti di regolamentazione impongano i loro punti di vista sui metodi di sviluppo da utilizzare.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Basic Concepts and Taxonomy of Dependable and Secure Computing.” Questo documento presenta un’approfondita discussione sui concetti di fidatezza descritti da alcuni dei principali pionieri che sono stati responsabili dello sviluppo di queste idee. A. Avizienis, J. C. Laprie, B. Randell e C. Landwehr, *IEEE Transactions on Dependable and Secure Computing*, 1 (1), 2004. <http://dx.doi.org/10.1109/TDSC.2004.2>

“Formal Methods: Practice and Experience.” Un’eccellente indagine sull’uso dei metodi formali nell’industria del software, oltre a una descrizione di alcuni progetti che hanno utilizzato questi metodi. Gli autori presentano una sintesi realistica degli ostacoli che impediscono il diffondersi dei metodi formali. J. Woodcock, P. G. Larsen, J. Bicarregui e J. Fitzgerald. *Computing Surveys*, 41 (1) January 2009. <http://dx.doi.org/10.1145/1592434.1592436>

The LSCITS Socio-technical Systems Handbook. Questo manuale presenta in modo accessibile i sistemi sociotecnici e suggerisce la lettura di altri documenti più dettagliati su questi sistemi. (2012) <http://archive.cs.st-andrews.ac.uk/STSE-Handbook/>

e11

Ingegneria dell'affidabilità

L'obiettivo di questo capitolo è spiegare come l'affidabilità del software può essere specificata, implementata e misurata. Dopo aver letto questo capitolo:

- capirete la differenza tra affidabilità e disponibilità del software;
- apprenderete le metriche per specificare l'affidabilità e come utilizzarle per specificare requisiti misurabili dell'affidabilità;
- capirete come sia possibile utilizzare stili architetturali differenti per implementare architetture di sistemi affidabili e fault-tolerant;
- conoscerete le buone pratiche di programmazione per l'ingegneria di sistemi software affidabili;
- capirete come l'affidabilità di un sistema software possa essere misurata mediante test statistici.

11.1 Disponibilità e affidabilità

11.2 Requisiti di affidabilità

11.3 Architetture fault-tolerant

11.4 Programmare per l'affidabilità

11.5 Misura dell'affidabilità

La nostra dipendenza dai sistemi software per quasi tutti gli aspetti personali ed economici della nostra vita significa che ci aspettiamo che il software sia disponibile ogni volta che ne abbiamo bisogno; potrebbe essere al mattino presto o a tarda sera, nel fine-settimana o durante le vacanze – il software deve essere disponibile tutto il giorno, ogni giorno dell'anno. Ci aspettiamo che il software operi correttamente, senza errori o guasti, preservando i nostri dati e le nostre informazioni personali. Abbiamo bisogno di fidarci del software che utilizziamo, e questo significa che il software deve essere affidabile.

L'uso delle tecniche di ingegneria del software, di linguaggi di programmazione migliori e di una gestione efficiente della qualità ha portato a significativi miglioramenti dell'affidabilità del software negli ultimi 20 anni. Nonostante questo, si verificano ancora malfunzionamenti nei sistemi che influiscono sulla loro disponibilità o determinano risultati errati. Nei casi in cui il software svolge un ruolo particolarmente critico – per esempio, in un aereo o in un'infrastruttura pubblica – si possono utilizzare alcune tecniche speciali di ingegneria dell'affidabilità per elevare i livelli di affidabilità e disponibilità che sono richiesti.

Purtroppo è facile fare confusione quando si parla di affidabilità dei sistemi; persone differenti che sostengono idee diverse quando discutono di guasti e malfunzionamenti dei sistemi. Brian Randell (Randell 200), un ricercatore pioniere dell'affidabilità del software, definì il modello difetto-errore-fallimento (fault-error-failure) basato sul concetto che gli errori umani generano difetti nel software; i difetti causano errori, e gli errori producono fallimenti. Randell ha definito esattamente i seguenti termini.

1. *Errore umano*. Comportamento umano che si traduce nell'introduzione di difetti nel sistema. Per esempio, nel sistema di controllo delle stazioni meteorologiche, un programmatore potrebbe decidere di calcolare il tempo della successiva trasmissione aggiungendo un'ora all'orario corrente. Questo funziona tranne quando l'ora di trasmissione è compresa tra le 23:00 e mezzanotte (che corrisponde alle 00:00 nel sistema a 24 ore).
2. *Difetto del sistema*. Caratteristica di un sistema software che può portare a un errore del sistema. Il difetto nell'esempio precedente è includere un codice per aggiungere 1 a una variabile chiamata *Tempo_di_trasmissione*, senza controllare se il valore di questa variabile è maggiore o uguale a 23:00.
3. *Errore del sistema*. Uno stato di errore durante l'esecuzione del sistema che può portare a un comportamento del sistema che gli utenti non si aspettano. In questo esempio, il valore della variabile *Tempo_di_trasmissione* viene impostata erroneamente a 24.XX, anziché a 00.XX, quando viene eseguito il codice errato.
4. *Fallimento del sistema*. Un evento che si verifica nel momento in cui il sistema non fornisce il servizio che gli utenti si aspettano. Nell'esempio in esame, non viene trasmesso alcun dato meteorologico perché l'ora non è valida.

I difetti di un sistema non necessariamente si traducono in errori, e gli errori non necessariamente si traducono in fallimenti del sistema:

1. non tutto il codice di un programma viene eseguito. Il codice che include il difetto (per esempio, la mancata inizializzazione di una variabile) non sarà mai eseguito per il modo in cui il software viene utilizzato;
2. errori transitori. Una variabile di stato può avere un valore sbagliato generato dall'esecuzione di un codice errato. Tuttavia, prima che questo valore sia utilizzato e causi un fallimento del sistema, potrebbe essere elaborato qualche altro valore di input che ripristina il valore corretto della variabile di stato. In questo caso, il valore sbagliato non ha alcun effetto pratico;
3. il sistema può includere dei meccanismi di identificazione e protezione dei difetti del software. Questi meccanismi assicurano che venga individuato e corretto un comportamento errato prima che siano influenzati i servizi del sistema.

Un altro motivo per cui i difetti in un sistema potrebbero non essere la causa di fallimenti del sistema è che gli utenti adattano il loro comportamento evitando di utilizzare input che provocano fallimenti dei programmi. Gli utenti esperti evitano quelle funzionalità del software di cui hanno scoperto l'inaffidabilità. Per esempio, io evito qualche funzionalità del mio word processor, come la numerazione automatica, perché ho sperimentato che spesso ottengo dei risultati errati. Riparare i difetti di funzionalità inutilizzate come questa non migliora praticamente l'affidabilità del sistema.

La distinzione tra difetti, errori e fallimenti porta a distinguere tre approcci complementari che vengono utilizzati per migliorare l'affidabilità di un sistema.

1. *Evitare i difetti (fault avoidance)*. Il processo di progettazione e implementazione dovrebbe utilizzare approcci allo sviluppo del software che aiutano ad evitare errori di progettazione e programmazione e così minimizzare la possibilità di introdurre difetti nel sistema. Minori difetti significa minori probabilità di fallimenti durante l'esecuzione del software. Queste tecniche includono l'uso di linguaggi di programmazione che permettono controlli più estesi del compilatore, evitando costrutti di programmazione inclini agli errori, come i puntatori.
2. *Identificazione e correzione dei difetti (fault detection and correction)*. I processi di verifica e convalida servono a scoprire e rimuovere i difetti in un programma, prima che questo sia consegnato agli utenti. I sistemi critici richiedono verifiche e convalide estese per scoprire il maggior numero possibile di difetti prima della consegna del software, e per convincere gli stakeholder e i responsabili della regolamentazione che il sistema è affidabile. Test e debugging sistematici e analisi statica sono esempi di tecniche di identificazione dei difetti.

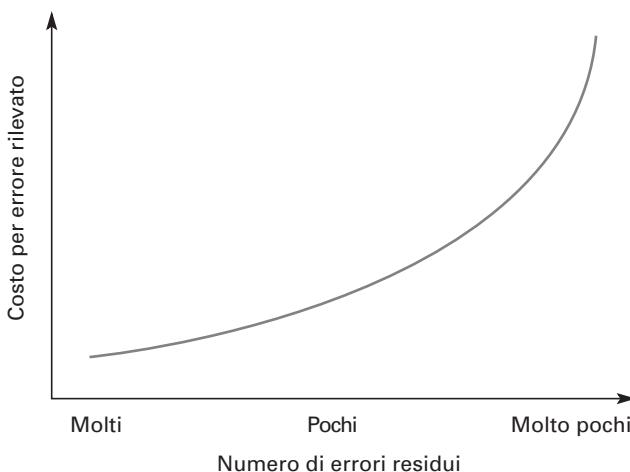


Figura 11.1 Costi per rimuovere i difetti residui di un sistema software.

3. *Tolleranza dei difetti (fault tolerance).* Il sistema è progettato in modo che i difetti o i comportamenti imprevisti siano rilevati durante l'esecuzione del software e siano gestiti in modo tale che non si verifichi un fallimento del sistema. In ogni sistema è possibile includere alcuni semplici approcci alla tolleranza dei difetti che si basano su controlli a runtime. Altre tecniche più avanzate, come l'uso delle architetture fault-tolerant, descritte nel Paragrafo 11.3, possono essere utilizzate quando è richiesto un sistema con un livello molto alto di disponibilità e affidabilità.

L'applicazione delle tecniche di fault avoidance, fault detection e fault tolerance purtroppo non è sempre economicamente vantaggiosa. I costi per trovare e rimuovere i difetti residui in un sistema software salgono esponenzialmente man mano che i difetti vengono identificati e rimossi (Figura 11.1). Al crescere dell'affidabilità del software, occorre sempre più tempo per scoprire un numero sempre più piccolo di difetti. A un certo punto, anche per i sistemi critici, i costi di questi sforzi aggiuntivi non sono più giustificabili.

Di conseguenza, le società di software accettano che il loro software includa qualche difetto residuo. Il livello di difetti dipende dal tipo di sistema. I prodotti software hanno un livello di difetti relativamente alto, mentre i sistemi critici di solito hanno un livello di difetti molto più basso.

Il motivo per cui si accettano i difetti è che, se e quando il sistema fallisce, è più economico pagare le conseguenze del fallimento che scoprire e rimuovere tutti i difetti prima che il sistema sia consegnato. Tuttavia, la decisione di rilasciare un software contenente difetti non è soltanto di natura economica; occorre valutare anche le conseguenze sociali e politiche di un fallimento del sistema.

11.1 Disponibilità e affidabilità

Nel Capitolo 10 ho introdotto i concetti di affidabilità e disponibilità dei sistemi. Se pensiamo ai sistemi come strumenti che forniscono qualche tipo di servizio (per esempio, erogare denaro, stabilire una connessione telefonica o gestire le chiamate telefoniche), allora per disponibilità di un servizio s'intende che il servizio deve essere attivo e in esecuzione, e per affidabilità s'intende che il servizio fornisce risultati corretti. Disponibilità e affidabilità possono essere espresse entrambe come probabilità. Se la disponibilità è 0,999, questo significa che, in un certo periodo di tempo, il sistema è disponibile per il 99,9% del tempo. Se, in media, 2 input su 1000 determinano fallimenti del sistema, allora l'affidabilità, espressa come tasso di fallimenti, è 0,002.

L'affidabilità e la disponibilità possono essere definite più precisamente in questo modo:

1. *affidabilità* – la probabilità che ha un'operazione di essere svolta senza che il sistema fallisca in un determinato periodo di tempo, in un dato ambiente, per uno scopo specifico;
2. *disponibilità* – la probabilità che un sistema, in un certo momento, sarà operativo e in grado di fornire i servizi richiesti.

L'affidabilità di un sistema non è un valore assoluto – dipende dall'ambiente e dal modo in cui il sistema viene utilizzato. Per esempio, supponiamo di misurare l'affidabilità di un'applicazione negli uffici di un'azienda dove la maggior parte degli utenti non sono interessati al funzionamento del software. Essi seguono le istruzioni d'uso del sistema e non tentano di fare esperimenti con il sistema. Se misuriamo l'affidabilità dello stesso sistema in un ambiente universitario, allora l'affidabilità potrebbe essere completamente differente. Qui, gli studenti potrebbero verificare i limiti del sistema e utilizzarlo nei modi più imprevedibili. Questo potrebbe causare dei fallimenti nel sistema che non si verificano nell'ambiente più formale degli uffici aziendali. Pertanto, le percezioni dell'affidabilità di un sistema nei due ambienti sono differenti.

La precedente definizione di affidabilità si basa sul concetto di funzionamento senza fallimenti, dove i fallimenti sono eventi esterni che influiscono sugli utenti di un sistema. Ma che cos'è un “fallimento”? Una definizione tecnica di fallimento è un comportamento che non è conforme alla specifica del sistema. Tuttavia, ci sono due problemi in questa definizione.

1. Le specifiche del software spesso sono incomplete e poco chiare, ed è compito degli ingegneri del software interpretare come il sistema dovrebbe comportarsi; poiché essi non sono esperti dei domini applicativi, potrebbero implementare il comportamento del sistema nel modo che gli utenti non si aspettano. Il software potrebbe comportarsi come specificato, ma per gli utenti non è corretto.

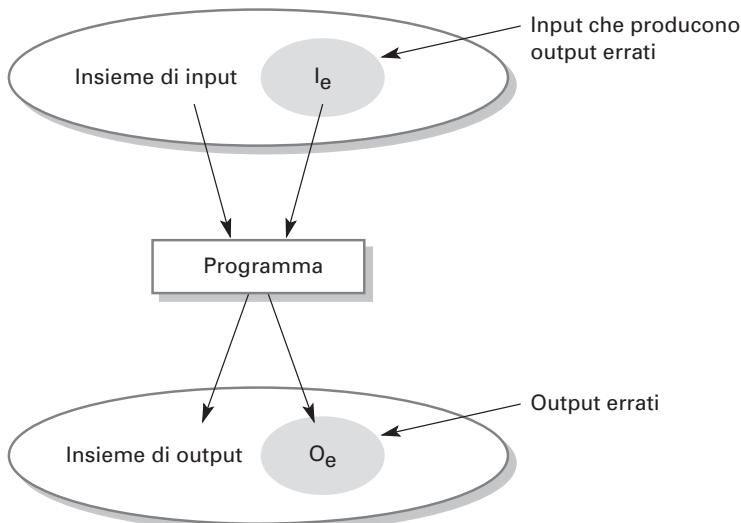


Figura 11.2 Un sistema rappresentato come una mappa di input/output.

2. Nessuno, tranne gli sviluppatori del sistema, legge i documenti della specifica del software. Gli utenti quindi potrebbero aspettarsi un certo comportamento del software, quando la specifica dice qualcosa di completamente differente.

Il fallimento dunque non è qualcosa che può essere definito con obiettività, ma è bensì un giudizio espresso dagli utenti di un sistema. È questo uno dei motivi per i quali non tutti gli utenti hanno la stessa percezione dell'affidabilità del sistema.

Per capire perché l'affidabilità è diversa in ambienti differenti, dobbiamo immaginare il sistema come una mappa di input/output. La Figura 11.2 mostra un sistema software che collega un insieme di input a un insieme di output. Data una sequenza di input, il programma risponde producendo un corrispondente output. Per esempio, dato l'input di un URL, un browser web produce un output che visualizza la pagina web richiesta.

Molti input non provocano fallimenti del sistema. Tuttavia, alcuni input o combinazioni di input, indicati dall'ellisse ombreggiata I_e nella Figura 11.2, provocano fallimenti del sistema o output errati. L'affidabilità del programma dipende dal numero di input che appartengono all'insieme degli input che generano output errati – in altre parole, l'insieme degli input che causano l'esecuzione di un codice errato o che determinano errori del sistema. Se gli input nell'insieme I_e sono eseguiti da parti del sistema utilizzate frequentemente, anche i fallimenti del sistema saranno frequenti. Se, invece, gli input di I_e vengono eseguiti da parti del codice raramente utilizzate, allora gli utenti difficilmente potranno notare i fallimenti del sistema.

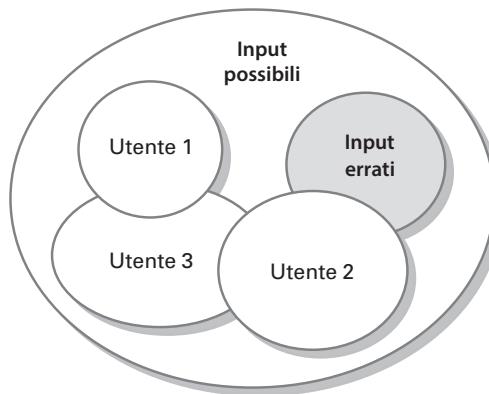


Figura 11.3 Schemi di utilizzo del software.

I difetti che influiscono sull'affidabilità del sistema per un utente potrebbero non manifestarsi mai a un altro utente che utilizza il sistema in modo differente dal primo. Nella Figura 11.3 l'insieme degli input errati corrisponde all'ellisse indicata con I_e nella Figura 11.2. L'insieme degli input prodotti dall'Utente 2 interseca questo insieme di input errati. Gli Utenti 1 e 3, invece, non usano mai input dell'insieme di input errati; per questi utenti il software sarà sempre affidabile.

La disponibilità di un sistema non dipende semplicemente dal numero di fallimenti, ma anche dal tempo richiesto per riparare un difetto che ha provocato un fallimento. Di conseguenza, se il sistema A fallisce una volta all'anno e il sistema B una volta al mese, allora A è apparentemente più affidabile di B. Supponiamo che A impieghi 6 ore per ripartire dopo un fallimento, mentre B richiede solo 5 minuti per ripartire. La disponibilità del sistema B in un anno (60 minuti di fuori servizio) è molto migliore di quella del sistema A (360 minuti di fuori servizio).

I danni provocati da un sistema indisponibile non si riflettono nella semplice metrica di disponibilità che specifica la percentuale del tempo in cui il sistema è disponibile. È importante anche il periodo in cui il sistema fallisce. Se un sistema è indisponibile per un'ora al giorno dalle 3 alle 4 della notte, questo potrebbe non influire su molti utenti. Se, invece, lo stesso sistema è indisponibile per 10 minuti durante il giorno, l'indisponibilità del sistema ha un effetto molto maggiore sugli utenti.

L'affidabilità e la disponibilità sono strettamente correlate, ma a volte una è più importante dell'altra. Se gli utenti si aspettano un servizio continuo da parte del sistema, allora il sistema deve avere una disponibilità molto alta. Deve essere disponibile ogni volta che c'è una richiesta dei suoi servizi da parte di un utente. Tuttavia, se un sistema può ripartire rapidamente dopo un fallimento senza perdere i dati dell'utente, allora i suoi fallimenti potrebbero non influire significativamente sugli utenti.

Un commutatore telefonico che smista le telefonate sulle varie linee è un esempio di sistema in cui la disponibilità è più importante dell'affidabilità. Gli utenti si aspettano di fare una telefonata ogni volta che alzano la cornetta o attivano un'app telefonica, ovvero il sistema deve avere una disponibilità molto alta. Se si verifica un guasto durante la preparazione della connessione telefonica, di solito questo guasto viene rapidamente risolto. Le centraline di commutazione possono ripristinare il sistema e tentare di ristabilire la connessione. Inoltre, se una telefonata viene interrotta, le conseguenze di solito non sono gravi, in quanto gli utenti semplicemente ripetono la telefonata.

11.2 Requisiti di affidabilità

Nel settembre 1993 un aereo atterrò all'aeroporto di Varsavia in Polonia durante un violento temporale. Per 9 secondi dopo l'atterraggio i freni controllati dal sistema software di frenatura non funzionarono. Il sistema di frenatura non aveva rilevato che l'aereo era atterrato e supponeva che era ancora in volo. Una funzione di sicurezza aveva bloccato il sistema di spinta inversa, che serve a rallentare la velocità dell'aereo, in quanto tale spinta è catastrofica quando l'aereo è in aria. L'aereo finì fuori pista, colpì un terrapieno e s'incendiò.

L'inchiesta dimostrò che il software del sistema di frenatura aveva funzionato perfettamente secondo le sue specifiche. Non ci furono errori nel sistema di controllo. La specifica del software era incompleta e non aveva preso in considerazione una situazione rara, che si verifica in questi casi. Il software svolse il suo compito, ma il sistema fallì.

L'incidente dimostra che l'affidabilità di un sistema non dipende semplicemente dalla buona ingegneria del software. Occorre prestare attenzione anche ai dettagli quando vengono definiti i requisiti che determinano la fidatezza del sistema. Questi requisiti di fidatezza sono di due tipi.

1. I *requisiti funzionali* che definiscono le funzioni di controllo e ripristino che devono essere incluse nel sistema e le funzioni che forniscono la protezione contro i fallimenti del sistema e gli attacchi esterni.
2. I *requisiti non funzionali* che definiscono l'affidabilità e la disponibilità del sistema.

Come detto nel Capitolo 10, l'affidabilità complessiva di un sistema dipende dall'affidabilità dell'hardware, del software e degli operatori del sistema. Il software del sistema deve tenere conto di questa esigenza. Oltre a includere i requisiti che compensano i fallimenti del software, ci devono essere anche i requisiti di affidabilità che aiutano a identificare e a risolvere un guasto dell'hardware o un errore dell'operatore.

11.2.1 Metriche di affidabilità

L'affidabilità può essere specificata come la probabilità che si verifichi un fallimento quando un sistema è utilizzato all'interno di determinato ambiente operativo. Per esempio, se accettiamo che una transazione su 1000 possa fallire, allora la probabilità di fallimento è 0,001. Questo non significa che ci sarà esattamente un fallimento ogni 1000 transazioni, ma se osserviamo N migliaia di transazioni, il numero di fallimenti che si verificheranno saranno circa N .

Tre metriche possono essere utilizzate per specificare l'affidabilità e la disponibilità.

1. *Probabilità di fallimento su richiesta* (POFOD, Probability Of Failure On Demand). Questa metrica definisce la probabilità che la richiesta di un servizio provochi il fallimento di un sistema. Quindi, se POFOD = 0,001, c'è una probabilità di 1/1000 che si verifichi un fallimento quando viene effettuata la richiesta di un servizio.
2. *Tasso di occorrenza dei fallimenti* (ROCOF, Rate Of Occurrence Of Failures). Questa metrica definisce il numero di fallimenti del sistema che probabilmente si verificheranno in un certo periodo di tempo (per esempio, in un'ora) o in un certo numero di esecuzioni del sistema. Nell'esempio precedente, ROCOF vale 1/1000. Il reciproco di ROCOF è il *tempo medio al fallimento* (MTTF, Mean Time To Failure), che a volte viene utilizzato come metrica di affidabilità. MTTF è il numero medio di unità di tempo tra due fallimenti del sistema. Un ROCOF di due fallimenti all'ora implica che il tempo medio al fallimento è 30 minuti.
3. *Disponibilità* (AVAIL, Availability). Questa metrica indica la probabilità che un sistema potrà essere operativo quando viene effettuata la richiesta di un servizio del sistema. Per esempio, una disponibilità di 0,9999 significa che, in media, il sistema sarà disponibile per il 99,99% del tempo di funzionamento. La Figura 11.4 mostra vari livelli di disponibilità di un sistema.

La metrica POFOD dovrebbe essere utilizzata nei casi in cui una richiesta possa provocare un grave fallimento del sistema. Questo vale indipendentemente dalla frequenza delle richieste. Per esempio, l'affidabilità di un sistema di protezione che controlla un reattore chimico e arresta la reazione in caso di surriscaldamento dovrebbe essere espressa dalla metrica POFOD. In generale, le richieste in un sistema di protezione sono rare, in quanto il sistema è l'ultima linea di difesa, dopo che tutte le altre funzioni di recovery sono fallite. Pertanto, un POFOD pari a 0,001 (1 fallimento su 1000 richieste) potrebbe sembrare rischioso. Tuttavia, se ci verificano solo due o tre richieste in tutta la vita del sistema, è improbabile che il sistema fallisca.

La metrica ROCOF dovrebbe essere usata quando le richieste dei servizi di un sistema vengono effettuate in modo regolare, anziché in modo intermittente. Per esempio, in un sistema che gestisce un numero notevole di transazioni, possiamo

Disponibilità	Spiegazione
0,9	Il sistema è disponibile per il 90% del tempo. Questo significa che, in un periodo di 24 ore (1440 minuti) il sistema non sarà disponibile per 144 minuti.
0,99	In un periodo di 24 ore, il sistema non sarà disponibile per 14,4 minuti.
0,999	Il sistema non è disponibile per 84 secondi in un periodo di 24 ore.
0,9999	Il sistema non è disponibile per 8,4 secondi in un periodo di 24 ore – circa un minuto alla settimana.

Figura 11.4 Specifica della disponibilità.

specificare un valore di ROCOF pari a 10 fallimenti al giorno. Questo significa che accettiamo che, in media, 10 transazioni al giorno possano fallire e, quindi, dovranno essere cancellate e ripetute. In alternativa, potremmo specificare ROCOF come il numero di fallimenti per 1000 transazioni.

Se è importante il tempo assoluto tra i fallimenti, possiamo specificare l'affidabilità come tempo medio al fallimento (MTTF). Per esempio, se stiamo specificando l'affidabilità per un sistema con lunghe transazioni (come nella progettazione assistita dall'elaboratore), è opportuno utilizzare questa metrica. Il valore di MTTF dovrebbe essere più grande del tempo medio che un utente lascia trascorrere senza salvare i risultati del suo lavoro. Questo significa che è molto improbabile che gli utenti perdano il loro lavoro per un fallimento del sistema durante una qualsiasi sessione di lavoro.

11.2.2 Requisiti di affidabilità non funzionali

I requisiti di affidabilità non funzionali sono specifiche della disponibilità e dell'affidabilità di un sistema espresse mediante una delle metriche di affidabilità (POFOD, ROCOF e AVAIL) descritte nel precedente paragrafo. Le specifiche quantitative della disponibilità e dell'affidabilità sono state utilizzate per molti anni nei sistemi a sicurezza critica, ma sono poco comuni nei sistemi critici aziendali. Tuttavia, poiché sempre più aziende richiedono dai loro sistemi un servizio di 24 ore/giorno per 7 giorni/settimana, ha senso che queste aziende siano più precise sulle loro aspettative di disponibilità e affidabilità.

La specifica quantitativa dell'affidabilità è utile per vari motivi.

1. Il processo per decidere il livello di affidabilità richiesto aiuta gli stakeholder a capire meglio di cosa hanno effettivamente bisogno. Gli stakeholder capiscono che ci sono vari tipi di fallimenti del sistema e che è costoso raggiungere alti livelli di affidabilità.
2. Consente di stabilire quando interrompere i test di un sistema. L'interruzione può avvenire quando il sistema ha raggiunto il livello di affidabilità richiesto.

Superspecifica dell'affidabilità

Superspecifica dell'affidabilità significa definire un livello di affidabilità che è maggiore di quello strettamente necessario per il funzionamento del software. La superspecifica dell'affidabilità aumenta i costi di sviluppo in modo spropositato. La ragione di questo è che i costi per ridurre i difetti del software e per verificare l'affidabilità aumentano esponenzialmente al crescere dell'affidabilità.

<http://software-engineering-book.com/web/over-specifying-reliability/>

3. Permette di definire le strategie di progettazione per migliorare l'affidabilità di un sistema. È possibile valutare in quale maniera ciascuna strategia può consentire di raggiungere il livello di affidabilità richiesto.
4. Se un ente di regolamentazione deve approvare un sistema prima che venga messo in servizio (per esempio, tutti i sistemi critici per la sicurezza di volo degli aerei sono regolamentati), è importante fornire la prova che è stato raggiunto il livello di affidabilità richiesto per ottenere la certificazione.

Per evitare di incorrere in costi eccessivi e non necessari, è importante specificare l'affidabilità effettivamente richiesta, anziché scegliere semplicemente un livello di affidabilità molto alto per l'intero sistema. Potreste avere requisiti differenti per parti diverse del sistema, se alcune parti sono più critiche di altre. Per specificare i requisiti di affidabilità è bene seguire queste tre linee guida.

1. Specificare i requisiti di affidabilità e disponibilità per vari tipi di fallimenti del sistema. Ci dovrebbero essere meno probabilità che si verifichino fallimenti con alti costi che fallimenti che non causano danni gravi.
2. Specificare i requisiti di affidabilità e disponibilità per vari tipi di servizi del sistema. I servizi critici dovrebbero avere la più alta affidabilità, ma si potrebbero tollerare più fallimenti nei servizi meno critici. Si potrebbe decidere che conviene utilizzare un'unica specifica quantitativa dell'affidabilità per la maggior parte dei servizi critici.
3. Valutare se è davvero necessario un alto livello di affidabilità. Per esempio, potreste creare dei meccanismi di identificazione degli errori per controllare gli output di un sistema e avere dei processi di correzione automatica degli errori. Così facendo, non sarebbe più necessario un alto livello di affidabilità per un sistema che genera gli output i cui errori possono essere scoperti e corretti.

Per illustrare queste linee guida, consideriamo i requisiti di affidabilità e disponibilità per una rete di sportelli automatici ATM di una banca, che eroga soldi in contanti e offre altri servizi. La banca ha due problemi con questo sistema.

1. Assicurare che il sistema offra ai clienti i servizi richiesti e che siano registrate correttamente le transazioni dei clienti in un apposito database.
2. Garantire che ogni sportello ATM sia disponibile quando richiesto.

La banca ha molti anni di esperienza nell'identificare e correggere gli errori nelle transazioni, utilizzando dei metodi contabili per rilevare quando si verifica un errore. Molte transazioni che falliscono possono essere semplicemente cancellate, senza perdite per la banca e disagi per il cliente. La banca che partecipa a una rete di sportelli ATM accetta che i fallimenti di questi sportelli possano significare che un piccolo numero di transazioni sono sbagliate, tuttavia ritiene che sia più conveniente correggere questi errori dopo che si sono verificati, anziché sostenere costi più elevati per evitare preventivamente tali errori. L'affidabilità assoluta richiesta da un sistema ATM è dunque relativamente bassa, in quanto sono accettabili molti fallimenti al giorno.

Per la banca (e per i suoi clienti), la disponibilità di una rete ATM è più importante rispetto al fatto che alcune transazioni possano fallire. La mancanza di disponibilità significa un aumento di richieste di servizi negli sportelli tradizionali, insoddisfazione dei clienti, maggiori costi per riparare la rete e così via. Di conseguenza, per i sistemi basati sulle transazioni, quali i sistemi bancari e di commercio elettronico, la specifica dell'affidabilità di solito è incentrata sulla disponibilità del sistema.

Per specificare la disponibilità di una rete ATM, occorre identificare i servizi del sistema e definire la disponibilità richiesta per ciascuno dei seguenti servizi:

- il servizio del database dei conti dei clienti;
- i singoli servizi forniti dalla rete ATM, come “prelievo contante” e “informazioni sul conto”.

Il servizio del database è il più critico, in quanto un suo fallimento significa che tutti gli sportelli della rete ATM sono disattivati. Quindi, occorre specificare questo servizio in modo da avere un livello elevato di disponibilità. Nell'esempio in esame, un valore accettabile per la disponibilità del database (ignorando problemi quali i periodi di manutenzione e aggiornamento) potrebbe essere 0,9999 circa, tra le ore 7:00 e le 23:00. Questo significa un'indisponibilità inferiore a un minuto alla settimana.

Per un singolo sportello ATM, la disponibilità globale dipende dall'affidabilità meccanica e dal fatto che possa finire il contante. I problemi del software probabilmente sono meno significativi di questi. Pertanto, è accettabile un livello inferiore di disponibilità per il software di uno sportello ATM. La disponibilità globale del software della rete ATM potrebbe essere quindi impostata a 0,999; questo significa che una macchina potrebbe essere indisponibile per 1 o 2 minuti al giorno. Questo consente al software ATM di riavviarsi nel caso si verifichi un problema.

L'affidabilità dei sistemi di controllo di solito è specificata come probabilità che il sistema possa fallire quando viene effettuata una richiesta (POFOD). Consideriamo i requisiti di affidabilità per il software di controllo della pompa di insulina (descritta nel Capitolo 1). Questo sistema rilascia insulina un certo numero di volte al giorno e controlla il livello di glucosio nel sangue del paziente parecchie volte al giorno.

Ci sono due tipi possibili di fallimenti nella pompa di insulina.

1. *Fallimenti transitori del software.* Possono essere corretti dall'azione dell'utente, per esempio ripristinare o ricalibrare la macchina. Per questo tipo di fallimenti, potrebbe essere accettabile un valore di POFOD relativamente basso, per esempio 0,002. Ciò significa che potrebbe verificarsi un fallimento ogni 500 richieste fatte dalla macchina, che corrisponde approssimativamente a una volta ogni 3,5 giorni, in quanto il livello di glucosio nel sangue viene controllato circa 5 volte ogni ora.
2. *Fallimenti permanenti del software.* Richiedono che il software venga reinstallato dal produttore. La probabilità che si verifichino questi fallimenti deve essere molto bassa. Il valore minimo potrebbe essere una volta all'anno circa, ovvero il valore di POFOD non dovrebbe essere maggiore di 0,000002.

La mancata somministrazione di insulina non ha implicazioni immediate sulla sicurezza, quindi sono i fattori commerciali, anziché quelli di sicurezza, che governano il livello di affidabilità richiesta. I costi dei servizi sono alti perché gli utenti richiedono una rapida riparazione o sostituzione. È interesse del produttore limitare il numero di fallimenti permanenti che richiedono riparazioni.

11.2.3 Requisiti di affidabilità funzionali

Per raggiungere un alto livello di affidabilità e disponibilità in un sistema che fa uso intensivo del software, si usa una combinazione di tecniche fault-avoidance, fault-detection e fault-tolerance. Questo significa che bisogna definire i requisiti di affidabilità funzionali per specificare come il sistema dovrà essere in grado di evitare, identificare e tollerare i difetti del software.

Questi requisiti di affidabilità funzionali dovrebbero specificare i difetti da identificare e le azioni da svolgere per garantire che tali difetti non provochino fallimenti del sistema. La specifica dell'affidabilità richiede, quindi, l'analisi dei requisiti non funzionali (se questi sono stati specificati), la valutazione dei rischi per l'affidabilità e la specifica delle funzioni che evitano questi rischi.

Ci sono quattro tipi di requisiti di affidabilità funzionali.

1. *Requisiti di controllo.* Specificano i controlli sugli input del sistema per garantire che gli input errati siano identificati prima che siano elaborati dal sistema.
2. *Requisiti di ripristino.* Definiscono le azioni che dovrà svolgere il sistema dopo che si è verificato un guasto. Questi requisiti di solito riguardano la gestione di copie del sistema e dei suoi dati e specificano come ripristinare i servizi dopo un fallimento del sistema.
3. *Requisiti di ridondanza.* Specificano le funzioni ridondanti del sistema che garantiscono che il fallimento di un singolo componente non provochi la perdita completa dei servizi. Questo argomento sarà descritto più dettagliatamente nel prossimo capitolo.

RR1	Dovrebbe essere stabilito un intervallo predefinito per tutti gli input dell'operatore, e il sistema dovrebbe verificare che tutti questi input ricadano nell'intervallo predefinito (controllo).
RR2	Dovrebbero essere conservate due copie del database dei pazienti su due server che non devono trovarsi nello stesso edificio (ripristino, ridondanza).
RR3	Per implementare il sistema di controllo della frenatura dovrebbe essere utilizzata una programmazione a N versioni (ridondanza).
RR4	Il sistema deve essere implementato in un sottoinsieme sicuro di Ada e controllato mediante l'analisi statica (processo).

Figura 11.5 Esempi di requisiti di affidabilità funzionali.

4. *Requisiti del processo.* Sono i requisiti di fault-avoidance; assicurano che siano sempre adottate delle buone tecniche nel processo di sviluppo del software. Le tecniche specificate dovrebbero ridurre il numero di difetti del sistema.

Alcuni esempi di requisiti di affidabilità sono illustrati nella Figura 11.5.

Non ci sono regole semplici per definire i requisiti di affidabilità funzionali. Le società che sviluppano sistemi critici di solito hanno le conoscenze appropriate sui possibili requisiti di affidabilità e su come questi requisiti riflettono la reale affidabilità di un sistema. Queste società sono specializzate in particolari tipi di sistemi, per esempio i sistemi di controllo del traffico ferroviario, in modo che i requisiti di affidabilità possano essere riutilizzati in più sistemi.

11.3 Architetture fault-tolerant

Il termine fault-tolerant indica un approccio alla fidatezza in cui i sistemi includono appositi meccanismi per continuare le operazioni, anche dopo che si è verificato un errore del software o un guasto dell'hardware, e il sistema si trova in uno stato di errore. I meccanismi fault-tolerant identificano e correggono questo stato di errore del sistema, in modo tale che, quando si verifica un guasto, non si abbia un fallimento del sistema. Questi meccanismi sono richiesti nei sistemi a sicurezza o a protezione critica, e quando il sistema non può passare a uno stato sicuro dopo che è stato rilevato un errore.

Affinché un sistema sia fault-tolerant, la sua architettura deve essere progettata per includere hardware e software ridondanti e diversi. Esempi di sistemi che possono richiedere un'architettura fault-tolerant sono i sistemi degli aerei che devono essere disponibili per tutta la durata del volo, i sistemi di telecomunicazioni e i sistemi critici di controllo e comando.

La più semplice realizzazione di un'architettura fidata si trova nei server replicati, dove due o più server possono svolgere lo stesso compito. Le richieste di

elaborazione vengono incanalate tramite un componente di gestione dei server, che indirizza ciascuna richiesta a un particolare server. Questo componente tiene traccia delle risposte dei server. Nel caso di guasto del server, che può essere rilevato dalla mancata risposta, il server guasto viene escluso dal sistema. Le richieste non elaborate vengono indirizzate ad altri server.

I server replicati sono largamente utilizzati nei sistemi di elaborazione delle transazioni, dove è facile conservare le copie delle transazioni da elaborare. I sistemi di elaborazione delle transazioni sono progettati in modo che i dati siano aggiornati solo quando una transazione è stata completata correttamente. I ritardi nell'elaborazione non influiscono sull'integrità del sistema. Una tecnica efficiente potrebbe essere quella di utilizzare l'hardware se il server di backup è quello che normalmente viene utilizzato per i compiti di bassa priorità. Se si verifica un problema con il server primario, le sue transazioni non elaborate vengono trasferite al server di backup, che le elabora con la massima priorità.

I server replicati forniscono la ridondanza, ma non sempre la diversità. L'hardware dei server di solito è sempre lo stesso, e i server eseguono la stessa versione del software. Quindi, è possibile gestire i guasti dell'hardware e gli errori del software che sono localizzati in una singola macchina. Non è possibile risolvere i problemi di progettazione del software che provocano il fallimento di tutte le versioni del software contemporaneamente. Per gestire i fallimenti dovuti alla progettazione del software, un sistema deve utilizzare software e hardware diversi.

Torres-Pomales ha studiato numerose tecniche fault-tolerant (Torres-Pomales 2000); Pullum (Pullum 2001) ha descritto vari tipi di architetture fault-tolerant. Nei prossimi paragrafi, presenterò tre schemi architetturali che sono stati utilizzati nei sistemi fault-tolerant.

11.3.1 Sistemi di protezione

Un sistema di protezione è un sistema specializzato che è associato a qualche altro sistema che, di solito è il sistema di controllo di qualche processo, come un processo chimico, o il sistema di controllo di un'apparecchiatura, come quello installato nei treni senza macchinista. Un esempio di sistema di protezione potrebbe essere quello che controlla se un treno si sta avvicinando a un semaforo rosso. Se non c'è alcuna indicazione che il sistema di controllo sta rallentando il treno, allora interviene automaticamente il sistema di protezione per attivare i freni e fermare il treno. I sistemi di protezione monitorano il loro ambiente in maniera autonoma. Se i sensori segnalano un problema che il sistema di controllo non sta gestendo, allora il sistema di protezione viene attivato per arrestare il processo o l'apparecchiatura.

La Figura 11.6 illustra la relazione tra un sistema di protezione e un sistema controllato. Il sistema di protezione monitora sia l'apparecchiatura controllata sia l'ambiente. Se viene rilevato un problema, invia dei comandi agli attuatori per

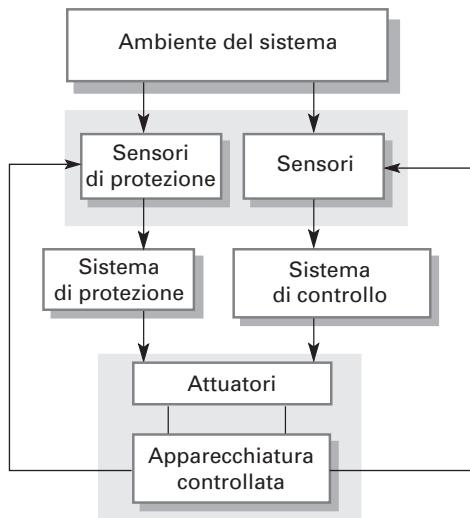


Figura 11.6 Architettura di un sistema di protezione.

arrestare il sistema o attiva altri meccanismi di protezione, quali l'apertura di una valvola di sfogo. Notate che ci sono altri due gruppi di sensori. Un gruppo serve per monitorare il sistema, l'altro per proteggerlo. Nel caso di fallimenti del sensore, i meccanismi di backup entrano in azione per consentire al sistema di protezione di continuare le operazioni. Il sistema potrebbe avere anche degli attuatori ridondanti.

Un sistema di protezione include soltanto le funzionalità critiche che sono richieste per commutare il sistema da uno stato di errore a uno stato sicuro (che potrebbe essere l'arresto del sistema). È un esempio di un'architettura fault-tolerant più generale nella quale il sistema principale è supportato da un sistema di backup più piccolo e semplice, che include soltanto le funzionalità essenziali. Per esempio, il software di controllo della navicella spaziale Shuttle aveva un sistema di backup con la funzionalità “portami a casa”. Ovvero, il sistema di backup era in grado di fare atterrare lo Shuttle in caso di guasto del sistema di controllo principale, ma non aveva altre funzionalità.

Il vantaggio di questo stile architettonico è che il software del sistema di protezione può essere molto più semplice del software che controlla il processo protetto. L'unica funzionalità del sistema di protezione è monitorare le operazioni e garantire che il sistema sia messo in sicurezza in caso di emergenza. Quindi, è possibile investire più risorse per evitare e identificare i guasti. È possibile controllare che la specifica del software sia corretta e coerente e che il software soddisfi tale specifica. L'obiettivo è garantire che l'affidabilità del sistema di protezione sia tale da garantire che il sistema abbia una probabilità di fallimento molto bassa (per esempio, 0,001). Dato che le richieste di intervento del sistema di

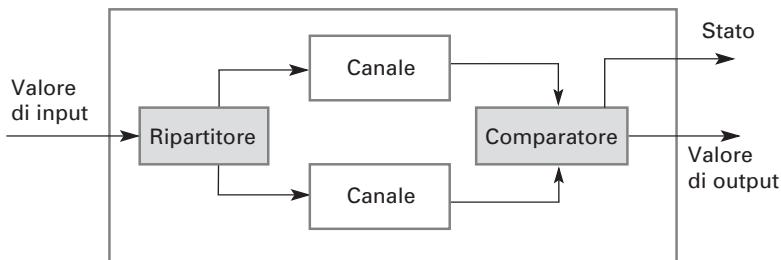


Figura 11.7 Architettura auto-monitorata.

protezione dovrebbero essere rare, una probabilità di fallimento su richiesta di 1/1000 significa che i fallimenti del sistema di protezione dovrebbero essere molto rari.

11.3.2 Architetture auto-monitorate

Un'architettura auto-monitorata (Figura 11.7) è un'architettura nella quale il sistema è progettato per monitorare le sue operazioni e svolgere qualche azione se rileva qualche problema. I calcoli sono svolti su canali separati, e i risultati di questi calcoli vengono confrontati. Se i risultati sono identici e disponibili nello stesso istante, allora si ritiene che il sistema stia operando correttamente. Se i risultati sono differenti, allora si presume che ci sia un guasto. Quando si verifica ciò, il sistema solleva un'eccezione di guasto sulla linea di stato, che segnala che il controllo deve essere trasferito a qualche altro sistema.

Per essere efficienti nel rilevare gli errori del software e i guasti dell'hardware, i sistemi auto-monitorati devono essere progettati seguendo queste linee guida.

1. L'hardware utilizzato in ciascun canale deve essere diverso. In pratica, questo potrebbe significare che ciascun canale deve utilizzare un tipo di processore differente per svolgere i calcoli richiesti, o che il chipset del sistema deve essere realizzato da costruttori differenti. In questo modo, si riduce la probabilità che si verifichino gli stessi fallimenti dovuti alla progettazione dei processori.
2. Il software utilizzato in ciascun canale deve essere diverso; altrimenti, lo stesso errore del software potrebbe verificarsi contemporaneamente nei due canali.

Questa architettura può essere utilizzata nei casi in cui sia importante che i calcoli siano corretti, mentre non è essenziale la disponibilità. Se le risposte dei due canali sono diverse, il sistema viene fermato. Per molti sistemi di diagnostica e cure mediche, l'affidabilità è più importante della disponibilità, in quanto una risposta errata del sistema potrebbe comportare una cura sbagliata per il paziente. Tuttavia, se il sistema viene fermato in caso di errore, questo è un inconveniente che non dovrebbe preoccupare il paziente.

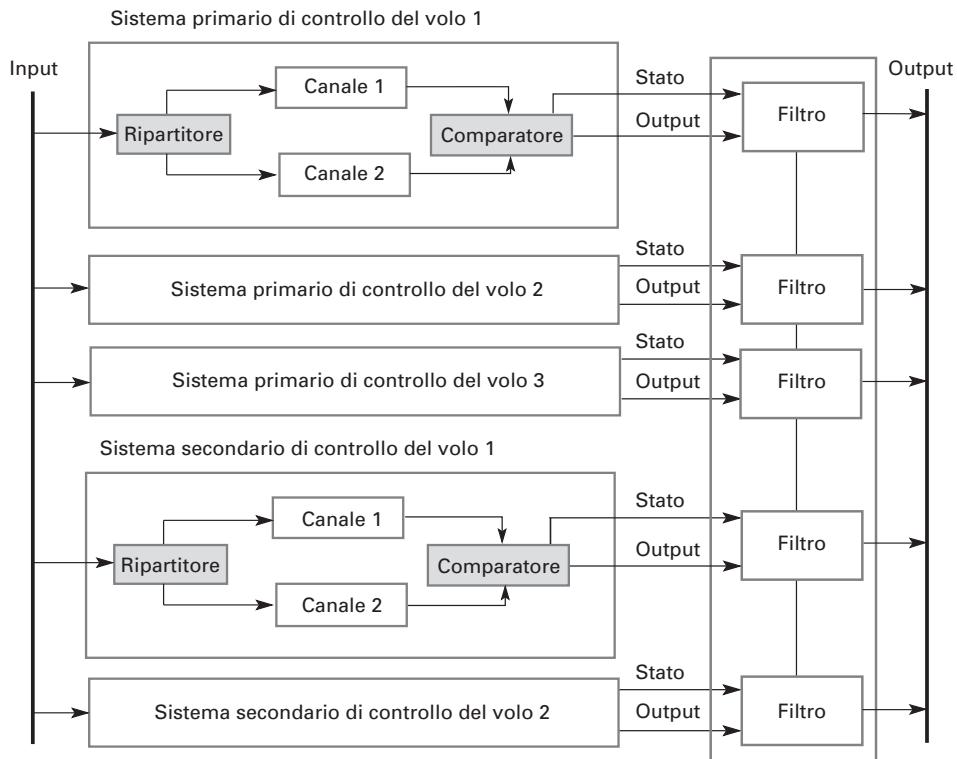


Figura 11.8 Architettura del sistema di controllo del volo negli Airbus 340.

Nei casi in cui sia richiesto un livello elevato di disponibilità, bisogna utilizzare più sistemi auto-monitorati in parallelo. Occorre un'unità di commutazione che rileva gli errori e seleziona i risultati da uno dei sistemi, nel quale entrambi i canali forniscono una risposta coerente. Questo approccio è utilizzato nel sistema di controllo del volo negli aerei della serie Airbus 340, che usa cinque computer auto-monitorati. La Figura 11.8 è un diagramma semplificato di questo sistema e mostra l'organizzazione di un sistema auto-monitorato.

Nel sistema di controllo del volo degli Airbus, ciascun computer di controllo svolge i calcoli in parallelo, utilizzando gli stessi input. Gli output sono collegati ai filtri hardware che rilevano se lo stato indica un errore e, in questo caso, l'output di quel computer viene bloccato. L'output viene prelevato da un altro sistema. Quindi, è possibile che quattro computer falliscano e che l'aereo continui a funzionare. In oltre 15 anni di servizio, non sono stati segnalati casi in cui il controllo di questi aerei sia stato perso a causa del fallimento completo del sistema di controllo del volo.

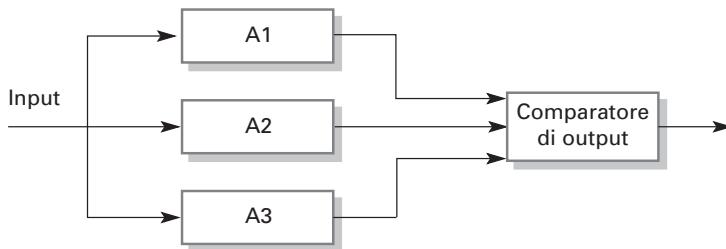


Figura 11.9 Ridondanza modulare triplice (TMR).

I progettisti del sistema di controllo degli Airbus hanno tentato di realizzare la diversità in vari modi.

1. I computer dei controlli primari del volo usano un processore differente dai sistemi secondari di controllo del volo.
2. Il chipset che viene utilizzato in ogni canale dei sistemi primari e secondari è fornito da un costruttore diverso.
3. Il software nei sistemi secondari di controllo del volo fornisce soltanto una funzionalità critica – è meno complesso del software primario.
4. Il software per ciascun canale nei sistemi primari e secondari è sviluppato utilizzando linguaggi di programmazione e team di sviluppo differenti.
5. Linguaggi di programmazione differenti sono utilizzati nei sistemi primari e secondari.

Come detto nel Paragrafo 11.3.4, tutto questo non garantisce la diversità, ma riduce la probabilità che si verifichino gli stessi guasti in canali differenti.

11.3.3 Programmazione a N versioni

Le architetture auto-monitorate sono esempi di sistemi in cui viene utilizzata la programmazione multiversione per fornire la ridondanza e la diversità del software. Questo concetto di programmazione multiversione è stato derivato dai sistemi hardware, dove per molti anni è stato applicato il concetto di ridondanza modulare triplice (TMR, Triple Modular Redundancy) per costruire sistemi che sono tolleranti ai guasti dell'hardware (Figura 11.9).

In un sistema TMR, l'unità hardware è replicata tre volte (o anche più). L'output di ciascuna unità viene passato a un comparatore di output, che di solito è implementato come un sistema di votazioni. Questo sistema confronta tutti i suoi input; se due o più input sono uguali, allora il loro valore diventa output. Se una di queste unità fallisce e non produce lo stesso output delle altre unità, il suo output viene ignorato. Un gestore di errori può tentare di riparare automaticamente l'unità in errore, ma, se ciò è impossibile, il sistema viene automaticamente riconfigurato per mettere fuori servizio tale unità. Il sistema continua a operare con due unità funzionanti.

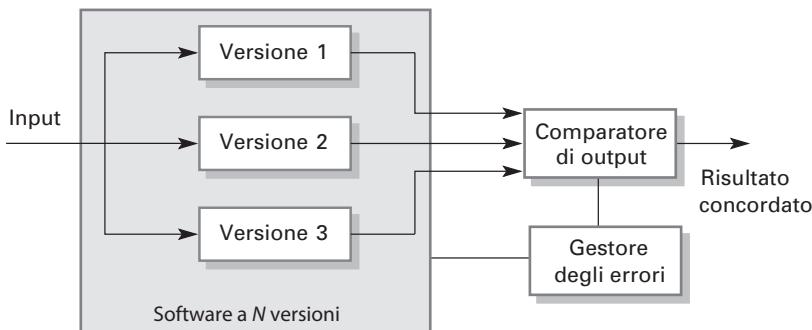


Figura 11.10 Programmazione a N versioni.

Questo approccio alla tolleranza agli errori si basa sul fatto che la maggior parte dei fallimenti delle unità hardware sono il risultato di fallimenti dei componenti, anziché di errori di progettazione. È probabile quindi che un componente fallisca in maniera indipendente da un altro componente. Questo presuppone che tutte le unità hardware, quando sono completamente operative, funzionino secondo le loro specifiche. Quindi, la probabilità che i componenti di tutte le unità hardware falliscano contemporaneamente è bassa.

Ovviamente, tutti i componenti potrebbero avere lo stesso errore di progettazione e, quindi, fornire la stessa risposta (errata). L'uso di unità hardware che hanno una specifica comune, ma che sono progettate e costruite da diversi produttori, riduce le probabilità di tale errore comune. Si suppone che la probabilità che team differenti commettano lo stesso errore di progettazione o di produzione sia bassa.

Un approccio simile può essere applicato al software fault-tolerant, dove N diverse versioni di un sistema software vengono eseguite in parallelo (Avizienis 1995). Questo approccio alla tolleranza agli errori del software, illustrato nella Figura 11.10, è stato utilizzato nei sistemi di segnalazione ferroviaria, nei sistemi a bordo degli aerei e nei sistemi di protezione dei reattori.

Adottando una specifica comune, lo stesso sistema software viene implementato da un certo numero di team. Queste versioni vengono eseguite su computer separati. I loro output vengono confrontati utilizzando un sistema di votazioni; gli output non coerenti o che non sono prodotti in tempo vengono scartati. Devono essere disponibili almeno tre versioni del sistema, in modo che due versioni possano essere coerenti nel caso si verifichi un singolo errore.

La programmazione a N versioni potrebbe essere meno costosa delle architetture auto-monitorate nei sistemi in cui è richiesto un alto livello di disponibilità. Tuttavia, essa richiede diversi team per sviluppare versioni differenti del software. Questo implica costi di sviluppo del software molto alti. Di conseguenza, questo approccio è utilizzato soltanto nei sistemi dove è impraticabile fornire un sistema di protezione in grado di contrastare i fallimenti della sicurezza critica.

11.3.4 Diversità del software

Tutte le precedenti architetture fault-tolerant si basano sulla diversità del software per realizzare la tolleranza agli errori. Ciò si basa sull'ipotesi che implementazioni differenti della stessa specifica (o di una parte della specifica per i sistemi di protezione) siano indipendenti. Queste implementazioni non dovrebbero includere errori comuni e non dovrebbero fallire nello stesso modo, contemporaneamente. Per questo, il software dovrebbe essere scritto da più team, che non dovrebbero comunicare tra loro durante il processo di sviluppo. Così facendo, si riduce la probabilità che si verifichino le stesse incomprensioni o cattive interpretazioni della specifica.

L'azienda che si sta procacciando il sistema potrebbe includere esplicite politiche di diversità che hanno lo scopo di massimizzare le differenze tra le versioni del sistema. Per esempio:

1. includendo il requisito che devono essere utilizzati metodi di progettazione differenti. Per esempio, un team potrebbe produrre un progetto orientato agli oggetti e un altro team potrebbe produrre un progetto orientato alle funzioni;
2. richiedendo che i programmi debbano essere implementati utilizzando linguaggi di programmazione differenti. Per esempio, in un sistema a 3 versioni, si potrebbero utilizzare i linguaggi Ada, C++ e Java per scrivere le versioni del software;
3. richiedendo l'utilizzo di strumenti e ambienti differenti per lo sviluppo del sistema;
4. richiedendo algoritmi differenti da utilizzare in alcune parti dell'implementazione. Tuttavia, questo limita la libertà del team di progettazione e potrebbe essere difficile da conciliare con i requisiti di performance del sistema.

Teoricamente, le diverse versioni del sistema non dovrebbero avere interdipendenze e, quindi, dovrebbero fallire in modi completamente differenti. Se questo è vero, allora l'affidabilità complessiva di un sistema diversificato si ottiene moltiplicando l'affidabilità di ciascun canale. Così, se ciascun canale ha una probabilità di fallimento su richiesta pari a 0,001, allora il POFOD totale di un sistema a 3 canali (con tutti i canali indipendenti) è un milione di volte più grande dell'affidabilità di un sistema a un solo canale.

In pratica, però, è impossibile ottenere un'indipendenza completa dei canali. È stato dimostrato sperimentalmente che team indipendenti di progettazione del software spesso commettono gli stessi errori o interpretano in modo sbagliato le stesse parti di una specifica (Brilliant, Knight e Leveson 1990; Leveson 1995). Le ragioni di queste cattive interpretazioni sono varie.

1. I membri di team differenti spesso hanno la stessa formazione culturale, avendo studiato sugli stessi libri di testo ed essendo stati addestrati a utilizzare gli stessi approcci. Questo significa che essi potrebbero incontrare le

stesse difficoltà nel capire una specifica e nel comunicare con gli esperti dei domini. È molto probabile che essi, in modo autonomo, commettano gli stessi errori e progettino gli stessi algoritmi per risolvere un problema.

2. Se i requisiti sono sbagliati o si basano su cattive interpretazioni dell'ambiente del sistema, allora questi errori si rifletteranno in ciascuna implementazione del sistema.
3. In un sistema critico, la specifica dettagliata del sistema che viene derivata dai requisiti del sistema dovrebbe fornire una definizione chiara del comportamento del sistema. Se, invece, la specifica è ambigua, allora i membri di team differenti possono commettere gli stessi errori nell'interpretare la specifica.

Un modo per ridurre la possibilità di commettere gli stessi errori consiste nello sviluppare più specifiche dettagliate per il sistema e nel definire le specifiche con linguaggi differenti. Un team di sviluppo potrebbe lavorare partendo da una specifica formale, un altro team da un modello del sistema basato sugli stati, e un terzo team da una specifica scritta nel linguaggio naturale. Questo approccio aiuta a evitare alcuni errori di interpretazione della specifica, ma non risolve il problema degli errori dei requisiti. Introduce inoltre la possibilità di commettere errori di traduzione dei requisiti, generando specifiche non coerenti.

In un'analisi degli esperimenti, Hatton (Hatton 1997) ha concluso che un sistema a 3 canali era da 5 a 9 volte più affidabile di un sistema a un canale. Secondo Hatton, i miglioramenti dell'affidabilità che potrebbero essere ottenuti dedicando più risorse a una singola versione del sistema, in generale, non potranno raggiungere i livelli di affidabilità di un sistema a N versioni.

Ciò che non è chiaro, tuttavia, è se i miglioramenti dell'affidabilità di un sistema multiversione giustificano i costi extra di sviluppo. Per molti sistemi, i costi extra non sono giustificabili, in quanto potrebbe essere sufficiente un sistema a singola versione ben progettato. È soltanto nei sistemi critici o a sicurezza critica, dove i costi dei fallimenti sono molto elevati, che potrebbe essere necessario il software multiversione. Tuttavia, anche in questi casi (per esempio, nel sistema di una navicella spaziale) potrebbe essere sufficiente utilizzare un semplice sistema di backup con funzionalità limitate, in attesa che il sistema principale venga riparato e riavviato.

11.4 Programmare per l'affidabilità

In questo libro ho messo deliberatamente in risalto gli aspetti dell'ingegneria del software che sono indipendenti dai linguaggi di programmazione. È quasi impossibile discutere di programmazione senza scendere nei dettagli di uno specifico linguaggio di programmazione. Tuttavia, quando si considera l'ingegneria dell'affidabilità, ci sono alcune buone pratiche di programmazione che sono quasi universali e che aiutano a ridurre il numero di guasti dei sistemi.

Linee guida per una programmazione fidata

1. Limitare la visibilità delle informazioni in un programma.
2. Controllare la validità di tutti gli input.
3. Fornire un gestore per tutte le eccezioni.
4. Minimizzare l'uso di costrutti inclini agli errori.
5. Creare funzioni di riavviamento.
6. Controllare i limiti degli array.
7. Includere un timeout quando si chiama un componente esterno.
8. Assegnare un nome a tutte le costanti che rappresentano entità del mondo reale.

Figura 11.11 Linee guida per una programmazione fidata.

La Figura 11.11 mostra una lista di otto linee guida per queste buone pratiche di programmazione, che possono essere applicate indipendentemente dal particolare linguaggio di programmazione utilizzato per sviluppare un sistema, sebbene il modo in cui sono utilizzati dipende dallo specifico linguaggio di programmazione e dalle notazioni che sono state utilizzate per sviluppare il sistema. Seguendo queste linee guida, si riduce anche la possibilità di introdurre punti di vulnerabilità della sicurezza dei programmi.

Linea guida 1

Controllare la visibilità delle informazioni in un programma

Un principio di sicurezza che è adottato dalle organizzazioni militari è il principio del “devono conoscere”. Un’informazione viene fornita soltanto agli individui che devono conoscere tale informazione per poter svolgere il loro compito. Le informazioni che non sono rilevanti ai loro compiti vengono nascoste.

Quando si programma, bisogna adottare un principio analogo per controllare l’accesso alle variabili e alle strutture di dati che vengono utilizzate. I componenti di un programma dovrebbero accedere soltanto a quei dati che sono necessari per la loro implementazione. Gli altri dati dovrebbero essere inaccessibili e nascosti a tali componenti. Se un’informazione è nascosta, non sarà danneggiata da un componente del programma che non la potrà utilizzare. Se l’interfaccia resta la stessa, la rappresentazione dei dati potrà essere modificata senza influire sugli altri componenti del sistema.

Questo può essere ottenuto implementando le strutture di dati come tipi di dati astratti. Un tipo di dati astratto è quello in cui la struttura interna e la rappresentazione di una variabile di quel tipo sono nascoste. La struttura e gli attributi del tipo non sono visibili esternamente, e ogni accesso ai dati avviene tramite un’operazione.

Per esempio, supponiamo di avere un tipo di dati che rappresenta una coda di richieste di servizi. Le operazioni dovrebbero includere i comandi *get* e *put*, che aggiungono e tolgono gli elementi nella coda, e un comando che restituisce il

numero di elementi nella coda. Inizialmente, si potrebbe implementare la coda come un array, ma successivamente si potrebbe cambiare l'implementazione in una lista collegata. Questa lista è accessibile senza modificare il codice che implementa la coda, in quanto la rappresentazione della coda non è mai accessibile in modo diretto.

In alcuni linguaggi orientati agli oggetti, è possibile implementare i tipi di dati astratti utilizzando le definizioni dell'interfaccia, dove si dichiara l'interfaccia con un oggetto senza fare riferimento alla sua implementazione. Per esempio, supponiamo di definire l'interfaccia *Coda*, che supporta i metodi per inserire gli oggetti nella coda, eliminare gli oggetti dalla coda e conoscere la dimensione della coda. Nella classe degli oggetti che implementa questa interfaccia, gli attributi e i metodi dovrebbero essere privati per la classe.

Linea guida 2

Controllare la validità di tutti gli input

Tutti i programmi ricevono input dal loro ambiente operativo e li elaborano. La specifica di un sistema software contiene alcune ipotesi su questi input che si basano sul loro significato reale. Per esempio, si può ipotizzare che un numero di conto corrente bancario sia sempre un numero intero positivo di 8 cifre. In molti casi, tuttavia, la specifica del sistema non definisce quali azioni devono essere svolte se l'input è sbagliato. A volte gli utenti digitano dati errati. Come dirò nel Capitolo 13, gli attacchi che gli hacker portano a un sistema spesso consistono nell'immettere deliberatamente dati non validi. Anche quando gli input provengono da sensori o altri dispositivi, i loro valori potrebbero essere errati.

Occorre, quindi, controllare sempre la validità degli input, non appena vengono immessi nell'ambiente operativo del programma. I controlli richiesti ovviamente dipendono dal tipo di input; qui si seguito sono elencati alcuni tra i più importanti controlli da effettuare.

1. *Controllo dell'intervallo.* I valori degli input di solito devono essere compresi entro un determinato intervallo. Per esempio, un input che rappresenta una probabilità dovrebbe essere compreso tra 0,0 e 1,0; un input che rappresenta la temperatura dell'acqua liquida dovrebbe essere compreso tra 0 e 100 gradi Celsius e così via.
2. *Controllo della dimensione.* Alcuni input devono essere formati da un certo numero di caratteri; per esempio, il numero di conto corrente bancario di solito è formato da 8 caratteri. In altri casi, la dimensione può essere variabile, ma potrebbe esserci un limite massimo. Per esempio, è poco probabile che il nome di una persona abbia più di 40 caratteri.
3. *Controllo della rappresentazione.* Alcuni input potrebbero essere di un particolare tipo, che ha una rappresentazione standard. Per esempio, i nomi delle persone non includono caratteri numerici, gli indirizzi e-mail sono formati da due parti, separate dal simbolo @, e così via.

4. *Controllo della ragionevolezza.* Se un input fa parte di una particolare serie e noi conosciamo la relazione tra i membri di questa serie, possiamo controllare se il suo valore può ragionevolmente far parte della serie. Per esempio, se il valore di un input rappresenta la lettura di un contatore di energia elettrica, allora si può ragionevolmente prevedere che il consumo di energia sia approssimativamente lo stesso nei corrispondenti periodi dell'anno precedente. Ovviamente, potranno esserci delle variazioni, ma se le differenze sono troppo grandi, allora è probabile che l'input sia sbagliato.

Le azioni da svolgere nel caso in cui un input non sia convalidato dipendono dal tipo di sistema che si sta implementando. In alcuni casi, il problema deve essere segnalato all'utente, chiedendogli di digitare di nuovo l'input. Quando l'input proviene da un sensore, si potrebbe utilizzare il valore valido più recente. Nei sistemi real-time integrati, potrebbe essere necessario stimare il valore sulla base dei dati precedenti, in modo che il sistema possa continuare a funzionare.

Linea guida 3

Fornire un gestore per tutte le eccezioni

Durante l'esecuzione di un programma, errori o eventi inaspettati possono sempre accadere, a causa di un errore del programma o come risultato di circostanze esterne imprevedibili. Un errore o un evento imprevisto che si verifica durante l'esecuzione di un programma è detto *eccezione*. Esempi di eccezioni sono l'interruzione di energia elettrica, il tentativo di accedere a dati inesistenti o un overflow o underflow numerico.

Le eccezioni possono essere causate da condizioni hardware e software. Quando si verifica un'eccezione, il sistema dovrà gestirla. Questo può essere fatto dal programma stesso o da un apposito meccanismo di gestione delle eccezioni. Tipicamente, il meccanismo di gestione delle eccezioni segnala l'errore e interrompe l'esecuzione del programma. Quindi, per garantire che le eccezioni del programma non provochino il fallimento dell'intero sistema, occorre definire un gestore per tutte le possibili eccezioni che possono verificarsi; occorre anche accertarsi che tutte le eccezioni siano rilevate ed esplicitamente gestite.

I linguaggi di programmazione come Java, C++ e Python hanno appositi costrutti per la gestione delle eccezioni. Quando si verifica una situazione eccezionale, questa viene segnalata e il sistema trasferisce il controllo al gestore delle eccezioni, che è una sezione di codice che definisce i nomi delle eccezioni e le azioni appropriate per ciascuna eccezione (Figura 11.12). Il gestore delle eccezioni è al di fuori del flusso normale di controllo, e questo flusso non viene ripreso dopo che un'eccezione è stata gestita.

Il gestore delle eccezioni di solito svolge una delle seguenti operazioni:

1. segnala a un componente di livello più alto che si è verificata un'eccezione e fornisce informazioni al componente sul tipo di eccezione. Questo approccio si usa quando un componente chiama un altro componente e vuole

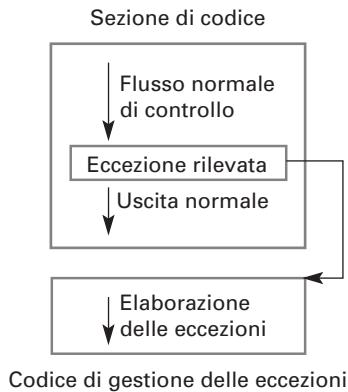


Figura 11.12 Gestione delle eccezioni.

sapere se il componente chiamato è stato eseguito correttamente; in caso contrario, è compito del componente chiamante svolgere un'azione per risolvere il problema;

2. eseguire un'elaborazione alternativa a quella originariamente prevista. Il gestore delle eccezioni svolge qualche azione per risolvere il problema. L'elaborazione normale potrebbe continuare. In alternativa, il gestore può segnalare che si è verificata un'eccezione, in modo che il componente chiamante sia informato e possa gestire l'eccezione;
3. passa il controllo a un sistema di supporto a runtime che gestisce l'eccezione. Questa è di solito l'operazione di default quando il guasto si verifica in un programma, per esempio, quando si ha un overflow di un valore numerico. L'azione usuale del sistema di supporto è interrompere l'elaborazione. Questo approccio dovrebbe essere utilizzato soltanto quando è possibile mettere il sistema in sicurezza, prima che il sistema di supporto assuma il controllo delle operazioni.

La gestione delle eccezioni all'interno di un programma consente di rilevare e correggere alcuni errori di input o riprendere l'esecuzione normale dopo che si è verificato un evento esterno inaspettato. Ciò garantisce un certo livello di tolleranza agli errori. Il programma rileva un problema ed è in grado di svolgere l'azione appropriata per risolverlo. Dal momento che molti errori ed eventi esterni inaspettati di solito sono transitori, spesso è possibile continuare le operazioni normali dopo che è stata elaborata un'eccezione.

Linea guida 4

Minimizzare l'uso di costrutti inclini agli errori

Gli errori nei programmi e, quindi, molti fallimenti dei programmi di solito sono una conseguenza dell'errore umano. I programmatori commettono errori perché perdono la traccia delle numerose relazioni tra le variabili di stato. Scrivono istru-

Costrutti inclini agli errori

Alcuni costrutti dei linguaggi di programmazione sono più inclini di altri a introdurre bug in un programma. L'affidabilità di un programma può essere migliorata evitando questi costrutti. Se possibile, dovreste ridurre al minimo l'uso del comando go nelle istruzioni, i numeri floating point, i puntatori, l'allocazione dinamica della memoria, il parallelismo, la ricorsione, gli interrupt, l'aliasing, gli array senza limiti e l'elaborazione degli input di default.

<http://software-engineering-book.com/web/error-prone-constructs/>

zioni che determinano un comportamento inaspettato del sistema o un cambiamento del suo stato. Le persone sbagliano sempre; alla fine degli anni '60 si scoprì che alcuni approcci alla programmazione erano più inclini di altri a introdurre errori in un programma.

Per esempio, dovreste evitare di usare i numeri floating point in quanto la precisione di questi numeri è limitata dalla loro rappresentazione hardware. I confronti di numeri molto grandi o molto piccoli non sono affidabili. Un altro costrutto che è potenzialmente incline all'errore è l'allocazione dinamica della memoria, mediante la quale il programmatore gestisce esplicitamente la memoria. È facile che un programmatore si dimentichi di liberare lo spazio della memoria quando non è più necessario, e questo può rendere difficile l'identificazione degli errori a runtime.

Alcuni standard per lo sviluppo di sistemi a sicurezza critica proibiscono nel modo più assoluto l'uso di costrutti inclini agli errori. Tuttavia, questa posizione estrema non è normalmente praticabile. Tutti questi costrutti e tecniche sono utili, sebbene debbano essere utilizzati con prudenza. Quando possibile, i loro effetti potenzialmente pericolosi dovrebbero essere controllati utilizzandoli all'interno di oggetti o tipi di dati astratti, i quali operano come barriere naturali che limitano i danni quando si verifica un errore.

Linea guida 5

Creare funzioni di riavviamento

Molti sistemi informatici aziendali si basano su brevi transazioni dove l'elaborazione degli input dell'utente richiede un tempo relativamente breve. Questi sistemi sono progettati in modo che le modifiche del database del sistema vengano completate soltanto dopo che tutti gli altri processi sono stati ultimati con successo. Se si verifica un errore durante un processo, il database non viene aggiornato per non creare incongruenze nei dati. Virtualmente, tutti i sistemi di commercio elettronico, dove l'utente conferma l'acquisto nella videata finale, operano in questo modo.

Le interazioni degli utenti con i sistemi di commercio elettronico di solito durano pochi minuti e richiedono un'elaborazione minima. Le transazioni dei database sono brevi e di norma vengono completate in meno di un secondo. Altri tipi di sistemi, invece, come i sistemi CAD o i word processor, richiedono lunghi

transazioni. In un sistema con transazioni lunghe, il tempo che passa tra l'inizio e il completamento di un lavoro può essere di parecchi minuti o di qualche ora. Se il sistema fallisce durante una lunga transazione, si potrebbe perdere tutto il lavoro. Analogamente, nei sistemi con calcoli intensivi, come quelli scientifici, un lavoro potrebbe richiedere minuti o ore per essere completato. Tutto questo tempo andrebbe sprecato se si verificasse un fallimento del sistema.

Per tutti questi tipi di sistemi è importante creare una funzione di riavviamento, che si basa sulla conservazione di copie di dati raccolti o generati durante i vari processi. La funzione di riavviamento dovrebbe consentire al sistema di ripartire utilizzando queste copie di dati, anziché ricominciare tutto daccapo. Queste copie sono dette "checkpoint". Per esempio:

1. in un sistema di commercio elettronico possiamo conservare le copie dei moduli compilati dagli utenti e consentire loro di accedere e inviare questi moduli senza costringerli a compilarli di nuovo;
2. in una lunga transazione o in un sistema con calcoli intensivi possiamo salvare automaticamente i dati ogni pochi minuti e, nel caso di fallimento del sistema, riavviarlo con gli ultimi dati salvati. Dovremmo anche consentire all'utente di sbagliare e fornirgli un modo per ritornare al checkpoint più recente e da qui ricominciare.

Se si verifica un'eccezione ed è impossibile continuare il normale funzionamento, si può gestire l'eccezione utilizzando il ripristino dello stato prima dell'errore. Questo significa che viene ripristinato lo stato del sistema che è stato salvato nell'ultimo checkpoint e riprendere le operazioni da questo punto.

Linea guida 6

Controllare i limiti degli array

Tutti i linguaggi di programmazione consentono di specificare le caratteristiche degli array – strutture di dati sequenziali accessibili tramite un indice numerico. Questi array di solito occupano aree contigue all'interno dell'area di memoria di un programma. Per ogni array viene specificata la dimensione, che rispecchia il modo in cui un array sarà utilizzato. Per esempio, se volete rappresentare le età di 10.000 persone, allora potete specificare un array con 10.000 locazioni di memoria in cui memorizzare i dati delle età delle persone.

Alcuni linguaggi di programmazione, come Java, controllano sempre che, quando viene immesso un valore in un array, l'indice sia all'interno dell'array. Per esempio, se un array A è indicizzato da 0 a 10.000, il tentativo di inserire dei valori negli elementi A[-5] o A[12345] provocherà un'eccezione. Tuttavia, alcuni linguaggi di programmazione, come C e C++, non includono il controllo automatico dei limiti degli array e calcolano semplicemente un offset dall'inizio dell'array. Pertanto, A[12345] identifica il valore che si trova 12345 locazioni dall'inizio dell'array, indipendentemente dal fatto che esso appartenga o meno all'array.

Questi linguaggi non includono il controllo automatico dei limiti degli array perché ciò introduce un overhead ogni volta che c'è un accesso al database, con conseguente aumento del tempo di esecuzione del programma. La mancanza di questo controllo provoca dei punti di vulnerabilità nella protezione, come un overflow dei buffer, di cui diremo nel Capitolo 13. Più in generale, si introducono delle vulnerabilità che possono portare al fallimento del sistema. Se state utilizzando un linguaggio come C o C++ che non effettua il controllo automatico dei limiti degli array, dovreste creare dei controlli per verificare che gli indici degli array siano sempre all'interno dei loro limiti.

Linea guida 7

Includere un timeout quando si chiama un componente esterno

Nei sistemi distribuiti, i componenti del sistema vengono eseguiti su computer differenti, e le chiamate tra i vari componenti avvengono attraverso la rete. Per ricevere qualche servizio, il componente A può chiamare il componente B. A aspetta la risposta di B prima di continuare la sua esecuzione. Se il componente B non risponde per un motivo qualsiasi, il componente A non può proseguire la sua esecuzione, perché resta in attesa della risposta di B. Un utente che è in attesa di una risposta da parte sistema assiste a un fallimento silenzioso del sistema, senza ricevere alcuna risposta. Per uscire da questa situazione di stallo, l'utente non ha altra alternativa che riavviare il sistema.

Per evitare questo stallo, bisogna includere dei timeout ogni volta che si chiama un componente esterno. Un timeout presuppone che il componente chiamato abbia fallito e non produrrà una risposta. Definite un periodo di tempo durante il quale prevedete di ricevere una risposta dal componente chiamato. Se la risposta non arriva entro tale periodo, si presume che sia avvenuto un guasto e il controllo viene restituito al componente chiamante. A questo punto, si può tentare di risolvere il problema oppure dire agli utenti del sistema che cosa è accaduto e consentire loro di decidere cosa fare.

Linea guida 8

Assegnare un nome a tutte le costanti che rappresentano entità del mondo reale

Tutti i programmi importanti includono un certo numero di valori costanti che rappresentano i valori delle entità del mondo reale. Questi valori non cambiano durante l'esecuzione dei programmi. A volte, si tratta di costanti assolute che non cambiano mai (per esempio, la velocità della luce), ma più spesso si tratta di valori che cambiano molto lentamente nel tempo. Per esempio, i programmi che calcolano le imposte includono alcune costanti che rappresentano i tassi percentuali da applicare ai redditi. Questi tassi potrebbero cambiare da un anno all'altro, e quindi i programmi devono essere aggiornati con i nuovi valori.

Dovreste includere sempre una sezione nel programma in cui assegnate un nome a tutti i valori costanti delle entità del mondo reale che vengono utilizzate. Per utilizzare queste costanti, fate riferimento ai loro nomi, non ai loro valori. In questo modo, otterrete due vantaggi relativamente alla fidatezza dei sistemi.

1. Ci sono meno probabilità di commettere errori e di utilizzare un valore errato. È facile sbagliare a digitare un numero, e il sistema spesso non sarà in grado di rilevare l'errore. Per esempio, supponiamo che il tasso sia 34%. Un semplice errore di trasposizione durante la digitazione potrebbe portare questo valore a 43%. Se, invece, sbagliate a digitare un nome (come Aliquota-fiscale-standard), questo errore sarà segnalato dal compilatore come variabile non dichiarata.
2. Quando un valore cambia, non occorre controllare tutto il programma per sapere dove avete utilizzato tale valore. Basta modificare il valore definito nella dichiarazione della costante. Il nuovo valore sarà automaticamente utilizzato tutte le volte che ci sarà un riferimento al nome della costante.

11.5 Misura dell'affidabilità

Per valutare l'affidabilità di un sistema, occorre raccogliere dati sul suo funzionamento. I dati richiesti possono includere:

1. il numero di fallimenti del sistema che si verificano per un determinato numero di richieste di servizi del sistema. Questo valore è utilizzato per misurare la metrica POFOD e si applica indipendentemente dal periodo di tempo durante il quale vengono fatte le richieste;
2. il tempo o il numero di transazioni tra i fallimenti del sistema più il tempo totale trascorso o il numero totale di transazioni. Questo valore è utilizzato per misurare le metriche ROCOF e MTTF;
3. il tempo di recovery (riparazione e riavviamento) dopo un fallimento del sistema che ha provocato la perdita di un servizio. Questo valore è una misura della disponibilità del sistema. La disponibilità non dipende soltanto dal tempo tra i fallimenti, ma anche dal tempo necessario per rimettere in funzione il sistema.

Le unità di tempo che possono essere utilizzate in queste metriche sono tempi di calendario o unità discrete, come il numero di transazioni. Dovreste utilizzare i tempi di calendario per i sistemi che sono sempre in funzione. I sistemi di monitoraggio, come quelli che controllano i processi, ricadono in questa categoria. Pertanto, ROCOF potrebbe essere il numero di fallimenti al giorno. I sistemi che elaborano le transazioni, come gli sportelli ATM delle banche o i sistemi di prenotazione dei voli aerei, hanno carichi di lavoro che variano durante il giorno. In questi casi, l'unità di "tempo" utilizzata potrebbe essere il numero di transazioni; ovvero, ROCOF potrebbe essere il numero di transazioni fallite per N migliaia di transazioni.

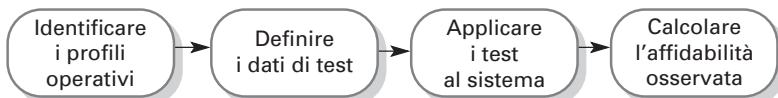


Figura 11.13 Test statistico per misurare l'affidabilità di un sistema.

Il test dell'affidabilità è un processo statistico che ha lo scopo di misurare l'affidabilità di un sistema. Le metriche dell'affidabilità, come POFOD, la probabilità di fallimento su richiesta, e ROCOF, il tasso di occorrenza dei fallimenti, possono essere utilizzate per specificare quantitativamente l'affidabilità richiesta per il software. Potete controllare il processo di test dell'affidabilità se il sistema ha raggiunto il livello richiesto di affidabilità.

Il processo che misura l'affidabilità di un sistema è anche detto test statistico (Figura 11.13). Questo processo statistico ha il compito di misurare l'affidabilità, non quello di trovare gli errori. Powell (Powell et al. 1999) descrive molto bene il test statistico nel libro *Cleanroom software engineering*.

Il test statistico è composto da quattro fasi.

1. Iniziate a studiare i sistemi esistenti dello stesso tipo per capire come vengono utilizzati in pratica. Questo è importante perché state cercando di misurare l'affidabilità così come è stata sperimentata dagli utenti del sistema. Lo scopo è definire un profilo operativo. Questo profilo identifica le classi degli input del sistema e la probabilità che questi input si verifichino durante l'utilizzo normale del sistema.
2. Definite una serie di dati di test che rispecchiano il profilo operativo. Questo significa che scegliete i dati con la stessa distribuzione di probabilità dei dati di test dei sistemi che avete studiato. Normalmente, si usa un generatore di dati di test a supporto di questo processo.
3. Provate il sistema utilizzando questi dati e contate il numero e il tipo di fallimenti. Registrate anche i tempi in cui i fallimenti si verificano. Come detto nel Capitolo 10, le unità di tempo scelte dovrebbero essere appropriate alla metrica utilizzata per misurare l'affidabilità.
4. Dopo avere osservato un numero statisticamente significativo di fallimenti, potete calcolare l'affidabilità del software e determinare il valore della metrica appropriata dell'affidabilità.

Questo approccio teoricamente interessante per misurare l'affidabilità non è facilmente applicabile in pratica. Le principali difficoltà sono dovute ai seguenti fattori.

1. *Incertezza del profilo operativo* – I profili operativi basati sulle esperienze con altri sistemi possono riflettere in modo non accurato l'utilizzo reale del sistema.

2. *Alti costi per generare i dati di test* – Può essere molto costoso generare l'enorme volume di dati richiesti da un profilo operativo, a meno che questa operazione non possa essere completamente automatizzata.
3. *Incertezza statistica quando viene specificata un'alta affidabilità* – Occorre generare un numero di fallimenti statisticamente significativo per garantire che le misure dell'affidabilità siano accurate. Quando il software è già affidabile, si verificano pochi fallimenti e, quindi, è difficile generarne di nuovi.
4. *Riconoscere i fallimenti* – Non è sempre ovvio che si sia verificato un fallimento. Se disponete di una specifica formale, potete identificare le deviazioni del sistema da tale specifica. Se, invece, la specifica è scritta nel linguaggio naturale, potrebbero esserci frasi ambigue che potrebbero indurre gli osservatori a giudicare in modo contrastante il comportamento del sistema.

Il modo migliore per generare i volumi di dati necessari per misurare l'affidabilità di un sistema consiste nell'utilizzare un apposito generatore di dati, che può essere configurato per generare automaticamente gli input corrispondenti al profilo operativo. Tuttavia, non sempre è possibile automatizzare la produzione di tutti i dati di test per i sistemi interattivi, in quanto gli input di solito sono una risposta agli output del sistema. I dataset di test per questi sistemi devono essere generati manualmente, con conseguente aumento dei costi. Anche quando è possibile generare i dataset in modo completamente automatico, scrivere i comandi per il generatore dei dati può richiedere una quantità di tempo significativa.

Il test statistico può essere utilizzato insieme a un generatore di errori per ottenere i dati sul livello di efficienza del processo di test dei difetti. Un generatore di errori (Voas e McGraw 1997) introduce deliberatamente degli errori in un programma. Quando il programma viene eseguito, gli errori introdotti causano fallimenti. Si analizzano i fallimenti per scoprire se la causa originale è uno degli errori che sono stati aggiunti nel programma. Se si scopre che l'X% degli errori introdotti ha causato fallimenti, allora, secondo Voas e McGraw, questo significa che il processo di test dei difetti scoprirà anche l'X% degli errori effettivi nel programma.

Questo approccio suppone che la distribuzione e il tipo di errori introdotti riflettano gli errori reali del sistema. Questo potrebbe essere vero per i fallimenti causati dagli errori di programmazione, ma è meno probabile per i fallimenti derivanti dai problemi connessi ai requisiti o alla progettazione del sistema. L'introduzione deliberata di errori in un programma non è efficace nel prevedere il numero di fallimenti che potrebbero derivare da cause diverse dagli errori di programmazione.

Modello di crescita dell'affidabilità

Un modello di crescita dell'affidabilità mostra come l'affidabilità di un sistema cambia nel tempo durante il processo di test. Quando si verifica un fallimento del sistema, l'errore che lo ha causato viene corretto; in questo modo, l'affidabilità del sistema migliora durante i test e il debugging del sistema. Per prevedere l'affidabilità, il modello di crescita dell'affidabilità deve quindi essere tradotto in un modello matematico.

<http://software-engineering-book.com/web/reliability-growth-modeling/>

11.5.1 Profili operativi

Il profilo operativo di un sistema software riflette il modo in cui esso sarà utilizzato in pratica. È composto da una specifica delle classi di input e delle probabilità delle loro occorrenze. Quando un nuovo sistema software sostituisce un sistema automatico esistente, è ragionevolmente semplice accedere ai probabili schemi di utilizzo del nuovo software, che dovrebbero corrispondere all'utilizzo esistente, eccezion fatta per le nuove funzionalità che presumibilmente sono incluse nel nuovo software. Per esempio, si può specificare un profilo operativo per un sistema di commutazione telefonica, in quanto le compagnie di telecomunicazioni conoscono gli schemi delle chiamate che questi sistemi devono gestire.

Tipicamente, il profilo operativo è tale che gli input con la probabilità più alta di essere generati ricadono in un piccolo numero di classi, come mostrato nella Figura 11.14. Ci sono molte classi i cui input sono altamente improbabili, ma non impossibili; queste sono mostrate a destra nella Figura 11.14. I puntini di sospensione (...) indicano che esistono molti di questi input insoliti rispetto a quelli mostrati.

Musa (Musa 1998) ha trattato lo sviluppo dei profili operativi per i sistemi di telecomunicazioni. Avendo raccolto una vasta serie di dati di utilizzo in questo dominio, il processo di sviluppo dei profili operativi è relativamente semplice. Per un sistema il cui sviluppo richiedeva 15 anni-uomo circa, fu sviluppato un profilo operativo in un mese-uomo circa. In altri casi, la generazione dei profili operativi richiese più tempo (2 o 3 anni-uomo), ma il costo era stato distribuito su un certo numero di release del sistema.

Quando un sistema software è nuovo e innovativo, però, è difficile prevedere come sarà utilizzato; di conseguenza, è praticamente impossibile generare un profilo operativo accurato. Il nuovo sistema potrebbe essere utilizzato da molti utenti con aspettative, background ed esperienze differenti. Non esiste un database con le informazioni sull'utilizzo del sistema. Questi utenti potrebbero far uso del sistema in modi che non erano stati previsti dagli sviluppatori.

Sviluppare un profilo operativo accurato è certamente possibile per alcuni tipi di sistemi, come quelli per le telecomunicazioni, che hanno schemi di utilizzo standardizzati. Per altri tipi di sistemi, invece, sviluppare un profilo operativo accurato potrebbe essere un compito difficile o addirittura impossibile:

1. un sistema potrebbe avere molti utenti diversi, ciascuno dei quali ha un suo modo di utilizzare il sistema. Come detto all'inizio di questo capitolo, utenti diversi hanno sensazioni diverse sull'affidabilità, in quanto usano lo stesso sistema in modi differenti. È difficile riassumere tutti questi schemi di utilizzo in un unico profilo operativo.
2. il modo in cui gli utenti utilizzano un sistema cambia nel tempo. Man mano che gli utenti imparano a utilizzare un nuovo sistema e acquisiscono familiarità con le sue funzioni, iniziano a utilizzare il sistema in modo più sofisticato. Un profilo operativo, quindi, che corrisponde allo schema di utilizzo iniziale di un sistema potrebbe non essere più valido dopo che gli utenti hanno acquisito familiarità con esso.

Per queste ragioni, spesso è impossibile sviluppare un profilo operativo affidabile. Se utilizzate un profilo operativo obsoleto o sbagliato, non potrete fidarvi della precisione delle misure di affidabilità che effettuate.

Punti chiave

- L'affidabilità di un sistema software può essere ottenuta evitando di introdurre errori, rilevando e correggendo gli errori prima di consegnare il sistema e includendo le funzioni di tolleranza agli errori che consentono al sistema di restare operativo dopo che un errore ha causato il fallimento del sistema.
- I requisiti di affidabilità possono essere definiti quantitativamente nella specifica dei requisiti di un sistema software. Le metriche di affidabilità includono la probabilità di fallimento su richiesta (POFOD), il tasso di occorrenza dei fallimenti (ROCOF) e la disponibilità (AVAIL).
- I requisiti di affidabilità funzionali riguardano le funzionalità di un sistema software, come i requisiti di controllo e ridondanza, che aiutano il sistema a soddisfare i suoi requisiti di affidabilità non funzionali.
- Le architetture dei sistemi fidati sono progettate per la tolleranza agli errori. Un certo numero di stili architettonici supportano la tolleranza agli errori, inclusi i sistemi di protezione, le architetture auto-monitorate e la programmazione a N versioni.
- La diversità del software è difficile da realizzare perché è praticamente impossibile garantire che ciascuna versione del software sia veramente indipendente.
- La programmazione fidata si basa sulla ridondanza sotto forma di controlli della validità degli input e dei valori delle variabili di programma.
- Il test statistico è utilizzato per stimare l'affidabilità di un sistema software. Consiste nel provare un sistema con dati di test che corrispondono a un profilo operativo, che rispecchia la distribuzione degli input immessi durante l'utilizzo del software.

Esercizi

- 11.1 Spiegate perché è praticamente impossibile convalidare le specifiche di affidabilità quando queste sono espresse in termini di un numero molto piccolo di fallimenti sulla vita complessiva di un sistema software.
- * 11.2 Suggerite alcune metriche di affidabilità che sono appropriate alle classi dei seguenti sistemi software. Spiegate perché avete scelto queste metriche. Prevedete l'utilizzo di questi sistemi e suggerite i valori appropriati per le metriche di affidabilità.
- Un sistema che monitora i pazienti in una unità di cura intensiva.
 - Un word processor.
 - Un sistema di controllo di un distributore automatico.
 - Un sistema per controllare la frenatura di un'automobile.
 - Un sistema per controllare un'unità di refrigerazione.
 - Un generatore di report gestionali.
- * 11.3 Un sistema di protezione dei treni aziona automaticamente i freni se viene superato il limite di velocità di un tratto ferroviario oppure se il treno entra in un tratto ferroviario con il semaforo rosso. Spiegate le vostre risposte e suggerite quale metrica di affidabilità utilizzereste per specificare l'affidabilità richiesta per tale sistema.
- * 11.4 Qual è la caratteristica comune di tutti gli stili architetturali che si basano sulla tolleranza agli errori del software?
- 11.5 Suggerite le circostanze nelle quali è appropriato utilizzare un'architettura tollerante agli errori quando si implementa un sistema di controllo basato sul software e spiegate perché è necessario questo tipo di approccio.
- * 11.6 Siete responsabili del progetto di un sistema di commutazione che deve essere disponibile 24 ore/giorno per 7 giorni/settimana; il sistema non è a sicurezza critica. Spiegate le vostre risposte e suggerite uno stile architettonico che potrebbe essere utilizzato per questo sistema.
- * 11.7 Il software di controllo per una macchina di radioterapia, utilizzata per curare i pazienti di cancro, deve essere implementato tramite la programmazione a N versioni. Secondo voi, questa soluzione è appropriata per il tipo di controllo da effettuare?
- 11.8 Spiegate perché tutte le versioni di un sistema progettato attorno alla diversità del software possono fallire nello stesso modo.
- 11.9 Spiegate perché occorre gestire esplicitamente tutte le eccezioni in un sistema che è stato progettato per avere un alto livello di disponibilità.
- 11.10 I fallimenti del software possono causare seri inconvenienti agli utenti del software. È giusto che le società rilascino software che include errori che potrebbero provocare fallimenti del software? Queste società dovrebbero ricompensare gli utenti per le perdite causate dai fallimenti del loro software? Dovrebbero garantire per legge il software nello stesso modo in cui i produttori di beni di consumo garantiscono i loro prodotti?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Software Fault Tolerance Techniques and Implementation. Una descrizione completa delle tecniche per realizzare software e architetture fault-tolerant. Il libro tratta anche i problemi generali della fidatezza del software. L'ingegneria dell'affidabilità è un'area matura e le tecniche qui descritte sono ancora attuali. L. L. Pullum, Artech House, 2001.

“Software Reliability Engineering: A Roadmap.” L'indagine condotta da un ricercatore esperto riassume lo stato dell'arte dell'ingegneria dell'affidabilità del software e tratta le sfide della ricerca in quest'area. M. R. Lyu, *Proc. Future of Software Engineering*, IEEE Computer Society, 2007. <http://dx.doi.org/10.1109/FOSE.2007.24>

“Mars Code.” Descrive l'approccio all'ingegneria dell'affidabilità adottato nello sviluppo del software del Mars Curiosity Rover, che si basava sull'uso di buone pratiche di progettazione, sulla ridondanza e sul controllo dei modelli (descritto nel Capitolo 12). G. J. Holzmann, *Comm. ACM*, 57 (2), 2014. <http://dx.doi.org/10.1145/2560217.2560218>

Ingegneria della sicurezza

L'obiettivo di questo capitolo è descrivere le tecniche utilizzate per garantire la sicurezza quando si sviluppano sistemi critici. Dopo aver letto questo capitolo:

- capirete che cosa s'intende per sistema a sicurezza critica e perché la sicurezza deve essere considerata separatamente dall'affidabilità nell'ingegneria dei sistemi critici;
- capirete come eseguire l'analisi dei rischi per ottenere i requisiti di sicurezza;
- conoscerete i processi e gli strumenti che si usano per garantire la sicurezza del software;
- apprenderete il concetto di caso di sicurezza che si usa per dimostrare la sicurezza di un sistema alle autorità di controllo, e come utilizzare gli argomenti formali nei casi di sicurezza.

12.1 Sistemi a sicurezza critica

12.2 Requisiti di sicurezza

12.3 Processi di ingegneria della sicurezza

12.4 Casi di sicurezza

Nel Paragrafo 11.2 ho descritto brevemente l'incidente aereo avvenuto nell'aeroporto di Varsavia, dove un Airbus finì fuori pista e si incendiò; ci furono 2 morti e 54 feriti. L'inchiesta dimostrò che la causa principale dell'incidente era imputabile al fallimento del software di controllo che riduceva la capacità del sistema di frenatura dell'aereo. Questo è uno degli esempi, per fortuna rari, di come il comportamento di un sistema software possa provocare morti e feriti. Dimostra che il software è diventato un componente centrale in molti sistemi che sono critici nel preservare e proteggere la vita. Sono detti sistemi software a sicurezza critica; sono stati sviluppati vari metodi e tecniche specializzati per l'ingegneria del software a sicurezza critica.

Come detto nel Capitolo 10, la sicurezza è una delle principali proprietà della fidatezza. Un sistema può essere considerato sicuro se opera senza fallimenti catastrofici, ovvero senza provocare morti o feriti tra le persone. Anche i sistemi i cui fallimenti possono causare danni ambientali possono essere considerati a sicurezza critica, in quanto un danno ambientale (come la dispersione di sostanze chimiche) può arrecare nel tempo morte e malattie alle persone.

Il software nei sistemi a sicurezza critica svolge un duplice ruolo nella realizzazione della sicurezza.

1. Il sistema può essere controllato dal software in modo che le decisioni prese dal software e le successive azioni siano a sicurezza critica. Pertanto, il comportamento del software è direttamente correlato alla sicurezza complessiva del sistema.
2. Il software è usato esclusivamente per controllare e monitorare altri componenti a sicurezza critica del sistema. Per esempio, tutti i componenti delle turbine di un aereo sono monitorati dal software, che cerca di rilevare preventivamente qualche indizio di fallimento dei componenti.

La sicurezza nei sistemi software si raggiunge cercando di capire le situazioni che potrebbero portare a fallimenti correlati alla sicurezza. Il software viene progettato in modo che tali fallimenti non possano verificarsi. Si potrebbe quindi pensare che, se un sistema a sicurezza critica è affidabile e si comporta come specificato, esso è sicuro; purtroppo, non è così. L'affidabilità di un sistema software è necessaria per ottenere la sicurezza, ma non è sufficiente. I sistemi affidabili potrebbero non essere sicuri e viceversa. L'incidente aereo di Varsavia è un esempio di questa situazione, che descriverò dettagliatamente nel Paragrafo 12.2.

I sistemi software che sono affidabili potrebbero non essere sicuri per quattro motivi.

1. Non possiamo essere certi al 100% che un sistema software sia privo di errori e fault-tolerant. Gli errori non rilevati possono restare dormienti per parecchio tempo, e i fallimenti del software possono verificarsi dopo molti anni di funzionamento affidabile del sistema.

2. La specifica potrebbe essere incompleta se non descrive il comportamento richiesto dal sistema in qualche situazione critica. Un'alta percentuale di malfunzionamenti del sistema è la conseguenza di una specifica ambigua, anziché di errori di progettazione. In uno studio degli errori nei sistemi integrati, Lutz (Lutz 1993) ha concluso che “le difficoltà con i requisiti sono la causa principale degli errori del software connessi alla sicurezza, che persistono fino all'integrazione e al test del sistema”¹ Uno studio più recente condotto da Veras (Veras et al. 2010) nei sistemi di controllo dello spazio conferma che gli errori della specifica sono un problema importante per i sistemi integrati.
3. I malfunzionamenti dell'hardware possono innescare comportamenti anomali di sensori e attuatori. Quando un componente è prossimo al guasto fisico, può comportarsi in modo irregolare e generare segnali che sono fuori dai valori che possono essere gestiti dal software. Il software quindi può fallire o interpretare in modo errato questi segnali.
4. Gli operatori di un sistema possono generare input che non sono singolarmente errati, ma che in qualche caso possono provocare un malfunzionamento del sistema. Un esempio classico di questo si è verificato quando il carrello di atterraggio di un aereo è crollato mentre l'aereo era fermo a terra. A quanto pare, un tecnico ha premuto un pulsante che indicava al software di gestione degli accessori di sollevare il carrello. Il software ha svolto il suo compito perfettamente, ma il sistema avrebbe dovuto disabilitare tale comando quando l'aereo si trova a terra.

La sicurezza, quindi, deve essere tenuta in considerazione come l'affidabilità quando si sviluppa un sistema a sicurezza critica. Le tecniche di ingegneria dell'affidabilità, che ho presentato nel Capitolo 11, sono ovviamente applicabili all'ingegneria dei sistemi a sicurezza critica. Pertanto, qui non tratterò le architetture dei sistemi e la progettazione fidata, ma illustrerò le tecniche per migliorare e garantire la sicurezza dei sistemi.

12.1 Sistemi a sicurezza critica

I sistemi a sicurezza critica sono sistemi nei quali è essenziale che le operazioni siano sempre sicure. In altre parole, il sistema non dovrebbe mai arrecare danni alle persone o all'ambiente, indipendentemente dal fatto che il sistema sia conforme o no alla sua specifica. Esempi di sistemi a sicurezza critica sono i sistemi di controllo e monitoraggio degli aerei, i sistemi di controllo dei processi negli stabilimenti chimici e farmaceutici e i sistemi di controllo delle automobili.

¹ Lutz, R R. 1993. “Analysing Software Requirements Errors in Safety-Critical Embedded Systems.” In RE’93, 126-133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.

I sistemi a sicurezza critica possono essere classificati in due categorie:

1. *Software primario a sicurezza critica.* Questo è il software che è integrato come controllore in un sistema. Il malfunzionamento di questo software può provocare un malfunzionamento dell'hardware, che potrebbe danneggiare le persone e l'ambiente. Il software per la pompa di insulina, descritto nel Capitolo 1, è un esempio di sistema primario a sicurezza critica. Un fallimento di questo sistema può arrecare danni all'utente.

Il sistema della pompa di insulina è un sistema semplice, ma il software di controllo è utilizzato anche in sistemi a sicurezza critica molto complessi. Il software, anziché il controllo dell'hardware, è essenziale perché occorre gestire un gran numero di sensori e attuatori, che hanno leggi di controllo complicate. Per esempio, un aereo militare tecnologicamente sofisticato e aerodinamicamente instabile richiede continue regolazioni, controllate dal software, delle superfici di volo per garantire che non si schianti a terra.

2. *Software secondario a sicurezza critica.* Questo software può provocare danni in modo indiretto. Un esempio è un sistema di progettazione assistita dall'elaboratore (CAD), il cui malfunzionamento potrebbe causare un errore di progettazione dell'oggetto che si sta sviluppando. Questo errore potrebbe arrecare danni alle persone nel caso di malfunzionamento di questo oggetto. Un altro esempio di software secondario a sicurezza critica è il sistema Mentcare per la gestione dei pazienti con problemi psichici. Un malfunzionamento di questo sistema, a causa del quale un paziente mentalmente instabile potrebbe essere trattato in modo improprio, potrebbe indurre il paziente a danneggiare se stesso o altre persone.

Alcuni sistemi di controllo, come quelli che controllano le infrastrutture pubbliche (energia elettrica, telecomunicazioni, trattamento delle acque reflue ecc.) , sono sistemi secondari a sicurezza critica. È poco probabile che un malfunzionamento di questi sistemi abbia conseguenze immediate sull'uomo. Tuttavia, un'interruzione prolungata dei sistemi che controllano queste infrastrutture potrebbe causare morti e feriti. Per esempio, un guasto del sistema di depurazione delle acque reflue potrebbe innalzare il livello di malattie infettive, in quanto i liquami vengono dispersi nell'ambiente.

Come ho spiegato nel Capitolo 11, la disponibilità e l'affidabilità dei sistemi software sono realizzate evitando i difetti (*fault avoidance*), identificando e correggendo i difetti (*fault detection and correction*) e tollerando i difetti (*fault tolerance*). Lo sviluppo dei sistemi a sicurezza critica usa questi tre approcci, potenziandoli mediante tecniche guidate dai rischi (*hazard-driven*), che considerano i potenziali incidenti che possono capitare a un sistema software.

1. *Evitare i rischi (hazard avoidance).* Il sistema è progettato in modo da evitare i rischi. Per esempio, un sistema per tagliare la carta richiede che l'operatore usi entrambe le mani per premere contemporaneamente due

pulsanti separati, evitando così il rischio che una mano dell'operatore possa trovarsi nel percorso della lama.

2. *Identificare ed eliminare i rischi (hazard detection and removal).* Il sistema è progettato in modo che i rischi siano individuati e rimossi prima che possano provocare incidenti. Per esempio, un sistema di controllo di un impianto chimico potrebbe rilevare una pressione eccessiva e aprire una valvola di sfogo per ridurre la pressione, prima che si verifichi un'esplosione.
3. *Limitazione dei danni.* Il sistema potrebbe includere delle funzioni di protezione che minimizzano i danni che un incidente potrebbe causare. Per esempio, il motore di un aereo di solito include degli estintori antincendio automatici. Se si incendia un motore, le fiamme vengono immediatamente estinte prima che possano danneggiare altre parti dell'aereo.

Un rischio è uno stato del sistema che può causare un incidente. Utilizzando il precedente esempio del sistema per tagliare la carta, un rischio si presenta quando la mano dell'operatore è in una posizione tale che può essere ferita dalla lama di taglio. I rischi non sono incidenti – spesso ci mettiamo in situazioni rischiose e ne usciamo senza problemi. Tuttavia, un incidente è sempre preceduto da un rischio, quindi riducendo i rischi, si riducono gli incidenti.

Un rischio è un esempio della terminologia specializzata che viene utilizzata nell'ingegneria dei sistemi a sicurezza critica. Altri termini della sicurezza sono riportati nella Figura 12.1.

Oggi è possibile costruire sistemi che possono affrontare condizioni di errore. Possiamo progettare meccanismi nel sistema che sono in grado di identificare e risolvere singoli problemi. Tuttavia, se più cose vanno male contemporaneamente, è più probabile che si verifichi un incidente. Poiché i sistemi diventano sempre più complessi, è difficile capire le relazioni tra le varie parti del sistema. Di conseguenza, non possiamo prevedere le conseguenze di una combinazione di eventi inaspettati o di fallimenti.

Analizzando una serie di incidenti gravi, Perrow (Perrow 1984) ha scoperto che quasi tutti gli incidenti erano dovuti a una combinazione di fallimenti in varie parti di un sistema.

Combinazioni impreviste di fallimenti dei sottosistemi portano a interazioni che causano il fallimento dell'intero sistema. Per esempio, il guasto del sistema di condizionamento dell'aria può provocare il surriscaldamento dell'hardware; se l'hardware è surriscaldato, il suo comportamento diventa imprevedibile, quindi il surriscaldamento potrebbe indurre il sistema a generare segnali sbagliati. Questi segnali potrebbero causare una reazione sbagliata del software.

Perrow ha notato che nei sistemi complessi è impossibile prevedere tutte le possibili combinazioni di guasti. Per questo, ha coniato il termine “normali incidenti”, nel senso che gli incidenti possono essere considerati inevitabili quando si realizza un sistema complesso a sicurezza critica.

Termino	Descrizione
Incidente (o contrattempo)	Un evento o una sequenza di eventi non pianificati che possono portare morte o infortuni alle persone o danni alle cose e all'ambiente. Un'overdose di insulina è un esempio di incidente.
Danno	La misura delle perdite risultanti da un incidente. Le perdite variano dalla morte di diverse persone, alle ferite lievi o ai danni alle cose. Un'overdose di insulina potrebbe arrecare gravi danni fisici o perfino la morte all'utente della pompa di insulina.
Pericolo	Una condizione che può causare un incidente. Un esempio di pericolo è un sensore che misura il livello di zuccheri nel sangue.
Probabilità del pericolo	La probabilità che si verifichino eventi che possono creare situazioni pericolose. I valori di probabilità tendono a essere arbitrari, ma variano da "probabile" (per esempio, una probabilità 1/100 che si presenti un pericolo) a "non plausibile" (non ci sono condizioni che possano creare situazioni pericolose). La probabilità che un sensore della pompa di insulina sovraffasti il livello di zuccheri nel sangue è bassa.
Gravità del pericolo	Una stima del danno peggiore che potrebbe risultare da un particolare pericolo. La gravità può variare da "catastrofica", quando molte persone perdono la vita, a "lieve", quando si hanno solo danni lievi. Se è possibile la morte di un singolo individuo, una stima appropriata della gravità del pericolo è "molto alta".
Rischio	Una misura della probabilità che il sistema provochi un incidente. Il rischio è stimato considerando la probabilità e la gravità del pericolo, e la probabilità che il pericolo cau si un incidente. Il rischio di un'overdose di insulina varia da medio a basso.

Figura 12.1 Terminologia della sicurezza.

Per ridurre la complessità, potremmo utilizzare semplici controlli hardware, anziché software. Tuttavia, i sistemi controllati dal software possono monitorare una gamma più ampia di condizioni rispetto ai sistemi elettromeccanici più semplici. Possono essere adattati abbastanza facilmente; utilizzano l'hardware del computer, che ha un'affidabilità intrinsecamente elevata, e che è fisicamente piccolo e leggero.

I sistemi controllati dal software possono fornire sofisticati blocchi di sicurezza. Possono supportare strategie di controllo che riducono la quantità di tempo che le persone devono dedicare agli ambienti pericolosi. Sebbene il controllo software possa introdurre più modi in cui un sistema può fallire, tuttavia esso consente un monitoraggio e una protezione migliori. Il controllo software quindi può contribuire a migliorare la sicurezza dei sistemi.

Specifica dei requisiti basata sui rischi

La specifica basata sui rischi è un approccio che è stato largamente adottato dagli sviluppatori di sistemi protetti e a sicurezza critica. Si basa principalmente su quegli eventi che potrebbero causare i danni maggiori o che si verificano con maggiore frequenza. Gli eventi che hanno conseguenze secondarie o che sono estremamente rari possono essere ignorati. Per scrivere una specifica basata sui rischi occorre conoscere i rischi cui potrà essere sottoposto il sistema, scoprirne le cause e definire i requisiti per gestire tali rischi.

<http://software-engineering-book.com/web/risk-based-specification/>

È importante mantenere un senso delle proporzioni nei sistemi a sicurezza critica. Questi sistemi operano senza problemi per la maggior parte del tempo. Relativamente poche persone nel mondo sono state uccise o ferite a causa di un errore del software. Perrow ha ragione nel dire che gli incidenti sono sempre possibili. È impossibile realizzare un sistema sicuro al 100%, e le società di software devono decidere se le conseguenze di un incidente occasionale valgano i benefici derivanti dall'uso di tecnologie avanzate.

12.2 Requisiti di sicurezza

All'inizio di questo libro ho descritto l'incidente aereo avvenuto nell'aeroporto di Varsavia a causa di un fallimento del sistema di frenatura di un Airbus. L'inchiesta sull'incidente dimostrò che il software di controllo aveva operato secondo la sua specifica; non c'erano stati errori nel programma. Tuttavia, la specifica del software era incompleta e non aveva tenuto conto di una situazione rara, che purtroppo si era verificata in questo caso. Il software aveva funzionato, ma il sistema aveva fallito.

Questo episodio dimostra che la sicurezza di un sistema non dipende soltanto dalle buone tecniche di ingegneria, ma occorre prestare attenzione ai dettagli quando vengono definiti i requisiti del sistema e vengono inseriti speciali requisiti del software che sono orientati a garantire la sicurezza di un sistema. I requisiti di sicurezza sono requisiti funzionali, che definiscono le funzionalità di controllo e recovery che dovrebbero essere incluse nel sistema e le caratteristiche che forniscono la protezione contro i fallimenti del sistema e gli attacchi esterni.

Per generare requisiti di sicurezza funzionali, di solito, si inizia studiando il dominio applicativo, i regolamenti e gli standard sicurezza. Da questo studio scaturiscono requisiti di livello più alto, che potrebbero essere chiamati requisiti "non deve". Diversamente dai normali requisiti funzionali che definiscono ciò che il sistema deve fare, i requisiti "non deve" definiscono il comportamento del sistema che non è accettabile. Esempi di requisiti "non deve" sono:

- il sistema non deve permettere di selezionare l'inversione di spinta quando l'aereo è in volo;

- il sistema non deve permettere l’attivazione simultanea di più di tre segnali d’allarme;
- il sistema di navigazione non deve consentire agli utenti di impostare la destinazione quando l’automobile è in movimento.

Questi requisiti “non deve” non possono essere implementati direttamente, ma devono essere scomposti in requisiti funzionali più specifici. In alternativa, le scelte di implementazione di questi requisiti possono essere rinviate fino alla progettazione del sistema, quando si deciderà per esempio il particolare tipo di dispositivo da utilizzare nel sistema.

I requisiti di sicurezza sono soprattutto requisiti di protezione e non riguardano il funzionamento normale del sistema. Possono specificare che il sistema deve essere fermato per motivi di sicurezza. Nel definire i requisiti di sicurezza, occorre quindi trovare un compromesso accettabile tra sicurezza e funzionalità per evitare eccessi di protezione. Non ha senso costruire un sistema molto sicuro se non opera in modo economicamente vantaggioso.

La specifica dei requisiti basata sui rischi è un approccio generale che è utilizzato nell’ingegneria dei sistemi critici, dove vengono identificati non solo i rischi cui sono sottoposti i sistemi, ma anche i requisiti per evitare o ridurre tali rischi. La specifica può essere utilizzata per tutti i tipi di requisiti di fidatezza. Per i sistemi a sicurezza critica, essa si traduce in un processo guidato dai rischi che sono stati identificati. Come detto nel precedente paragrafo, un rischio è qualcosa che potrebbe causare la morte o il ferimento di una persona.

Il processo di specifica della sicurezza guidata dai rischi è composto da quattro attività.

1. *Identificazione dei rischi.* Questa attività identifica i rischi potenziali di un sistema. Questi rischi possono essere memorizzati in un apposito registro, che è un documento formale che riporta le analisi e le valutazioni sulla sicurezza. Questo registro può essere sottoposto all’esame di un ente di regolamentazione come parte di un caso di sicurezza.
2. *Valutazione dei rischi.* Questa attività decide quali rischi sono più pericolosi e/o più frequenti. I rischi dovrebbero essere classificati in base alla loro pericolosità e frequenza quando si definiscono i requisiti della sicurezza.
3. *Analisi dei rischi.* È il processo che analizza le cause principali dei rischi e gli eventi che possono concretizzarli.
4. *Riduzione dei rischi.* Questo processo si basa sui risultati dell’analisi dei rischi e permette di stabilire i requisiti di sicurezza. Questi requisiti definiscono le tecniche per evitare i rischi o per impedire che un rischio degeneri in un incidente o, in caso di incidente, per ridurre al minimo i danni risultanti.

La Figura 12.2 illustra questo processo di specifica dei requisiti della sicurezza guidata dai rischi.

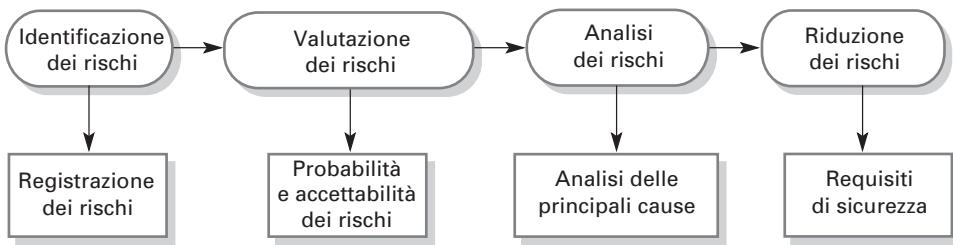


Figura 12.2 Specifica dei requisiti guidata dai rischi.

12.2.1 Identificazione dei rischi

Nei sistemi a sicurezza critica, l'identificazione dei rischi inizia individuando varie classi di rischi, come i rischi fisici, elettrici, biologici, radioattivi e fallimenti dei servizi. Ciascuna di queste classi può essere analizzata per scoprire i rischi specifici che potrebbero concretizzarsi. Devono essere identificate anche le possibili combinazioni di rischi che sono potenzialmente pericolose.

Gli ingegneri, che lavorano con gli esperti dei domini applicativi e con i professionisti della sicurezza, identificano i rischi in base alle loro precedenti esperienze e analizzando i domini. Possono essere utilizzate varie tecniche di lavoro di gruppo, come il brainstorming, dove alcune persone si incontrano per uno scambio di idee. Per il sistema della pompa di insulina, le persone che potrebbero essere coinvolte sono i dottori, gli assistenti medici, gli ingegneri e i progettisti del software.

Il sistema della pompa di insulina, descritto nel Capitolo 1, è un sistema a sicurezza critica, in quanto un guasto potrebbe causare danni fisici o perfino la morte dell'utente di questo sistema. Gli incidenti che potrebbero verificarsi quando si usa questa macchina includono alcuni danni agli occhi, al cuore e ai reni, come conseguenza del mancato controllo del livello di zuccheri nel sangue, disfunzioni cognitive per il basso livello di zuccheri nel sangue o qualche altro problema fisico, come una reazione allergica.

Alcuni dei potenziali rischi connessi alla pompa di insulina sono:

- overdose di insulina (fallimento del servizio);
- dose insufficiente di insulina (fallimento del servizio);
- guasto nel sistema hardware di monitoraggio (fallimento del servizio);
- mancanza di corrente per batteria esaurita (elettrico);
- interferenze elettriche con altre apparecchiature medicali, come un pacemaker (elettrico);
- falso contatto tra sensore e attuatore a causa di un collegamento difettoso (fisico);
- rottura di parti della macchina collegate al corpo del paziente (fisico);

- infezione causata dall'introduzione della macchina (biologico);
- reazione allergica all'insulina o ai materiali utilizzati dalla macchina (biologico).

I rischi correlati al software di solito riguardano servizi che non vengono forniti dal sistema o l'assenza di monitoraggio e protezione da parte del sistema. I sistemi di monitoraggio e protezione possono essere inclusi in un dispositivo per rilevare quelle condizioni, quali un basso livello di carica della batteria, che potrebbero causare un fallimento del dispositivo.

Si può utilizzare un registro dei rischi per memorizzare i rischi che sono stati identificati, spiegando i motivi per i quali ciascun rischio è stato incluso nel registro. Questo registro è un documento legale importante perché contiene tutte le decisioni relative alla sicurezza per ciascun rischio. Può essere utilizzato per dimostrare che gli ingegneri dei requisiti hanno analizzato con attenzione e cura tutti i potenziali rischi. Nel caso di incidente, il registro dei rischi può essere utilizzato in una successiva inchiesta o procedimento legale per dimostrare che gli sviluppatori non sono stati negligenti nell'analisi della sicurezza del sistema.

12.2.2 Valutazione dei rischi

Il processo di valutazione dei rischi consiste essenzialmente nel capire i fattori che potrebbero rendere reale un rischio e le conseguenze di un incidente associato al concretizzarsi di tale rischio. Occorre fare questa analisi per capire se un rischio è una seria minaccia al sistema o all'ambiente. L'analisi deve fornire anche una base per decidere come gestire le conseguenze di un rischio reale.

Per ogni rischio, il risultato dell'analisi e il processo di classificazione producono una dichiarazione di accettabilità, che è espressa in termini di rischio, dove il rischio tiene conto della probabilità che si verifichi un incidente e delle sue conseguenze. Esistono tre categorie di rischi.

1. *Rischi intollerabili* – Nei sistemi a sicurezza critica sono i rischi che minacciano la vita umana. Il sistema deve essere progettato in modo che questi rischi non possano verificarsi; ma se questo non possibile, il sistema deve avere apposite funzioni che garantiscono che tali rischi siano identificati prima che possano causare un incidente. Nel caso della pompa di insulina, un rischio intollerabile è che venga somministrata una dose eccessiva di insulina.
2. *Rischi bassi o con conseguenze accettabili (ALARP, As Low As Reasonably Practical)* – Sono i rischi che hanno conseguenze lievi oppure gravi ma con una bassa probabilità di realizzazione. Il sistema dovrebbe essere progettato in modo che la probabilità che si verifichi un incidente sia ridotta al minimo, compatibilmente con altri fattori, quali i costi e la consegna del sistema software. Un rischio ALARP per la pompa di insulina potrebbe essere il

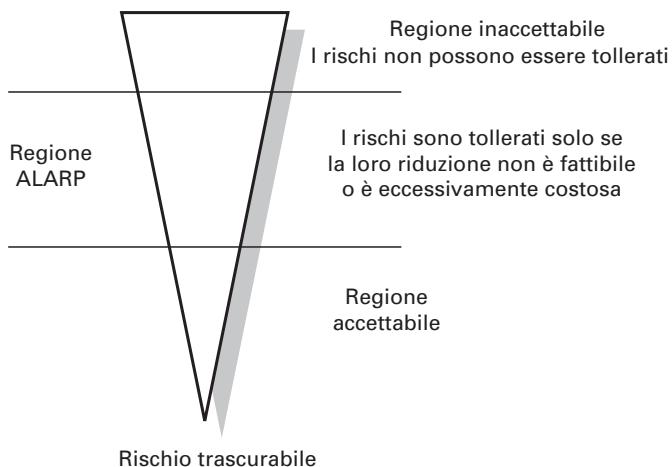


Figura 12.3 Il triangolo dei rischi.

guasto dell'hardware di monitoraggio. Le conseguenze di questo guasto sono, nel caso peggiore, una somministrazione insufficiente di insulina, che non provoca gravi danni.

3. *Rischi accettabili* – Sono quelli i cui incidenti associati di solito producono danni molto lievi. I progettisti dei sistemi dovrebbero compiere tutti i passi necessari per ridurre il numero dei rischi “accettabili”, purché ciò non comporti un significativo aumento dei costi, dei tempi di consegna o degli attributi non funzionali del sistema. Un rischio accettabile nel caso della pompa di insulina potrebbe essere l’eventualità di una reazione allergica nel paziente. Questa reazione di solito provoca soltanto una lieve irritazione cutanea. Potrebbe non essere economicamente conveniente utilizzare materiali speciali troppo costosi per ridurre questo tipo di rischio.

La Figura 12.3 mostra queste tre regioni. La larghezza del triangolo rispecchia i costi necessari per garantire che i rischi non producano incidenti. I costi più alti si hanno in corrispondenza dei rischi nella parte superiore del diagramma; i costi più bassi sono correlati ai rischi nel vertice inferiore del triangolo.

I confini tra le regioni della Figura 12.3 non sono fissi, ma dipendono dal grado di accettazione dei rischi da parte delle aziende nelle quali il sistema sarà installato. Questo varia da paese a paese – alcune aziende sono più avverse al rischio di altre. Col passare del tempo, però, tutte le aziende sono diventate più avverse al rischio, quindi i confini si sono spostati verso il basso. Per eventi rari, i costi economici di accettazione dei rischi e di risarcimento in caso di incidenti potrebbero essere minori dei costi richiesti per prevenire gli incidenti. Tuttavia, i clienti potrebbero richiedere sistemi con probabilità di incidenti molto basse, indipendentemente dai costi necessari per realizzare tali sistemi.

Pericolo rilevato	Probabilità del pericolo	Probabilità del pericolo	Rischio stimato	Accettabilità
1. Calcolo di una dose eccessiva di insulina	Media	Alta	Alto	Intollerabile
2. Calcolo di una dose insufficiente di insulina	Media	Bassa	Basso	Accettabile
3. Mancanza di corrente nell'hardware di monitoraggio	Media	Media	Basso	ALARP
4. Interruzione dell'energia elettrica	Alta	Bassa	Basso	Accettabile
5. Collegamento errato della macchina	Alta	Alta	Alto	Intollerabile
6. Rottura di parti collegate al paziente	Bassa	Alta	Medio	ALARP
7. Macchina che causa infezione	Media	Media	Medio	ALARP
8. Interferenza elettrica	Bassa	Alta	Medio	ALARP
9. Reazione allergica	Bassa	Bassa	Basso	Accettabile

Figura 12.4 Classificazione dei rischi per la pompa di insulina.

Per esempio potrebbe essere più economico per un'azienda rimediare all'inquinamento nei rari casi in cui questo avvenga, anziché installare sistemi di antinquinamento. Tuttavia, poiché l'opinione pubblica e i media non tollerano più questo tipo di incidenti, non è più accettabile rimediare al danno, anziché prevenirlo. Eventi in altri sistemi possono portare a una riclassificazione dei rischi. Per esempio, i rischi che erano ritenuti improbabili (quelli nella regione ALARP) potrebbero essere riclassificati come intollerabili a causa di eventi esterni, come gli attacchi terroristici, o fenomeni naturali, come gli tsunami.

La Figura 12.4 riporta una classificazione dei rischi per i pericoli identificati nel precedente paragrafo per il sistema della pompa di insulina. Ho separato i pericoli che sono correlati a calcoli errati dell'insulina (sovradosaggio o sottodosaggio). Un sovradosaggio di insulina è potenzialmente più pericoloso di un sottodosaggio nel breve termine. Un'overdose di insulina può provocare disfunzioni cognitive, stati di coma e perfino la morte. Una dose insufficiente di insulina può aumentare i livelli degli zuccheri nel sangue. Nel breve termine, questi alti livelli possono causare stanchezza, ma senza gravi conseguenze; nel lungo termine, però, possono provocare seri problemi al cuore, ai reni e agli occhi.

I rischi 4-9 nella Figura 12.4 non riguardano il software, ma nonostante questo il software svolge un ruolo nell'identificazione del rischio. Il software di monitoraggio dell'hardware dovrebbe monitorare lo stato del sistema e avvertire in caso

di potenziali problemi. L'avvertimento spesso permette di rilevare il pericolo prima che si verifichi un incidente. Esempi di pericoli che potrebbero essere rilevati sono l'interruzione di energia elettrica, che viene rilevata monitorando la batteria, e l'errato collegamento della macchina, che può essere rilevato monitorando i segnali provenienti dal sensore degli zuccheri nel sangue.

Il software di monitoraggio del sistema, ovviamente, è correlato alla sicurezza. Se un pericolo non viene rilevato, si può verificare un incidente. Se il software di monitoraggio fallisce, ma l'hardware funziona correttamente, il fallimento non è grave. Se, invece, il software di monitoraggio fallisce e il guasto dell'hardware non viene rilevato, le conseguenze potrebbero essere più serie.

La valutazione dei rischi richiede la stima delle probabilità e della severità dei pericoli. Questo è difficile in quanto i pericoli e gli incidenti non sono comuni. Di conseguenza, gli ingegneri potrebbero non avere un'esperienza diretta con gli incidenti che si sono verificati in precedenza. Nel calcolo delle probabilità e della severità dei pericoli, ha senso utilizzare termini relativi, come *probabile*, *improbabile*, *raro*, *alto*, *medio* e *basso*. Quantificare questi termini è praticamente impossibile, perché non sono disponibili sufficienti dati statistici per i principali tipi di incidenti.

12.2.3 Analisi dei rischi

L'analisi dei rischi ha lo scopo di scoprire le cause principali dei pericoli che gravano su un sistema a sicurezza critica. È compito dell'analista scoprire quali eventi o combinazioni di eventi potrebbero provocare il fallimento del sistema. Per fare questo, si può adottare un approccio top-down (dall'alto verso il basso) o bottom-up (dal basso verso l'alto). Le tecniche di analisi top-down deduttive, che sono più facili da usare, partono dal rischio e da qui cercano di identificare possibili fallimenti del sistema. Le tecniche bottom-up induittive partono da un possibile fallimento del sistema e da qui cercano di identificare quali rischi potrebbero derivare da tale fallimento.

Varie tecniche sono state proposte come possibili approcci alla scomposizione o analisi dei rischi (Storey 1996). Una delle tecniche più utilizzate è l'analisi dell'albero dei guasti, una tecnica top-down che è stata sviluppata per analizzare i rischi software e hardware (Leveson, Cha e Shimeall 1991). È una tecnica abbastanza facile da capire perché non richiede conoscenze specialistiche dei domini applicativi.

Per eseguire un'analisi dell'albero dei guasti, si parte dai rischi che sono stati identificati. Per ciascun rischio si procede all'indietro per scoprire le possibili cause. Si pone il rischio alla radice dell'albero e si identificano gli stati del sistema che possono produrre tale rischio. Per ciascuno di questi stati si identificano gli ulteriori stati del sistema che possono condurre allo stato che si sta analizzando. Questa scomposizione continua finché non si raggiungono le cause principali di un rischio. I rischi che possono nascere soltanto da una combinazione di cause

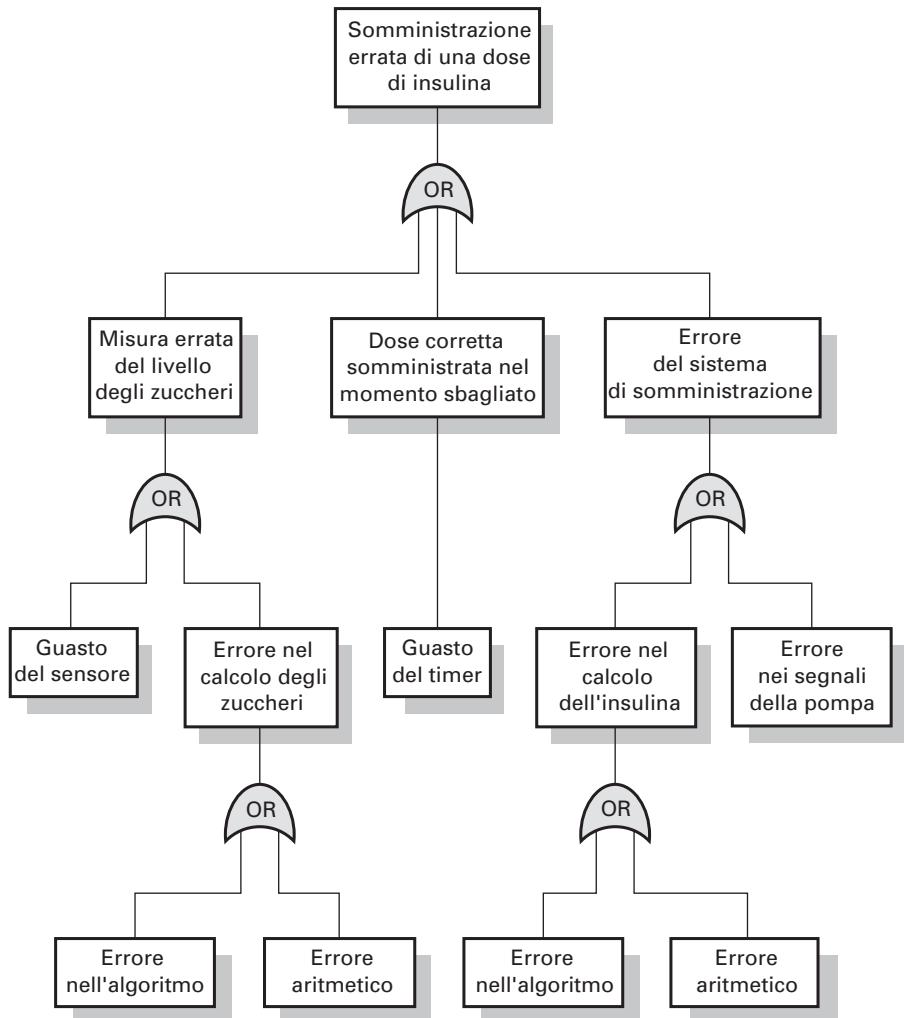


Figura 12.5 Esempio di albero dei guasti.

principali di solito hanno meno probabilità di causare incidenti rispetto a quelli che hanno una sola causa principale.

La Figura 12.5 illustra un albero dei guasti per i rischi correlati al software nel sistema che controlla la somministrazione dell'insulina. In questo caso, ho unito in un unico rischio il sovradosaggio e il sottodosaggio dell'insulina, ovvero “somministrazione errata di una dose di insulina”. Questo riduce il numero di alberi dei guasti che sono richiesti. Ovviamente, quando specifichiamo come dovrebbe reagire il software a questo rischio, dobbiamo distinguere una dose eccessiva di insulina da una dose insufficiente. Come detto in precedenza, i due casi non hanno le stesse conseguenze – in breve, un’overdose è più pericolosa.

Osservando la Figura 12.5 possiamo le seguenti considerazioni.

1. Tre condizioni potrebbero portare alla somministrazione errata di una dose di insulina. (1) Il livello di zuccheri nel sangue può essere misurato in modo errato, e quindi la quantità di insulina viene calcolata in base a un input sbagliato. (2) Il sistema di somministrazione dell'insulina potrebbe non rispondere correttamente ai comandi che specificano la quantità di insulina da iniettare. (3) La dose è calcolata correttamente, ma viene somministrata troppo presto o troppo tardi.
2. Il ramo sinistro dell'albero dei guasti, relativo alla misura errata del livello di zuccheri nel sangue, illustra come questo potrebbe accadere. Le cause di questo errore della misura potrebbero essere due: il guasto del sensore che fornisce l'input per calcolare il livello di zuccheri; un errore nel calcolo del livello di zuccheri. Questo valore viene calcolato da qualche parametro misurato, come la conduttività della pelle. Le cause di un errore nel calcolo potrebbero essere due: un errore nell'algoritmo; un errore aritmetico derivante dall'uso di numeri floating point.
3. Il ramo centrale dell'albero dei guasti riguarda i problemi connessi ai tempi di somministrazione, che possono derivare soltanto da un guasto del timer del sistema.
4. Il ramo destro dell'albero, che riguarda gli errori del sistema di somministrazione dell'insulina, mostra le possibili cause di questo tipo di errori. Le cause potrebbero essere un errore nel calcolo dell'insulina o un errore nei segnali della pompa che inietta l'insulina. L'errore nel calcolo potrebbe essere causato da un errore nell'algoritmo o da un errore aritmetico.

Gli alberi dei guasti sono utilizzati anche per identificare i potenziali problemi dell'hardware. L'albero dei guasti hardware mostra i requisiti che dovrebbe avere il software per rilevare e correggere questi problemi. Per esempio, le dosi di insulina non vengono somministrate frequentemente – non più di cinque o sei volte all'ora e, in alcuni casi, anche meno. Quindi, la capacità del processore è disponibile per eseguire i programmi di diagnostica. Gli errori hardware, che si verificano nei sensori, nella pompa o nel timer, possono essere identificati e segnalati prima che producano gravi effetti sul paziente.

12.2.4 Riduzione dei rischi

Una volta identificati i rischi e le loro cause principali, siamo in grado di definire i requisiti di sicurezza che gestiscono i rischi e assicurano che non si verifichino incidenti. Ci sono per questo tre possibili strategie.

1. *Evitare i rischi* – Il sistema viene progettato in modo che il rischio non possa presentarsi.
2. *Individuazione e rimozione dei rischi* – Il sistema è progettato in modo che i rischi siano individuati e neutralizzati prima che possano causare incidenti.

3. *Limitazione dei danni* – Il sistema viene progettato in modo da minimizzare le conseguenze di un incidente.

Normalmente i progettisti dei sistemi critici usano una combinazione di questi approcci. Nei sistemi a sicurezza critica i rischi intollerabili possono essere gestiti minimizzandone la probabilità e aggiungendo un sistema di protezione (Capitolo 11) che fornisca un backup di sicurezza. Per esempio, il sistema di controllo di un impianto chimico tenterà di identificare ed evitare una pressione eccessiva nei reattori. Sarebbe necessario anche un sistema di protezione indipendente che monitorizza la pressione e apre una valvola di sfogo se viene rilevata una pressione troppo alta.

Nel sistema della pompa di insulina, uno “stato sicuro” è uno stato di spegnimento in cui non viene somministrata insulina. In un periodo di tempo breve, questa condizione non è una minaccia alla salute del diabetico. Per gli errori del software che potrebbero determinare una dose errata di insulina, potrebbero essere sviluppate le seguenti soluzioni.

1. *Errore aritmetico* – Si verifica quando alcuni calcoli aritmetici causano un errore nella rappresentazione. La specifica deve identificare tutti gli errori aritmetici che potrebbero verificarsi e definire il gestore delle eccezioni che dovrà essere incluso per ciascuno di tali errori; dovrà anche descrivere l’azione da svolgere per ciascuno di questi errori. L’azione sicura di default è spegnere il sistema di somministrazione dell’insulina e attivare un allarme d’avvertimento.
2. *Errore nell’algoritmo* – Questo è un caso più difficile, in quanto non c’è un’eccezione chiara del programma che può essere gestita. Questo tipo di errore potrebbe essere identificato confrontando la dose di insulina richiesta con quella precedentemente somministrata; se la differenza è molto grande, allora potrebbe esserci stato un errore nel calcolo della quantità. Il sistema potrebbe anche registrare la sequenza delle dosi; se il sistema ha somministrato un numero di dosi con quantità superiori alla media, si potrebbe attivare un allarme e ridurre i dosaggi successivi.

Alcuni requisiti di sicurezza per il software della pompa di insulina sono riportati nella Figura 12.6; si tratta di requisiti dell’utente. Naturalmente, questi requisiti dovrebbero essere espressi in modo più dettagliato in una specifica più dettagliata dei requisiti del sistema.

12.3 Processi di ingegneria della sicurezza

I processi utilizzati per sviluppare il software a sicurezza critica si basano sui processi utilizzati nell’ingegneria dell’affidabilità del software. In generale, occorre molta cura per sviluppare una specifica del sistema completa e dettagliata. La progettazione e l’implementazione del sistema di solito seguono un modello a cascata basato su piani, con revisioni e verifiche in ogni fase del processo.

SR1	Il sistema non dovrebbe somministrare dosi di insulina superiori alla dose massima specificata per un utente del sistema.
SR2	Il sistema non dovrebbe somministrare una dose cumulativa giornaliera superiore alla dose massima specificata per un utente del sistema.
SR3	Il sistema dovrebbe includere una funzionalità di diagnostica dell'hardware da eseguire almeno quattro volte all'ora.
SR4	Il sistema dovrebbe includere un gestore di tutte le eccezioni identificate nella Tabella 3.
SR5	L'allarme acustico dovrebbe essere attivato ognqualvolta viene individuata un'anomalia hardware o software; dovrebbe essere visualizzato anche un messaggio diagnostico come definito nella Tabella 4.
SR6	Nel caso di un allarme del sistema, il sistema di somministrazione dell'insulina dovrebbe essere sospeso finché l'utente non avrà resettato il sistema e rimosso l'allarme.

Nota: le Tabelle 3 e 4 sono incluse nella documentazione dei requisiti; non sono qui riportate.

Figura 12.6 Esempi di requisiti di sicurezza.

Questo processo è realizzato applicando le tecniche di *fault avoidance* (evitare l'introduzione di errori nel software) e *fault detection* (identificare gli errori). Per alcuni tipi di sistemi, come quelli installati negli aerei, possono essere utilizzate le architetture fault-tolerant, descritte nel Capitolo 11.

L'affidabilità è un prerequisito per i sistemi a sicurezza critica. A causa dei costi molto alti e delle conseguenze potenzialmente tragiche dei fallimenti dei sistemi, è possibile aggiungere altre attività di verifica nello sviluppo dei sistemi a sicurezza critica. Queste attività possono includere lo sviluppo di modelli formali per un sistema, l'analisi per scoprire errori e incongruenze, e l'utilizzo di strumenti di analisi statica che controllano il codice sorgente per scoprire potenziali errori.

I sistemi sicuri devono essere affidabili, ma, come ho detto in precedenza, l'affidabilità non è sufficiente. Errori e omissioni nei requisiti e nelle verifiche potrebbero rendere insicuri i sistemi affidabili. Pertanto, i processi di sviluppo dei sistemi a sicurezza critica dovrebbero includere un'attività di revisione della sicurezza, nella quale gli ingegneri e gli stakeholder esaminano il lavoro svolto e controllano esplicitamente i problemi potenziali che potrebbero compromettere la sicurezza del sistema.

Alcuni tipi di sistemi a sicurezza critica sono regolamentati, come ho spiegato nel Capitolo 10. Gli enti di regolamentazione nazionali e internazionali richiedono prove dettagliate sulla sicurezza dei sistemi. Queste prove possono includere i seguenti elementi.

1. La specifica del sistema che è stato sviluppato e le registrazioni delle verifiche effettuate in base a tale specifica.

2. La prova che sono stati svolti i processi di verifica e convalida e i risultati ottenuti da questi processi.
3. La prova che le società che hanno sviluppato il sistema hanno definito e seguito processi software fidati che includono le revisioni della garanzia di sicurezza. Devono esserci anche le registrazioni che dimostrano che questi processi sono stati appropriatamente applicati.

Non tutti i sistemi a sicurezza critica sono regolamentati. Per esempio, non sono stati ancora regolamentati i sistemi software installati nelle automobili, sebbene queste oggi abbiano molti sistemi software integrati. La sicurezza dei sistemi installati nelle auto è affidata alla responsabilità del costruttore delle auto. Tuttavia, a causa della possibilità di un'azione legale nel caso di incidente, gli sviluppatori di sistemi non regolamentati devono mantenere le stesse informazioni dettagliate sulla sicurezza. Se viene loro contestato il comportamento del sistema, devono essere in grado di dimostrare che non sono stati negligenti nello sviluppo del software per quel tipo di auto.

La necessità di ampliare questo processo e di documentare i prodotti software è un'altra ragione per la quale non è possibile applicare i processi agili, senza modifiche significative, allo sviluppo dei sistemi a sicurezza critica. I processi agili si concentrano sul software e i loro sostenitori affermano (giustamente) che la maggior parte della documentazione sui processi non sarà mai utilizzata dopo che il software sarà stato prodotto. Tuttavia, ci sono casi in cui per motivi legali o normativi, occorre conservare la documentazione sui processi utilizzati e sul sistema stesso.

I sistemi a sicurezza critica, come altri tipi di sistemi che richiedono elevati livelli di fidatezza, devono basarsi su processi fidati (Capitolo 10). Un processo fidato di solito include attività, quali la gestione dei requisiti, la gestione delle modifiche, il controllo della configurazione, la modellazione del sistema, le revisioni e le ispezioni, la pianificazione e l'analisi dei test. Quando un sistema è a sicurezza critica, potrebbero essere richiesti alcuni processi aggiuntivi per garantire la sicurezza e per la verifica e l'analisi del software.

12.3.1 Processi per la garanzia della sicurezza

Per garanzia della sicurezza s'intende una serie di attività che controllano se un sistema potrà operare in sicurezza. Queste attività dovrebbero essere inserite in tutte le fasi di sviluppo del software. Dovrebbero essere registrate tutte le analisi della sicurezza e le persone che le hanno eseguite. Le attività di garanzia della sicurezza devono essere dettagliatamente documentate. Questa documentazione potrebbe far parte delle prove che dimostrano al responsabile di un ente di regolamentazione o al proprietario del sistema che il sistema è stato sviluppato per operare in sicurezza.

Ecco alcuni esempi di attività di garanzia della sicurezza.

1. *Analisi e monitoraggio dei rischi.* I rischi vengono tracciati partendo da un'analisi preliminare fino al test e alla convalida del sistema.
2. *Revisioni della sicurezza.* Sono attività che si svolgono durante tutto il processo di sviluppo.
3. *Certificazione della sicurezza.* Viene certificata la sicurezza dei componenti critici. Questa attività riguarda un gruppo esterno al team di sviluppo del sistema che esamina le prove disponibili e decide se il sistema o un suo componente può essere considerato sicuro prima che sia consegnato agli utenti.

Per supportare questi processi di garanzia della sicurezza, gli ingegneri del progetto dovrebbero scegliere i responsabili della sicurezza di un sistema. Queste persone, in caso di fallimento del sistema relativo alla sicurezza, dovranno dimostrare che le attività di garanzia della sicurezza sono state svolte in modo corretto.

Gli ingegneri della sicurezza collaborano con i manager della qualità per assicurare che venga utilizzato un sistema di gestione della configurazione dettagliato per tracciare tutta la documentazione relativa alla sicurezza e verificare che questa sia conforme alla documentazione tecnica associata. Non avrebbe senso avere procedure di convalida rigorose se un errore nella gestione della configurazione significa consegnare al cliente il sistema sbagliato. La gestione della qualità e la gestione della configurazione sono trattate, rispettivamente, nei Capitoli 21 e 22.

L'analisi dei rischi è una parte essenziale dello sviluppo dei sistemi a sicurezza critica. Essa richiede l'identificazione dei rischi, la loro probabilità di concretizzarsi e la probabilità che un rischio possa provocare un incidente. Se c'è un codice di programma che controlla e gestisce i singoli rischi, allora si può ritenere che questi rischi non provocheranno incidenti. Se è richiesta una certificazione esterna prima che un sistema venga utilizzato (per esempio, in un aereo), di solito una condizione della certificazione è che sia dimostrata questa tracciabilità.

Il documento di sicurezza più importante da produrre è il registro dei rischi. Questo documento fornisce le prove di come sono stati tenuti in considerazione i potenziali rischi durante lo sviluppo del software. Il registro dei rischi è utilizzato durante il processo di sviluppo del software per documentare come ogni singolo rischio è stato considerato in ogni fase di questo processo.

La Figura 12.7 mostra un esempio semplificato di registro dei rischi per il sistema di somministrazione dell'insulina. Il registro documenta il processo di analisi dei rischi e mostra i requisiti di progettazione che sono stati generati durante questo processo. Questi requisiti di progettazione hanno lo scopo di garantire che il sistema di controllo non possa mai rilasciare un'overdose di insulina a un paziente.

Registro dei rischi		Pagina 4: stampa 20.02.2012
Sistema: Pompa di insulina		File: InsulinePump/Safety/HazardLog
Ingegnere della sicurezza: Giorgio Rossi		Versione: 1/3
Pericolo individuato:	Somministrazione al paziente di un'overdose di insulina	
Identificato da:	Giovanna Bianchi	
Classe di gravità	1	
Rischio identificato	Alto	
Identificazione nell'albero dei guasti: Si	Data: 24.01.11	Posizione: Registro dei rischi, Pagina 5
Creatori dell'albero dei guasti: Giovanna Bianchi e Paolo Verdi		
Albero dei guasto controllato: Si	Data: 28.01.11	Verificatore: Giorgio Rossi
Requisiti di progettazione della sicurezza del sistema		
1.	Il sistema deve includere il software di test automatico che provi il sistema dei sensori, l'orologio e il sistema di somministrazione dell'insulina.	
2.	Il software di test automatico deve essere eseguito una volta al minuto.	
3.	Se il software di test automatico rileva un guasto in un componente del sistema, deve essere emesso un allarme acustico e il display della pompa dovrà indicare il nome del componente in cui è stato rilevato il guasto. La somministrazione dell'insulina deve essere sospesa.	
4.	Il sistema deve incorporare un sistema di riscrittura (<i>override</i>) che permette all'utente del sistema di modificare la dose calcolata di insulina che deve essere fornita dal sistema.	
5.	La dose di override non deve essere maggiore di un valore prestabilito (<i>maxOverride</i>); viene abilitata quando il sistema viene configurato dal personale medico.	

Figura 12.7 Esempio di pagina del registro dei rischi.

Le persone che sono responsabili della sicurezza dovrebbero essere esplicitamente identificate nel registro dei rischi. L'identificazione delle persone è importante per due motivi.

1. Quando le persone sono identificate, possono essere ritenute responsabili delle loro azioni. È probabile che esse siano più attente perché è più facile risalire al responsabile di un problema.
2. In caso di incidente, potrebbero esserci inchieste o procedimenti legali. È importante essere in grado di identificare i responsabili della garanzia della sicurezza in modo che possano difendere il loro operato in un eventuale processo legale.

I revisori della sicurezza controllano anche la specifica del software, il progetto e il codice sorgente, con l'obiettivo di scoprire condizioni potenzialmente rischiose. Non sono processi automatici, ma richiedono persone in grado di controllare attentamente gli errori che potrebbero essere stati commessi, verificando omissioni o cattive interpretazioni della specifica che potrebbero influire sulla sicurezza di un sistema. Per esempio, in un incidente aereo che ho descritto in precedenza,

Abilitazione degli ingegneri del software

In alcune aree dell'ingegneria, i tecnici che si occupano di sicurezza devono essere ingegneri abilitati. Non è consentito che ingegneri inesperti o poco qualificati possano assumersi la responsabilità della sicurezza. In 30 stati degli Stati Uniti, esiste una forma di abilitazione per gli ingegneri del software che si occupano di sviluppo di sistemi a sicurezza critica. Questi stati richiedono che le persone coinvolte nello sviluppo di software a sicurezza critica siano ingegneri abilitati, con un livello minimo di qualifica ed esperienza. È un tema controverso; per questo l'abilitazione non è richiesta in molti altri paesi.

<http://software-engineering-book.com/safety-licensing/>

un revisore della sicurezza avrebbe potuto discutere l'ipotesi che un aereo è a terra quando c'è un peso su entrambe le ruote e le ruote girano.

I revisori della sicurezza dovrebbero essere guidati dal registro dei rischi. Per ogni rischio identificato, un team di revisori esamina il sistema e giudica se esso può affrontare un rischio in modo sicuro. Qualsiasi dubbio deve essere riportato nel rapporto del team dei revisori e deve essere presentato al team di sviluppo del sistema. Il Capitolo 21 descrive dettagliatamente vari tipi di revisioni, che riguardano la garanzia della qualità del software.

La certificazione della garanzia della qualità serve quando vengono incorporati dei componenti esterni in un sistema a sicurezza critica. Se tutte le parti di un sistema sono sviluppate localmente, è possibile conservare tutte le informazioni sui processi di sviluppo. Tuttavia, non è economicamente conveniente sviluppare componenti che sono già disponibili presso altri fornitori. Quando si sviluppano sistemi a sistema, il problema è che i componenti esterni potrebbero essere stati sviluppati con standard differenti da quelli sviluppati localmente. La loro sicurezza non è nota.

Di conseguenza, potrebbe essere necessario che tutti i componenti esterni debbano essere certificati prima che possano essere integrati nel sistema. Il team di certificazione della sicurezza, che è distinto da quello di sviluppo, svolge un'attenta analisi per verificare e convalidare i componenti. Se necessario, contattano gli sviluppatori dei componenti per controllare che questi abbiano seguito processi fidati per creare i componenti e per esaminare il codice sorgente dei componenti. Una volta che i membri del team di certificazione della sicurezza sono convinti che un componente soddisfa la sua specifica e non ha funzionalità "nascoste", possono emettere un certificato che permette al componente di essere impiegato in un sistema a sicurezza critica.

12.3.2 Verifica formale

I metodi formali per lo sviluppo del software, come detto nel Capitolo 10, si basano su un modello formale del sistema che opera come una specifica del sistema. Questi metodi formali riguardano principalmente l'analisi matematica della specifica, trasformando la specifica in una rappresentazione semanticamente

equivalente più dettagliata o verificando formalmente che una trasformazione del sistema sia semanticamente equivalente a un'altra rappresentazione.

La necessità di garantire la sicurezza nei sistemi a sicurezza critica è stata una delle linee guida principali nello sviluppo dei metodi formali. Eseguire tutti i possibili test per un sistema è estremamente costoso e non si ha la certezza di scoprire tutti gli errori del software. Numerosi sistemi ferroviari a sicurezza critica sono stati sviluppati utilizzando metodi formali negli anni '90 (Dehbonei e Mejia 1995; Behm et al. 1999). Società come Airbus utilizzano normalmente i metodi formali per sviluppare il software per i loro sistemi critici (Souyris et al. 2009).

I metodi formali possono essere utilizzati in varie fasi del processo di verifica e convalida.

1. Una specifica formale del sistema può essere sviluppata e analizzata matematicamente per rilevare eventuali incongruenze. Questa tecnica è efficace per scoprire errori e omissioni nella specifica. La verifica dei modelli, descritta nel prossimo paragrafo, è un metodo particolarmente efficace per l'analisi della specifica di un sistema.
2. È possibile verificare formalmente, tramite dimostrazioni matematiche, che il codice di un sistema software è coerente con la sua specifica. Questo richiede una specifica formale. È efficace per scoprire errori di programmazione e di progettazione.

A causa del notevole gap semantico tra la specifica formale di un sistema e il codice del programma, è difficile e costoso dimostrare che un programma sviluppato separatamente sia coerente con la sua specifica. La verifica di un programma oggi si basa principalmente su un processo di sviluppo trasformativo, una specifica formale viene sistematicamente trasformata attraverso una serie di rappresentazioni in codice di programma. Gli strumenti software supportano lo sviluppo delle trasformazioni e aiutano a verificare che le corrispondenti rappresentazioni del sistema siano coerenti. Il metodo B è probabilmente il metodo trasformativo formale più largamente utilizzato (Abrial 2010). È stato utilizzato nello sviluppo dei sistemi di controllo dei treni e del software avionico.

I sostenitori dei metodi formali ritengono che l'uso di questi metodi permette di realizzare sistemi più affidabili e sicuri. La verifica formale dimostra che il programma sviluppato soddisfa la sua specifica e che gli errori di implementazione non comprometteranno la fidatezza del sistema. Se sviluppati un modello formale di sistemi concorrenti utilizzando una specifica scritta in un linguaggio come CSP (Schneider 1999), potrete scoprire le condizioni che potrebbero causare un deadlock nel programma finale, e sarete in grado di risolvere questo tipo di problema. Questo è molto difficile da fare effettuando soltanto i test.

Tuttavia, le specifiche e le verifiche formali non garantiscono che il software sia affidabile nell'uso pratico.

1. La specifica potrebbe non riflettere i reali requisiti degli utenti e di altri stakeholder del sistema. Come detto nel Capitolo 10, gli stakeholder raramente capiscono le notazioni formali, quindi non possono leggere direttamente la specifica formale per trovare errori ed omissioni. Questo significa che è probabile che la specifica formale non sia una rappresentazione accurata dei requisiti del sistema.
2. La verifica può contenere errori. Le verifiche dei programmi sono estese e complesse, quindi, come i programmi estesi e complessi, di solito contengono errori.
3. La verifica può contenere ipotesi sbagliate sul modo in cui il sistema viene utilizzato. Se il sistema non viene utilizzato come previsto, il comportamento del sistema non rientra nel campo di azione della verifica.

La verifica di un sistema software complesso richiede una notevole quantità di tempo; occorrono matematici esperti e strumenti software specializzati, come i programmi che dimostrano i teoremi. È un processo costoso e, all'aumentare della dimensione del sistema, i costi della verifica formale crescono in modo sproporzionato.

Molti ingegneri del software quindi ritengono che la verifica formale non sia economicamente conveniente. Essi credono che lo stesso livello di fiducia nel sistema può essere raggiunto in modo più economico utilizzando altre tecniche di convalida, come le ispezioni e i test del sistema. Tuttavia, società come Airbus che utilizzano la verifica formale sostengono che non è richiesto il test dei singoli componenti, ottenendo così un notevole risparmio di costi (Moy et al. 2013).

Io sono convinto che i metodi formali e la verifica formale giochino un ruolo importante nello sviluppo dei sistemi a sicurezza critica. Le specifiche formali sono molto efficaci per identificare alcuni tipi di problemi che potrebbero causare fallimenti del sistema. Sebbene la verifica formale resti impraticabile per i grandi sistemi, tuttavia può essere utilizzata per verificare i componenti principali a sicurezza critica e protezione critica.

12.3.3 Verifica dei modelli

La verifica formale dei programmi tramite un approccio deduttivo è difficile e costosa, tuttavia sono stati sviluppati approcci alternativi all'analisi formale che si basano su un concetto più rigoroso di correttezza. Il più diffuso di questi approcci è la “verifica dei modelli” (Jhala e Majumdar 2009), che richiede la creazione di un modello formale degli stati di un sistema e la verifica della correttezza di questo modello tramite speciali strumenti software. Le fasi di questo processo di verifica sono illustrate nella Figura 12.8.

La verifica dei modelli è stata largamente utilizzata per controllare i progetti dei sistemi hardware. Il suo impiego si sta progressivamente diffondendo nei sistemi software critici, come il software di controllo nei veicoli della NASA per

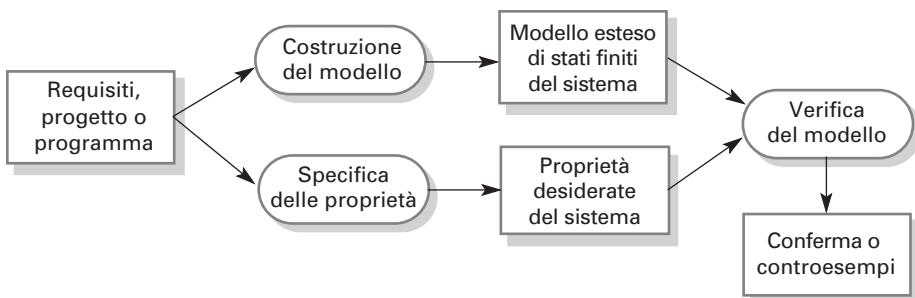


Figura 12.8 Verifica dei modelli.

l'esplorazione di Marte (Regan e Hamilton 2004; Holzmann 2014) e nello sviluppo di software avionico per gli aerei Airbus (Bochot et al. 2009).

Sono stati sviluppati vari strumenti di verifica dei modelli. SPIN è stato uno dei primi esempi di strumenti per il controllo dei modelli (Holzmann 2003). Tra i sistemi più recenti figurano SLAM di Microsoft (Ball, Levin e Rajamani 2011) e PRISM (Kwiatkowska, Norman e Parker 2011).

I modelli utilizzati dai sistemi di verifica dei modelli sono modelli estesi di stati finiti del software. I modelli sono espressi nel linguaggio del sistema utilizzato per la verifica dei modelli – per esempio, SPIN usa un linguaggio chiamato Promela. Le proprietà desiderate per il sistema vengono identificate e scritte in una notazione formale, di solito basata su una logica temporale. Per esempio, nel sistema di controllo delle stazioni meteorologiche in un'area selvaggia, una proprietà da controllare potrebbe essere quella che il sistema potrà sempre passare dallo stato di “registrazione” allo stato di “trasmissione”.

Il software che controlla il modello esamina poi tutti i percorsi all'interno del modello (ovvero tutte le possibili transizioni di stato), verificando che la proprietà sia mantenuta in ciascun percorso. Se ciò è vero, il software di controllo conferma che il modello è corretto rispetto a tale proprietà. Se, invece, la proprietà non è mantenuta in un particolare percorso, il software di controllo produce un controesempio che illustra dove la proprietà non è presente. La verifica dei modelli è particolarmente utile nella convalida dei sistemi concorrenti, i cui test sono notoriamente difficili da eseguire a causa della loro sensibilità al tempo. Il software di controllo può esplorare le transizioni concorrenti intrecciate e scoprire i problemi potenziali.

Un elemento chiave nella verifica dei modelli è la creazione del modello del sistema. Se il modello deve essere creato manualmente (dal documento dei requisiti o di progettazione), il processo è costoso, in quanto la creazione del modello richiede parecchio tempo. In aggiunta, c'è il rischio che il modello creato non sarà un modello accurato dei requisiti o del progetto. Pertanto, sarebbe meglio creare il modello automaticamente dal codice sorgente del programma. I programmi di controllo dei modelli disponibili possono operare direttamente con i linguaggi Java, C, C++ e Ada.

La verifica dei modelli è molto costosa in termini di elaborazioni da svolgere, in quanto usa un approccio integrale per controllare tutti i percorsi all'interno del modello del sistema. Al crescere della dimensione del sistema, aumenta il numero di stati, con conseguente aumento del numero di percorsi da controllare. Per i grandi sistemi, quindi, la verifica dei modelli potrebbe essere impraticabile, a causa del tempo richiesto per eseguire tutti questi controlli. Tuttavia, alcune società stanno sviluppando algoritmi migliori che sono in grado di identificare le parti dello stato di un sistema che non necessitano di essere esplorate durante la verifica di una particolare proprietà. Quando questi algoritmi saranno integrati nel software di controllo, sarà possibile utilizzare la verifica dei modelli anche nello sviluppo di sistemi critici su larga scala.

12.3.4 Analisi statica dei programmi

Gli analizzatori statici automatici sono strumenti software che esaminano il testo sorgente di un programma e rilevano i possibili errori e anomalie. Analizzano il testo sorgente e riconoscono i vari tipi di istruzioni in un programma. Possono capire se un'istruzione è formata correttamente, possono fare delle inferenze sul flusso di controllo di un programma e, in molti casi, sono in grado di calcolare l'insieme di tutti i possibili valori dei dati di un programma. Svolgono un compito complementare al rilevamento degli errori svolto dal compilatore di un linguaggio di programmazione e possono essere utilizzati come parte del processo di ispezione o come attività distinta del processo di verifica e convalida.

L'analisi statica automatica è più veloce ed economica delle revisioni dettagliate del codice ed è molto efficace per scoprire alcuni tipi di errori dei programmi. Tuttavia, non è in grado di scoprire alcune classi di errori che potrebbero essere identificate nelle ispezioni dei programmi.

Gli strumenti di analisi statica (Lopes, Vicente e Silva 2009) operano sul codice sorgente di un sistema e, almeno per qualche tipo di analisi, non richiedono input aggiuntivi. Questo significa che i programmatore non hanno bisogno di imparare notazioni particolari per scrivere le specifiche dei programmi, quindi i benefici dell'analisi sono subito chiari; ne consegue che è più semplice introdurre l'analisi statica automatica in un processo di sviluppo rispetto alla verifica formale o alla verifica dei modelli.

L'obiettivo dell'analisi statica automatica è quello di focalizzare l'attenzione di un lettore di codice sulle anomalie di un programma, come le variabili che sono utilizzate senza essere inizializzate, le variabili inutilizzate o i dati i cui valori potrebbero eccedere i limiti consentiti. La Figura 12.9 riporta alcuni esempi di problemi che possono essere identificati dall'analisi statica.

Ovviamente, le verifiche specifiche fatte dall'analizzatore statico dipendono dal linguaggio di programmazione utilizzato e da ciò che è consentito nel linguaggio. Le anomalie spesso sono il risultato di omissioni o errori di programmazione, quindi mettono in evidenza gli elementi che potrebbero fallire quando viene ese-

Classe di errore	Verifica dell'analisi statica
Errori dei dati	Variabili utilizzate senza inizializzazione Variabili dichiarate ma inutilizzate Variabili assegnate due volte ma mai utilizzate tra le assegnazioni Violazioni dei limiti degli array Variabili non dichiarate
Errori di controllo	Codice irraggiungibile Salti incondizionati nei cicli
Errori di input/output	Variabili restituite due volte senza un'assegnazione che si interpone
Errori di interfaccia	Tipi di parametri non corrispondenti Numero di parametri non corrispondente Risultati di una funzione inutilizzati Funzioni e procedure mai chiamate
Errori di gestione della memoria	Puntatori non assegnati Aritmetica dei puntatori Memory leak (perdita di memoria)

Figura 12.9 Verifiche dell'analisi statica automatica.

guito il programma. Queste anomalie, tuttavia, non sono necessariamente errori del programma; potrebbero essere costrutti appositamente introdotti dal programmatore; inoltre, non tutte le anomalie hanno conseguenze negative.

Tre livelli di verifica possono essere implementati negli analizzatori statici.

1. *Verifica degli errori caratteristici.* A questo livello, l'analizzatore statico conosce gli errori più comuni che vengono commessi dai programmatori nei linguaggi come Java o C. Lo strumento analizza il codice per trovare schemi che sono tipici di un particolare problema e li segnala al programmatore. Anche se relativamente semplice, l'analisi basata sugli errori caratteristici può essere economicamente molto conveniente. Zheng e i suoi collaboratori (Zheng et al. 2006) hanno analizzato un'ampia base di codice in C e C++; hanno scoperto che il 90% degli errori nei programmi derivava da 10 errori caratteristici.
2. *Verifica degli errori definiti dall'utente.* In questo approccio, gli utenti dell'analizzatore statico definiscono gli schemi di errore da rilevare. Questi tipi di errori sono correlati al dominio delle applicazioni oppure si basano sulla conoscenza del particolare sistema che si sta sviluppando. Un esempio di schema di errore è “mantenere l'ordine”; per esempio, il metodo A deve essere chiamato sempre prima del metodo B. Una società potrebbe raccogliere le informazioni sugli errori più comuni che si verificano nei suoi programmi ed estendere gli strumenti di analisi statica con gli schemi di errori per mettere in evidenza questi errori.

3. *Verifica delle asserzioni.* Questo è il metodo più generale e potente dell'analisi statica. Gli sviluppatori includono nel loro programma alcune asserzioni formali (spesso scritte come commenti stilizzati) che stabiliscono le relazioni che devono essere rispettate in quel punto del programma. Per esempio, il programma potrebbe includere un'asserzione che stabilisce che il valore di una variabile deve essere compreso nell'intervallo $x..y$. L'analizzatore esegue simbolicamente il codice e mette in evidenza le istruzioni nelle quali tale asserzione non è vera.

L'analisi statica è efficace nel trovare gli errori nei programmi, ma di solito genera un gran numero di falsi positivi, che sono sezioni di codice in cui non ci sono errori, ma dove le regole dell'analizzatore statico fanno sì che vengano rilevati potenziali errori. Il numero di falsi positivi può essere ridotto aggiungendo più informazioni al programma sotto forma di asserzioni, ma questo richiede un lavoro aggiuntivo da parte dello sviluppatore del codice. Il lavoro deve essere fatto selezionando questi falsi positivi prima che il codice stesso possa essere sottoposto al controllo degli errori.

Molte società oggi usano l'analisi statica nei loro processi di sviluppo del software. Microsoft ha introdotto l'analisi statica nello sviluppo dei driver dei dispositivi, nei quali un errore del programma può avere effetti molto gravi. Hanno esteso l'approccio a una gamma più ampia del loro software per identificare i problemi di protezione come pure gli errori che influiscono sull'affidabilità dei programmi (Ball, Levin e Rajamani 2011). La verifica dei problemi più noti, come l'overflow dei buffer, è efficace per migliorare la protezione, in quanto gli hacker spesso basano i loro attacchi su queste tipiche vulnerabilità. Gli attacchi possono essere rivolti a sezioni di codice poco utilizzate che non sono state integralmente testate. L'analisi statica è un modo economicamente conveniente per trovare questi tipi di vulnerabilità.

12.4 Casi di sicurezza

Come detto in precedenza, molti sistemi a sicurezza critica, con uso intensivo del software, sono regolamentati. Gli enti di regolamentazione hanno una significativa influenza sui processi di sviluppo e sull'installazione di questi sistemi. Questi enti sono istituzioni governative che hanno la responsabilità di garantire che le aziende private non consegnino sistemi che possano minacciare la sicurezza pubblica e ambientale o l'economia nazionale. I proprietari dei sistemi a sicurezza critica devono convincere le autorità di regolamentazione che hanno fatto il massimo possibile per garantire che i loro sistemi sono sicuri. Gli enti di regolamentazione definiscono il caso di sicurezza per un sistema critico, specificando la prova da fornire per dimostrare che il funzionamento normale del sistema non potrà arrecare danni all'utente.

Gli elementi della prova vengono raccolti durante il processo di sviluppo del sistema. La prova può includere informazioni sull'analisi e sulla riduzione dei rischi, risultati di test, l'analisi statistica, informazioni sui processi di sviluppo, verbali delle riunioni di revisione del software e così via. Tutti questi elementi vengono assemblati e organizzati in un caso di sicurezza, che è una rappresentazione dettagliata del perché i proprietari e gli sviluppatori di un sistema ritengono che il sistema sia sicuro.

Un caso di sicurezza è un insieme di documenti che include una descrizione del sistema da certificare, le informazioni sui processi utilizzati per sviluppare il sistema e le argomentazioni logiche che dimostrano che il sistema possa essere ritenuto sicuro. Bishop e Bloomfield (Bishop e Bloomfield 1998)² hanno definito in questo modo un caso di sicurezza:

“Un insieme di prove documentate che costituiscono una valida e convincente dimostrazione che il sistema è adeguatamente sicuro per una particolare applicazione in un determinato ambiente”.

L'organizzazione e i contenuti di un caso di sicurezza dipendono dal tipo di sistema che si sta certificando e dal contesto in cui il sistema dovrà operare. La Figura 12.10 mostra una possibile struttura di un caso di sicurezza. Non esistono standard universali per i casi di sicurezza. Le strutture possono variare in funzione del tipo e della maturità del dominio applicativo. Per esempio, i casi di sicurezza per le applicazioni nucleari esistono da molti anni. Sono molto complessi e completi e sono presentati in un modo che è familiare agli ingegneri nucleari. I casi di sicurezza per le apparecchiature medicali sono stati introdotti più di recente. La struttura dei casi di sicurezza è più flessibile, e i casi stessi sono meno dettagliati di quelli nucleari.

Un caso di sicurezza è riferito a un sistema nel suo complesso e, come parte del caso, potrebbe esserci un caso di sicurezza sussidiario. Quando si costruisce un caso di sicurezza per il software, occorre mettere in relazione i fallimenti del software con i fallimenti più generali del sistema e dimostrare che questi fallimenti non potranno mai verificarsi o non potranno propagarsi in modo tale che provocare altri fallimenti pericolosi per il sistema.

I casi di sicurezza sono documenti estesi e complessi e quindi sono costosi da produrre e mantenere. A causa di questi alti costi, gli sviluppatori dei sistemi a sicurezza critica devono tenere in considerazione i requisiti del caso di sicurezza durante il processo di sviluppo.

1. Graydon e altri (Graydon, Knight e Strunk 2007) hanno dimostrato che lo sviluppo di un caso di sicurezza dovrebbe essere integrato nella progettazione e nell'implementazione del sistema. Questo significa che le decisioni relative alla progettazione del sistema potrebbero essere influenzate dai

² Bishop, P. e R. E. Bloomfield. 1998. “A Methodology for Safety Case Development.” In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. <http://www.adelard.com/papers/ss98web.pdf>

Componente	Descrizione
Descrizione del sistema	Una panoramica del sistema e una descrizione dei suoi componenti critici.
Requisiti di sicurezza	I requisiti di sicurezza estratti dalla specifica dei requisiti del sistema. È possibile includere anche i dettagli di altri requisiti importanti del sistema.
Analisi dei pericoli e dei rischi	Documenti che descrivono i pericoli e i rischi che sono stati identificati e le contromisure per ridurre i rischi. Analisi e registro dei rischi.
Analisi del progetto	Un insieme di argomentazioni strutturate (si veda il Paragrafo 12.4.1) che dimostrano che il progetto è sicuro.
Verifica e convalida	Una descrizione delle procedure di verifica e convalida utilizzate e, ove appropriato, i piani di test per il sistema. Sintesi dei risultati dei test che mostrano i difetti che sono stati identificati e corretti. Se sono stati utilizzati i metodi formali, una specifica formale del sistema e un'analisi di questa specifica. Registrazioni delle analisi statistiche del codice sorgente.
Rapporti di revisione	Registrazione di tutte le revisioni del progetto e della sicurezza.
Competenze del team	Prove della competenza di tutto il team coinvolto nello sviluppo e nella convalida dei sistemi connessi alla sicurezza.
Processo di garanzia della qualità (QA)	Registrazioni dei processi di garanzia (si veda il Capitolo 21) svolti durante lo sviluppo del sistema.
Processi di gestione delle modifiche	Registrazioni di tutte le proposte di modifica, delle azioni svolte e, ove appropriato, la dimostrazione che tali modifiche non compromettono la sicurezza. Informazioni sulle procedure e sulle registrazioni della gestione della configurazione.
Casi di sicurezza associati	Riferimenti ad altri casi di sicurezza che possono influire sul caso di sicurezza in esame.

Figura 12.10 Possibili componenti di un caso di sicurezza.

requisiti del caso di sicurezza. Si dovrebbero evitare, quindi, quelle scelte di progettazione che potrebbero aumentare le difficoltà e i costi del caso di sicurezza.

2. I responsabili della certificazione hanno le loro opinioni su ciò che è accettabile o inaccettabile in un caso di sicurezza. Un team di sviluppo dovrebbe collaborare con loro fin dalle fasi iniziali del processo di sviluppo per capire che cosa si aspettano i responsabili dal caso di sicurezza del sistema.

Lo sviluppo dei casi di sicurezza è costoso perché richiede la registrazione di varie attività e lo svolgimento di particolari processi di verifica e convalida della sicurezza. Le modifiche del sistema e le conseguenti revisioni aumentano i costi

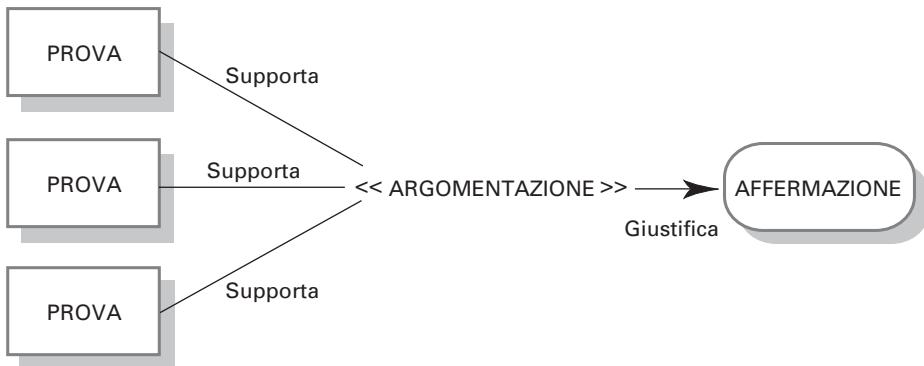


Figura 12.11 Argomentazioni strutturate.

dei casi di sicurezza. Quando viene apportata una modifica al software o all’hardware del sistema, potrebbe essere necessario riscrivere gran parte dei casi di sicurezza per dimostrare che la sicurezza del sistema non è stata influenzata dalla modifica.

12.4.1 Argomentazioni strutturate

Per stabilire se un sistema è sicuro da utilizzare, ci si deve basare su argomentazioni logiche. Queste argomentazioni dovrebbero dimostrare che le prove presentate supportano le affermazioni sulla sicurezza e sulla fidatezza di un sistema. Queste affermazioni possono essere assolute (l’evento X non accadrà mai) o probabilistiche (la probabilità che si verifichi l’evento Y è 0,n). Un’argomentazione collega la prova all’affermazione. Come mostra la Figura 12.11, un’argomentazione è una relazione tra ciò che si pensa sia il caso (l’affermazione) e le prove che sono state raccolte. L’argomentazione spiega essenzialmente perché l’affermazione, che è un’asserzione sulla sicurezza o sulla fidatezza del sistema, può essere dedotta dalle prove disponibili.

Le argomentazioni in un caso di sicurezza di solito vengono presentate come argomentazioni “basate su affermazioni”. Si fa un’affermazione sulla sicurezza di un sistema e, sulla base delle prove disponibili, viene presentata un’argomentazione per giustificare tale affermazione. Per esempio, la seguente argomentazione potrebbe essere utilizzata per giustificare l’affermazione che i calcoli svolti dal software di controllo in una pompa di insulina non produrranno mai un’overdose di insulina. Ovviamente, questa è una descrizione molto semplificata dell’argomentazione. In un caso di sicurezza reale dovranno essere presentati riferimenti più dettagliati delle prove.

Affermazione – La singola dose massima calcolata dalla pompa di insulina non supererà il valore di *maxDose*, dove *maxDose* è la dose sicura per un paziente.

Prova – Dimostrazione della sicurezza del programma che controlla la pompa di insulina (descritto nel paragrafo successivo).

Prova – Insieme dei dati di test per la pompa di insulina. In 400 test, che forniscono una copertura completa del codice, il valore della dose di insulina che è stata somministrata, *currentDose*, non ha mai superato *maxDose*.

Prova – Rapporto dell’analisi statica del programma di controllo della pompa di insulina. L’analisi statica del software di controllo non ha rilevato anomalie che possono influire sul valore di *currentDose*, la variabile che contiene la dose corretta di insulina da somministrare.

Argomentazione – La prova presentata dimostra che la dose massima di insulina che può essere calcolata è uguale a *maxDose*.

È ragionevole supporre, con un alto livello di confidenza, che le prove giustifichino l’affermazione che la pompa di insulina non calcolerà una dose di insulina maggiore della dose di sicurezza.

Le prove presentate sono ridondanti e diverse. Il software viene verificato utilizzando più meccanismi differenti, con una significativa sovrapposizione tra loro. Come detto nel Capitolo 10, utilizzando processi diversi e ridondanti, aumenta il livello di confidenza. Se non vengono rilevati errori e omissioni da un processo di convalida, ci sono buone probabilità che essi saranno identificati da uno degli altri processi.

Di norma si fanno molte affermazioni sulla sicurezza di un sistema; spesso la validità di un’affermazione dipende dalla validità delle altre affermazioni. È possibile quindi organizzare le affermazioni in modo gerarchico. La Figura 12.12 mostra una parte di questa struttura gerarchica di affermazioni per la pompa di insulina. Per dimostrare che un’affermazione di alto livello è valida, occorre prima esaminare le argomentazioni per le affermazioni di livello inferiore. Se si può dimostrare che ciascuna di queste affermazioni di livello inferiore è giustificata, allora si può desumere che anche le affermazioni di livello più alto sono giustificate.

12.4.2 Argomentazioni sulla sicurezza del software

Un’ipotesi generale che permette di valutare il lavoro svolto dal software per la sicurezza di un sistema è che il numero di errori che possono mettere in pericolo la sicurezza sia notevolmente più piccolo di tutti gli errori che potrebbero esistere nel sistema. La garanzia della sicurezza può quindi concentrarsi sugli errori che costituiscono un potenziale pericolo per il sistema. È possibile dimostrare che questi errori non possono verificarsi o, se si verificano, i rischi associati non si tradurranno in incidenti, e quindi il sistema è sicuro. Questa è la base delle argomentazioni sulla sicurezza del software.

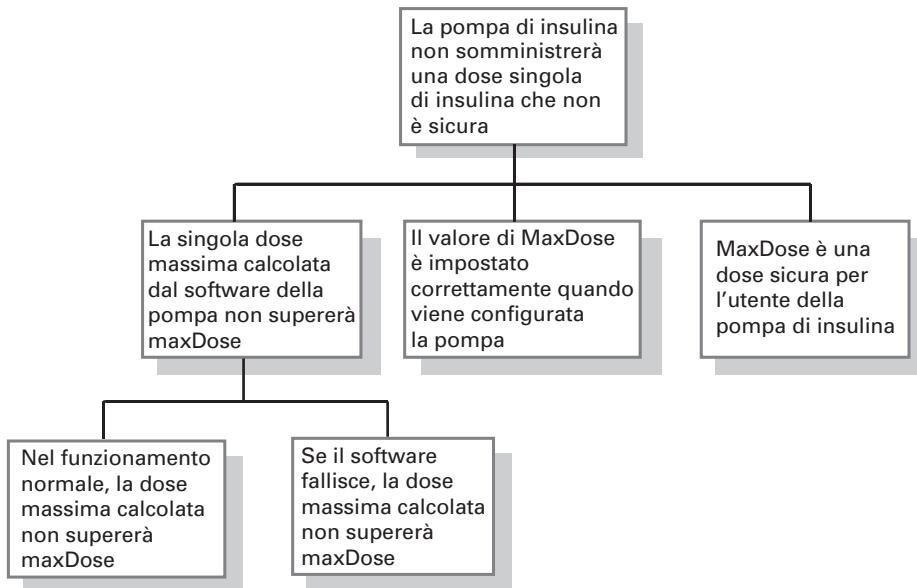


Figura 12.12 Gerarchia delle argomentazioni sulla sicurezza per la pompa di insulina.

Le argomentazioni sulla sicurezza del software sono un tipo di argomentazioni strutturate che dimostrano che un programma soddisfa i requisiti della sicurezza. In un'argomentazione sulla sicurezza non è necessario dimostrare che il programma opera come previsto; è sufficiente dimostrare che l'esecuzione di un programma non potrà portare il sistema in uno stato potenzialmente non sicuro. Le argomentazioni sulla sicurezza sono quindi economicamente più convenienti di quelle sulla correttezza dei sistemi. Non occorre considerare tutti gli stati di un programma – basta concentrarsi semplicemente sugli stati che potrebbero essere pericolosi per il sistema.

Le argomentazioni sulla sicurezza dimostrano che, in condizioni normali di esecuzione, un programma è sicuro. Di solito, si basano sulla dimostrazione per assurdo, ovvero si suppone che il sistema non sia sicuro e, poi, si dimostra che è impossibile che il sistema raggiunga uno stato che non è sicuro. I passaggi per formulare un'argomentazione sulla sicurezza sono qui elencati.

1. Si inizia supponendo che l'esecuzione del programma possa portare il sistema in uno stato non sicuro, che è stato identificato dall'analisi dei rischi del sistema.
2. Si scrive un predicato (un'espressione logica) che definisce questo stato non sicuro.
3. Si analizza sistematicamente un modello del sistema o il programma, e si dimostra che, per tutti i percorsi che conducono a tale stato, la condizione finale di questi percorsi, anche questa definita predicato, contraddice il pre-

```

– La dose di insulina da somministrare è una funzione del livello
– di zuccheri nel sangue, della dose precedente che è stata somministrata
– e del tempo trascorso dalla dose precedente

currentDose = computelnsulin () ;

// Controllo delle sicurezza – regolare currentDose se necessario.

// istruzione if 1
if (previousDose == 0)
{
    if (currentDose > maxDose/2)
        currentDose = maxDose/2 ;
}
else
    if (currentDose > (previousDose * 2) )
        currentDose = previousDose * 2 ;

// istruzione if 2
if ( currentDose < minimumDose )
    currentDose = 0 ;
else if ( currentDose > maxDose )
    currentDose = maxDose ;
administerInsulin (currentDose) ;

```

Figura 12.13 Calcolo della dose di insulina con i controlli sulla sicurezza.

dicato dello stato non sicuro. Se questo è vero, si può dedurre che l’ipotesi iniziale non è vera.

- Se si ripete questa procedura per tutti i rischi che sono stati identificati, allora si ha la prova che il sistema è sicuro.

Le argomentazioni sulla sicurezza possono essere applicate a vari livelli, dai requisiti ai modelli di progettazione fino al codice. A livello dei requisiti, bisogna dimostrare che tutti i requisiti della sicurezza sono soddisfatti e che i requisiti non invalidano le ipotesi sul sistema. A livello di progettazione, occorre analizzare un modello di stati del sistema per trovare gli stati non sicuri. A livello del codice, si considerano tutti i percorsi all’interno del codice a sicurezza critica per dimostrare che l’esecuzione di tutti questi percorsi porta a una contraddizione.

Come esempio consideriamo il codice riportato nella Figura 12.13, che è una descrizione semplificata della parte del codice che implementa il processo di somministrazione dell’insulina. Il codice calcola la dose di insulina da somministrare e poi esegue alcuni controlli per verificare che il risultato del calcolo non sia un’overdose per il paziente. Per formulare un’argomentazione sulla sicurezza per questo codice, bisogna dimostrare che la dose di insulina somministrata non sarà mai maggiore del livello massimo di sicurezza per una singola dose. Questa dose viene stabilita dal medico per ciascun paziente diabetico.

Per dimostrare la sicurezza non occorre provare che il sistema rilascia la dose “corretta”, ma semplicemente che non rilascerà mai un’overdose per ogni paziente. Si parte dal presupposto che *maxDose* sia il livello di sicurezza per il paziente.

Per formulare l’argomentazione sulla sicurezza, identifichiamo il predicato che definisce lo stato non sicuro, che è rappresentato dalla relazione *currentDose > maxDose*. Poi dimostriamo che tutti i percorsi del programma portano a una contraddizione di questa ipotesi di stato non sicuro. Se questo è il caso, allora la condizione iniziale di stato non sicuro non può essere vera. Se possiamo provare una contraddizione, allora possiamo essere sicuri che il programma non calcolerà mai una dose di insulina dannosa. Possiamo strutturare e rappresentare graficamente le argomentazioni sulla sicurezza, come illustra la Figura 12.14.

L’argomentazione sulla sicurezza illustrata nella Figura 12.14 presenta tre possibili percorsi del programma che portano alla chiamata del metodo *administerInsulin*. Dobbiamo dimostrare che la quantità di insulina rilasciata non supera *maxDose*. Consideriamo tutti i possibili percorsi del codice che portano al metodo *administerInsulin*.

1. Nessuno dei due rami della seconda istruzione *if* viene eseguito; questo può capitare solo se *currentDose* è fuori dall’intervallo *minimumDose ... maxDose*. La postcondizione è quindi:

currentDose >= minimumDose e currentDose <= maxDose

2. Viene eseguito il ramo “vero” della seconda istruzione *if*; in questo caso viene eseguita l’assegnazione che pone *currentDose* a zero. Quindi, la postcondizione è *currentDose = 0*.
3. Viene eseguito il ramo “falso” della seconda istruzione *if*; in questo caso viene eseguita l’assegnazione che pone *currentDose* pari a *maxDose*. Quindi, la postcondizione è *currentDose = maxDose*.

In tutti e tre i casi le postcondizioni contraddicono la precondizione di non sicurezza *currentDose > maxDose*. Possiamo quindi affermare che l’ipotesi iniziale è falsa e che il calcolo della dose è sicuro.

Per formulare un’argomentazione sulla sicurezza che un programma non esegue calcoli errati, identifichiamo tutti i possibili percorsi all’interno del codice che potrebbero portare a un’assegnazione potenzialmente non sicura. Procediamo all’inverse partendo dallo stato non sicuro, e consideriamo l’ultima assegnazione di tutte le variabili di stato in ogni percorso che porta a questo stato non sicuro. Se riusciamo a dimostrare che nessuno di questi valori delle variabili non è sicuro, allora abbiamo dimostrato che l’ipotesi iniziale (che il calcolo non è sicuro) è sbagliata.

Procedere all’inverse è importante perché significa che possiamo ignorare tutti gli stati intermedi, tranne quelli finali che portano alla condizione di uscita del codice. I valori precedenti non riguardano la sicurezza del sistema. In questo

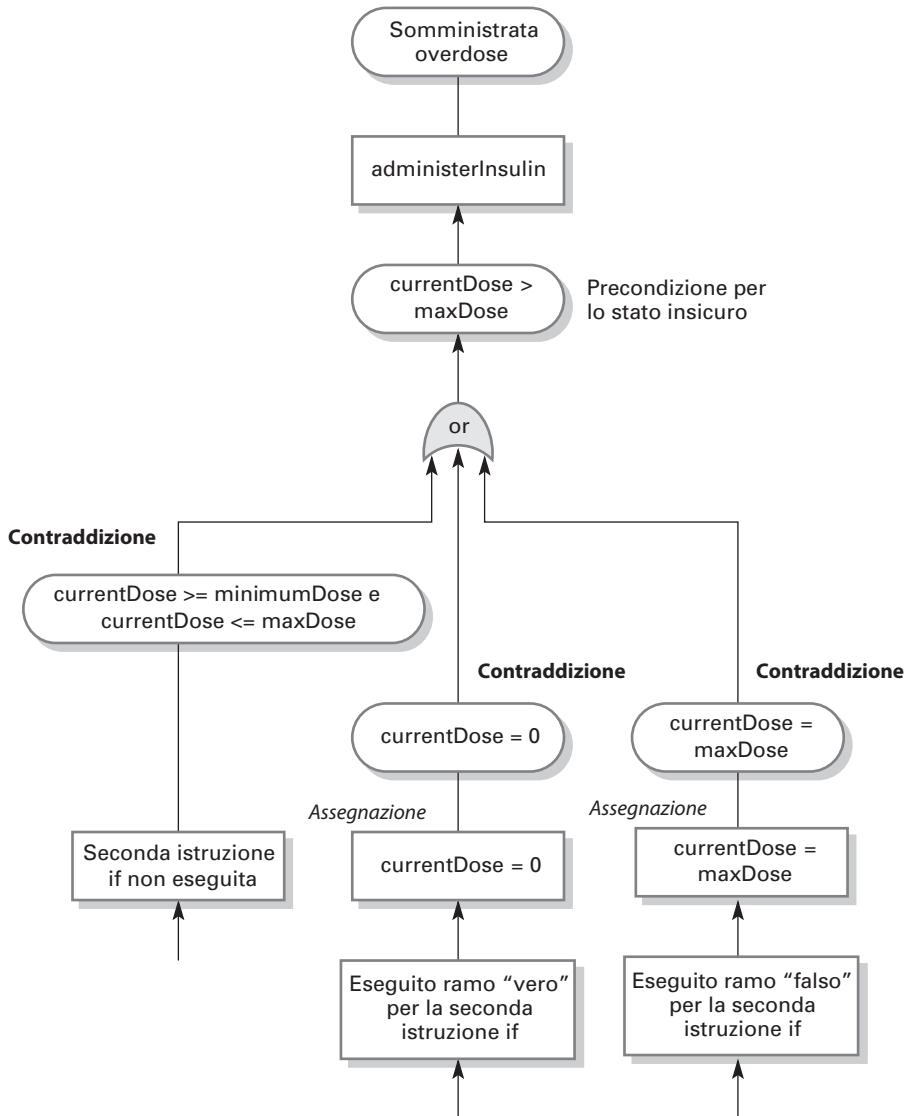


Figura 12.14 Argomentazione sulla sicurezza basata sulla dimostrazione delle contraddizioni.

esempio, ciò che interessa è l'insieme dei possibili valori di *currentDose* immediatamente prima di eseguire il metodo *administerInsulin*. Nell'argomentazione sulla sicurezza possiamo ignorare i calcoli della prima istruzione *if* nella Figura 12.13, in quanto i loro risultati vengono sovrascritti nelle successive istruzioni del programma.

Punti chiave

- I sistemi a sicurezza critica sono sistemi i cui fallimenti possono causare la morte o il ferimento delle persone.
- Un approccio guidato dai rischi può essere utilizzato per capire i requisiti della sicurezza per i sistemi a sicurezza critica. Bisogna prima identificare i potenziali rischi e poi scomporli (adottando metodi come l’analisi dell’albero dei guasti) per scoprire le loro cause principali.
- È importante avere un processo certificato e ben definito per lo sviluppo dei sistemi a sicurezza critica. Il processo dovrebbe includere l’identificazione e il monitoraggio dei potenziali rischi.
- L’analisi statica è un approccio alla verifica e alla convalida dei sistemi software che esamina il codice sorgente (o altra rappresentazione) cercando errori e anomalie. Consente di controllare tutte le parti di un programma, non soltanto quelle che sono sottoposte ai test.
- La verifica dei modelli è un approccio formale all’analisi statica che controlla tutti gli stati di un sistema per identificare potenziali errori.
- I casi di sicurezza e fidatezza raccolgono tutte le prove che dimostrano che un sistema è sicuro e fidato. I casi di sicurezza sono richiesti quando un ente di regolamentazione esterno deve certificare il sistema software prima che sia consegnato agli utenti.

Esercizi

- 12.1 Identificate sei prodotti di largo consumo che potrebbero essere controllati da sistemi software a sicurezza critica.
 - 12.2 Spiegate perché i confini nel triangolo dei rischi illustrato nella Figura 12.3 possono cambiare col tempo e con gli orientamenti delle aziende.
- * 12.3 Nel sistema della pompa di insulina, il paziente deve cambiare l’ago e aggiungere l’insulina a intervalli regolari di tempo; può anche modificare la quantità massima di insulina per una singola dose e la dose giornaliera massima che può essere somministrata. Indicate tre errori tipici che potrebbe commettere il paziente e proponete i requisiti di sicurezza necessari per evitare che questi errori si traducano in gravi incidenti.
- * 12.4 Un sistema software a sicurezza critica per il trattamento del cancro è composto da due componenti principali:
- una macchina per la terapia radioattiva che rilascia dosi controllate di radiazioni nelle zone tumorali. Questa macchina è controllata da un sistema software integrato;
 - un database che contiene i dati per il trattamento di ciascun paziente. I dettagli dei trattamenti vengono immessi in questo database e vengono automaticamente scaricati dalla macchina.
- Identificate tre rischi che potrebbero nascere in questo sistema. Suggerite i requisiti per ridurre le probabilità che ciascun rischio possa tradursi in un incidente. Spiegate perché i vostri requisiti sono efficaci contro tali rischi.

* 12.5 Un sistema di protezione dei treni aziona automaticamente i freni di un treno se viene superato il limite di velocità per una determinata lunghezza del binario oppure se il treno entra in un tratto del binario dove il semaforo è rosso (che vieta l'ingresso in tale tratto). Ci sono due requisiti di sicurezza per questo sistema di protezione:

- il treno non dovrà entrare in un tratto di binario con il semaforo rosso;
- il treno non dovrà superare il limite di velocità specificato per ogni tratto di binario.

Supponendo che lo stato dei semafori e i limiti di velocità per i tratti di binari siano trasmessi automaticamente al software a bordo del treno prima di entrare in un tratto di binario, proponete cinque possibili requisiti funzionali per il software che possano essere ottenuti dai requisiti di sicurezza del sistema.

* 12.6 Spiegate perché potrebbe essere economicamente vantaggioso utilizzare specifiche e verifiche formali nello sviluppo di sistemi software a sicurezza critica. Perché pensate che alcuni ingegneri di sistemi critici siano contrari all'uso dei metodi formali?

12.7 Spiegate perché la verifica dei modelli in alcuni casi sia un metodo economicamente più conveniente della verifica della correttezza di un programma rispetto a una specifica formale.

* 12.8 Elencate quattro tipi di sistemi che potrebbero richiedere i casi di sicurezza, spiegando perché sono necessari i casi di sicurezza.

12.9 Il meccanismo di controllo della serratura di un deposito di scorie nucleari è progettato per funzionare in modo sicuro. Esso garantisce che l'ingresso al deposito sia consentito solo quando gli schermi radioattivi sono attivati o quando il livello di radioattività nel locale è al di sotto di un determinato valore (livello di guardia); pertanto:

- (i) se gli schermi radioattivi controllati a distanza sono attivati all'interno del locale, un operatore autorizzato può aprire la porta;
- (ii) se il livello di radioattività nel locale è al di sotto di un determinato valore, un operatore autorizzato può aprire la porta;
- (iii) un operatore autorizzato è identificato da un codice da immettere all'ingresso della porta.

Il codice della Figura 12.15 controlla il meccanismo della serratura della porta di accesso al deposito. Notate che lo stato di sicurezza è quello che vieta l'ingresso al locale. Utilizzando l'approccio descritto in questo capitolo, formulate un'argomentazione sulla sicurezza per questo codice. Utilizzate i numeri di riga del codice per fare riferimento a specifiche istruzioni. Se pensate che il codice non sia sicuro, suggerite le modifiche appropriate per renderlo sicuro.

12.10 Gli ingegneri del software che si occupano della specifica e dello sviluppo di sistemi a sicurezza critica dovrebbero essere professionalmente abilitati oppure no a svolgere questi compiti? Spiegate le vostre ragioni.

* *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

```
1  entryCode = lock.getEntryCode () ;
2  if (entryCode == lock.authorizedCode)
3  {
4      shieldStatus = Shield.getStatus ();
5      radiationLevel = RadSensor.get ();
6      if (radiationLevel < dangerLevel)
7          state = safe;
8      else
9          state = unsafe;
10     if (shieldStatus == Shield.inPlace() )
11         state = safe;
12     if (state == safe)
13     {
14         Door.locked = false ;
15         Door.unlock ();
16     }
17     else
18     {
19         Door.lock ( );
20         Door.locked := true ;
21     }
22 }
```

Figura 12.15 Codice di controllo della serratura della porta del deposito.

Ulteriori letture

Safeware: System Safety and Computers. Sebbene abbia più di 20 anni, questo libro offre ancora oggi la migliore e più completa trattazione dei sistemi a sicurezza critica. È particolarmente interessante ed efficace la descrizione dell'analisi dei rischi e del processo di derivazione dei requisiti da tale analisi. N. Leveson, Addison-Wesley, 1995.

“Safety-Critical Software.” Un’edizione speciale della rivista *IEEE Software* che descrive i sistemi a sicurezza critica. Include alcuni articoli sullo sviluppo, basato su modelli, dei sistemi a sicurezza critica, la verifica dei modelli e i metodi formali. *IEEE Software*, 30 (3), May/June 2013.

“Constructing Safety Assurance Cases for Medical Devices.” Questo breve documento presenta un esempio pratico di come un caso di sicurezza possa essere formulato per un'apparecchiatura medica. A. Ray e R. Cleaveland, Proc. Workshop on Assurance Cases for Software-Intensive Systems, San Francisco, 2013. <http://dx.doi.org/10.1109/ASSURE.2013.6614270>

Ingegneria della protezione

L'obiettivo di questo capitolo è presentare gli aspetti della protezione che devono essere considerati durante lo sviluppo dei sistemi applicativi. Dopo aver letto questo capitolo:

- capirete l'importanza dell'ingegneria della protezione e la differenza fra protezione delle applicazioni e protezione delle infrastrutture;
- saprete utilizzare un approccio basato sui rischi per ottenere i requisiti di protezione e analizzare i progetti dei sistemi;
- conoscerete gli schemi architetturali del software e le linee guida di progettazione per l'ingegneria di sistemi protetti;
- capirete perché testare e assicurare la protezione di un sistema è un'operazione difficile e costosa.

- 13.1 Protezione e fidatezza
- 13.2 Protezione e organizzazioni
- 13.3 Requisiti di protezione
- 13.4 Progettazione di sistemi protetti
- 13.5 Test e garanzia della protezione

La vastissima diffusione di Internet negli anni '90 ha introdotto una nuova sfida per gli ingegneri del software – progettare e implementare sistemi protetti. Man mano che un numero sempre maggiore di sistemi si collegava a Internet, vari tipi di attacchi esterni venivano progettati per minacciare questi sistemi. La progettazione di sistemi fidati è diventata molto più difficile. Oltre ai problemi derivanti da errori accidentali nel processo di sviluppo, gli ingegneri hanno dovuto affrontare le minacce alla sicurezza dei sistemi architettate da persone dotate di grandi capacità tecniche.

Oggi è essenziale progettare sistemi in grado di resistere agli attacchi esterni; senza difese appropriate, è inevitabile che il funzionamento di sistema collegato alla rete risulterà compromesso da tali attacchi. Gli hacker potranno utilizzare il sistema per scopi illeciti, accedere a informazioni confidenziali o interrompere i servizi offerti dal sistema.

Per progettare sistemi protetti occorre tenere in considerazione tre fattori essenziali.

1. *Riservatezza* – Le informazioni in un sistema potrebbero diventare accessibili a persone o a programmi che non sono autorizzati ad eccedere a tali informazioni. Per esempio, il furto dei dati delle carte di credito da un sistema di commercio elettronico è un problema di riservatezza.
2. *Integrità* – Le informazioni di un sistema possono danneggiarsi, diventando inutilizzabili o inaffidabili. Per esempio, un virus che cancella i dati è un problema di integrità.
3. *Disponibilità* – L'accesso a un sistema o ai suoi dati potrebbe diventare impossibile. Un attacco denial-of-service che sovraccarica un server è un tipico caso in cui la disponibilità del sistema è compromessa.

Questi tre fattori sono strettamente correlati. Se un attacco rende indisponibile il sistema, non sarà possibile aggiornare le informazioni che cambiano nel tempo. Questo significa che l'integrità del sistema potrebbe essere compromessa. Se un attacco ha successo e l'integrità del sistema è compromessa, potrebbe essere necessario arrestare il sistema per risolvere il problema, riducendo in questo modo la disponibilità del sistema.

Da un punto di vista organizzativo, la protezione deve essere considerata a tre livelli.

1. *Protezione delle infrastrutture* – Occorre garantire la protezione di tutti i sistemi e delle reti che forniscono le infrastrutture e i servizi all'organizzazione.
2. *Protezione delle applicazioni* – Occorre proteggere i singoli sistemi o i gruppi di sistemi applicativi.
3. *Protezione delle operazioni* – Occorre proteggere il funzionamento e l'utilizzo dei sistemi dell'organizzazione.

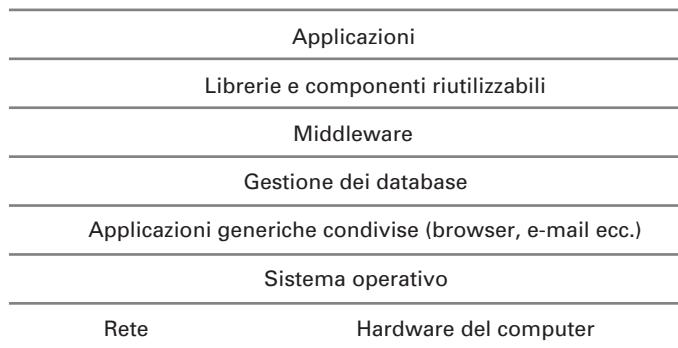


Figura 13.1 Livelli di un sistema dove può essere compromessa la protezione.

La Figura 13.1 riporta uno schema dello stack di un sistema applicativo che mostra come il sistema si affida a un'infrastruttura di altri sistemi per il suo funzionamento. Il livello più basso dell'infrastruttura è rappresentato dall'hardware, ma l'infrastruttura software per i sistemi applicativi può includere i seguenti elementi:

- un sistema operativo, come Linux o Windows;
- generiche applicazioni che vengono eseguite nel sistema, come i browser del Web e i client per la posta elettronica;
- un sistema per la gestione dei database;
- il middleware che supporta il calcolo distribuito e l'accesso ai database;
- librerie di componenti riutilizzabili che vengono utilizzate dal software delle applicazioni.

I sistemi in rete sono controllati dal software, e la protezione delle reti può essere minacciata quando un hacker intercetta, legge e modifica i pacchetti di dati trasmessi nelle reti. Questa operazione richiede attrezzi speciali, quindi la maggior parte degli attacchi sono rivolti alle infrastrutture software dei sistemi. Gli hacker si concentrano sulle infrastrutture software in quanto i loro componenti, come i browser, sono disponibili a tutti. Gli hacker possono esplorare i punti deboli di questi sistemi e condividerne le informazioni sulle vulnerabilità che hanno scoperto. Poiché molte persone usano lo stesso software, gli attacchi ai sistemi hanno una vasta applicabilità.

La protezione delle infrastrutture è un problema importante nella gestione dei sistemi, i cui manager configurano le infrastrutture in modo che possano resistere agli attacchi. La gestione della protezione dei sistemi include varie attività, quali la gestione degli utenti e delle autorizzazioni agli accessi, lo sviluppo e la manutenzione del software, l'identificazione e il monitoraggio degli attacchi e il ripristino del funzionamento del sistema dopo un attacco.

1. La gestione degli utenti e delle autorizzazioni include l’iscrizione e la cancellazione degli utenti del sistema, la garanzia che siano utilizzati meccanismi di autenticazione appropriati e la configurazione delle autorizzazioni in modo che ogni utente abbia accesso solo alle risorse strettamente necessarie.
2. L’installazione e la manutenzione del software includono anche la configurazione del middleware in modo da evitare potenziali punti di vulnerabilità della protezione. È importante aggiornare regolarmente il software con le versioni o patch più recenti, per risolvere i problemi di protezione che sono stati scoperti.
3. Il monitoraggio e il rilevamento degli attacchi e il ripristino del funzionamento dopo un attacco richiedono l’osservazione costante del sistema per identificare eventuali accessi non autorizzati, la capacità di riconoscere gli attacchi esterni, l’impiego di strategie appropriate per resistere a questi attacchi, l’organizzazione delle attività di backup in modo da poter ripristinare il corretto funzionamento del sistema dopo un attacco.

La protezione delle operazioni è principalmente un problema umano e sociale, perché bisogna garantire che le persone che usano il sistema non si comportino in modo da compromettere la protezione del sistema. Per esempio, gli utenti potrebbero restare connessi al sistema mentre questo è incustodito; un hacker potrebbe facilmente accedere al sistema. Gli utenti spesso si comportano in modo poco sicuro per svolgere il loro lavoro con più efficienza. Per salvaguardare la protezione delle operazioni occorre sensibilizzare gli utenti sui problemi della protezione e trovare un giusto compromesso tra protezione del sistema ed efficienza del lavoro.

Il termine *protezione cibernetica* oggi si usa comunemente quando si parla di protezione dei sistemi software. Questo termine riguarda tutti gli aspetti della protezione dei cittadini, delle aziende e delle infrastrutture contro le minacce che possono nascere utilizzando i computer e Internet. La protezione cibernetica riguarda tutti i livelli di un sistema, dall’hardware e le reti, i sistemi applicativi fino alle unità mobili che possono essere utilizzate per accedere a questi sistemi. Il Capitolo 14, dedicato all’ingegneria della resilienza, tratta i problemi generali della protezione cibernetica, inclusa la protezione delle infrastrutture.

In questo capitolo tratterò in particolare i problemi dell’ingegneria della protezione delle applicazioni – i requisiti, la progettazione e i test della protezione. Non descriverò le tecniche generali di protezione che vengono utilizzate, come la crittografia, né i meccanismi di controllo degli accessi o i vettori degli attacchi, come virus e worm. Queste tecniche sono descritte dettagliatamente nei libri di testo sulla protezione dei computer (Pfleeger e Pfleeger 2007; Anderson 2008; Stallings e Brown 2012).

13.1 Protezione e fidatezza

La protezione è un attributo del sistema che rispecchia la capacità di autoprotezione del sistema dagli attacchi esterni o interni. Gli attacchi esterni sono possibili perché molti computer e unità mobili sono collegati alla rete e quindi accessibili agli estranei. Esempi tipici di attacchi al sistema sono l'installazione di virus e cavalli di Troia, l'uso non autorizzato dei servizi del sistema o la modifica non autorizzata delle funzioni del sistema o dei suoi dati.

Se davvero volete un sistema che sia quanto più possibile protetto, la soluzione migliore consiste nel non collegarlo a Internet. In questo caso, i problemi della protezione sono limitati a garantire che gli utenti autorizzati non abusino del sistema e a controllare l'uso di dispositivi esterni come le unità USB. In pratica, però, l'accesso a Internet offre grandi vantaggi alla maggior parte dei sistemi, quindi scollegarsi dalla rete non è un'opzione praticabile per la sicurezza.

Per molti sistemi la protezione è l'attributo più importante della fidatezza. I sistemi per applicazioni militari, i sistemi per il commercio elettronico e quelli che richiedono l'elaborazione e l'interscambio di informazioni riservate devono essere progettati in modo che raggiungano un alto livello di protezione. Per esempio, se non è disponibile il sistema per le prenotazioni online di una compagnia aerea, si provocano disagi e ritardi nell'emissione dei biglietti. Tuttavia, se il sistema non è protetto, un hacker potrebbe cancellare tutte le prenotazioni e sarebbe praticamente impossibile garantire il funzionamento normale di una compagnia aerea.

Come per altri aspetti della fidatezza, anche alla protezione è associata una terminologia specifica (Pfleeger e Pfleeger 2007). Questa terminologia è spiegata nella Figura 13.2. La Figura 13.3 è una storia di protezione per il sistema Mentcare che ho utilizzato per illustrare alcuni di questi termini. La Figura 13.4 usa i concetti definiti nella Figura 13.2 per illustrare come si applicano a questa storia di protezione.

Le vulnerabilità del sistema possono nascere per problemi nella definizione dei requisiti, nella progettazione o nell'implementazione del software, oppure possono derivare da errori umani, sociali o aziendali. Le persone possono scegliere password facili da indovinare o scrivere le loro password in posti dove possono essere facilmente trovate. Gli amministratori dei sistemi potrebbero commettere degli errori quando configurano i file di controllo degli accessi al sistema; molti utenti non installano o non usano il software di protezione. Tuttavia, non possiamo classificare questi problemi come semplici errori umani. Gli errori o le omissioni degli utenti spesso rispecchiano alcune scelte di progettazione inappropriate che richiedono, per esempio, di cambiare spesso le password (questo induce gli utenti a scrivere le loro password su fogli di carta) o di configurare complessi meccanismi di autenticazione.

Termino	Definizione
Risorsa	Qualcosa che merita di essere protetta. La risorsa può essere lo stesso sistema software o i dati utilizzati dal sistema.
Attacco	Un hacker sfrutta una vulnerabilità del sistema con lo scopo di provocare qualche danno alle risorse del sistema. Gli attacchi possono essere portati dall'esterno del sistema (attacchi esterni) o da utenti autorizzati del sistema (attacchi interni).
Controllo	Una misura protettiva che riduce la vulnerabilità del sistema. La crittografia può essere un esempio di controllo che riduce le vulnerabilità di un sistema che ha un controllo degli accessi debole.
Esposizione	Possibile perdita o danno a un sistema di calcolo. Il termine può riferirsi alla perdita o al danneggiamento dei dati oppure al tempo e alle energie spese per ripristinare le funzionalità del sistema dopo una violazione della protezione.
Minaccia	Circostanza che ha il potenziale di causare perdite o danni. Il termine identifica una vulnerabilità del sistema che è sottoposto a un attacco.
Vulnerabilità	Un punto debole in un sistema informatico che può essere sfruttato per arrecare perdite o danni al sistema.

Figura 13.2 Terminologia della protezione.

Esistono quattro tipi di minacce alla protezione.

1. *Minacce di intercettazione* – Un hacker ha guadagnato l'accesso a una risorsa del sistema. Per esempio, una possibile minaccia al sistema Mentcare potrebbe essere quella in cui un hacker riesce ad accedere alla cartella clinica di un paziente.
2. *Minacce di interruzione* – Un hacker rende indisponibile una parte del sistema. Per esempio, una possibile minaccia potrebbe essere un attacco denial-of-service a un server del database del sistema.

Accesso non autorizzato al sistema Mentcare

Il personale medico si collega al sistema Mentcare utilizzando un nome utente e una password. Il sistema richiede che la password sia composta da almeno otto lettere; non effettua altri controlli sulle password digitate. Un criminale viene a sapere che un famoso campione sportivo è sotto trattamento per problemi psichiatrici; vuole accedere illegalmente alle informazioni riservate per poter ricattare il campione.

Fingendo di essere un parente preoccupato e parlando con le infermiere nella clinica psichiatrica, scopre come accedere al sistema e alle informazioni personali sulle infermiere e le loro famiglie. Osservando i cartellini di riconoscimento, legge i nomi di alcune persone che sono autorizzate ad accedere al sistema. Poi tenta di collegarsi al sistema usando questi nomi e cercando di indovinare le password, digitando i nomi dei figli delle infermiere.

Figura 13.3 Una storia di protezione per il sistema Mentcare.

Termino	Esempio
Risorsa	Le cartelle cliniche dei pazienti in cura o che hanno ricevuto cure in passato.
Attacco	Una persona che accede al sistema spacciandosi per un utente autorizzato.
Controllo	Un sistema di controllo delle password che non permette di utilizzare nomi propri o parole presenti in un qualsiasi dizionario.
Esposizione	Potenziali perdite economiche derivanti dalla scelta di futuri pazienti di rivolgersi a un'altra clinica, perché non credono che la clinica sia in grado di proteggere i loro dati personali. Perdita finanziaria in seguito all'azione legale del campione sportivo. Danno di reputazione.
Minaccia	Un utente non autorizzato accede al sistema ricostruendo le credenziali (nome utente e password) di un utente autorizzato.
Vulnerabilità	L'autenticazione degli utenti si basa su un sistema di controllo debole, perché non richiede password complesse. Gli utenti possono scegliere password facili da indovinare.

Figura 13.4 Esempi di terminologia della protezione.

3. *Minacce di modifica* – Un hacker danneggia una risorsa del sistema. Nel sistema Mentcare una minaccia di modifica potrebbe verificarsi quando un hacker modifica o distrugge la cartella clinica di un paziente.
4. *Minacce di falsificazione* – Un hacker inserisce informazioni false nel sistema. Probabilmente questa non è una minaccia reale per il sistema Mentcare, ma potrebbe esserlo per un sistema bancario, nel quale potrebbero essere aggiunte false transazioni per trasferire denaro sul conto corrente di un truffatore.

I controlli che potremmo mettere in atto per migliorare la protezione del sistema si basano su tre concetti fondamentali: evitare i rischi (*avoidance*); identificare i rischi (*detection*) e ripristinare le normali funzionalità del sistema (*recovery*).

1. *Evitare le vulnerabilità* – I controlli che hanno il compito di garantire che gli attacchi al sistema non abbiano successo. La strategia qui consiste nel progettare il sistema in modo da evitare problemi di protezione. Per esempio, i sistemi per applicazioni militari sensibili non sono connessi a Internet, in modo da rendere più difficili gli accessi dall'esterno. Anche la crittografia è un controllo che si basa sul concetto di avoidance. Qualsiasi accesso non autorizzato ai dati crittografati non permetterà all'hacker di leggere i dati; il processo per decifrarli è lungo, difficile e costoso.
2. *Identificazione e neutralizzazione degli attacchi* – I controlli che hanno il compito di identificare e respingere gli attacchi. Questi controlli richiedono funzionalità che siano in grado di monitorare le operazioni del sistema e segnalare attività insolite del sistema. Se vengono rilevati questi attacchi,

allora può essere intrapresa un’azione appropriata, come sospendere le operazioni di alcune parti del sistema o limitare l’accesso a pochi utenti.

3. *Limitare l’esposizione e ripristinare il sistema* – I controlli che supportano il ripristino del funzionamento normale del sistema dopo un problema. Questi controlli variano dalle strategie di backup automatico e di “mirroring” delle informazioni alla stipula di polizze assicurative per coprire i costi associati a un attacco condotto con successo.

La protezione è strettamente correlata con gli attributi della fidatezza: affidabilità, disponibilità, sicurezza e resilienza.

1. *Protezione e affidabilità* – Se un sistema subisce un attacco e il sistema o i suoi dati vengono danneggiati in seguito a tale attacco, questo potrebbe provocare un fallimento del sistema che, a sua volta, potrebbe compromettere l’affidabilità del sistema.

Gli errori durante lo sviluppo di un sistema possono provocare delle fallo nei meccanismi di protezione. Se un sistema non rifiuta gli input imprevisti o non controlla i limiti degli array, gli hacker possono sfruttare questi punti di debolezza per guadagnare l’accesso al sistema. Per esempio, un mancato controllo della validità di un input potrebbe consentire a un hacker di inserire ed eseguire un codice dannoso per il sistema.

2. *Protezione e disponibilità* – Un tipico attacco che viene portato a un sistema basato sul Web è l’attacco denial-of-service, nel quale un server del Web viene inondato di richieste di servizi effettuate da varie sorgenti. Scopo di questo attacco è rendere indisponibile il sistema. Una variante di questo attacco consiste nel minacciare un sito economicamente redditizio con questo tipo di attacco se non viene pagato un compenso agli hacker.
3. *Protezione e sicurezza* – Il problema chiave è un attacco che danneggia il sistema o i suoi dati. I controlli sulla sicurezza si basano sull’ipotesi che sia possibile analizzare il codice sorgente del software a sicurezza critica e che l’esecuzione del codice sia una traduzione assolutamente accurata del codice sorgente. Se questo non è vero, perché un hacker ha modificato il codice, potrebbero verificarsi dei fallimenti nei meccanismi di sicurezza e il caso di sicurezza creato per il software non sarebbe più valido.

Analogamente alla sicurezza, non possiamo assegnare un valore numerico alla protezione di un sistema né possiamo testare in modo completo la protezione di un sistema. Sicurezza e protezione possono essere pensate come caratteristiche “negative” o “non deve”, nel senso che esse riguardano cose che non devono accadere. In altri termini, non potremo mai dimostrare che un sistema è sicuro o protetto.

4. *Protezione e resilienza* – La resilienza, trattata nel Capitolo 14, è una caratteristica che rispecchia la capacità di un sistema software di resistere a eventi dannosi e di ripristinare il suo funzionamento normale. L’evento

dannoso più probabile per i sistemi software collegati in rete è un attacco cibernetico, per cui la maggior parte del lavoro svolto per migliorare la resilienza consiste nell'impedire e identificare tali attacchi e nel ripristinare le operazioni normali del sistema dopo che si è verificato un simile attacco.

La protezione deve essere garantita se vogliamo creare sistemi affidabili, disponibili e sicuri. Non è una caratteristica facoltativa, che può essere aggiunta in un secondo momento, ma deve essere tenuta in considerazione in tutte le fasi del ciclo di sviluppo, dalla definizione dei requisiti fino alla messa in funzione del sistema.

13.2 Protezione e organizzazioni

Realizzare sistemi protetti è un'operazione costosa e incerta. È impossibile prevedere i costi di un fallimento della protezione, quindi per le società e altre organizzazioni è difficile valutare quanto devono investire sulla protezione dei sistemi. Da questo punto di vista, protezione e sicurezza sono differenti. Ci sono leggi che governano la sicurezza degli operatori e dei luoghi di lavoro; gli sviluppatori di sistemi a sicurezza critica devono applicare queste leggi, indipendentemente dai costi. Essi potrebbero essere perseguiti legalmente se installano sistemi che non sono sicuri. Tuttavia, a meno che un fallimento della protezione non sveli informazioni personali riservate, non ci sono leggi che impediscono di installare un sistema non protetto.

Le società valutano i rischi e le perdite che potrebbero provocare alcuni tipi di attacchi alle risorse dei loro sistemi; poi possono decidere se sia economicamente più conveniente accettare questi rischi, anziché realizzare sistemi protetti in grado di impedire o respingere gli attacchi esterni. Le società che gestiscono le carte di credito applicano questo approccio per proteggersi dalle frodi. Di solito è possibile introdurre nuove tecnologie per ridurre le frodi effettuate con le carte di credito. Tuttavia, spesso è più conveniente risarcire gli utenti truffati che acquistare e installare costosi e sofisticati dispositivi tecnologici per ridurre le frodi.

La gestione dei rischi della protezione è quindi un problema economico più che tecnico. Quando un attacco al sistema ha successo, bisogna considerare non solo le perdite economiche e i danni della reputazione, ma anche i costi delle procedure di protezione e delle tecnologie necessarie per ridurre questi costi. Affinché la gestione dei rischi possa essere efficace, le società dovrebbero adottare politiche di protezione che si basano sui seguenti criteri.

1. *Le risorse che devono essere protette* – Non ha senso applicare rigorose procedure di protezione a tutte le risorse di una società. Molte risorse non sono riservate, e una società potrebbe migliorare la propria immagine rendendo gratuitamente accessibili queste risorse. I costi per garantire la protezione delle informazioni che si trovano nel dominio pubblico sono molto minori di quelli richiesti per proteggere le informazioni riservate.

2. *Il livello di protezione che è richiesto per i vari tipi di risorse* – Non tutte le risorse richiedono lo stesso livello di protezione. In alcuni casi (per esempio, per le informazioni riservate sul personale), è richiesto un alto livello di protezione; per altre informazioni, le conseguenze di un fallimento della loro protezione potrebbero essere irrilevanti, quindi è accettabile un livello di protezione più basso. Pertanto, alcune informazioni potrebbero essere messe a disposizione di tutti gli utenti autorizzati che sono collegati al sistema; altre informazioni più sensibili, invece, dovrebbero essere a disposizione esclusivamente degli utenti che svolgono determinati ruoli di responsabilità.
3. *Le responsabilità dei singoli utenti, manager e società* – Le politiche di protezione dovrebbero definire che cosa si aspettano gli utenti – per esempio l'utilizzo di password complesse, la disconnessione di computer e l'esclusione di determinati uffici. Dovrebbero definire inoltre che cosa gli utenti si aspettano dalla società, per esempio i servizi di backup e di archiviazione delle informazioni e la fornitura di apparecchiature.
4. *Procedure e tecnologie di protezione esistenti che dovrebbero essere mantenute* – Per ragioni pratiche ed economiche potrebbe essere essenziale continuare a utilizzare le misure di protezione esistenti, anche se presentano alcune limitazioni. Per esempio, una società potrebbe richiedere l'utilizzo di un nome utente e di una password per l'autenticazione, semplicemente perché altri metodi potrebbero essere rifiutati dagli utenti.

Le politiche di protezione spesso definiscono le strategie generali di accesso alle informazioni che dovrebbero essere applicate alla società. Per esempio, una strategia di accesso potrebbe basarsi sui gradi di autorevolezza o anzianità delle persone che richiedono l'accesso. Per esempio, una strategia di protezione militare potrebbe stabilire che “gli utenti possono esaminare soltanto quei documenti la cui classificazione è uguale o inferiore al livello delle loro credenziali”. Questo significa che, se un lettore è stato accreditato con il livello “riservato”, potrà accedere ai documenti che sono classificati come “confidenziali”, “riservati” o “liberi”, ma non a quelli classificati come “top secret”.

Lo scopo principale delle politiche di protezione è informare tutto il personale di una società sui problemi connessi alla protezione, evitando di produrre documenti tecnici lunghi e dettagliati. Dal punto di vista dell'ingegneria della protezione, queste politiche devono definire, in termini generali, gli obiettivi della protezione. Gli ingegneri della protezione si occupano dell'implementazione di questi obiettivi.

13.2.1 Valutazione dei rischi di protezione

La valutazione e la gestione dei rischi di protezione sono attività che hanno come obiettivi l'identificazione e la comprensione dei rischi delle risorse (sistemi e dati) di una società. In teoria, la valutazione di un singolo rischio dovrebbe essere

svolta per tutte le risorse; in pratica, però, questo potrebbe essere impossibile a causa dell'alto numero di sistemi e database esistenti da valutare. In questi casi, si potrebbe effettuare una valutazione generica di tutte queste risorse. Le valutazioni dei singoli rischi dovrebbero essere eseguite sempre per i nuovi sistemi.

La valutazione e la gestione dei rischi di protezione sono attività aziendali, più che attività squisitamente tecniche che fanno parte del ciclo di vita dello sviluppo del software. La ragione di questo è che alcuni tipi di attacchi non si basano sulla tecnologia, ma sui punti di debolezza dei meccanismi di protezione di un'azienda. Per esempio, un hacker potrebbe guadagnare l'accesso alle apparecchiature fingendo di esser un ingegnere accreditato. Se un'azienda ha un processo che verifica con il fornitore delle apparecchiature se è stata programmata la visita di un ingegnere, è possibile impedire questo tipo di attacco. Questo approccio è molto più semplice che tentare di risolvere il problema utilizzando una soluzione tecnologica.

Quando si avvia lo sviluppo di un nuovo sistema software, la valutazione e la gestione dei rischi di protezione dovrebbero accompagnare tutto il processo di sviluppo, dalla specifica iniziale fino all'installazione del sistema. Le fasi del processo di valutazione dei rischi sono le seguenti.

1. *Valutazione preliminare dei rischi* – Obiettivo della valutazione preliminare dei rischi è identificare i rischi generici che sono applicabili al sistema e decidere se è possibile raggiungere un livello adeguato di protezione a costi ragionevoli. In questa fase, non devono essere effettuate scelte sui requisiti dettagliati del sistema, sul suo progetto o sulla tecnologia di implementazione. Non sono ancora note le vulnerabilità potenziali della tecnologia o dei controlli che sono inclusi nei componenti riutilizzabili o nel middleware del sistema. La valutazione dei rischi deve quindi focalizzarsi sull'identificazione e sull'analisi dei rischi di alto livello per il sistema. I risultati di questo processo sono poi utilizzati per agevolare la definizione dei requisiti di protezione.
2. *Valutazione dei rischi di progettazione* – Questa valutazione dei rischi si svolge durante il ciclo di sviluppo del sistema e si basa sulle scelte tecniche di progettazione e implementazione del sistema. I risultati della valutazione possono portare a modifiche dei requisiti di protezione e all'aggiunta di nuovi requisiti. Vengono analizzate le vulnerabilità note e quelle potenziali, e queste informazioni vengono utilizzate per scegliere le funzionalità del sistema e per stabilire come implementare, provare e installare il sistema.
3. *Valutazione dei rischi d'impiego* – Questo processo di valutazione si occupa dei rischi che potrebbero presentarsi durante l'utilizzo di un sistema. Per esempio, quando un sistema viene utilizzato in un ambiente in cui le interruzioni sono comuni, la protezione del sistema potrebbe essere a rischio quando un utente collegato al sistema lascia incustodito il suo computer per risolvere un problema. Per contrastare questo rischio, si potrebbe specificare

care un requisito di timeout, in modo che un utente venga automaticamente scollegato dal sistema dopo un certo periodo di inattività.

La valutazione dei rischi d'impiego dovrebbe proseguire dopo che un sistema è stato installato presso i clienti, per capire come il sistema viene effettivamente utilizzato e per prendere in considerazione eventuali proposte di nuovi requisiti o di modifica dei requisiti esistenti. Le ipotesi sui requisiti d'impiego che furono fatte quando fu definita la specifica potrebbero rivelarsi sbagliate. Le modifiche richieste dai clienti potrebbero indicare che il sistema viene utilizzato in modo diverso da come fu originariamente progettato. Queste modifiche potrebbero richiedere nuovi requisiti di protezione che devono essere implementati durante l'evoluzione del sistema.

13.3 Requisiti di protezione

La specifica dei requisiti di protezione per i sistemi ha molto in comune con quella dei requisiti di sicurezza. Non è possibile specificare i requisiti di sicurezza o di protezione come probabilità. I requisiti di protezione, analogamente a quello di sicurezza, sono spesso requisiti di tipo “non deve”, che definiscono un comportamento inaccettabile del sistema, anziché una funzionalità richiesta per il sistema.

La protezione, però, rispetto alla sicurezza è un problema più difficile da affrontare per diversi motivi.

1. Quando consideriamo la sicurezza, possiamo supporre che l'ambiente in cui il sistema è installato non è ostile. Nessuno tenterà di provocare un incidente correlato alla sicurezza. Quando consideriamo la protezione, invece, dobbiamo supporre che gli attacchi al sistema saranno intenzionali e che gli hacker possono conoscere i punti di debolezza del sistema.
2. Quando si verifica un fallimento che potrebbe mettere a rischio la sicurezza, cerchiamo gli errori o le omissioni che hanno causato il fallimento. Quando un attacco deliberato provoca il fallimento del sistema, trovare la causa potrebbe essere più difficile, in quanto l'hacker potrebbe nascondere la causa del fallimento.
3. Di solito è accettabile interrompere il funzionamento di un sistema o ridurre i suoi servizi per evitare un fallimento correlato alla sicurezza. Gli attacchi a un sistema, invece, possono essere attacchi denial-of-service, che hanno lo scopo di compromettere la disponibilità del sistema. L'arresto del sistema significa che l'attacco ha avuto successo.
4. Gli eventi correlati alla sicurezza sono accidentali e non sono creati da un avversario intelligente. Un hacker può provare le difese di un sistema tramite una serie di attacchi, modificando gli attacchi dopo che ha acquisito maggiori informazioni sul comportamento del sistema e sulle sue risposte.

Queste distinzioni implicano che i requisiti di protezione devono essere più estesi dei requisiti di sicurezza. La sicurezza richiede una serie di requisiti funzionali che assicurano la protezione contro eventi ed errori che potrebbero provocare fallimenti correlati alla sicurezza. Questi requisiti di solito riguardano l'identificazione di problemi e le azioni da intraprendere quando si verifica uno di questi problemi. I requisiti di protezione, invece, riguardano vari tipi di minacce che potrebbe subire un sistema.

Firesmith (Firesmith 2003) ha identificato 10 tipi di requisiti di protezione che possono essere inclusi nella specifica di un sistema.

1. I requisiti di identificazione specificano se un sistema deve identificare i suoi utenti prima di interagire con loro.
2. I requisiti di autenticazione specificano come identificare gli utenti.
3. I requisiti di autorizzazione specificano i privilegi e le autorizzazioni di accesso degli utenti identificati.
4. I requisiti di immunità specificano come un sistema dovrebbe proteggersi da virus, worm e minacce simili.
5. I requisiti di integrità specificano come evitare di danneggiare i dati.
6. I requisiti di identificazione delle intrusioni specificano quali meccanismi utilizzare per scoprire gli attacchi portati al sistema.
7. I requisiti di non-ripudio specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento.
8. I requisiti di riservatezza specificano come deve essere mantenuta la riservatezza delle informazioni.
9. I requisiti di controllo della protezione specificano come può essere controllato l'utilizzo di un sistema.
10. I requisiti di protezione della manutenzione del sistema specificano come un'applicazione può impedire che siano apportate le modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione.

Ovviamente, non tutti i sistemi richiedono tutti questi requisiti di protezione. I requisiti specifici dipendono dal tipo di sistema, dalle condizioni d'uso e dal tipo di utenti.

Scopo della valutazione preliminare dei rischi è identificare i rischi generici della protezione per un sistema e i suoi dati. La valutazione dei rischi è un input importante per il processo di ingegneria dei requisiti della protezione. Questi requisiti possono essere proposti per supportare le strategie generali della gestione dei rischi: evitare, identificare e mitigare i rischi.

1. I requisiti per evitare i rischi definiscono i rischi che dovrebbero essere evitati da una buona progettazione del sistema, in modo che questi rischi non possano assolutamente presentarsi.

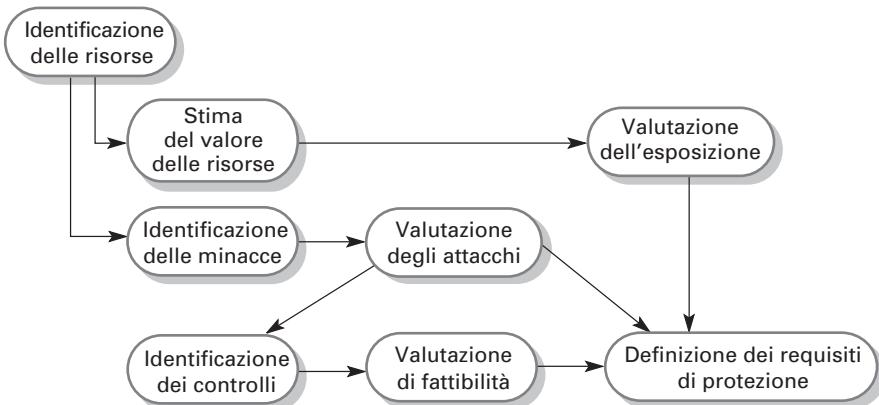


Figura 13.5 Valutazione preliminare dei rischi per definire i requisiti di protezione.

2. I requisiti per identificare i rischi definiscono i meccanismi che identificano i rischi quando questi si presentano e li neutralizzano prima che possano danneggiare il sistema o i suoi dati.
3. I requisiti per mitigare i rischi definiscono come un sistema dovrebbe essere progettato in modo che possa ripristinare il suo normale funzionamento e le sue risorse, dopo che si è verificata qualche perdita.

La Figura 13.5 mostra un processo di valutazione preliminare dei rischi per definire i requisiti di protezione. Le fasi di questo processo sono le seguenti.

1. *Identificazione delle risorse* – Vengono identificate le risorse del sistema che potrebbero richiedere protezione. Il sistema stesso, alcune sue funzioni particolari e i suoi dati potrebbero essere inclusi fra queste risorse.
2. *Stima del valore delle risorse* – Si stima il valore delle risorse identificate.
3. *Valutazione dell'esposizione* – Si stimano le potenziali perdite associate a ogni risorsa. Questo processo dovrebbe tenere conto delle perdite dirette, come il furto di informazioni, i costi di ripristino e il possibile deterioramento della reputazione.
4. *Identificazione delle minacce* – Si definiscono le potenziali minacce alle risorse del sistema.
5. *Valutazione degli attacchi* – Si scomponete ciascuna minaccia nei vari attacchi che potrebbe subire il sistema, analizzando i modi in cui questi attacchi potrebbero essere portati. Per analizzare questi attacchi, si possono utilizzare gli alberi degli attacchi (Schneier 1999), che sono simili agli alberi dei guasti (Capitolo 12), in quanto si parte da una minaccia alla radice di un albero, e poi si identificano i possibili attacchi casuali e le modalità in cui potrebbero essere portati.

Risorsa	Valore	Esposizione
Il sistema informatico	Alto. Necessario per supportare tutti i consulti medici. Potenzialmente a sicurezza critica.	Alta. La cancellazione dei dati degli ambulatori medici potrebbe causare perdite economiche. Costi per ripristinare il funzionamento normale del sistema. Possibili danni per i pazienti per l'impossibilità di prescrivere le cure.
Il database dei pazienti	Alto. Necessario per supportare tutti i consulti medici. Potenzialmente a sicurezza critica.	Alta. La cancellazione dei dati degli ambulatori medici potrebbe causare perdite economiche. Costi per ripristinare il funzionamento normale del sistema. Possibili danni per i pazienti per l'impossibilità di prescrivere le cure.
La cartella clinica di un paziente	Normalmente basso, ma potrebbe essere alto per alcuni pazienti di alto profilo.	I danni diretti sono bassi, ma potrebbe essere compromessa la reputazione.

Figura 13.6 Analisi delle risorse in un rapporto di valutazione preliminare dei rischi per il sistema Mentcare.

6. *Identificazione dei controlli* – Si definiscono i controlli che dovrebbero essere realizzati per proteggere le risorse del sistema. I controlli sono meccanismi tecnici, come la crittografia, che sono in grado di proteggere le risorse.
7. *Valutazione di fattibilità* – Si esamina la fattibilità tecnica dei controlli proposti e si valutano i loro costi. Non è conveniente realizzare controlli costosi per proteggere risorse di poco valore.
8. *Definizione dei requisiti di protezione* – Dopo aver valutato l'esposizione del sistema, le sue potenziali minacce e i relativi controlli, è possibile definire i requisiti di protezione del sistema. Questi requisiti possono essere applicati alle infrastrutture del sistema o al sistema applicativo.

Il sistema Mentcare, che gestisce i pazienti con problemi psichiatrici, è un sistema a sicurezza critica. Le Figure 13.6 e 13.7 sono frammenti di un rapporto sull'analisi dei rischi di questo sistema software. La Figura 13.6 è un'analisi delle risorse che descrive le risorse del sistema e il loro valore. La Figura 13.7 riporta alcune minacce che potrebbe subire il sistema.

Una volta che è stata completata la valutazione preliminare dei rischi, è possibile proporre i requisiti che stabiliscono come evitare, identificare e mitigare i rischi del sistema. Tuttavia, la definizione di questi requisiti non è un processo banale o automatico. Esso richiede gli input sia dagli ingegneri sia dagli esperti del dominio che consigliano i requisiti funzionali per il sistema software in base

Minaccia	Probabilità	Controllo	Fattibilità
Un utente non autorizzato accede al sistema come amministratore e lo rende indisponibile	Bassa	Consentire la gestione del sistema solo da alcune postazioni che sono fisicamente protette.	Bassi costi di implementazione, ma occorre fare attenzione alla distribuzione delle chiavi e assicurarsi che una chiave sia sempre disponibile in caso di emergenza.
Un utente non autorizzato accede al sistema come utente normale e acquisisce informazioni riservate	Alta	Tutti gli utenti devono autenticarsi mediante un meccanismo biometrico. Tutte le modifiche apportate alle informazioni di un paziente devono essere registrate per consentirne la tracciabilità.	Soluzione tecnicamente fattibile, ma molto costosa. È possibile qualche resistenza da parte degli utenti. Semplice da implementare; inoltre agevola il ripristino del sistema.

Figura 13.7 Analisi delle minacce e dei controlli in un rapporto di valutazione preliminare dei rischi.

alla loro interpretazione dell'analisi dei rischi. Alcuni esempi di requisiti di protezione per il sistema Mentcare con i rischi associati sono qui di seguito elencati.

1. All'inizio di una seduta clinica, le informazioni relative a ogni paziente devono essere scaricate dal database in un'area protetta del client del sistema.

Rischio: danni per possibile attacco denial-of-service. Mantenere le copie locali significa che l'accesso è ancora possibile.

2. Tutte le informazioni sui pazienti sul client del sistema devono essere crittografati.

Rischio: accesso esterno ai record dei pazienti. Se i dati sono crittografati, l'hacker deve avere la chiave per decifrare le informazioni sui pazienti.

3. Le informazioni sui pazienti devono essere caricate nel database alla fine di una seduta clinica e cancellate dal client.

Rischio: accesso esterno ai record dei pazienti tramite un computer portatile rubato.

4. Tutte le modifiche apportate al database del sistema e gli autori di queste modifiche devono essere memorizzati in un apposito registro in un computer diverso dal server del sistema.

Rischio: attacchi interni ed esterni possono danneggiare i dati correnti. Un registro dovrebbe consentire di ripristinare i record aggiornati dalla copia di backup del sistema.

I primi due requisiti sono correlati: le informazioni sui pazienti vengono scaricate su una macchina locale, in modo che un consulto medico possa continuare anche se il server del database è sotto attacco o non è disponibile. Queste informazioni devono essere cancellate alla fine di una seduta clinica per evitare che gli utenti successivi del client possano accedere a tali informazioni. Il quarto requisito riguarda il recupero dei dati e la verifica degli accessi. La presenza del registro permette di ricostruire le modifiche apportate al database e rintracciare gli autori di tali modifiche. Questo dovrebbe scoraggiare l'uso improprio del sistema da parte del personale autorizzato.

13.3.1 Casi di uso improprio

Derivare i requisiti di protezione dall'analisi dei rischi è un processo creativo che coinvolge ingegneri ed esperti di domini. Un approccio che è stato sviluppato per supportare questo processo per gli utenti del linguaggio UML si basa sul concetto di caso d'uso improprio (Sindre e Opdahl 2005). I casi d'uso improprio sono scenari che rappresentano interazioni dannose con un sistema. Questi scenari possono essere utilizzati per discutere e identificare possibili minacce e, quindi, determinare i requisiti di protezione di un sistema. Possono essere utilizzati insieme ai casi d'uso per derivare i requisiti del sistema (Capitoli 4 e 5).

I casi d'uso improprio sono associati alle istanze dei casi d'uso e rappresentano minacce o attacchi correlati a questi casi d'uso. Possono essere inclusi in un diagramma di casi d'uso, ma devono avere anche una descrizione più dettagliata e completa. Nella Figura 13.8 ho riportato i casi d'uso per un addetto all'accettazione che usa il sistema Mentcare e ho aggiunto i casi d'uso improprio, che di solito sono rappresentati da ellissi scure.

Analogamente ai casi d'uso, anche i casi d'uso improprio possono essere descritti in vari modi. Io credo che sia più utile descriverli come un supplemento alla descrizione originale dei casi d'uso. Ritengo inoltre che sia meglio avere un formato flessibile per i casi d'uso improprio, in quanto ci sono vari tipi di attacchi che possono essere descritti in modi differenti. La Figura 13.9 mostra la descrizione originale del caso d'uso di trasferimento dei dati (Figura 5.4), con l'aggiunta della descrizione di un caso d'uso improprio.

Il problema dei casi d'uso improprio è speculare al problema generale dei casi d'uso, in quanto le interazioni tra utente finale e sistema non esprimono tutti i requisiti del sistema. I casi d'uso improprio possono essere utilizzati come parte

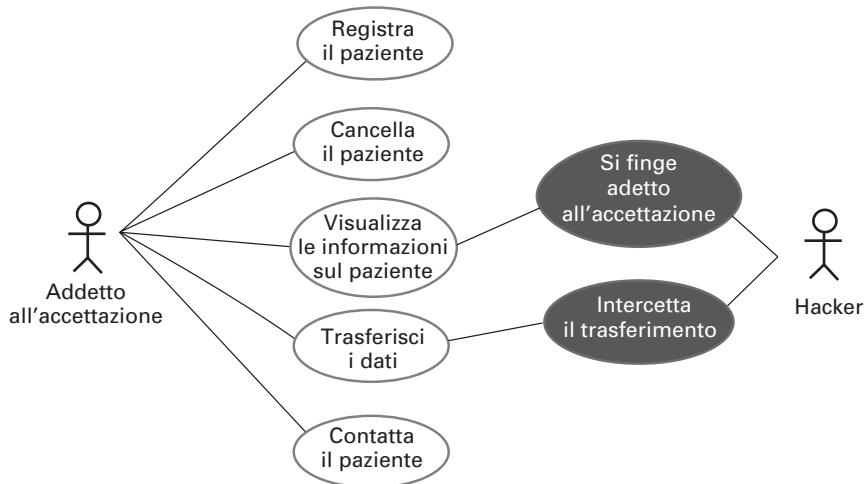


Figura 13.8 Casi d'uso improprio.

del processo di ingegneria dei requisiti di protezione, ma occorre anche considerare i rischi che sono associati agli stakeholder che non interagiscono direttamente con il sistema.

13.4 Progettazione di sistemi protetti

È molto difficile proteggere un sistema dopo che è stato implementato. Pertanto, occorre prendere in considerazione i problemi della protezione durante lo sviluppo di un sistema e fare le scelte appropriate di progettazione per migliorare la protezione del sistema. In questo paragrafo tratterò due importanti argomenti, indipendenti dalle applicazioni, che riguardano la progettazione di sistemi protetti.

1. *Progettazione architettonica* – Come influiscono le scelte di progettazione sulla protezione di un sistema?
2. *Buone pratiche di progettazione* – Quali sono le buone pratiche da adottare durante la progettazione di un sistema protetto?

Ovviamente, questi non sono gli unici temi di progettazione che sono importanti per la protezione. Ogni applicazione è diversa dalle altre, e la progettazione della protezione deve tener conto dello scopo, della criticità e dell'ambiente operativo di ciascuna applicazione. Per esempio, se stiamo progettando un sistema per applicazioni militari, dobbiamo adottare il loro modello di classificazione della protezione (*secret*, *top secret* ecc.). Se stiamo progettando un sistema che gestisce le informazioni personali, dobbiamo tener conto delle leggi sulla privacy che pongono dei vincoli all'uso di dati riservati.

Utilizzando la ridondanza e la diversità, caratteristiche essenziali per la fidatezza, si può ritenere che un sistema sia in grado di resistere agli attacchi che

Sistema Mentcare – Trasferimento dei dati	
Attori	Personale di accettazione, sistema di registrazione dei pazienti (SRP).
Descrizione	Un addetto all'accettazione può trasferire i dati dal sistema Mentcare a un database generale di registrazione dei pazienti che è mantenuto da un'autorità sanitaria. Le informazioni trasferite possono essere informazioni personali aggiornate (indirizzo, numero di telefono ecc.) o una sintesi della diagnosi e della cura dei pazienti.
Dati	Informazioni personali dei pazienti, sintesi della cura.
Stimolo	Comando emesso dal personale di accettazione.
Risposta	Conferma che l'SRP è stato aggiornato.
Commenti	L'addetto all'accettazione deve avere le autorizzazioni appropriate per accedere alle informazioni dei pazienti e all'SRP.
Sistema Mentcare – Intercettazione del trasferimento (caso d'uso improprio)	
Attori	Personale di accettazione, sistema di registrazione dei pazienti (SRP), hacker.
Descrizione	Un addetto all'accettazione trasferisce i dati dal suo PC al sistema Mentcare sul server. Un hacker intercetta il trasferimento dei dati e fa una copia dei dati.
Dati (risorsa)	Informazioni personali dei pazienti, sintesi della cura.
Attacchi	Viene aggiunto nel sistema un monitor in rete; vengono intercettati i pacchetti di dati che sono trasferiti dal PC dell'addetto all'accettazione al server del sistema. Viene configurato un server fantoccio tra il PC dell'addetto all'accettazione e il server del database, in modo che l'addetto all'accettazione creda di interagire con il vero sistema.
Mitigare i rischi	Tutte le unità connesse in rete devono essere tenute in una stanza chiusa a chiave. Gli ingegneri che accedono a queste unità devono essere autorizzati. Tutti i trasferimenti di dati tra client e server devono essere crittografati. Le comunicazioni tra client e server devono avvenire tramite certificazione.
Requisiti	Tutte le comunicazioni tra client e server devono usare il protocollo SSL (Secure Socket Layer). Il protocollo https usa procedure di autenticazione crittografate e certificate.

Figura 13.9 Descrizione di un caso d'uso improprio.

vengono portati verso specifiche funzionalità di progettazione o implementazione. I meccanismi che supportano un alto livello di disponibilità possono aiutare il sistema a ripristinare il suo normale funzionamento dopo un attacco denial-of-service, con il quale un hacker ha l'obiettivo di compromettere le operazioni del sistema e di metterlo fuori servizio.

Attacchi denial-of-service

Gli attacchi denial-of-service tentano di bloccare il funzionamento di un sistema connesso in rete, bombardandolo con un notevole numero di richieste di servizi, di solito centinaia di richieste. Questi attacchi costituiscono un carico di lavoro per il quale il sistema non è stato progettato e, quindi, il sistema è costretto a escludere alcune legittime richieste di servizi. Il sistema potrebbe diventare indisponibile sia perché interrompe le sue operazioni per l'eccessivo sovraccarico sia perché i manager del sistema decidono di fermare il flusso eccessivo di richieste di servizi.

<http://software-engineering-book.com/web/denial-of-service/>

La progettazione di un sistema protetto, inevitabilmente, richiede dei compromessi. Di solito, è possibile progettare più misure di protezione che siano in grado di ridurre le probabilità che un attacco abbia successo. Purtroppo, queste misure di protezione di norma richiedono calcoli aggiuntivi e, quindi, influiscono negativamente sulle prestazioni complessive del sistema. Per esempio, per ridurre le probabilità che le informazioni riservate possano essere lette da un hacker, potremmo crittografarle. Ma questo significa che gli utenti del sistema, per poter utilizzare queste informazioni, devono attendere che vengano decifrate, con conseguente rallentamento del loro lavoro.

Ci sono anche contrasti fra protezione e utilizzabilità – un'altra proprietà emergente per i sistemi software. Le misure di protezione a volte richiedono che l'utente si ricordi determinate informazioni e le immetta nel sistema (per esempio, più password). A volte gli utenti dimenticano queste informazioni e, quindi, aggiungere ulteriori misure di protezione significa rendere inutilizzabile il sistema per questi utenti.

I progettisti dei sistemi devono trovare un giusto compromesso tra protezione, prestazioni e utilizzabilità. Questo dipende dal tipo di sistema che si sta sviluppando, dalle aspettative dei suoi utenti e dal suo ambiente operativo. Per esempio, gli utenti di un sistema militare hanno familiarità con i meccanismi di protezione di alto livello, quindi accettano ed eseguono le procedure che richiedono frequenti verifiche. In un sistema per il trading azionario, dove è essenziale la velocità, le interruzioni delle operazioni per eseguire le verifiche di protezione sarebbero del tutto inaccettabili.

13.4.1 Valutazione di rischi di progettazione

La valutazione dei rischi di protezione durante l'ingegneria dei requisiti identifica una serie di requisiti di protezione di alto livello per il sistema. Durante la progettazione e l'implementazione del sistema, le scelte architettoniche e tecnologiche possono influire sulla protezione di un sistema. Queste scelte generano nuovi requisiti di progettazione e potrebbero richiedere la modifica dei requisiti esistenti.

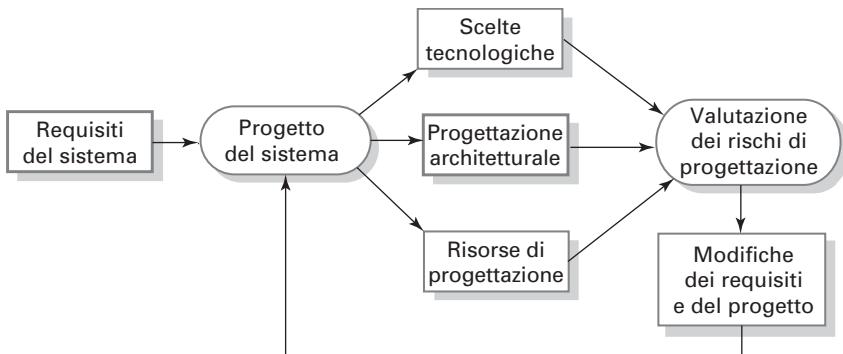


Figura 13.10 Progettazione del sistema e valutazione dei rischi sono processi intrecciati.

La progettazione del sistema e la valutazione dei rischi correlati alla progettazione sono processi intrecciati (Figura 13.10). Fatte le scelte di progettazione preliminare, si valutano i rischi associati a queste scelte. Questa valutazione potrebbe richiedere nuovi requisiti per mitigare i rischi che sono stati identificati oppure potrebbe essere necessario modificare il progetto per ridurre questi rischi. All'evolversi del sistema, mentre si sviluppano nuovi dettagli del sistema, i rischi vengono valutati di nuovo e i risultati di questa valutazione vengono sottoposti ai progettisti del sistema. Il processo di valutazione dei rischi di progettazione termina quando il progetto è completo e i rischi restanti sono accettabili.

Quando si valutano i rischi durante la progettazione e l'implementazione, si hanno maggiori informazioni su cosa deve essere protetto e sulle vulnerabilità del sistema. Alcune di queste vulnerabilità riguardano le scelte di progettazione. Per esempio, una vulnerabilità intrinseca dell'autenticazione basata su password è che un utente autorizzato svela la sua password a un utente non autorizzato. Quindi, se si usa questo tipo di autenticazione, il processo di valutazione dei rischi potrebbe suggerire nuovi requisiti per mitigare questo rischio. Per esempio, potrebbe essere proposto il requisito di autenticazione multifattoriale, dove gli utenti per identificarsi devono digitare non solo una password, ma qualche dato personale.

La Figura 13.11 è un modello del processo di valutazione dei rischi di progettazione. La differenza chiave tra l'analisi preliminare dei rischi e la valutazione dei rischi di progettazione è che, nella fase di progettazione, si hanno le informazioni sulla rappresentazione e sulla distribuzione dei dati e sull'organizzazione del database per le risorse di alto livello che devono essere protette. Inoltre, si conoscono le scelte importanti di progettazione, come il software da riutilizzare, i controlli e la protezione delle infrastrutture e così via. Sulla base di queste informazioni, è possibile identificare le modifiche da apportare ai requisiti della protezione e al progetto del sistema per migliorare la protezione delle principali risorse del sistema.

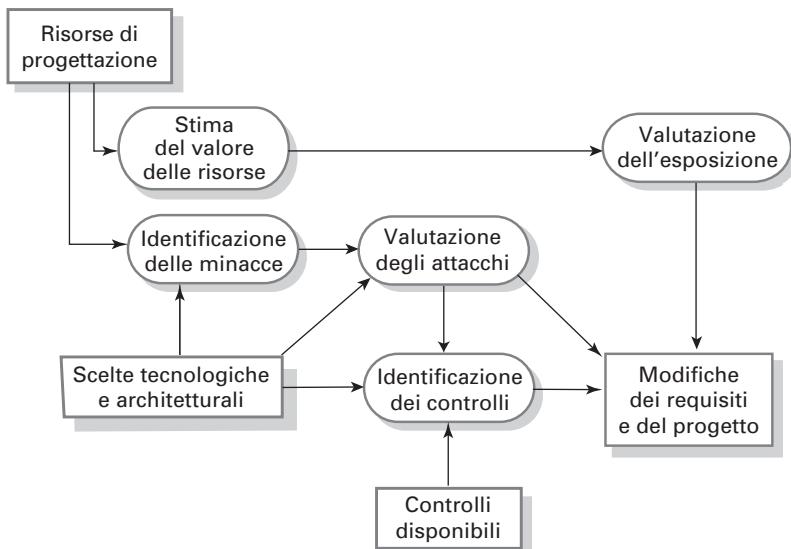


Figura 13.11 Valutazione dei rischi di progettazione.

Due esempi del sistema Mentcare illustrano come i requisiti di protezione sono influenzati dalle scelte di progettazione relative alla rappresentazione e alla distribuzione dei dati.

1. I progettisti hanno deciso di separare le informazioni personali dei pazienti da quelle delle cure ricevute dai pazienti, con una chiave che collega questi due tipi di record. Le informazioni sulle cure sono tecniche e quindi molto meno sensibili di quelle personali dei pazienti. Se la chiave è protetta, un hacker sarà in grado di accedere soltanto alle informazioni di routine, ma potrà associarle ai singoli pazienti.
2. Supponiamo che i progettisti abbiano deciso che, all'inizio di ogni sessione, i record dei pazienti siano copiati in un client locale. Questo consente al personale medico di continuare a lavorare anche quando il server diventa indisponibile. Il personale medico può accedere ai record dei pazienti da un computer portatile, anche se manca la connessione di rete. Tuttavia, adesso ci sono due insiemi di record da proteggere; inoltre le copie sul client sono soggette a rischi addizionali, per esempio il furto del portatile. Quindi, bisogna studiare i controlli da utilizzare per ridurre questo rischio. Per esempio, si potrebbe includere il requisito che i record dei dati memorizzati nei portatili o in altri personal computer devono essere crittografati.

Per illustrare come le scelte sulle tecnologie di sviluppo influenzino la protezione, supponiamo che il responsabile dell'assistenza sanitaria abbia deciso di realizzare un sistema Mentcare utilizzando un sistema informatico off-the-shelf per la

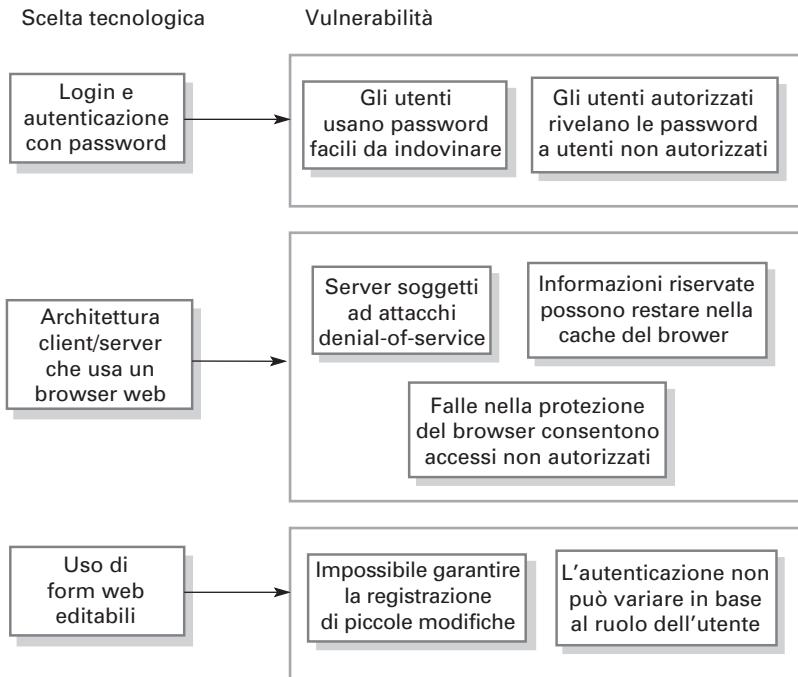


Figura 13.12 Vulnerabilità associate alle scelte tecnologiche.

gestione dei record dei pazienti. Questo sistema deve essere configurato per ciascun tipo di clinica in cui viene utilizzato. Questa decisione è stata presa perché sembra offrire le più ampie funzionalità con costi e tempi di sviluppo minori.

Quando si sviluppa un'applicazione riutilizzando un sistema esistente, occorre accettare le scelte di progettazione che sono state fatte dagli sviluppatori di quel sistema. Supponiamo che alcune di queste scelte siano le seguenti:

1. gli utenti del sistema sono autenticati tramite una combinazione di nome-utente e password. Non sono supportati altri metodi di autenticazione;
2. l'architettura del sistema è di tipo client-server; i client accedono ai dati del server tramite un browser standard installato nei computer client;
3. le informazioni vengono presentate agli utenti come form web editabili; gli utenti possono modificare le informazioni localmente e poi inviare le informazioni aggiornate al server.

Per un generico sistema, queste scelte di progettazione sono perfettamente accettabili; tuttavia, la valutazione dei rischi di progettazione ha identificato alcune vulnerabilità. La Figura 13.12 riporta alcuni esempi di queste potenziali vulnerabilità.

Una volta identificate le vulnerabilità, occorre studiare i rimedi per ridurre i rischi associati. Questo spesso richiede nuovi requisiti di protezione del sistema o la modifica delle modalità di utilizzo del sistema. Esempi di questi requisiti sono elencati qui di seguito.

1. Dovrà essere messo a disposizione un programma che controlla le password. Questo programma dovrà essere eseguito giornalmente per verificare tutte le password degli utenti. Dovranno essere identificate le password che figurano nel dizionario del sistema; gli utenti con password deboli dovranno essere segnalati agli amministratori del sistema.
2. Sarà consentito l'accesso al sistema soltanto ai computer client che sono stati approvati e registrati dagli amministratori del sistema.
3. Su ogni computer client potrà essere installato un solo browser approvato.

Poiché viene utilizzato un sistema off-the-shelf, non è possibile includere il programma che controlla le password nel stesso sistema delle applicazioni; quindi deve essere utilizzato un sistema separato. Questo programma analizza la forza delle password degli utenti quando vengono impostate per la prima volta e avverte immediatamente l'utente che ha scelto una password debole. Una password vulnerabile può essere identificata abbastanza rapidamente dopo che l'utente l'ha impostata; dovrà essere svolta un'apposita azione per garantire che l'utente abbia cambiato la password.

Il secondo e il terzo requisito richiedono che tutti gli utenti potranno accedere al sistema tramite lo stesso browser. Possiamo scegliere il browser che riteniamo più protetto durante lo sviluppo del sistema, e poi installarlo su tutti i computer client. Gli aggiornamenti dei meccanismi di protezione sono semplificati, in quanto non c'è bisogno di aggiornare browser di diversi tipi quando viene identificata e corretta qualche vulnerabilità.

Il modello illustrato nella Figura 13.10 presuppone un processo di progettazione nel quale il progetto viene sviluppato a un livello abbastanza dettagliato prima di avviare l'implementazione. Questo non si verifica nei processi agili, nei quali la progettazione e l'implementazione si svolgono insieme, con il codice che viene migliorato durante lo sviluppo del progetto. Le frequenti consegne degli incrementi del sistema non consentono di valutare dettagliatamente i rischi, anche se sono disponibili le informazioni sulle risorse da proteggere e si conoscono le scelte tecnologiche.

I problemi relativi alla protezione e allo sviluppo agile sono stati ampiamente discussi (Lane 2010; Schoenfeld 2013). Questi problemi sono ancora irrisolti – alcuni ritengono che esista un conflitto naturale tra protezione e sviluppo agile; altri credono che questo conflitto possa essere risolto utilizzando le storie incentrate sulla protezione (Safecode 2012). Questo resta un importante problema per i sostenitori dei metodi agili. Nel frattempo, molte società sensibili ai problemi della protezione si rifiutano di utilizzare i metodi agili, perché sono in conflitto con le loro strategie di analisi dei rischi e di protezione.

13.4.2 Progettazione architettonale

Le scelte di progettazione dell'architettura del software possono avere un impatto notevole sulle proprietà di un sistema software. Se si sceglie un'architettura inappropriata, potrebbe essere molto difficile mantenere la riservatezza e l'integrità delle informazioni o assicurare il livello richiesto di disponibilità del sistema.

Quando si progetta l'architettura di un sistema che garantisce la protezione, occorre considerare due punti fondamentali.

1. *Protezione* – In che modo si deve organizzare il sistema per far sì che le risorse critiche possano essere protette da attacchi esterni?
2. *Distribuzione* – Come devono essere distribuite le risorse all'interno del sistema per ridurre al minimo le conseguenze di un attacco che ha successo?

Questi punti sono potenzialmente in conflitto. Se poniamo tutte le risorse in un unico luogo, possiamo costruire più strati di protezione attorno ad esse. Poiché dobbiamo realizzare un unico sistema di protezione, possiamo progettare un sistema robusto con diversi strati di protezione. Tuttavia, se questo sistema di protezione dovesse fallire, tutte le risorse sarebbero compromesse. Aggiungendo più strati di protezione, si riduce anche l'utilizzabilità del sistema, e questo rende più difficile soddisfare i requisiti di prestazioni e utilizzabilità.

D'altra parte, se distribuiamo le risorse, diventa più costoso proteggerle, in quanto i sistemi di protezione devono essere implementati per ciascuna risorsa distribuita. Tipicamente, quindi, non è economicamente conveniente implementare più strati di protezione. Le probabilità che i meccanismi di protezione siano violati sono maggiori. Tuttavia, se questo accade, non si avranno danni generalizzati per tutte le risorse. Potrebbe essere possibile duplicare e distribuire le informazioni, in modo che se una loro copia viene danneggiata o resa inaccessibile, si potrà utilizzare la copia di riserva. Tuttavia, se le informazioni sono riservate, mantenendo più copie, aumenta il rischio che un hacker possa accedere a tali informazioni.

Per il sistema Mencare è stata scelta un'architettura client-server con un database centrale condiviso. Per assicurare la protezione, il sistema ha un'architettura a strati con le risorse critiche protette al livello più basso nel sistema. La Figura 13.13 mostra questa architettura a più livelli, dove le risorse critiche da proteggere sono i record dei singoli pazienti.

Per accedere ai record e modificarli, un hacker deve penetrare tre strati di protezione del sistema.

1. *Protezione a livello della piattaforma* – Lo strato più alto controlla l'accesso alla piattaforma sulla quale viene eseguito il sistema. Di solito, questo richiede che l'utente debba collegarsi con un particolare computer. La piattaforma include anche un supporto dedicato al mantenimento dell'integrità dei file, delle copie di backup e così via.

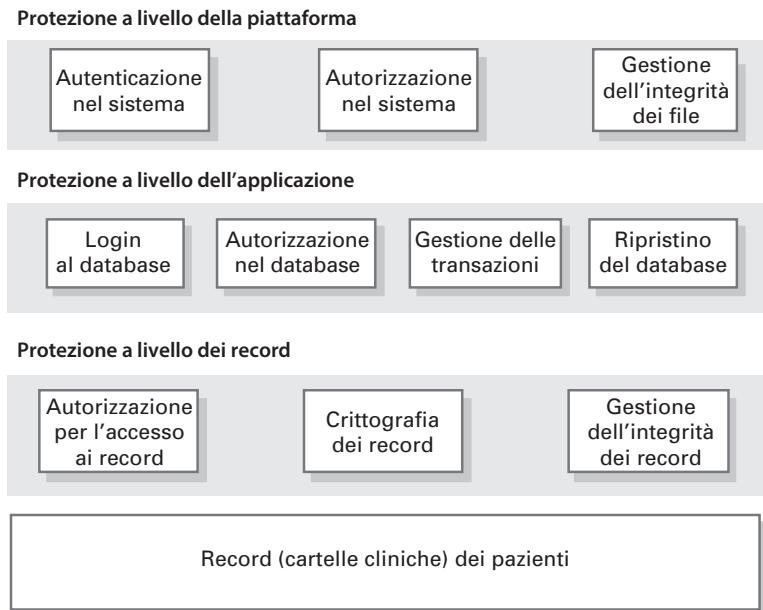


Figura 13.13 Architettura di protezione a più strati.

2. *Protezione a livello dell'applicazione* – Lo strato successivo è integrato nella stessa applicazione. L'utente deve accedervi mediante autenticazione e ricevere l'autorizzazione per svolgere alcune operazioni, come la lettura o la modifica dei dati. In alcuni casi, potrebbe essere disponibile un supporto per la gestione dell'integrità dell'applicazione.
3. *Protezione a livello dei record* – Questo strato è attivo quando un utente richiede l'accesso a specifiche cartelle cliniche dei pazienti; ha il compito di verificare che l'utente sia autorizzato a svolgere le operazioni richieste sui record dei pazienti. A questo livello, la protezione potrebbe anche includere un sistema di crittografia per garantire che i record non possano essere letti tramite un browser. I sistemi di verifica dell'integrità, come quelli che utilizzano i valori di checksum, sono in grado di rilevare eventuali modifiche apportate al di fuori dei normali meccanismi di aggiornamento dei record.

Il numero di strati di protezione necessari per una particolare applicazione dipende dalla criticità dei dati. Non tutte le applicazioni richiedono un sistema di protezione a livello dei singoli record; per questo, di solito, si utilizza un controllo degli accessi più generico. Per garantire la protezione, non si deve consentire che le credenziali di uno stesso utente possano essere utilizzate in ogni strato di protezione. Teoricamente, se il sistema si basa sulle password, la password per acce-

dere all'applicazione dovrà essere diversa da quella di accesso al sistema e da quella di accesso ai record. Tuttavia, le password multiple sono difficili da ricordare per gli utenti; molti utenti inoltre non gradiscono le ripetute richieste di autenticazione. Pertanto, occorre trovare un compromesso tra la protezione e l'utilizzabilità del sistema.

Se la protezione dei dati è un requisito critico, di solito un'architettura client-server centralizzata è la scelta più efficace per la protezione. Il server ha la responsabilità di proteggere i dati sensibili. Se la protezione è compromessa, le perdite associate a un attacco sono alte, in quanto si potrebbero perdere o danneggiare tutti i dati. Anche i costi di ripristino del normale funzionamento del sistema potrebbero essere alti (per esempio, si dovrebbero rifare le credenziali di accesso di tutti gli utenti). I sistemi centralizzati sono anche più vulnerabili agli attacchi denial-of-service che sovraccaricano il server e rendono impossibile a chiunque l'accesso al database del sistema.

Se le conseguenze di una violazione del server sono gravi, è meglio utilizzare un'architettura distribuita per l'applicazione. Così facendo, le risorse del sistema sono distribuite su più piattaforme, ognuna dotata di meccanismi di protezione differenti. Un attacco a un nodo potrebbe rendere indisponibili alcune risorse, ma sarebbe ancora possibile fornire una parte dei servizi del sistema. I dati possono essere duplicati su più nodi del sistema, facilitando il ripristino del sistema dopo un attacco.

La Figura 13.14 illustra l'architettura di un sistema bancario per il trading di azioni e fondi nelle borse di New York, Londra, Francoforte e Hong Kong. Il sistema è distribuito e, quindi, i dati delle singole borse sono conservati separatamente. Le risorse necessarie per supportare l'attività critica di trading (account degli utenti e prezzi delle azioni) sono duplicate e disponibili in tutti i nodi. Se un nodo, dopo un attacco, non è più disponibile, l'attività critica di trading può essere trasferita a un altro nodo, restando a disposizione degli utenti.

Ho già accennato al problema di trovare un compromesso tra protezione e prestazioni del sistema. Un problema che può presentarsi quando si progetta un sistema protetto è che, in molti casi, lo stile architettonico più adeguato ai requisiti di protezione potrebbe non essere appropriato per soddisfare i requisiti delle prestazioni del sistema. Per esempio, supponiamo che un'applicazione debba garantire assolutamente la riservatezza di un grosso database e consentire anche un accesso ai dati molto veloce. Un livello elevato di protezione suggerisce di utilizzare più strati di protezione, e questo implica che tra i vari strati ci dovranno essere delle comunicazioni, con conseguente rallentamento dell'accesso ai dati.

Se si usa un'architettura alternativa, l'implementazione della protezione e della riservatezza dei dati potrebbe risultare più difficile e costosa. In questi casi, occorre discutere con il cliente che ha pagato il sistema i conflitti intrinseci del sistema e trovare un accordo sulle modalità di risoluzione di questi conflitti.

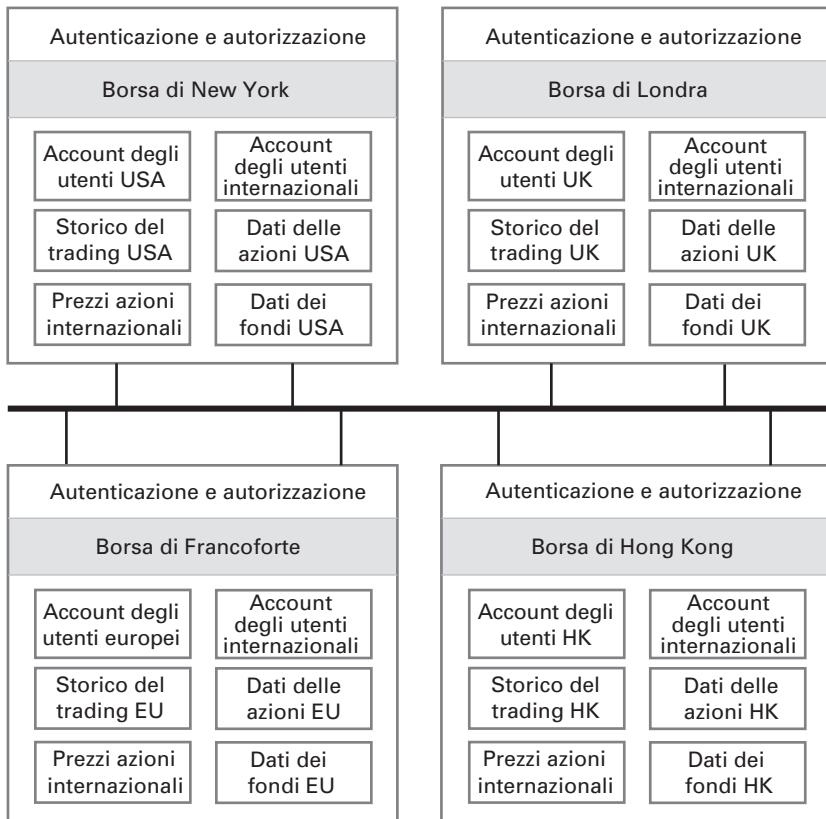


Figura 13.14 Risorse distribuite in un sistema di trading azionario.

13.4.3 Linee guida di progettazione

Non ci sono metodi facili per garantire la protezione di un sistema. Tipi differenti di sistemi richiedono misure di protezione differenti per raggiungere un livello di protezione accettabile per il proprietario del sistema. I requisiti e le attitudini dei vari gruppi di utenti influiscono profondamente su ciò che si può ritenere accettabile. Per esempio, gli utenti di un sistema bancario possono ritenere accettabile un livello di protezione molto alto, con procedure di protezione più invadenti che non sarebbero accettate degli utenti di un sistema universitario.

È possibile, tuttavia, indicare alcune linee guida o direttive generali che hanno un'ampia applicazione quando si progettano misure di protezione dei sistemi software. Queste linee guida includono alcune buone pratiche di progettazione per l'ingegneria dei sistemi protetti. Le direttive generali per la protezione, come quelle descritte in seguito, hanno due scopi principali.

Direttive di progettazione per la protezione	
1	Scelte di protezione basate su politiche di protezione esplicite
2	Il sistema non deve avere un singolo punto di fallimento
3	Il sistema deve fallire in modo protetto
4	Bilanciare protezione e utilizzabilità
5	Registrare le azioni degli utenti collegati al sistema
6	Applicare la ridondanza e la diversità per ridurre i rischi
7	Specificare il formato degli input del sistema
8	Suddividere le risorse in compartimenti separati
9	Progettare per la consegna del sistema
10	Progettate per il ripristino del sistema

Figura 13.15 Direttive di progettazione per l'ingegneria di sistemi protetti.

1. Migliorare la conoscenza dei problemi di protezione in un team di ingegneria del sistema. Gli ingegneri del software spesso si concentrano su obiettivi a breve termine, come quello di consegnare in tempo al cliente un programma funzionante; questo li induce facilmente a trascurare i problemi di protezione. La conoscenza di queste direttive potrà aiutarli a dare maggiore peso ai problemi di protezione durante la progettazione del software.
2. Fornire una lista di elementi da considerare durante il processo di convalida di un sistema. Partendo dalle direttive di alto livello qui presentate è possibile formulare delle domande più specifiche, utili per esaminare come la protezione è stata integrata in un sistema.

Le direttive di protezione, a volte, sono principi molto generali, come “proteggere la connessione più debole del sistema”, “semplificare il metodo” o “evitare misure ambigue di protezione”. Io credo che queste direttive generali siano troppo vaghe per essere realmente utili in un processo di progettazione. Per questo, ho elencato una serie di direttive di progettazione più pratiche. Le 10 direttive di progettazione, riassunte nella Figura 13.15, sono state estratte da diverse fonti (Schneier 2000; Viega e McGraw 2002; Wheeler 2004).

Direttiva 1 – Scelte di protezione basate su politiche di protezione esplicite

Le politiche di protezione sono definizioni di alto livello che stabiliscono le condizioni di protezione fondamentali di una società. Definiscono “cosa” va fatto, non “come” va fatto, quindi non indicano i meccanismi da utilizzare per ottenere o migliorare la protezione di un sistema. In teoria, i temi trattati dalle politiche di

protezione dovrebbero riflettere i requisiti di protezione di un sistema. In pratica, specialmente quando si adottano tecniche di sviluppo agile del software, ciò accade molto raramente.

I progettisti quindi dovrebbero utilizzare le politiche di protezione come base di riferimento su cui fondare e valutare le scelte di progettazione. Per esempio, supponiamo di progettare un sistema di controllo degli accessi per il sistema Mentcare. Le politiche di protezione stabiliscono che solo il personale medico accreditato può modificare i record dei pazienti. Questo porta a definire alcuni requisiti per controllare le credenziali di chiunque tenti di modificare i record del database e di rifiutare qualsiasi aggiornamento proposto da persone non autorizzate.

Un tipico problema da affrontare è che molte società non hanno politiche di protezione esplicite. Col passare del tempo, potrebbe essere necessario apportare alcune modifiche al sistema per risolvere determinati problemi, senza disporre di un documento di strategie generali che guidano l'evoluzione di un sistema. In questi casi, occorre risolvere i problemi, documentando le soluzioni appropriate con alcuni esempi e chiedendo l'approvazione dei manager della società.

Direttiva 2 – Il sistema non deve avere un singolo punto di fallimento

Per qualsiasi sistema critico, è buona prassi tentare di evitare che ci sia un solo punto di fallimento. In altre parole, un singolo fallimento in una parte del sistema non dovrebbe mai comportare il fallimento dell'intero sistema. In termini di protezione, questo significa che non ci si deve affidare a un unico meccanismo per garantire la protezione; piuttosto occorre adottare varie tecniche di protezione. Questo concetto è a volte chiamato “difesa in profondità”.

Un esempio di difesa in profondità è l'autenticazione multifattoriale. Per esempio, se si usa una password per autenticare gli utenti di un sistema, si potrebbe aggiungere anche un meccanismo di domande/risposte, che consiste in alcune domande e risposte che gli utenti hanno preregistrato nel sistema. Dopo che gli utenti hanno digitato le loro credenziali, devono rispondere correttamente alle domande per poter accedere al sistema.

Direttiva 3 – Il sistema deve fallire in modo protetto

I fallimenti sono inevitabili in tutti i sistemi software. Allo stesso modo in cui i sistemi a sicurezza critica devono essere a prova di fallimento, anche i sistemi a protezione critica devono essere “a prova di fallimento”. Quando un sistema fallisce, non si dovrebbero utilizzare procedure di protezione alternative che sono meno protette dello stesso sistema. Inoltre, in caso di fallimento, non dovrà essere consentito l'accesso ai dati a una persona che, in condizioni normali, non è autorizzata.

Per esempio, per il sistema Mentcare avevo suggerito un requisito che prevedeva di scaricare i dati dei pazienti su un sistema client all'inizio di ogni sessione.

Questo meccanismo accelera l'accesso ai dati e ne garantisce la continuità anche nel caso di indisponibilità del server. Di norma, il server cancella questi dati alla fine di ogni sessione, ma, in caso di fallimento del server, i dati vengono conservati sul client. Un approccio a prova di fallimento in questi casi consiste nel crittografare i dati dei pazienti sul client; questo significa che una persona non autorizzata non sarà in grado di leggere i dati.

Direttiva 4 – Bilanciare protezione e utilizzabilità

Le esigenze di protezione e utilizzabilità sono spesso in contrasto. Per proteggere un sistema, occorre introdurre appositi controlli per garantire che gli utenti siano autorizzati a utilizzare il sistema e che agiscano secondo le politiche di protezione. Tutto ciò implica inevitabilmente qualche adempimento aggiuntivo per gli utenti, che dovranno ricordare il nome e la password per il login, potranno accedere al sistema soltanto tramite alcuni computer e così via. Questo significa che gli utenti impiegheranno più di tempo per accedere e utilizzare il sistema. Se si aggiungono nuovi livelli di protezione, di solito l'utilizzabilità del sistema si riduce. Consiglio di leggere il libro di Cranor e Garfinkel (Cranor e Garfinkel 2005) che tratta vari problemi relativi alla protezione e all'utilizzabilità dei sistemi software.

Si potrebbe arrivare un punto in cui diventa controproducente aggiungere nuove misure di protezione a spese dell'utilizzabilità. Per esempio, se gli utenti devono digitare più password o cambiare frequentemente le password scegliendo stringhe di caratteri impossibili da ricordare, è molto probabile che scrivano le password su un foglio di carta. Un utente non autorizzato (specialmente se lavora all'interno della società) potrebbe leggere queste password e utilizzarle per tentare di accedere al sistema.

Direttiva 5 – Registrare le azioni degli utenti collegati al sistema

Se è possibile farlo, è bene mantenere sempre un registro con le azioni svolte dagli utenti collegati al sistema. Questo registro dovrebbe riportare almeno il nome dell'utente, l'azione che ha svolto, le risorse utilizzate e la data e l'ora dell'azione. Mantenendo questo registro come una lista di comandi eseguibili, è possibile esaminare le azioni svolte per ripristinare il normale funzionamento del sistema dopo un fallimento. Occorrono anche gli strumenti che consentono di analizzare il registro e identificare le azioni anomale, consentendo così di scoprire gli attacchi e il modo in cui un hacker sia riuscito a guadagnare l'accesso al sistema.

Oltre ad agevolare il ripristino del sistema dopo un fallimento, il registro delle azioni degli utenti è utile anche come deterrente agli attacchi interni. Se le persone sanno che le loro azioni vengono registrate, è difficile che svolgano operazioni illecite. Questo registro è particolarmente efficace contro gli attacchi casuali, come un'infermiera che vuole leggere le cartelle cliniche degli amici, o per rile-

vare gli attacchi portati utilizzando le credenziali di un utente che sono state rubate da un hacker ingannando l’utente in rete. Ovviamente, questo approccio non è infallibile, in quanto gli hacker più esperti potrebbero modificare il contenuto dello stesso registro.

Direttiva 6 – Applicare la ridondanza e la diversità per ridurre i rischi

Ridondanza significa mantenere più copie del software e dei dati di un sistema. La diversità, applicata al software, significa che le diverse versioni del software non devono basarsi sulla stessa piattaforma o essere implementate con la stessa tecnologia. In questo modo, la vulnerabilità di una piattaforma o di una tecnologia non influenzerà tutte le versioni del software e, quindi, non potrà provocare un fallimento dell’intero sistema.

Ho già descritto alcuni esempi di ridondanza – mantenere le informazioni sia sul server sia sul client, prima nel sistema Mentcare e poi nel sistema distribuito di trading azionario (illustrato nella Figura 13.14). Nel sistema Mentcare, si potrebbero utilizzare sistemi operativi differenti sul client e sul server (per esempio, Linux sul server e Windows sul client); questo assicura che un attacco basato sulla vulnerabilità di un sistema operativo non avrà effetto sul server e sul client. Ovviamente, disponendo di più sistemi operativi, aumentano i costi di gestione dei sistemi. Occorre trovare un compromesso tra i benefici della protezione e questo incremento dei costi.

Direttiva 7 – Specificare il formato degli input del sistema

Un tipico attacco consiste nel fornire al sistema input inaspettati, che possono causare comportamenti del sistema non previsti in fase di progettazione. Questi input possono causare il blocco del sistema, con conseguente interruzione dei servizi. Gli input potrebbero essere frammenti di codice dannoso che vengono eseguiti dal sistema. Le vulnerabilità connesse all’overflow dei buffer di memoria, dimostrate per la prima volta da un famoso worm di Internet (Spafford 1989) e comunemente utilizzate dagli hacker, possono essere sfruttate utilizzando come input lunghe stringhe di caratteri. Un altro tipico attacco è il cosiddetto “avvelenamento di SQL”, nel quale un hacker immette come input un frammento di codice SQL che viene interpretato dal server.

È possibile evitare molti di questi problemi specificando il formato e la struttura degli input che sono previsti per il sistema. Questa specifica dovrebbe basarsi sulla conoscenza degli input previsti. Per esempio, se l’utente deve digitare il suo cognome, si potrebbe specificare che tutti i caratteri devono essere alfabetici, senza numeri né segni di punteggiatura (tranne il trattino). Si potrebbe anche impostare un limite alla lunghezza dei nomi e degli indirizzi degli utenti; per esempio, nessun nome è formato da più di 40 caratteri e nessun indirizzo supera i 100 caratteri. Se si prevede come input un valore numerico, allora devono essere

esclusi tutti i caratteri alfabetici. Queste informazioni saranno poi utilizzate nei controlli degli input, quando sarà implementato il sistema.

Direttiva 8 – Suddividere le risorse in compartimenti separati

Suddividere le risorse di un sistema in compartimenti separati significa che gli utenti non possono avere l'accesso a tutte le informazioni del sistema. Applicando un principio di protezione generale che suggerisce di limitare l'accesso alle informazioni strettamente necessarie, è bene organizzare le informazioni di un sistema in compartimenti tra loro isolati. Gli utenti dovrebbero avere l'accesso soltanto a quelle informazioni che sono necessarie per svolgere il loro lavoro, anziché a tutte le informazioni del sistema. Questo permette di ridurre gli effetti di un eventuale attacco portato al sistema utilizzando le credenziali di un utente autorizzato. Alcune informazioni potranno essere perdute o danneggiate, ma tutte le altre saranno salvaguardate.

Per esempio, il sistema Mentcare potrebbe essere progettato in modo tale che i medici possano accedere non a tutti i record del database, ma soltanto a quelli dei pazienti che hanno fissato un appuntamento nella loro clinica. Questo non soltanto riduce le potenziali perdite derivanti da attacchi interni, ma significa anche che un hacker esterno, che ha rubato le credenziali di un medico, non potrà danneggiare tutti i record dei pazienti.

Detto questo, è possibile prevedere meccanismi che permettano accessi di emergenza; per esempio, se un paziente molto grave ha bisogno di cure urgenti, non dovrà fissare un appuntamento con il suo medico curante. In queste circostanze, si potrebbe adottare un meccanismo di protezione alternativo per superare le limitazioni derivanti dalla suddivisione del sistema in compartimenti. Nei casi in cui i vincoli di protezione vengano esclusi per assicurare la disponibilità del sistema, è essenziale che venga utilizzato un meccanismo di autenticazione che regista le informazioni sull'utilizzo del sistema. Ciò permetterà di risalire a chi ha utilizzato il sistema in modo illecito.

Direttiva 9 – Progettare per la consegna del sistema

Molti problemi di protezione derivano da un'errata configurazione del sistema quando questo viene consegnato al cliente. La consegna prevede l'installazione del software nei computer che lo eseguiranno e l'impostazione di parametri appropriati all'ambiente operativo e alle preferenze degli utenti del sistema. Alcuni errori, come dimenticarsi di disattivare le funzioni di debugging o lasciare inalterata la password dell'amministratore, possono introdurre delle vulnerabilità nel sistema.

Una buona pratica di gestione può evitare molti problemi di protezione derivanti dagli errori commessi durante l'installazione e la configurazione del sistema. I progettisti del software hanno la responsabilità di “progettare per la consegna”, nel senso che dovranno fornire un supporto che riduca al minimo le proba-

bilità che gli utenti o gli amministratori del sistema commettano errori durante la configurazione del software.

Io consiglio quattro modi per integrare in un sistema un supporto alla consegna.

1. *Includere le funzioni per vedere e analizzare le configurazioni* – Dovreste includere sempre in un sistema apposite funzioni che consentano agli amministratori o ad altri utenti autorizzati di esaminare la configurazione corrente del sistema.
2. *Ridurre al minimo i privilegi di default* – Dovreste progettare il software in modo che la configurazione di default del sistema fornisca i privilegi minimi essenziali.
3. *Localizzare i parametri di configurazione* – Quando progettate il supporto alla configurazione del sistema, dovreste accertarvi che tutti i parametri che riguardano la configurazione di un'unica parte del sistema possano essere impostati nello stesso luogo.
4. *Fornire metodi semplici per eliminare le vulnerabilità di protezione* – Dovreste includere alcune semplici procedure di aggiornamento del sistema per eliminare le vulnerabilità che vengono scoperte dopo l'installazione del software.

I problemi di consegna del software non sono così diffusi, in quanto un numero sempre crescente di sistemi software non richiede di essere installato dai clienti; infatti, spesso il software viene eseguito come un servizio, accessibile tramite un browser web. Tuttavia, il software del server resta vulnerabile a causa di errori e omissioni durante l'installazione; inoltre alcuni tipi di sistemi richiedono un software dedicato da eseguire sui computer degli utenti.

Direttiva 10 – Progettare per il ripristino del sistema

Indipendentemente dall'impegno profuso per assicurare la protezione di un sistema, dovreste progettare il sistema prevedendo sempre che la protezione possa essere violata. Di conseguenza, occorre pensare alle azioni che ripristinano lo stato di funzionamento protetto del sistema dopo che si è verificato un fallimento. Per esempio, si potrebbe includere un sistema di autenticazione di backup nel caso in cui siano danneggiate le password di autenticazione.

Supponiamo, per esempio, che una persona non autorizzata, esterna alle cliniche, riesca ad accedere al sistema Mentcare; noi non sappiamo come questa persona abbia ottenuto una valida combinazione di login/password. In questo caso, sarà necessario inizializzare di nuovo tutto il sistema di autenticazione; non basta cambiare solo le credenziali utilizzate da questa persona. Ciò è essenziale perché questa persona potrebbe conoscere altre password. Occorre, quindi, assicurarsi che tutti gli utenti autorizzati abbiano cambiato le loro password. Occorre anche assicurarsi che la persona non autorizzata non possa accedere al meccanismo di cambio delle password.

Il sistema dovrà quindi essere progettato in modo da negare l'accesso a tutti gli utenti, finché non avranno modificato le loro password; dovrà essere inviata una e-mail a tutti gli utenti per chiedere loro di cambiare la password. Occorre inoltre un meccanismo alternativo per autenticare gli utenti che chiedono di cambiare la password, considerando che le password che hanno scelto potrebbero non essere protette. Per fare questo, si potrebbe utilizzare un meccanismo di domande/risposte, che chiede agli utenti autorizzati di rispondere a domande di cui conoscono le risposte esatte. Questo meccanismo viene attivato solo quando vengono cambiate le password e consente di ripristinare il sistema dopo un attacco riducendo al minimo il disservizio per gli utenti.

La progettazione di meccanismi di recovery è essenziale per realizzare sistemi resilienti. Questo argomento è trattato dettagliatamente nel Capitolo 14.

13.4.4 Programmare sistemi protetti

Progettare un sistema protetto significa integrare la protezione nelle applicazioni software. Oltre a garantire la protezione in fase di progettazione, è anche importante considerare la protezione quando si programma un sistema software. Molti attacchi portati al software con successo si basano sulle vulnerabilità dei programmi che sono state introdotte durante lo sviluppo dei programmi.

Il primo attacco famoso portato ai sistemi basati su Internet avvenne nel 1988, quando un worm venne introdotto nei sistemi Unix attraverso la rete (Spafford 1989). Questo attacco aveva sfruttato una ben nota vulnerabilità della programmazione. Se i sistemi sono programmati in C, non c'è alcun controllo automatico sui limiti degli array. Un hacker può includere come input una lunga stringa di caratteri in un comando del programma; questa stringa viene sovrascritta sullo stack del programma e può causare il passaggio del controllo a un codice dannoso. Da allora, questa vulnerabilità è stata sfruttata in molti altri sistemi programmati in C o C++.

Questo esempio illustra due importanti aspetti della programmazione dei sistemi protetti.

1. Le vulnerabilità spesso dipendono dal linguaggio utilizzato per programmare. Il controllo dei limiti degli array è automatico nei linguaggi come Java, quindi questa non è una vulnerabilità che può essere sfruttata nei programmi Java. Tuttavia, milioni di programmi sono scritti in C e C++, in quanto questi linguaggi consentono di sviluppare un software più efficiente. Pertanto, evitare semplicemente di utilizzare questi linguaggi non è un'opzione realistica.
2. Le vulnerabilità della protezione sono strettamente correlate all'affidabilità dei programmi. Il precedente esempio ha causato il crash del programma, quindi devono essere adottate delle misure appropriate per migliorare l'affidabilità del programma e, quindi, anche la protezione del sistema.

Linee guida per una programmazione fidata

1. Limitare la visibilità delle informazioni in un programma.
2. Controllare la validità di tutti gli input.
3. Fornire un gestore per tutte le eccezioni.
4. Minimizzare l'uso di costrutti inclini agli errori.
5. Creare funzioni di riavviamento.
6. Controllare i limiti degli array.
7. Includere un timeout quando si chiama un componente esterno.
8. Assegnare un nome a tutte le costanti che rappresentano entità del mondo reale.

Figura 13.16 Linee guida per una programmazione fidata.

Nel Capitolo 11 ho presentato le linee guida per la programmazione di sistemi fidati. Queste linee guida, riportate nella Figura 13.16, possono anche migliorare la protezione di un programma, in quanto gli hacker cercano di sfruttare le vulnerabilità dei programmi per guadagnare l'accesso a un sistema software. Per esempio, un attacco di “avvelenamento di SQL” si basa sul fatto che un hacker compila un form digitando i comandi SQL, anziché il testo atteso dal sistema. Questo può danneggiare il database o consentire l'accesso a informazioni riservate. È possibile evitare completamente questo problema implementando la verifica degli input (linea guida 2), basandosi sul formato e sulla struttura previsti per gli input.

13.5 Test e garanzia della protezione

La valutazione della protezione dei sistemi sta diventando sempre più importante, al punto che possiamo ritenere che i sistemi che utilizziamo siano protetti. I processi di verifica e convalida dei sistemi basati sul Web devono concentrarsi sulla valutazione della protezione, testando la capacità dei sistemi di resistere a diversi tipi di attacchi. Tuttavia, come spiega Anderson (Anderson 2008), questo tipo di valutazione della protezione è molto difficile da eseguire; di conseguenza, i sistemi spesso sono consegnati con alcune fallo nel sistema di protezione. Gli hacker usano queste vulnerabilità per guadagnare l'accesso al sistema e, poi, per danneggiare il sistema stesso o i suoi dati.

Le ragioni per le quali è difficile testare la protezione di un sistema sono essenzialmente due.

1. I requisiti di protezione, come quelli di sicurezza, sono requisiti del tipo “non deve”, ovvero specificano cosa non deve accadere, anziché una funzionalità del sistema o un comportamento richiesto. Di solito non è possibile definire un comportamento indesiderato sotto forma di semplici vincoli che possono essere controllati dal sistema.

Se sono disponibili le risorse, si può dimostrare, almeno in teoria, che un sistema soddisfa i requisiti funzionali, ma è impossibile provare che un sistema non fa qualcosa. Indipendentemente dal numero di test, le vulnerabilità della protezione possono restare in un sistema anche dopo che è stato consegnato.

Ovviamente, possiamo generare i requisiti funzionali che servono a proteggere il sistema contro qualche tipo di attacco che conosciamo. D'altra parte, non siamo in grado di ricavare i requisiti per tipi di attacchi non noti o imprevisti. Perfino nei sistemi che sono in uso da molti anni, un hacker ingegnoso può ideare una nuova forma di attacco e violare un sistema che era considerato protetto.

2. Gli hacker sono persone intelligenti che cercano attivamente le vulnerabilità che possono sfruttare per accedere a un sistema. Mettono alla prova il sistema, sperimentando operazioni che sono molto lontane dalla normale attività del sistema o dal suo tipico utilizzo. Per esempio, in un campo riservato ai cognomi possono immettere 1000 caratteri con una combinazione di lettere, numeri e segni di punteggiatura, semplicemente per vedere come risponde il sistema.

Una volta che hanno trovato una vulnerabilità, la rendono pubblica in modo che possa aumentare il numero di possibili hacker. Utilizzano i forum di Internet per scambiarsi le informazioni sulle vulnerabilità dei sistemi. Questo è anche un mercato fiorente di malware, dove gli hacker possono ottenere kit che li aiutano a sviluppare software dannoso, come worm o keystroke logger (codice in grado di intercettare tutto ciò che viene digitato sulla tastiera senza che l'utente se ne accorga).

Gli hacker possono tentare di scoprire le ipotesi fatte dagli sviluppatori di un sistema e poi provare queste ipotesi per vedere che cosa accade. Utilizzano e provano un sistema, analizzandone il comportamento per un determinato periodo di tempo tramite appositi strumenti software che sono in grado di scoprire eventuali vulnerabilità. Gli hacker, in effetti, hanno più tempo da destinare alla ricerca di vulnerabilità di quanto ne abbiano gli ingegneri per provare il sistema.

Possiamo utilizzare una combinazione di test, analisi supportata da strumenti e verifiche formali per controllare e analizzare la protezione di un sistema di applicazioni.

1. *Test basati sull'esperienza* – In questi casi, il sistema viene analizzato considerando tipi di attacchi noti al team di convalida. Questo potrebbe richiedere lo sviluppo di casi di test o l'esame del codice sorgente del sistema. Per esempio, per verificare che il sistema non può subire il noto attacco di “avvelenamento di SQL”, potremmo testare il sistema utilizzando input che includono comandi SQL. Per controllare che non si verifichino errori di overflow dei buffer, possiamo esaminare tutti i buffer di input per vedere se il programma ha controllato che le assegnazioni degli elementi dei buffer siano entro i limiti.

Lista di controllo della protezione
1. Tutti i file creati nell'applicazione hanno i permessi di accesso appropriati? I permessi di accesso sbagliati possono permettere l'accesso a questi file da parte di utenti non autorizzati.
2. Il sistema chiude automaticamente le sessioni dell'utente dopo un periodo di inattività? Le sessioni che rimangono attive possono permettere accessi non autorizzati tramite un computer lasciato incustodito.
3. Se il sistema è scritto in un linguaggio di programmazione che non verifica i limiti degli array, ci sono situazioni in cui può essere sfruttato l'overflow di un buffer? Gli overflow dei buffer possono permettere agli hacker di inviare stringhe di codice al sistema e di farle eseguire.
4. Quando vengono impostate le password, il sistema controlla che siano "forti"? Le password forti sono composte da un insieme di lettere, numeri e segni di punteggiatura, e di solito non sono voci del dizionario. Sono più difficili da violare delle password semplici.
5. Gli input provenienti dall'ambiente operativo del sistema vengono sempre controllati per verificare che abbiano il formato specificato per gli input? L'elaborazione di input di forma errata è una tipica causa di vulnerabilità della protezione.

Figura 13.17 Esempio di lista di controllo della protezione di un sistema.

Potremmo definire una lista di controllo per i problemi noti della protezione. La Figura 13.17 riporta alcuni esempi di domande che potrebbero essere utilizzate per guidare i test basati sull'esperienza. Questa lista potrebbe includere anche i controlli per verificare che siano state applicate le direttive di progettazione e programmazione.

2. *Test di violazione* – È un tipo di test basato sull'esperienza dove è possibile sfruttare l'esperienza di persone esterne al team di sviluppo per testare la protezione di un sistema di applicazioni. I team che eseguono questi test hanno l'obiettivo di violare la protezione del sistema. Simulano attacchi al sistema e usano lo loro ingenuità per scoprire nuovi modi di compromettere la protezione del sistema. I membri del team devono avere una certa esperienza con le tecniche di prova e identificazione delle vulnerabilità della protezione dei sistemi.
3. *Analisi supportata da strumenti* – In questo approccio sono utilizzati diversi strumenti di protezione (come i controllori delle password) per analizzare il sistema. I controllori delle password individuano le password deboli, che di solito sono formate da nomi comuni o da stringhe di lettere consecutive. Si tratta in realtà di un'estensione dei test basati sull'esperienza, dove le conoscenze delle violazioni della protezione sono integrate negli strumenti utilizzati. L'analisi statica è, ovviamente, un altro tipo di analisi supportata da strumenti, il cui impiego è in costante crescita.

L'analisi statica supportata da strumenti (Capitolo 12) è un approccio particolarmente utile per verificare la protezione di un sistema. Grazie a questa analisi, il team che esegue i test può disporre di una guida efficace tra le aree di un programma che potrebbero includere errori e vulnerabilità. Le anomalie rilevate dall'analisi statica possono essere direttamente eliminate oppure possono servire per progettare i test che verificano se esse rappresentano un rischio reale per il sistema. Microsoft usa regolarmente l'analisi statica per identificare possibili vulnerabilità della protezione (Jenney 2013). Hewlett-Packard offre uno strumento chiamato Fortify (Hewlett-Packard 2012) appositamente progettato per controllare la presenza di vulnerabilità della protezione nei programmi Java.

4. *Verifica formale* – Ho descritto l'utilizzo della verifica formale dei programmi nei Capitoli 10 e 12. Si tratta essenzialmente di formulare delle argomentazioni matematiche formali che dimostrano che un programma è conforme alla sua specifica. Hall e Chapman (Hall e Chapman 2002) hanno dimostrato che è possibile dimostrare che un sistema soddisfa i suoi requisiti formali di protezione più di 10 anni fa; da allora sono stati eseguiti numerosi esperimenti. Tuttavia, come accade in altre aree, la verifica formale della protezione non è molto utilizzata; richiede specialisti esperti e, quindi, è improbabile che sia economicamente conveniente come l'analisi statica.

I test della protezione richiedono molto tempo e, di solito, il tempo a disposizione del team di test è limitato. Questo significa che dobbiamo adottare un approccio basato sui rischi per testare la protezione e concentrare la nostra attenzione sui rischi più significativi per il sistema. Se abbiamo eseguito l'analisi dei rischi di protezione del sistema, possiamo utilizzare i risultati di questa analisi come guida nel processo di test. Oltre a verificare che il sistema soddisfa i requisiti di protezione derivati da questi rischi, il team di test dovrà anche mettere a dura prova il sistema mediante approcci alternativi che minacciano le risorse del sistema.

Punti chiave

- L'ingegneria della protezione si occupa delle tecniche di sviluppo e mantenimento di sistemi software che siano in grado di resistere ad attacchi che hanno lo scopo di danneggiare il sistema stesso o i suoi dati.
- Le minacce alla protezione sono rivolte alla riservatezza, all'integrità e alla disponibilità di un sistema o dei suoi dati.
- La gestione dei rischi di protezione si occupa di valutare le perdite che potrebbero derivare da attacchi al sistema e di definire i requisiti di protezione necessari per eliminare o ridurre queste perdite.

- Per specificare i requisiti di protezione bisogna identificare le risorse che devono essere protette e definire i metodi e le tecnologie da adottare per proteggere queste risorse.
- I problemi chiave da affrontare quando si progetta l’architettura di un sistema protetto sono l’organizzazione della struttura del sistema per proteggere le risorse più importanti e la distribuzione delle risorse per ridurre al minimo le perdite dovute a un attacco condotto con successo.
- Le linee guida per progettare sistemi protetti sensibilizzano i progettisti dei sistemi sui problemi della protezione che non sono stati considerati. Forniscono una base per creare liste di controllo della protezione dei sistemi.
- La convalida della protezione è un processo difficile, in quanto i requisiti di protezione stabiliscono ciò che non deve accadere in un sistema, non ciò che dovrebbe accadere. Inoltre, gli hacker sono persone intelligenti che, di solito, hanno più tempo per ricercare eventuali vulnerabilità di quanto ne abbiano gli ingegneri per verificare la protezione di un sistema.

Esercizi

- * 13.1 Spiegate le differenze principali tra ingegneria della protezione delle applicazioni e ingegneria della protezione delle infrastrutture.
- * 13.2 Suggerite per il sistema Mentcare un esempio di risorsa, esposizione, vulnerabilità, attacco, minaccia e controllo, oltre a quelli descritti in questo capitolo.
- 13.3 Spiegate perché c’è bisogno sia della valutazione preliminare dei rischi della protezione sia della valutazione dei rischi di progettazione durante lo sviluppo di un sistema.
- 13.4 Estendete la tabella della Figura 13.7 per identificare due ulteriori minacce per il sistema Mentcare, insieme ai controlli associati; utilizzate la nuova tabella per formulare i requisiti di protezione del software che implementano i controlli proposti.
- 13.5 Utilizzando un’analogia tratta da un contesto di ingegneria diverso dal software, spiegate perché dovrebbe essere utilizzato un approccio a strati per la protezione delle risorse.
- * 13.6 Spiegate perché è importante utilizzare tecnologie differenti per supportare sistemi distribuiti nei casi in cui la disponibilità dei sistemi sia critica.
- * 13.7 Per il sistema di trading azionario descritto nel Paragrafo 13.4.2, la cui architettura è illustrata nella Figura 13.14, suggerite due ulteriori possibili attacchi al sistema e proponete le strategie appropriate per contrastare questi attacchi.
- 13.8 Spiegate perché in un sistema protetto è importante convalidare tutti gli input dell’utente per verificare che abbiano il formato previsto.
- * 13.9 Suggerite una tecnica per convalidare un sistema di protezione delle password per un’applicazione che avete sviluppato. Spiegate la funzione di qualsiasi strumento che ritenete possa essere utile.

- 13.10 Il sistema Mentcare deve essere protetto contro gli attacchi che potrebbero svelare le informazioni riservate dei pazienti. Suggerite tre attacchi che potrebbero essere portati a questo sistema. Utilizzando queste informazioni, estendete la lista di controllo della Figura 13.17 per guidare coloro che eseguiranno i test del sistema Mentcare.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd ed. Una trattazione completa dei problemi relativi alla costruzione di sistemi protetti. L'autore focalizza l'attenzione sui sistemi, anziché sull'ingegneria del software, analizzando dettagliatamente l'hardware e le reti; sono descritti alcuni eccellenti esempi tratti da fallimenti di sistemi reali. (R. Anderson, John Wiley & Sons, 2008) <http://www.cl.cam.ac.uk/~rja14/book.html>

24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them. È uno dei migliori libri pratici sulla programmazione di sistemi protetti. Gli autori descrivono le vulnerabilità più comuni della protezione e spiegano come possono essere evitate nella pratica. (M. Howard, D. LeBlanc e J. Viega, McGraw-Hill, 2009).

Computer Security: Principles and Practice. Un buon testo sui problemi generali della protezione dei computer. Tratta le tecnologie di protezione, i sistemi fidati, la gestione della protezione e la crittografia. (W. Stallings e L. Brown, Addison-Wesley, 2012).

CAPITOLO

e14

Ingegneria della resilienza

L'obiettivo di questo capitolo è spiegare il concetto di ingegneria della resilienza, ovvero progettare sistemi in modo che possano resistere a eventi esterni avversi, come gli errori degli operatori e gli attacchi cibernetici. Dopo aver letto questo capitolo:

- capirete la differenza tra resilienza, affidabilità e protezione, e perché la resilienza è importante per i sistemi in rete;
- apprenderete i concetti fondamentali per realizzare sistemi resilienti, ovvero identificazione dei problemi, resistenza ai guasti e agli attacchi, ripristino dei servizi critici e delle funzioni del sistema;
- capirete perché la resilienza è un problema sociotecnico, anziché tecnico, e conoscerete il ruolo degli operatori e dei manager di sistema nel fornire la resilienza;
- apprenderete un metodo di progettazione che supporta la resilienza dei sistemi.

14.1 Protezione cibernetica

14.2 Resilienza sociotecnica

14.3 Progettazione di sistemi resilienti

Nell’aprile 1970, la navicella Apollo 13 in missione verso la luna ebbe un guasto gravissimo. Un serbatoio di ossigeno esplose, causando notevoli perdite di ossigeno atmosferico e di ossigeno per le celle a combustibile che fornivano energia elettrica alla navicella. La situazione era molto pericolosa per la vita degli uomini dell’equipaggio, perché non c’era alcuna possibilità di soccorrerli. Non c’erano piani di emergenza per questo tipo di incidente. Tuttavia, utilizzando le apparecchiature in modi alternativi al loro normale utilizzo e adattando le procedure standard alla situazione di emergenza, l’equipaggio e il personale di terra riuscirono a risolvere i problemi. La navicella rientrò sulla Terra e i membri dell’equipaggio si salvarono. L’intero sistema (persone, apparecchiature e processi) era *resiliente*. Fu modificato per resistere al guasto e furono ripristinate le sue funzioni normali.

Nel Capitolo 10 ho introdotto il concetto di resilienza come uno degli attributi fondamentali della fidatezza dei sistemi, definendo così la resilienza:

La resilienza di un sistema è la stima delle probabilità che il sistema possa garantire la continuità dei servizi critici al verificarsi di eventi dannosi, come il guasto di un componente o un attacco cibernetico.

Questa non è la definizione “standard” di resilienza – diversi autori, come Laprie (Laprie 2008) e Hollnagel (Hollnagel 2006), hanno proposto definizioni che si basano sulla capacità di un sistema di resistere ai cambiamenti; ovvero un sistema resiliente è quello che può operare con successo quando alcune delle ipotesi fondamentali fatte dai progettisti del sistema non sono più valide.

Per esempio, una tipica ipotesi iniziale di progettazione è che gli utenti potranno commettere degli errori utilizzando il sistema, ma non cercheranno deliberatamente le vulnerabilità del sistema per poterle sfruttare in modo illecito. Se il sistema è utilizzato in un ambiente in cui può subire attacchi cibernetici, questa ipotesi non è più valida. Un sistema resiliente è in grado di far fronte a un cambiamento dell’ambiente operativo e può continuare a funzionare correttamente.

Queste definizioni sono generiche. La mia definizione di resilienza, invece, è più vicina al modo in cui il termine viene oggi adottato nella pratica da istituzioni pubbliche e private. Essa esprime tre idee essenziali:

1. L’idea che qualcuno dei servizi offerti da un sistema sia un servizio critico, la cui interruzione può avere conseguenze umane, sociali o economiche gravi.
2. L’idea che qualche evento sia dannoso e possa influire sulla capacità di un sistema di fornire i suoi servizi critici.
3. L’idea che la resilienza sia una stima – non ci sono metriche che esprimano il livello di resilienza di un sistema, ovvero la resilienza non può essere misurata. La resilienza di un sistema può essere stimata soltanto da esperti, che esaminano il sistema e i suoi processi operativi.

Lo studio della resilienza dei sistemi ha avuto inizio nella comunità degli esperti dei sistemi a sicurezza critica, con l’obiettivo di capire quali erano i fattori che provocavano incidenti, le tecniche per evitare questi incidenti e salvaguardare i sistemi. Il numero sempre crescente di attacchi cibernetici portati ai sistemi connessi in rete ha indotto gli esperti a considerare la resilienza come una caratteristica fondamentale della protezione. La resilienza è essenziale per costruire sistemi che siano in grado di resistere agli attacchi cibernetici, continuando a fornire i servizi ai loro utenti.

Ovviamente, l’ingegneria della resilienza è strettamente correlata all’ingegneria dell’affidabilità e della protezione. Scopo dell’ingegneria dell’affidabilità è garantire che i sistemi non falliscano. Un fallimento del sistema è un evento osservabile dall’esterno, che spesso è la conseguenza di un errore del sistema. Per ridurre il numero di errori del sistema e per identificare gli errori prima che possano provocare fallimenti, sono state sviluppate varie tecniche, come quelle di fault avoidance e fault tolerance descritte nel Capitolo 11.

Nonostante il nostro massimo impegno, gli errori si verificano sempre in un sistema grande e complesso, e possono causare un fallimento del sistema. I piani di consegna dei sistemi sono sempre più stringenti e i budget per i loro test sono limitati. I membri dei team di sviluppo lavorano sotto pressione, ed è praticamente impossibile scoprire tutti gli errori e le vulnerabilità della protezione in un sistema software. Costruiamo sistemi talmente complessi che non riusciamo a capire tutte le interazioni tra i numerosi componenti dei sistemi. Alcune di queste interazioni possono innescare il fallimento di un intero sistema.

L’ingegneria della resilienza non si occupa delle tecniche per evitare i fallimenti, ma riconosce che i fallimenti possono accadere. Si basa su due importanti ipotesi.

1. L’ingegneria della resilienza suppone che sia impossibile evitare i fallimenti di un sistema; quindi, si occupa di limitare i costi dovuti a questi fallimenti e di ripristinare il funzionamento normale del sistema.
2. L’ingegneria della resilienza suppone che siano state adottate le buone pratiche dell’ingegneria dell’affidabilità per ridurre al minimo il numero di errori in un sistema. Di conseguenza, si concentra più sulle tecniche per limitare il numero di fallimenti che possono essere provocati da eventi esterni, come gli errori degli operatori o gli attacchi cibernetici.

In pratica, i fallimenti tecnici di un sistema sono innescati spesso da eventi che sono esterni al sistema. Questi eventi possono essere azioni svolte dagli operatori o errori degli utenti che sono imprevedibili. Negli ultimi anni, tuttavia, essendo cresciuto il numero di sistemi connessi in rete, questi eventi sono spesso attacchi cibernetici. In un attacco cibernetico, uno o più hacker tentano di danneggiare il sistema o rubare informazioni riservate. Questi eventi oggi sono più significativi degli errori degli utenti o degli operatori come fonte potenziale di fallimenti del sistema.

Sulla base dell’ipotesi che i fallimenti sono inevitabili, l’ingegneria della resilienza si occupa sia dell’immediata riparazione dei guasti per garantire i servizi critici sia del ripristino di tutti i servizi del sistema; quest’ultimo richiede tempi un po’ più lunghi della riparazione. Come dirò nel Paragrafo 14.3, questo significa che i progettisti del sistema devono realizzare alcune funzioni che permettano di preservare lo stato del software e dei dati del sistema. Se si verifica un fallimento, le informazioni essenziali devono essere ripristinate.

Le attività principali correlate alla resilienza sono quattro:

1. *Identificazione* – Il sistema o i suoi operatori dovrebbero essere in grado di riconoscere i sintomi di un problema che potrebbe causare il fallimento del sistema. Teoricamente, questa identificazione dovrebbe avvenire prima che si verifichi un fallimento.
2. *Resistenza* – Se i sintomi di un problema o di un attacco cibernetico vengono rilevati in tempo, possono essere attuate apposite strategie di resistenza che riducono la probabilità che il sistema possa fallire. Queste strategie hanno il compito di identificare le parti critiche del sistema che devono essere isolate in modo che non siano influenzate da altri problemi. La resistenza prevede un’attività di prevenzione, tramite apposite difese in grado di identificare i problemi, e un’attività di reazione nella quale vengono svolte le azioni appropriate per ogni problema che è stato identificato.
3. *Riparazione* – Se si verifica un fallimento, l’attività di riparazione ha il compito di garantire che i servizi critici del sistema siano rapidamente ripristinati, in modo che gli utenti del sistema non siano coinvolti nel guasto in modo significativo.
4. *Ripristino* – In questa attività finale, tutti i servizi del sistema vengono ripristinati e il funzionamento normale del sistema può continuare.

Queste attività implicano dei cambiamenti nello stato del sistema, come illustra la Figura 14.1, che mostra questi cambiamenti nel caso di attacco cibernetico. Oltre al normale funzionamento, il sistema controlla costantemente il traffico in rete per identificare eventuali attacchi di hacker. Nel caso di attacco, il sistema passa in uno stato di resistenza, in cui i servizi normali possono essere limitati.

Se la resistenza respinge con successo l’attacco, i servizi normali vengono ripristinati. Altrimenti, il sistema passa in uno stato di riparazione, dove sono forniti soltanto i servizi critici; qui vengono riparati i danni causati dall’attacco. Dopo che le riparazioni sono state completate, il sistema passa in uno stato di ripristino, dove i servizi vengono gradualmente ripristinati. Infine, dopo che tutte le operazioni di ripristino sono state completate, il sistema riprende il suo funzionamento normale.

Come insegnava l’esperienza dell’Apollo 13, la resilienza di un sistema non può essere “preventivamente programmata”. È impossibile prevedere tutto ciò che potrebbe guastarsi e tutti i contesti che potrebbero diventare problematici. La chiave della resilienza è quindi la flessibilità e l’adattabilità. Come dirò nel Para-

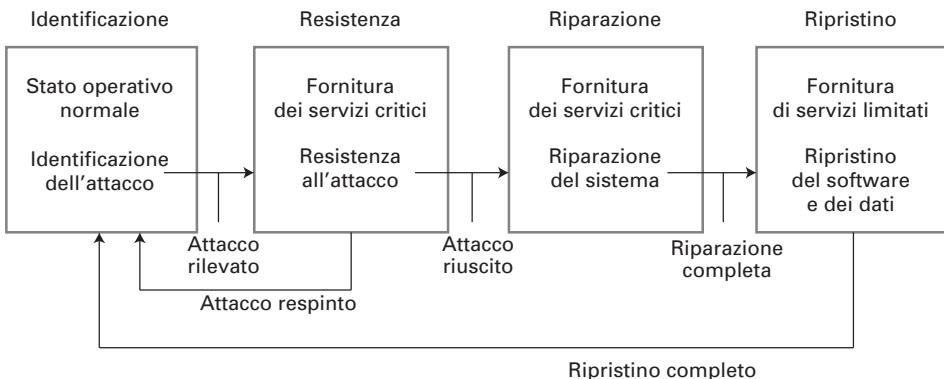


Figura 14.1 Attività della resilienza.

grafo 14.2, gli operatori e i manager di un sistema dovrebbero essere in grado di svolgere le azioni appropriate per proteggere e riparare il sistema, anche se queste azioni sono anomale o normalmente non consentite.

Migliorare la resilienza di un sistema, ovviamente, comporta dei costi. Potrebbe essere necessario acquistare un apposito software o modificare quello esistente; ulteriori investimenti potrebbero essere richiesti dall'hardware o dai servizi cloud per realizzare un sistema di backup da utilizzare nel caso di guasto del sistema principale. I vantaggi di questi costi sono impossibili da calcolare, perché le perdite provocate da un fallimento o da un attacco cibernetico possono essere valutate soltanto dopo che si sono verificati tali eventi.

Le società possono quindi essere riluttanti a investire nella resilienza, se non hanno mai sperimentato un attacco che ha causato gravi perdite. Tuttavia, il numero crescente di attacchi cibernetici di alto profilo che hanno danneggiato i sistemi software di aziende pubbliche e private ha accresciuto il bisogno di resilienza. È chiaro che le perdite possono essere molto significative, e alcune aziende potrebbero non sopravvivere a un attacco riuscito. Per questo, stanno progressivamente crescendo gli investimenti nell'ingegneria della resilienza per ridurre i rischi associati ai fallimenti dei sistemi.

14.1 Protezione cibernetica

Mantenere la protezione delle infrastrutture e dei sistemi software collegati in rete è uno dei problemi più significativi da affrontare. La vastissima diffusione di Internet e la nostra dipendenza dai sistemi computerizzati hanno creato nuove opportunità per i criminali informatici. È molto difficile quantificare le perdite causate da un crimine informatico. Tuttavia, le perdite dell'economia globale causate da crimini informatici nel 2013 sono state stimate tra 100 e 500 miliardi di dollari. (InfoSecurity).

Come detto nel Capitolo 13, la protezione cibernetica è un problema più ampio dell’ingegneria della protezione dei sistemi. L’ingegneria della protezione del software è un’attività principalmente tecnica che si occupa essenzialmente delle tecniche e delle tecnologie che sono in grado di assicurare che i sistemi applicativi siano protetti.

La protezione cibernetica è un argomento sociotecnico. Riguarda tutti gli aspetti della protezione dei cittadini, delle aziende e delle infrastrutture critiche contro le minacce che possono nascere dall’utilizzo dei computer e di Internet. Sebbene gli aspetti tecnici siano importanti, la tecnologia da sola non può garantire la protezione. I principali fattori che contribuiscono ai fallimenti della protezione cibernetica sono i seguenti:

- ignoranza della gravità del problema;
- progettazione inadeguata e applicazione permissiva delle procedure di protezione;
- incuria umana;
- compromessi inappropriati tra protezione e utilizzabilità.

La protezione cibernetica riguarda tutte le risorse informatiche di una società, dalle reti ai sistemi applicativi. La massima parte di queste risorse provengono dall’esterno, e le società non capiscono il loro funzionamento dettagliato. Sistemi come i browser web sono programmi complessi e, inevitabilmente, contengono bug che possono essere fonti di vulnerabilità. Sistemi differenti all’interno di una società sono collegati l’uno con l’altro in molti modi diversi; possono essere memorizzati sullo stesso disco, possono condividere dati, possono utilizzare componenti comuni del sistema operativo e così via. L’organizzazione dei “sistemi di sistemi” è incredibilmente complessa. È impossibile garantire che la sua protezione sia priva di vulnerabilità.

In generale, dovremmo dunque supporre che i nostri sistemi siano vulnerabili agli attacchi cibernetici e che, a un certo punto, sia probabile che si verifichi un attacco. Se un attacco cibernetico riesce, un’azienda può subire gravi perdite economiche. È essenziale quindi che gli attacchi siano contenuti e le perdite ridotte al minimo. Un’ingegneria efficiente della resilienza a livello delle aziende e dei sistemi può respingere tali attacchi e riportare rapidamente i sistemi al loro funzionamento normale, limitando significativamente le perdite.

Nel Capitolo 13, dove ho trattato l’ingegneria della protezione, ho introdotto i concetti fondamentali per la pianificazione della resilienza. Alcuni di questi concetti sono elencati qui di seguito.

1. *Risorse* – Sono i sistemi e i dati da proteggere. Alcune risorse sono più vulnerabili di altre e quindi richiedono un livello di protezione più elevato.
2. *Minacce* – Circostanze che possono causare perdite o danni alle infrastrutture di un’azienda o alle risorse di un sistema.

3. *Attacchi* – Sono manifestazioni di minacce, mediante le quali un hacker tenta di danneggiare una risorsa del sistema o rubare informazioni riservate, come i dati personali o di un sito web.

Tre tipi di minacce devono essere considerati nella pianificazione della resilienza.

1. *Minacce all'integrità delle risorse* – Sono le minacce nelle quali i sistemi o i dati vengono danneggiati in qualche modo da un attacco cibernetico. Un attacco può essere costituito dall'installazione di un virus o di un worm nel software del sistema o del database.
2. *Minacce alla disponibilità delle risorse* – Sono minacce che hanno il compito di negare l'uso delle risorse agli utenti autorizzati. L'esempio più noto è l'attacco denial-of-service che tenta di mettere fuori servizio un intero sito web in modo da renderlo indisponibile agli utenti esterni.

Queste non sono classi di minacce indipendenti. Un hacker potrebbe compromettere l'integrità del sistema di un utente immettendo un malware, come un componente di una botnet. Questo potrebbe essere chiamato da un'unità remota come parte di un attacco denial-of-service distribuito su un altro sistema. Altri tipi di malware possono essere utilizzati per catturare informazioni dettagliate su un utente, che permettono a un hacker di accedere alle risorse riservate all'utente.

Per contrastare queste minacce, le società dovrebbero dotarsi di controlli che complicano notevolmente l'accesso alle risorse da parte di persone non autorizzate. È anche importante innalzare i livelli di sensibilità sui problemi della protezione cibernetica, in modo che gli utenti dei sistemi sappiano perché questi controlli sono importanti e quindi siano meno disposti a rivelare informazioni personali a estranei.

Seguono alcuni esempi di controlli che potrebbero essere utilizzati.

1. *Autenticazione* – Gli utenti di un sistema devono dimostrare di essere autorizzati ad accedere al sistema. Il familiare metodo di autenticazione formato dal nome utente e dalla password, anche se debole, è un controllo universalmente utilizzato.
2. *Crittografia* – I dati vengono confusi da un apposito algoritmo, in modo che un lettore non autorizzato non riesca a interpretarli. Molte società adesso richiedono che i dati sui dischi dei loro computer portatili siano crittografati. In questo modo, se un dipendente perde il computer o glielo rubano, si riducono notevolmente le probabilità che le informazioni riservate siano violate.
3. *Firewall* – I pacchetti di dati che arrivano dalla rete vengono esaminati, poi accettati o respinti sulla base di determinate regole aziendali. I firewall possono essere utilizzati per garantire che solo i dati provenienti da fonti affidabili possano passare da Internet a una rete locale aziendale.

Una serie di diversi controlli in un’azienda realizza un sistema di protezione a strati. Un hacker dovrà superare tutti gli strati di protezione affinché un suo attacco possa avere successo. Tuttavia, occorre trovare un giusto compromesso tra protezione ed efficienza del sistema. Al crescere del numero di strati di protezione, si riduce la velocità del sistema. I sistemi di protezione consumano una crescente quantità di memoria e risorse del processore, lasciandone meno per lo svolgimento delle normali operazioni. Quanto più crescono le misure di protezione, tanto più è probabile che gli utenti adottino pratiche poco sicure per aumentare l’utilizzabilità del sistema.

Analogamente ad altri aspetti della fidatezza di un sistema, gli strumenti fondamentali di protezione contro gli attacchi cibernetici dipendono dalla ridondanza e dalla diversità. Ricordiamo che ridondanza significa avere capacità di riserva e risorse duplicate in un sistema. Diversità significa che vengono utilizzati vari tipi di software, procedure e dispositivi, in modo tale che un eventuale fallimento possa influire soltanto sulle parti che usano lo stesso tipo di sistema. Seguono alcuni esempi in cui la ridondanza e la diversità sono applicate alla resilienza cibernetica.

1. Per ogni sistema dovrebbero essere mantenute copie di dati e software su computer distinti. I dischi condivisi dovrebbero essere evitati, se possibile. In questo modo, si agevola il processo di recovery del sistema dopo un attacco cibernetici riuscito (riparazione e ripristino).
2. L’autenticazione a più stadi potrebbe essere una protezione contro gli attacchi tramite password. Oltre alla classica procedura di autenticazione nome utente/password, dovrebbero essere adottate altre procedure di autenticazione che richiedono agli utenti di fornire informazioni personali o un codice generato dalle loro unità mobili (resistenza).
3. I server critici potrebbero essere sovradimensionati, nel senso che potrebbero essere più potenti di quanto sia necessario per svolgere il loro normale carico di lavoro. La capacità in eccesso può essere utilizzata per resistere agli attacchi senza ridurre le prestazioni delle normali attività del server. Inoltre, se altri server vengono danneggiati, la capacità in eccesso di un server critico può essere messa a disposizione per eseguire il software dei server danneggiati mentre vengono riparati (resistenza e riparazione).

La pianificazione della protezione cibernetica deve basarsi sulle risorse, sui controlli e sulle quattro caratteristiche dell’ingegneria della resilienza: identificazione, resistenza, riparazione e ripristino. La Figura 14.2 mostra un processo di pianificazione che potrebbe essere utilizzato come modello. Le fasi chiave di questo processo sono qui di seguito descritte.

1. *Classificazione delle risorse* – Le risorse umane, software e hardware vengono esaminate e classificate in base a quanto siano essenziali per lo svolgimento delle normali operazioni. Possono essere classificate come critiche, importanti o utili.

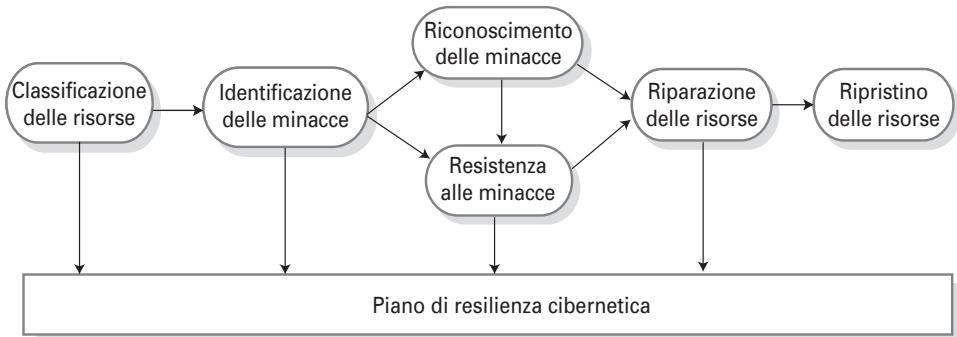


Figura 14.2 Pianificazione della protezione cibernetica.

2. *Identificazione delle minacce* – Occorre identificare e classificare le minacce per ogni risorsa (o almeno per le risorse critiche e importanti). In alcuni casi, si potrebbe provare a stimare la probabilità che ha una minaccia di presentarsi, anche se queste stime sono spesso poco precise in quanto non si hanno informazioni sufficienti sui potenziali hacker.
3. *Riconoscimento delle minacce* – Per ciascuna minaccia o, in alcuni casi, per ogni coppia di risorsa/minaccia, occorre identificare come potrebbe essere riconosciuto un attacco basato su tale minaccia. Potrebbe essere necessario acquistare o sviluppare un software addizionale per il riconoscimento delle minacce o mettere in atto delle apposite procedure di verifica.
4. *Resistenza alle minacce* – Occorre identificare le possibili strategie di resistenza per ciascuna minaccia o coppia di risorsa/minaccia. Queste strategie possono essere integrate nel sistema (strategie tecniche) o in apposite procedure operative. Potreste anche studiare delle strategie di neutralizzazione delle minacce per impedire che una minaccia possa ripetersi.
5. *Riparazione delle risorse* – Per ciascuna risorsa critica o coppia di risorsa/minaccia, occorre definire come deve essere riparata una risorsa dopo che ha subito un attacco cibernetico. Questo potrebbe richiedere l'utilizzo di unità hardware aggiuntive o la modifica delle procedure di backup per semplificare l'accesso alle copie di dati ridondanti.
6. *Ripristino delle risorse* – Questo è un processo più generale nel quale vengono definite le procedure per ripristinare il funzionamento normale del sistema. Il ripristino delle risorse dovrebbe occuparsi di tutte le risorse, non semplicemente di quelle che sono critiche per una società.

Le informazioni raccolte in tutte queste fasi dovrebbero essere conservate in un piano di resilienza cibernetica. Questo piano dovrebbe essere aggiornato periodicamente e, se possibile, le strategie identificate dovrebbero essere provate simulando degli attacchi al sistema.

Un'altra parte importante della pianificazione della resilienza cibernetica riguarda le scelte sulle azioni da svolgere nel caso di attacco cibernetico. Paradossalmente, i requisiti di resilienza e protezione sono spesso in conflitto. Scopo della protezione di solito è limitare quanto più possibile i privilegi, in modo che gli utenti possano fare soltanto ciò che le politiche aziendali di protezione consentono. Tuttavia, per risolvere un problema, un utente o un operatore del sistema potrebbe prendere l'iniziativa per svolgere delle azioni che normalmente sono riservate a persone con privilegi di alto livello.

Per esempio, il manager di un sistema medico di solito non può modificare i diritti di accesso del personale medico ai record del database. Per motivi di protezione, i permessi di accesso devono essere formalmente autorizzati, e due persone devono essere coinvolte nella modifica di questi permessi. Così facendo, si riducono i rischi che un manager del sistema possa colludere con gli hacker e consentire l'accesso a informazioni mediche riservate.

Adesso, supponiamo che il manager abbia notato che un utente collegato al sistema stia accedendo a un gran numero di record al di fuori del normale orario di lavoro. Il manager ha il sospetto che un account di accesso sia stato violato e che l'utente che sta accedendo ai record non sia un utente legalmente autorizzato. Per limitare i danni, i diritti di accesso di questo utente dovrebbero essere immediatamente annullati; successivamente, si potrebbe verificare se l'utente autorizzato abbia realmente effettuato l'accesso fuori orario. Tuttavia, le procedure di protezione, che non consentono al manager di modificare i diritti di accesso degli utenti, rendono impossibile questo intervento.

La pianificazione della resilienza dovrebbe tenere conto di queste situazioni. Un modo per farlo consiste nell'includere nei sistemi una modalità di "emergenza" nella quale i normali controlli sono ignorati. Anziché proibire le operazioni, il sistema registra che cosa viene fatto e chi è il responsabile. Successivamente, la lettura delle operazioni di emergenza che sono state registrate può essere utilizzata per verificare se le azioni svolte dal manager erano giustificate. Ovviamente, anche qui c'è spazio per un uso improprio del sistema; la modalità di emergenza è essa stessa una potenziale vulnerabilità. Quindi, le società devono trovare un giusto compromesso tra le possibili perdite e i benefici di aggiungere più funzioni a supporto della resilienza.

14.2 Resilienza sociotecnica

L'ingegneria della resilienza è essenzialmente un'attività sociotecnica, anziché puramente tecnica. Come detto nel Capitolo 10, un sistema sociotecnico è formato da software, hardware e persone, ed è influenzato dalla cultura, dalle politiche e dalle procedure della società che possiede e usa il sistema. Per progettare un sistema resiliente, occorre considerare tutti i componenti che formano questo tipo di sistema, non soltanto il software. L'ingegneria della resilienza si occupa di

eventi esterni ostili al sistema che possono causare il suo fallimento. La gestione di questi eventi spesso è più facile ed efficace se vengono trattati all'interno del più ampio contesto dei sistemi sociotecnici.

Per esempio, il sistema Mentcare gestisce le informazioni riservate dei pazienti; un possibile attacco cibernetico potrebbe rubare tali informazioni. È possibile adottare alcune misure tecniche, come l'autenticazione e la crittografia, per proteggere questi dati, ma queste misure non sono efficaci se un hacker è in possesso delle credenziali di un utente regolarmente autorizzato. Potremmo tentare di risolvere questo problema a livello tecnico, utilizzando procedure di autenticazione più complesse. Tuttavia, queste procedure di solito infastidiscono gli utenti e possono causare delle vulnerabilità, in quanto gli utenti tendono a scrivere i dati di autenticazione su un foglio di carta. Una strategia migliore potrebbe essere quella di introdurre delle politiche e procedure aziendali che danno particolare enfasi all'importanza di non condividere le credenziali di accesso al sistema e che spiegano agli utenti alcuni metodi semplici per creare e gestire password più complesse e robuste.

I sistemi resilienti sono flessibili e adattabili, quindi possono affrontare eventi imprevisti. È molto difficile creare un software che possa adattarsi in modo automatico per risolvere problemi che non possono essere previsti. Tuttavia, come abbiamo visto nel caso del guasto verificatosi sull'Apollo 13, le persone sono molto indicate per svolgere questo compito. Per ottenere la resilienza, si dovrebbe sfruttare il fatto che le persone sono una parte importante dei sistemi sociotecnici. Anziché tentare di prevedere e gestire tutti i problemi del software, si dovrebbe lasciare la soluzione di alcuni di questi problemi alle persone che hanno la responsabilità del funzionamento e della gestione del sistema software.

Per capire perché si dovrebbe lasciare la soluzione di alcuni di questi problemi alle persone, bisogna considerare la struttura gerarchica dei sistemi sociotecnici che include sistemi tecnici con uso intensivo del software. La Figura 14.3 mostra che i sistemi tecnici S1 e S2 sono parte di un più ampio sistema sociotecnico ST1; quest'ultimo include gli operatori che monitorano le condizioni di S1 e S2, e che possono decidere di intervenire per risolvere alcuni problemi di questi sistemi. Se, per esempio, il sistema S1 fallisce, gli operatori di ST1 possono rilevare questo problema e intraprendere le azioni appropriate di recovery prima che il fallimento locale di S1 si traduca in un fallimento del più ampio sistema sociotecnico. Gli operatori possono anche avviare le procedure di riparazione e ripristino per riportare il sistema S1 al suo normale funzionamento.

I processi che riguardano il funzionamento e la gestione sono l'interfaccia tra la società e i sistemi tecnici che sono utilizzati. Se questi processi sono ben progettati, consentono alle persone di scoprire e risolvere i problemi dei sistemi tecnici, e di garantire che gli errori degli operatori siano ridotti al minimo. Come dirò nel Paragrafo 14.2.2, i processi rigidi superautomatici non sono intrinsecamente resilienti, in quanto non consentono alle persone di sfruttare le proprie

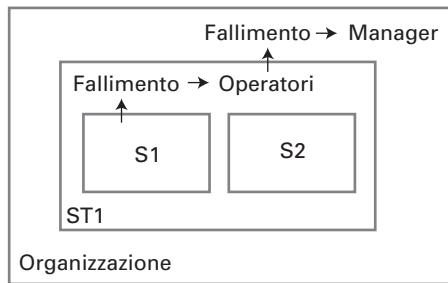


Figura 14.3 Sistemi sociotecnici e tecnici annidati.

capacità e conoscenze per adattare e modificare i processi nel caso in cui si verifichi un evento imprevisto.

Il sistema ST1 è uno dei tanti sistemi sociotecnici di una società. Se gli operatori non possono risolvere un problema di un sistema tecnico, questo potrebbe tradursi in un fallimento del sistema sociotecnico ST1. I manager responsabili dell'organizzazione devono quindi rilevare il problema e svolgere le azioni appropriate per risolverlo. La resilienza è dunque una caratteristica che riguarda sia l'organizzazione sia il sistema.

Hollnagel (Hollnagel 2010), uno dei primi sostenitori dell'ingegneria della resilienza, ritiene che sia importante per le organizzazioni studiare e imparare dai successi ottenuti, ma anche dagli insuccessi. I fallimenti di alto profilo delle misure di protezione e sicurezza portano inevitabilmente a cambiare le pratiche e le procedure esistenti. Tuttavia, anziché cercare soluzioni per questi fallimenti, è meglio evitarli osservando come le persone trattano i problemi e garantiscono la resilienza. Questa buona pratica può quindi essere diffusa in tutta l'organizzazione. La Figura 14.4 mostra le quattro caratteristiche, suggerite da Hollnagel, che rispecchiano la resilienza di un'organizzazione.

1. *Capacità di rispondere* – Le organizzazioni devono essere in grado di adattare i loro processi e procedure per rispondere ai problemi che si presentano. Questi problemi possono essere rischi previsti o minacce rilevate. Per esempio, se viene rilevata una nuova minaccia alla protezione, un'organizzazione resiliente può apportare rapidamente le modifiche appropriate in modo che tale minaccia non possa danneggiare le sue operazioni.
2. *Capacità di monitorare* – Le organizzazioni dovrebbero monitorare le loro operazioni interne e il loro ambiente esterno per rilevare eventuali minacce. Per esempio, una società potrebbe monitorare come i suoi dipendenti applicano le politiche di protezione. Se viene rilevato un comportamento che viola le norme di protezione da parte di un dipendente, la società dovrebbe svolgere le azioni appropriate per capire perché ciò è avvenuto e poi corregere il comportamento del dipendente.

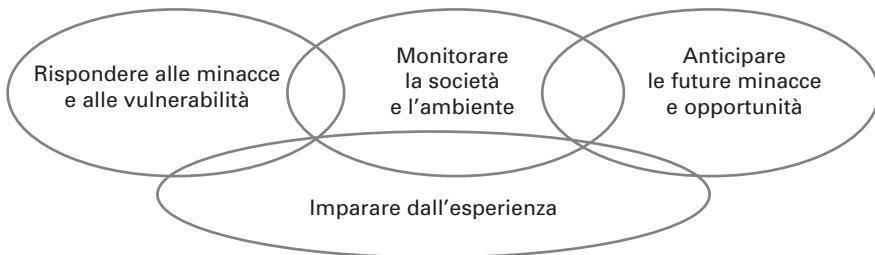


Figura 14.4 Caratteristiche di un'organizzazione resiliente.

3. *Capacità di prevedere* – Un'organizzazione resiliente non dovrebbe concentrarsi semplicemente sulle sue operazioni correnti, ma dovrebbe prevedere i possibili eventi e cambiamenti futuri che potrebbero influire sul funzionamento e sulla resilienza dei suoi sistemi. Questi eventi possono essere innovazioni tecnologiche, modifiche di regolamenti o leggi, e cambiamenti nei comportamenti degli utenti. Per esempio, la tecnologia indossabile sta iniziando a diffondersi; le società dovrebbero quindi capire come questo fenomeno possa influire sulle loro attuali politiche e procedure di protezione.
4. *Capacità di imparare* – La resilienza di un'organizzazione può essere migliorata imparando dall'esperienza. È particolarmente importante imparare dai successi ottenuti contro eventi avversi o attacchi cibernetici. Sfruttando l'esperienza positiva di questi successi, si possono definire delle buone pratiche di protezione che possono essere diffuse in tutta l'organizzazione.

Come sostiene Hollnagel, per diventare resilienti, le organizzazioni devono affrontare in qualche modo tutti questi problemi. Alcune organizzazioni si concentreranno sulla qualità più di altre. Per esempio, una società che usa un centro dati su larga scala potrebbe concentrarsi principalmente sul monitoraggio e sulla reattività del centro. D'altra parte, una libreria digitale che gestisce informazioni di archiviazione a lungo termine potrebbe essere più interessata a prevedere come le modifiche future possano influire sulla sua attività e come rispondere a qualsiasi minaccia alla protezione dei suoi sistemi.

14.2.1 Errore umano

I primi studi dell'ingegneria della resilienza riguardavano gli incidenti che si verificavano nei sistemi a sicurezza critica e le azioni degli operatori che potevano provocare un fallimento nella protezione dei sistemi. Questi studi spiegarono come difendere i sistemi che devono resistere alle azioni umane che sono accidentalmente o deliberatamente dannose per i sistemi.

Sappiamo che le persone possono commettere degli errori e, a meno che un sistema non sia completamente automatizzato, è inevitabile che gli utenti e gli operatori del sistema a volte sbagliano. Purtroppo, questi errori umani in alcuni casi provocano gravi fallimenti del sistema. Reason (Reason 2000) ritiene che il problema dell'errore umano può essere considerato in due modi.

1. *Approccio alle persone* – Gli errori sono da attribuire alla responsabilità dei singoli individui e le “azioni non sicure” (come un operatore che non chiude una barriera di protezione) sono una conseguenza dell’incuria o dell’imprudenza degli individui. Chi sostiene questo approccio, ritiene che gli errori umani possono essere ridotti minacciando azioni disciplinari, attuando procedure più rigorose, riqualificando il personale e così via. In altri termini, responsabile dell’errore è l’individuo che ha sbagliato.
2. *Approccio ai sistemi* – L’ipotesi di base è che le persone sono fallibili e, quindi, commetteranno qualche errore. Le persone sbagliano perché sono sotto pressione, essendo oberate di lavoro, perché non sono adeguatamente addestrate o perché il sistema non è progettato bene. I buoni sistemi riconoscono la possibilità dell’errore umano e includono barriere e meccanismi in grado di rilevare l’errore umano che consentono di salvaguardare un sistema dal fallimento. Se si verifica un fallimento, il modo migliore per evitare che si ripeta è capire come e perché le difese del sistema non abbiano rilevato preventivamente l’errore che lo ha causato. Incolpare e punire la persona che ha provocato il fallimento non migliora la sicurezza a lungo termine del sistema.

I credo che l’approccio ai sistemi sia quello giusto e che gli ingegneri debbano supporre che ci sarà sempre l’errore umano durante il funzionamento di un sistema. Pertanto, per migliorare la resilienza di un sistema, i progettisti dovrebbero studiare le difese e le barriere agli errori umani che potrebbero far parte del sistema; dovrebbero decidere se sia opportuno integrare queste difese nei componenti tecnici del sistema; se questo non è possibile, le difese potrebbero essere integrate nei processi, nelle procedure e nelle linee guida all’utilizzo del sistema. Per esempio, potrebbero essere necessari due operatori per controllare gli input di un sistema critico.

Le difese che proteggono dagli errori umani possono essere tecniche o sociotecniche. Per esempio, il codice per convalidare tutti gli input è una difesa tecnica; una procedura per approvare gli aggiornamenti dei sistemi critici che richiede la conferma di due operatori è una difesa sociotecnica. Utilizzando barriere difensive differenti, ci sono meno probabilità che una vulnerabilità possa essere condivisa tra più componenti del sistema, ma è più probabile che l’errore di un utente venga rilevato prima che si verifichi il fallimento del sistema.

In generale, i progettisti dovrebbero applicare i concetti di ridondanza e diversità per creare una serie di strati difensivi (Figura 14.5), dove ogni strato usa un approccio differente per respingere gli attacchi degli hacker o per rilevare gli

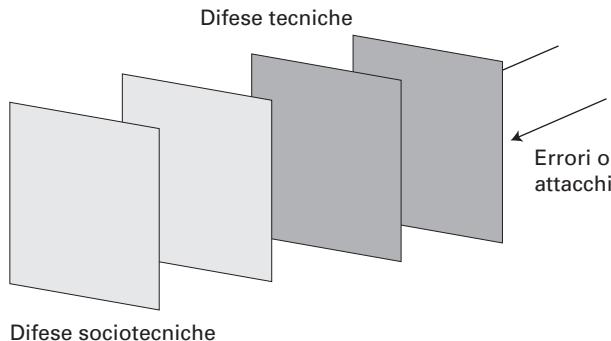


Figura 14.5 Strati difensivi.

errori dei componenti o degli utenti. Le barriere più scure rappresentano i controlli svolti dal software; le barriere più chiare rappresentano i controlli svolti dalle persone.

Come esempio di questo metodo di difesa in profondità, riportiamo alcuni controlli per rilevare gli errori che potrebbero far parte di un sistema di controllo del traffico aereo.

1. *Avviso di conflitto come parte di un sistema di controllo del traffico aereo* – Quando un controllo indica all'aereo di modificare la velocità o l'altitudine, il sistema estrapola la nuova traiettoria per vedere se essa interagisce con altri aerei in volo; in caso affermativo, il sistema emette un allarme acustico.
2. *Procedure formali di registrazione per la gestione del traffico aereo* – Lo stesso sistema di controllo potrebbe avere una procedura ben definita che stabilisce come registrare le istruzioni di controllo che vengono impartite. Queste registrazioni mettono a disposizione di terzi le informazioni sulle istruzioni di controllo e agevolano le eventuali verifiche sulla correttezza delle istruzioni emesse.
3. *Verifiche reciproche* – Il controllo del traffico aereo richiede un team di controllori, ciascuno dei quali monitora costantemente il lavoro degli altri. Quando un controllore commette un errore, gli altri controllori lo rilevano e lo coraggono prima che si verifichi un incidente.

Reason (Reason 200) ha utilizzato il concetto di strati difensivi per elaborare una teoria su come l'errore umano può portare al fallimento di un sistema. Ha introdotto il modello del formaggio svizzero, suggerendo che gli strati difensivi non sono solide barriere, ma piuttosto somigliano alle fette del formaggio svizzero. Alcuni tipi di formaggio svizzero, come l'Emmenthal, hanno buchi di varie dimensioni al loro interno. Reason ritiene che le vulnerabilità, o ciò che egli chiama condizioni latenti negli strati, sono simili a questi buchi.

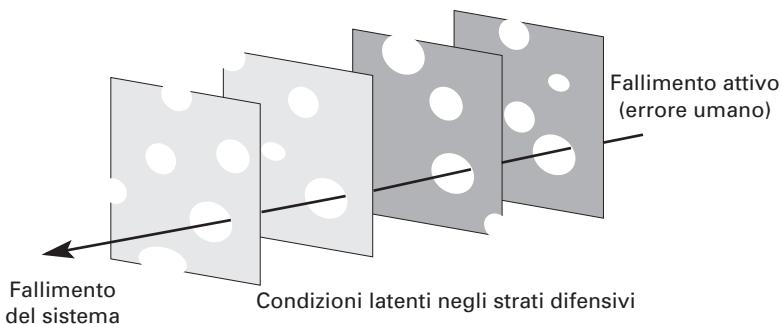


Figura 14.6 Fallimento di un sistema secondo il modello del formaggio svizzero di Reason.

Queste condizioni latenti non sono statiche – cambiano in base allo stato del sistema e alle persone coinvolte nell'utilizzo del sistema. Continuando con l'analogia del formaggio svizzero, i buchi cambiano dimensione e si spostano all'interno degli strati difensivi durante il funzionamento del sistema. Per esempio, se un sistema si affida alle verifiche reciproche di due operatori, una possibile vulnerabilità potrebbe essere che entrambi gli operatori commettano lo stesso errore. Questo è poco probabile in condizioni normali e quindi, secondo il modello del formaggio svizzero, significa che il buco è piccolo. Tuttavia, quando il sistema è sovraccaricato e il carico di lavoro per entrambi gli operatori è alto, allora gli errori sono più probabili; aumenta quindi la dimensione del buco che rappresenta questa vulnerabilità.

Il fallimento di un sistema con strati difensivi si verifica quando c'è qualche evento esterno che ha la potenzialità di provocare qualche danno. Questo evento potrebbe essere un errore umano (che Reason chiama fallimento attivo) o un attacco cibernetico. Se tutte le barriere difensive falliscono, il sistema nel suo insieme fallirà. Teoricamente, questo corrisponde al caso dei buchi allineati nelle fette del formaggio svizzero, come illustra la Figura 14.6.

Questo modello suggerisce che è possibile adottare diverse strategie per aumentare la resilienza agli eventi esterni avversi.

1. Ridurre la probabilità che si verifichi un evento esterno che potrebbe innescare il fallimento del sistema. Per ridurre gli errori umani, si potrebbero introdurre dei corsi di addestramento per gli operatori oppure si potrebbe dare maggiore autonomia agli operatori, in modo che possano abbassare il loro carico di lavoro, evitando condizioni di stress eccessivo. Per ridurre gli attacchi cibernetici, si potrebbe ridurre il numero di persone che hanno informazioni privilegiate sul sistema, limitando così i rischi che gli hacker possano venire in possesso di tali informazioni.
2. Aumentare il numero di strati difensivi. Come regola generale, quanto più è alto il numero di strati difensivi, tanto meno probabile è che i buchi possano trovarsi allineati provocando il fallimento del sistema. Tuttavia, se

questi strati non sono indipendenti, allora potrebbero avere in comune la stessa vulnerabilità. Quindi, è probabile che le barriere abbiano lo stesso “buco” nello stesso punto; in questo caso, il beneficio di aggiungere un nuovo strato è molto limitato.

3. Progettare i sistemi in modo da includere vari tipi di barriere difensive. Questo significa che i “buchi” molto probabilmente si troveranno in posti differenti e, quindi, è più difficile che siano allineati; è anche meno probabile che un errore non venga rilevato.
4. Ridurre al minimo il numero di condizioni latenti in un sistema. In effetti, questo significa che vengono ridotte le dimensioni dei “buchi” del sistema. Tuttavia, questo potrebbe aumentare notevolmente i costi di ingegneria del sistema. Se si riduce il numero di bug nel sistema, crescono i costi dei test e quelli di verifica e convalida del sistema.

Quando si progetta un sistema, occorre considerare tutte queste opzioni e scegliere le soluzioni economicamente più convenienti per migliorare le difese del sistema. Se si sta realizzando un software personalizzato, allora la soluzione migliore potrebbe essere quella di utilizzare le verifiche del software per aumentare il numero e la diversità degli strati difensivi. Se, invece, si sta utilizzando un software off-the-shelf, allora potrebbe essere conveniente aggiungere nuove difese sociotecniche. Si potrebbero cambiare le procedure di addestramento per ridurre le probabilità che si presentino i problemi e per semplificare la gestione degli incidenti quando si verificano.

14.2.2 Processi operativi e gestionali

Tutti i sistemi software sono associati ad alcuni processi operativi che riflettono le ipotesi dei progettisti su come questi sistemi saranno utilizzati. Alcuni sistemi software, in particolare quelli che controllano o sono interfacciati con apparecchiature speciali, hanno operatori addestrati che sono una parte intrinseca del sistema di controllo. Le decisioni su quali funzioni devono far parte del sistema tecnico e quali funzioni devono essere affidate alla responsabilità degli operatori vengono prese durante la fase di progettazione. Per esempio, in un sistema di imaging di un ospedale, l’operatore potrebbe avere la responsabilità di controllare la qualità delle immagini subito dopo che sono state elaborate. Questo controllo consente alla procedura di imaging di essere ripetuta se c’è qualche problema.

I processi operativi sono quelli che interessano il normale utilizzo del sistema. Per esempio, gli operatori di un sistema di controllo del traffico aereo seguono specifici processi quando un aereo entra o esce dallo spazio aereo di loro competenza, quando devono correggere la velocità o l’altitudine del velivolo, quando c’è un’emergenza e così via. Per i nuovi sistemi, questi processi operativi devono essere definiti e documentati durante il processo di sviluppo del sistema. Potrebbe essere necessario addestrare gli operatori e adattare altri processi in modo da rendere più efficiente l’uso di un nuovo sistema.

Funzionamento efficiente dei processi	Gestione dei problemi
Ottimizzazione e controllo dei processi	Flessibilità e adattabilità dei processi
Nascondere e proteggere le informazioni	Condivisione e visibilità delle informazioni
Automazione per ridurre il carico di lavoro con un minor numero di operatori e manager	Processi manuali e operatori/manager di riserva per gestire i problemi
Specializzazione dei ruoli	Condivisione dei ruoli

Figura 14.7 Efficienza e resilienza.

Molti sistemi software, tuttavia, non hanno operatori addestrati, ma hanno utenti che li usano come parte del loro lavoro o a supporto dei loro interessi personali. Per i sistemi personali, i progettisti potrebbero descrivere l'uso previsto dei sistemi, ma non hanno alcun controllo sul modo in cui gli utenti si comporteranno realmente. Per i sistemi informatici aziendali, invece, gli utenti potrebbero essere addestrati in modo da insegnare loro come utilizzare i sistemi. Sebbene il comportamento degli utenti non possa essere controllato, tuttavia è ragionevole aspettarsi che essi, di solito, seguano il processo definito.

I sistemi informatici aziendali, di solito, hanno anche amministratori o manager che sono responsabili della loro gestione. Sebbene essi non facciano parte del processo aziendale supportato dal sistema, il loro compito è monitorare il sistema software per rilevare errori e problemi. Se si presenta un problema, i manager devono svolgere le azioni appropriate per risolvere il problema e riportare il sistema al suo normale stato operativo.

Nel precedente paragrafo ho descritto l'importanza del metodo di difesa in profondità e l'uso di vari meccanismi per controllare gli eventi avversi che potrebbero causare il fallimento del sistema. I processi operativi e gestionali sono un meccanismo importante della difesa e, nel progettare un processo, occorre trovare un giusto compromesso tra funzionamento efficiente e gestione dei problemi; queste esigenze sono spesso in conflitto, come illustra la Figura 14.7, in quanto per aumentare l'efficienza dei processi, di solito occorre ridurre la ridondanza e la diversità.

Per 25 anni le società hanno concentrato i loro sforzi nel cosiddetto miglioramento dei processi. Per migliorare l'efficienza dei processi operativi e gestionali, le società hanno studiato come i loro processi venivano attuati, ricercando pratiche particolarmente efficienti o inefficienti. La pratica efficiente viene codificata e documentata, e il software può essere sviluppato per supportare questo processo “ottimale”. La pratica inefficiente viene sostituita da una più efficiente. A volte, i meccanismi di controllo dei processi vengono introdotti per fare in modo che gli operatori e i manager seguano questa “pratica ideale”.

Il problema del miglioramento dei processi è che, spesso, per le persone diventa più difficile affrontare i problemi. Ciò che sembra una pratica “inefficiente” spesso deriva dal fatto che le persone conservano informazioni ridondanti o condividono le informazioni perché sanno che, così facendo, è più facile gestire i problemi quando le cose vanno male. Per esempio, i controllori del traffico aereo potrebbero stampare i dettagli di volo, nonostante abbiano il database dei voli, perché così potranno avere le informazioni sui voli nel caso in cui il database non sia disponibile.

Le persone hanno una capacità unica nel rispondere con efficienza a situazioni impreviste, anche se non hanno avuto alcuna esperienza diretta con tali situazioni. Quindi, quando le cose vanno male, gli operatori e i manager spesso riescono a superare una situazione imprevista, sebbene a volte siano costretti a infrangere le regole o “aggirare” le procedure prestabilite. Per questo motivo, i processi operativi dovrebbero essere progettati in modo che siano flessibili e adattabili. I processi operativi non dovrebbero essere troppo vincolanti; non dovrebbero richiedere operazioni da svolgere in un determinato ordine; e il sistema software non dovrebbe fare alcun affidamento su uno specifico processo da seguire.

Per esempio, un sistema per servizi di emergenza viene utilizzato per gestire le telefonate di emergenza e avviare le azioni appropriate a ciascuna telefonata. Il processo “normale” di gestione di una telefonata consiste nel registrare le informazioni dettagliate di chi ha telefonato e poi inviare un messaggio al servizio di emergenza appropriato, fornendo i dettagli e l’indirizzo dell’incidente. Questa procedura genera dei record che tengono traccia delle azioni svolte. Una successiva indagine potrà verificare se una chiamata di emergenza è stata gestita correttamente.

Supponiamo adesso che questo sistema subisca un attacco denial-of-service, che rende indisponibile il sistema di messaging. Anziché rispondere semplicemente alle chiamate, gli operatori possono utilizzare i loro telefoni cellulari e le conoscenze dei loro colleghi centralinisti per chiamare direttamente le unità dei servizi di emergenza, in modo che queste possano intervenire nel caso di incidenti gravi.

La gestione e la disponibilità delle informazioni sono caratteristiche importanti per la resilienza delle operazioni. Per rendere più efficiente un processo, potrebbe essere utile presentare agli operatori le informazioni di cui potrebbero avere bisogno. Da un punto di vista della protezione, le informazioni non dovrebbero essere accessibili, a meno che gli operatori o i manager non abbiano bisogno di tali informazioni. Tuttavia, un approccio più liberale all’accesso alle informazioni potrebbe migliorare la resilienza dei sistemi.

Se agli operatori vengono presentate soltanto quelle informazioni che il progettista del sistema ritiene che essi “debbano conoscere”, gli operatori potrebbero non essere in grado di rilevare i problemi che non influiscono direttamente sui loro compiti. Se le cose vanno male, gli operatori del sistema non hanno una visione ampia su ciò che sta accadendo nel sistema, quindi è più difficile per loro

attuare le strategie per gestire i problemi. Se non riescono ad accedere ad alcune informazioni per motivi di protezione, allora non saranno in grado di interrompere un attacco e riparare i danni causati da tale attacco.

Automatizzare il processo di gestione dei sistemi significa che un singolo manager può gestire un gran numero di sistemi. I sistemi automatizzati sono in grado di rilevare problemi comuni e svolgere le azioni appropriate per risolvere questi problemi. Le operazioni e la gestione del sistema richiedono meno persone e, quindi, i costi si riducono. Tuttavia, l'automazione dei processi ha due svantaggi.

1. I sistemi di gestione automatizzati possono fallire e svolgere azioni sbagliate. Al persistere di un problema, il sistema potrebbe svolgere azioni impreviste che peggiorano la situazione e che non possono essere capite dai manager del sistema.
2. La soluzione dei problemi è tipicamente un processo collaborativo. Se sono disponibili meno manager, è probabile che occorra più tempo per scegliere la strategia da adottare per affrontare un problema o un attacco cibernetico.

L'automazione dei processi, quindi, può avere effetti positivi e negativi sulla resilienza del sistema. Se il sistema automatizzato opera correttamente, può rilevare i problemi, invocare la resistenza agli attacchi cibernetici se necessario, e avviare le procedure di ripristino automatico. Se, invece, il sistema automatizzato non è in grado di risolvere un problema, poche persone saranno in grado di farlo e il sistema potrebbe essere danneggiato dall'automazione dei processi che svolge un'azione sbagliata.

In un ambiente in cui ci sono vari tipi di sistemi e apparecchiature, è praticamente impossibile che tutti gli operatori e i manager siano in grado di gestire tutti i vari sistemi. Le persone quindi potrebbero specializzarsi in modo da diventare esperte di un piccolo numero di sistemi. Questo porta a operazioni più efficienti, ma ha conseguenze negative sulla resilienza del sistema.

Il problema della specializzazione dei ruoli è che, in un particolare momento, potrebbe essere indisponibile proprio quella persona che è esperta di interazioni fra sistemi. Di conseguenza, è difficile affrontare determinati problemi se manca lo specialista. Se le persone si occupano di più sistemi, conoscono le interdipendenze e le relazioni tra i sistemi e quindi possono risolvere i problemi comuni a più sistemi. Se non è disponibile uno specialista, diventa molto più difficile contenere un problema e riparare i danni che ha provocato.

Si potrebbe utilizzare la tecnica della valutazione dei rischi, descritta nel Capitolo 13, per agevolare le scelte relative al bilanciamento tra efficienza e resilienza dei processi. Si dovrebbero considerare tutte le situazioni rischiose che potrebbero richiedere l'intervento degli operatori o dei manager e valutare le probabilità che questi rischi si realizzino e le conseguenti perdite. Per i rischi che possono causare danni gravi e perdite significative e per i rischi che hanno alte probabilità di verificarsi, si dovrebbe preferire la resilienza all'efficienza dei processi.

14.3 Progettazione di sistemi resilienti

I sistemi resilienti sono in grado di resistere agli eventi ostili e ripristinare il loro normale funzionamento dopo un errore del software o un attacco cibernetico. Possono offrire servizi critici con poche interruzioni e ripristinare rapidamente il loro stato operativo normale dopo che si è verificato un incidente. Quando si progetta un sistema resiliente, bisogna supporre che i fallimenti del sistema o le violazioni da parte di hacker potranno verificarsi e, quindi, bisogna prevedere apposite misure difensive ridondanti e diverse per contrastare questi eventi avversi.

La progettazione dei sistemi resilienti prevede due attività strettamente correlate.

1. *Identificare le risorse e i servizi critici* – Le risorse e i servizi critici sono quegli elementi che consentono a un sistema di svolgere completamente il suo compito primario. Per esempio, il compito primario di un sistema che gestisce l'invio delle ambulanze in seguito alle telefonate di emergenza è quello di aiutare il più presto possibile le persone che ne hanno bisogno. I servizi critici sono quelli che si occupano di ricevere le telefonate e inviare le ambulanze sul luogo in cui è richiesto il soccorso medico. Altri servizi quali la registrazione delle telefonate o il monitoraggio delle ambulanze sono meno importanti.
2. *Progettare i componenti del sistema che supportano l'identificazione, la resistenza, la riparazione e il ripristino* – Per esempio, nel sistema che gestisce l'invio delle ambulanze, si potrebbe includere un timer di guardia per rilevare se il sistema non sta rispondendo agli eventi. Agli operatori potrebbe essere richiesto di autenticarsi con un token hardware per proteggere il sistema da eventuali accessi non autorizzati. Se il sistema fallisce, le telefonate potrebbero essere smistate a un altro centro di emergenza, in modo da assicurare la continuità dei servizi essenziali. Potrebbero essere conservate copie del database e del software del sistema su unità hardware separate in modo da facilitare il ripristino dei servizi dopo un'interruzione temporanea.

I concetti fondamentali di identificazione, resistenza e ripristino furono alla base dei primi studi di ingegneria della resilienza condotti da Ellison e altri (Ellison et al. 1999, 2002); progettarono un metodo per analizzare la capacità di sopravvivenza dei sistemi. Questo metodo è utilizzato per valutare le vulnerabilità dei sistemi e per supportare la progettazione di architetture e funzionalità che migliorano la sopravvivenza dei sistemi.

L'analisi di sopravvivenza dei sistemi è formata da quattro fasi (Figura 14.8), nelle quali vengono esaminati i requisiti e l'architettura attuali o proposti del sistema, i servizi critici, gli scenari di possibili attacchi, i punti deboli di un sistema e le modifiche proposte per migliorare la sopravvivenza di un sistema. Le attività chiave in ciascuna di queste fasi sono quattro.

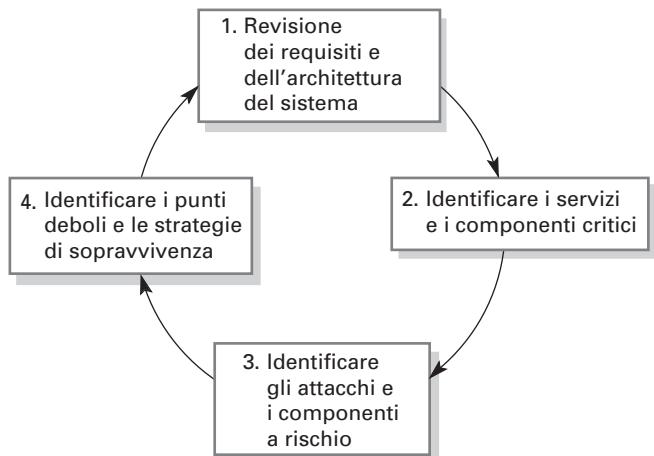


Figura 14.8 Fasi dell'analisi di sopravvivenza.

1. *Capire il sistema*: per un sistema esistente o in fase di progetto, si rivedono i suoi obiettivi (detti anche “obiettivi della missione”), i requisiti e l’architettura.
2. *Identificare i servizi critici*: si identificano i servizi che devono essere sempre garantiti e i componenti necessari per fornire tali servizi.
3. *Simulare gli attacchi*: si definiscono gli scenari o i casi d’uso per possibili attacchi, e si identificano i componenti del sistema che potrebbero essere coinvolti in questi attacchi.
4. *Analisi di sopravvivenza*: si identificano i componenti critici che potrebbero essere danneggiati da un attacco; si definiscono le strategie di sopravvivenza in base alle capacità di resistenza, identificazione e ripristino dei componenti.

Il problema fondamentale dell’analisi di sopravvivenza è che il suo punto di partenza è la documentazione dei requisiti e dell’architettura di un sistema. Questa è un’ipotesi accettabile per i sistemi di difesa militari (il lavoro fu sponsorizzato dal Dipartimento della Difesa degli Stati Uniti), ma pone due problemi per i sistemi aziendali.

1. Non c’è una esplicita relazione con i requisiti aziendali della resilienza. Io credo che questi requisiti siano un punto di partenza più appropriato rispetto ai requisiti tecnici del sistema.
2. Si suppone che ci sia una definizione dettagliata dei requisiti di un sistema. In pratica, la resilienza potrebbe essere adattata al sistema, quando non c’è un documento completo o aggiornato dei requisiti. Per i nuovi sistemi, la resilienza stessa potrebbe essere un requisito, o i sistemi potrebbero essere sviluppati utilizzando un approccio agile. L’architettura del sistema potrebbe essere progettata per tenere conto della resilienza.

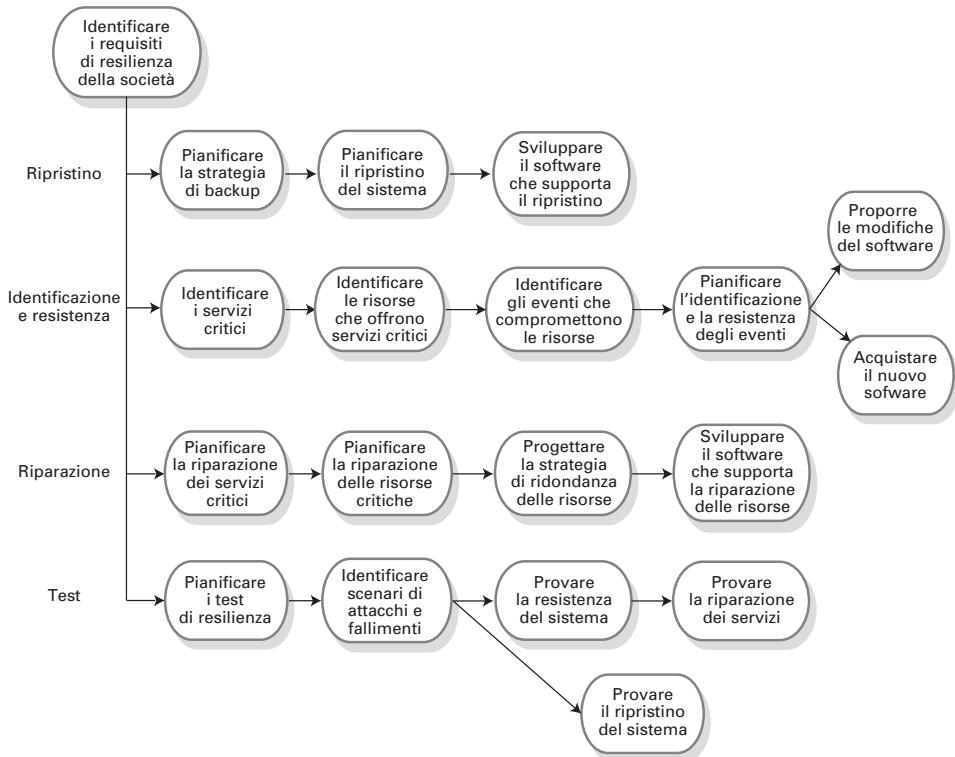


Figura 14.9 Ingegneria della resilienza.

Un metodo più generale di ingegneria della resilienza, illustrato nella Figura 14.9, prende atto della mancanza di requisiti dettagliati e progetta esplicitamente le funzioni di riparazione e ripristino del sistema. Per la maggior parte dei componenti di un sistema, non dovete accedere al loro codice sorgente e non sarete in grado di modificarlo. La vostra strategia per realizzare la resilienza deve essere progettata tenendo conto di questa limitazione.

Ci sono cinque attività intercorrelate in questo approccio all'ingegneria della resilienza.

1. Identificare i requisiti aziendali della resilienza. Questi requisiti definiscono come l'azienda deve garantire i servizi che fornisce ai clienti e, da qui, stabilire i requisiti di resilienza per i singoli sistemi da sviluppare. Realizzare la resilienza è un processo costoso; quindi bisogna evitare i supporti superflui alla resilienza.
2. Definire un piano per ripristinare lo stato di funzionamento normale di un sistema o di un insieme di sistemi dopo un evento avverso. Questo piano deve essere integrato con la normale strategia di backup e archiviazione dell'azienda che consente il ripristino delle informazioni dopo un errore tecnico o umano. Il piano dovrebbe far parte anche di una strategia più

ampia di recovery dopo un fallimento disastroso. Bisogna considerare la possibilità di eventi fisici, come un incendio o un'inondazione, e studiare come conservare le informazioni critiche in luoghi separati. Si potrebbero utilizzare i backup su cloud per questo piano.

3. Identificare i fallimenti e gli attacchi cibernetici che potrebbero compromettere un sistema. Progettare le strategie di identificazione e resilienza per contrastare questi eventi avversi.
4. Definire un piano per ripristinare rapidamente i servizi critici dopo che sono stati danneggiati o interrotti da un fallimento o attacco cibernetico. Per far questo, di solito occorre creare delle copie ridondanti delle risorse critiche che forniscono questi servizi; in caso di emergenza, i servizi critici possono essere forniti da queste copie.
5. Verificare tutti gli aspetti del piano di resilienza. Questa verifica consiste nell'identificare i possibili scenari di fallimenti e attacchi e nell'applicare questi scenari al sistema.

Garantire la disponibilità dei servizi critici è l'essenza della resilienza. Di conseguenza, occorre conoscere:

- i servizi più critici per un'azienda;
- la qualità minima dei servizi da garantire;
- in che modo questi servizi potrebbero essere compromessi;
- in che modo questi servizi possono essere protetti;
- come ripristinare rapidamente i servizi se dovessero diventare indisponibili.

Come parte dell'analisi dei servizi critici, dovreste identificare le risorse del sistema che sono essenziali per fornire questi servizi. Queste risorse possono essere l'hardware (server, rete ecc.), il software, i dati e le persone. Per realizzare un sistema resiliente, dovreste trovare il modo di utilizzare la ridondanza e la diversità per garantire che queste risorse restino disponibili nel caso si verifichi un fallimento del sistema.

Per tutte queste attività, la chiave per fornire una risposta rapida e un piano di ripristino dopo un evento avverso consiste nel disporre di un software addizionale che supporta la resistenza, la riparazione e il ripristino delle risorse. Se si dispone del software di supporto appropriato, i processi di riparazione e ripristino possono essere parzialmente automatizzati ed eseguiti rapidamente dopo un fallimento del sistema.

Il test della resilienza richiede di simulare tutti i possibili fallimenti e attacchi cibernetici, per verificare che i piani di resilienza che sono stati predisposti operino secondo le previsioni. Il test è essenziale perché sappiamo per esperienza che le ipotesi fatte sulla pianificazione della resilienza spesso non sono valide e che le azioni pianificate non sempre sono adeguate. Il test della resilienza può identificare questi problemi e, quindi, il piano della resilienza può essere ridefinito.

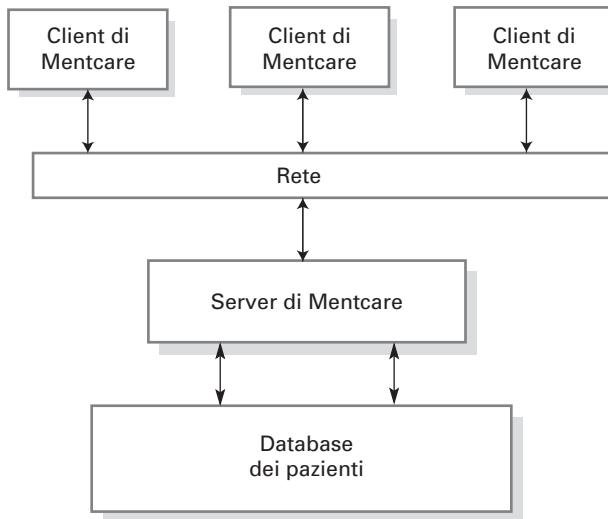


Figura 14.10 Architettura client-server del sistema Mentcare.

I test potrebbero essere molto difficili e costosi perché, ovviamente, non possono essere eseguiti sul sistema reale. Il sistema e il suo ambiente potrebbero essere duplicati per eseguire i test; il personale potrebbe essere temporaneamente sospeso dal suo normale lavoro per eseguire i test sul sistema. Per ridurre i costi, si potrebbero utilizzare i “test da scrivania”. Il team di test suppone che si sia verificato un problema e verifica le sue reazioni a questo problema; non simulano il problema su un sistema reale. Sebbene questo approccio possa fornire utili informazioni sulla resilienza del sistema, tuttavia è meno efficiente dei test per scoprire i difetti in un piano di resilienza.

Come esempio di questo approccio, vediamo un’applicazione dell’ingegneria della resilienza al sistema Mentcare. Per riassumere, questo sistema viene utilizzato per supportare i medici che curano pazienti che hanno problemi di salute mentale e che risiedono in varie città. Il sistema fornisce informazioni sui pazienti e registra i consulti con dottori e infermieri specializzati. Include un certo numero di controlli che segnalano quei pazienti che potrebbero diventare pericolosi per gli altri e per se stessi. La Figura 14.10 mostra l’architettura di questo sistema.

Il sistema può essere utilizzato da dottori e infermieri prima e dopo un consulto; le informazioni sui pazienti vengono aggiornate dopo ogni consulto. Per garantire l’efficienza delle cliniche, i requisiti di resilienza stabiliscono che i servizi critici del sistema siano disponibili durante il normale orario di lavoro, che i dati dei pazienti non subiscano danni o perdite permanenti a causa di un fallimento del sistema o di un attacco cibernetico, e che le informazioni sui pazienti non siano accessibili a persone non autorizzate.

Due servizi critici devono essere sempre garantiti nel sistema:

1. *un servizio informativo* che fornisce informazioni sul piano corrente di diagnosi e cura di ciascun paziente;
2. *un servizio di allerta* che segnala quei pazienti che potrebbero rappresentare un pericolo per gli altri e per se stessi.

Si noti che il servizio critico non richiede la disponibilità dei record completi dei pazienti. I dottori e gli infermieri hanno bisogno solo occasionalmente di rivedere i precedenti trattamenti dei pazienti, quindi l'assistenza medica non viene seriamente compromessa se i record completi non sono disponibili. Pertanto, è possibile garantire un'assistenza efficiente utilizzando record di sintesi che includono soltanto le informazioni sui pazienti e sulle cure recenti.

Le risorse richieste per fornire questi servizi durante il normale funzionamento del sistema sono:

1. il database che contiene tutte le informazioni sui pazienti;
2. un server che fornisce l'accesso al database per i computer client locali;
3. una rete per le comunicazioni client/server;
4. computer desktop o portatili utilizzati dal personale medico per accedere alle informazioni sui pazienti;
5. un insieme di regole che identificano i pazienti potenzialmente pericolosi e che contrassegnano i corrispondenti record del database. Il software su ogni client mette in evidenza i pazienti pericolosi.

Per pianificare le strategie di identificazione, resistenza e riparazione, occorre sviluppare una serie di scenari che anticipano gli eventi avversi che potrebbero compromettere i servizi critici offerti dal sistema. Ecco alcuni esempi di eventi avversi:

1. indisponibilità del server del database a causa di un fallimento del sistema o della rete o in seguito a un attacco denial-of-service;
2. danneggiamento deliberato o accidentale del database dei pazienti o delle regole che definiscono che cosa significa “paziente pericoloso”;
3. immissione di malware nei computer client;
4. accesso ai computer client da parte di persone non autorizzate che guadagnano l'accesso ai record dei pazienti.

La Figura 14.11 mostra alcune strategie di identificazione e resistenza per questi eventi avversi. Le strategie non sono soltanto approcci tecnici, ma suggeriscono anche alcuni seminari di sensibilizzazione sui problemi della protezione. Sappiamo che molte violazioni delle difese protettive si verificano perché gli utenti rivelano inavvertitamente informazioni privilegiate a un hacker; questi seminari riducono la probabilità che ciò accada. Non ho spazio qui per trattare tutte le possibili opzioni riportate nella Figura 14.11. Piuttosto, ho messo in evidenza come l'architettura del sistema possa essere modificata per essere più resiliente.

Evento	Identificazione	Resistenza
Indisponibilità del server	<ol style="list-style-type: none"> 1. Timer di guardia sul client che sospende le attività se non ci sono risposte all'accesso del client 2. Messaggi di testo dai manager del sistema al personale medico 	<ol style="list-style-type: none"> 1. Progettare l'architettura del sistema per mantenere le copie locali delle informazioni critiche 2. Creare una funzione per ricercare nei client le informazioni sui pazienti tramite una connessione peer-to-peer 3. Fornire al personale medico gli smartphone che possono essere utilizzati per accedere alla rete nel caso di fallimento del server 4. Utilizzare un server di backup
Danneggiamento del database dei pazienti	<ol style="list-style-type: none"> 1. Confrontare i checksum a livello dei record 2. Controllo periodico automatico dell'integrità del database 3. Sistema di reporting per informazioni errate 	<ol style="list-style-type: none"> 1. Registro di transazioni ripetibili per aggiornare il backup del database con le transazioni più recenti 2. Mantenere copie locali del software e delle informazioni sui pazienti per poter ripristinare il database dai backup e dalle copie locali
Presenza di malware nei computer client	<ol style="list-style-type: none"> 1. Sistema di reporting in modo che gli utenti dei computer possano segnalare comportamenti insoliti 2. Verifica automatica della presenza di malware all'avviamento 	<ol style="list-style-type: none"> 1. Seminari per sensibilizzare tutti gli utenti del sistema sui problemi di protezione 2. Disabilitare le porte USB nei computer client 3. Configurazione automatica del sistema per i nuovi client 4. Accesso al sistema dalle unità mobili 5. Installazione del software di protezione
Accesso non autorizzato alle informazioni dei pazienti	<ol style="list-style-type: none"> 1. Messaggi di allerta da parte degli utenti per segnalare possibili attacchi di hacker 2. Analisi del registro per verificare attività insolite 	<ol style="list-style-type: none"> 1. Processo di autenticazione a più livelli 2. Disabilitare le porte USB nei computer client 3. Registrazione degli accessi e analisi delle registrazioni in tempo reale 4. Seminari per sensibilizzare tutti gli utenti del sistema sui problemi di protezione

Figura 14.11 Strategie di identificazione e resistenza per eventi avversi.

Nella Figura 14.11 ho suggerito che mantenere le informazioni sui pazienti sui computer client è una possibile strategia di ridondanza che potrebbe aiutare ad assicurare i servizi critici. Questo comporta la modifica dell'architettura del software, come illustra la Figura 14.12. Le caratteristiche chiave di questa architettura sono riportate qui di seguito.

1. *Record di sintesi dei pazienti che sono mantenuti sui client locali* – I computer locali possono comunicare direttamente tra loro e scambiarsi le informazioni utilizzando sia la rete del sistema sia, se necessario, una rete appositamente creata tramite telefoni cellulari. Quindi, se il database non è disponibile, i dottori e gli infermieri possono ancora accedere alle informazioni essenziali sui pazienti (resistenza e riparazione).

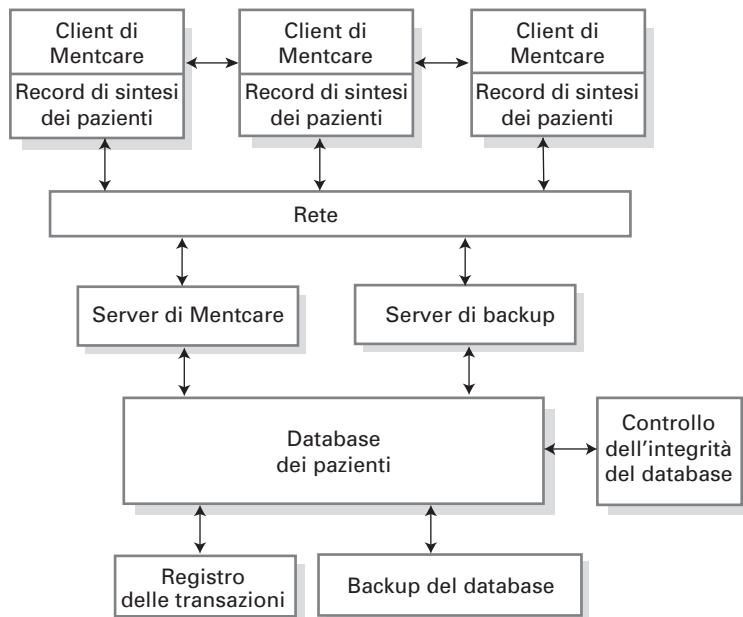


Figura 14.12 Architettura per un sistema Mentcare resiliente.

2. *Server di backup che entra in azione nel caso di fallimento del server principale* – Questo server ha il compito di effettuare periodici backup del server principale. Nel caso in cui quest'ultimo fallisse, il server di backup agisce come server principale per l'intero sistema. Questo consente di garantire la continuità dei servizi e la riparazione del server principale (resistenza e riparazione).
3. *Controllo dell'integrità del database* – Il controllo dell'integrità del database viene eseguito in background per verificare che non ci siano segni di danneggiamento nei record del database. Se viene scoperto un record danneggiato, viene automaticamente avviata la procedura di riparazione di alcuni o tutti i dati utilizzando la copia di backup del database. Il registro delle transazioni consente a questa copia di backup di essere aggiornata con i dettagli delle modifiche più recenti (identificazione e riparazione).

Per mantenere i servizi chiave di accesso alle informazioni dei pazienti e di allerta del personale medico, possiamo sfruttare la ridondanza intrinseca di un sistema client-server. Scaricando le informazioni sul client all'inizio di una sessione, il consulto medico può continuare senza bisogno di accedere al server. Dovranno essere scaricate soltanto le informazioni di quei pazienti che hanno fissato un consulto medico nel giorno della sessione. Se ci fosse bisogno di accedere alle informazioni di altri pazienti e il server fosse indisponibile, allora potrebbero essere contattati altri computer client tramite la connessione peer-to-peer per vedere se tali informazioni sono disponibili su questi client.

Il servizio che invia al personale medico un messaggio di allerta per quei pazienti che potrebbero essere pericolosi può essere facilmente implementato utilizzando questo approccio. I record dei pazienti che potrebbero ferire se stessi o altre persone vengono identificati prima del processo di download. Quando il personale medico accede a questi record, il software può mettere in evidenza i record per indicare i pazienti che richiedono un trattamento speciale.

Le caratteristiche di questa architettura che supportano la resistenza agli eventi avversi sono utili anche per supportare il processo di recovery dopo che si sono verificati questi eventi. Mantenendo più copie delle informazioni e avendo a disposizione l'hardware di backup, i servizi critici di un sistema possono essere rapidamente riportati al loro normale funzionamento. Poiché il sistema deve essere disponibile soltanto durante l'orario di lavoro normale (dalle 8:00 alle 18:00), esso può essere ripristinato nelle ore notturne, in modo che sia disponibile il giorno successivo a quello in cui si è verificato un fallimento.

Oltre ai servizi critici, devono essere garantite anche la riservatezza e l'integrità dei dati sui pazienti. L'architettura illustrata nella Figura 14.12 include un sistema di backup e un controllo esplicito dell'integrità del database per ridurre le probabilità che le informazioni sui pazienti siano danneggiate accidentalmente o da un attacco cibernetico. Le informazioni che si trovano nei computer client sono disponibili e possono essere utilizzate per supportare il processo di riparazione dei dati danneggiati.

Sebbene le copie multiple dei dati possano essere una difesa contro il danneggiamento di dati, tuttavia esse costituiscono un rischio per la riservatezza, in quanto tutte queste copie devono essere protette. In questo caso, il rischio può essere controllato nei seguenti modi:

1. scaricando soltanto i record di sintesi di quei pazienti che hanno fissato un appuntamento in clinica. Questo limita il numero di record che potrebbero essere danneggiati;
2. crittografando i dati nei dischi dei computer client. Gli hacker che non hanno la chiave di crittografia non saranno in grado di leggere tali dati se dovessero riuscire ad accedere ai computer;
3. cancellando le informazioni scaricate alla fine di ogni sessione medica. Questo riduce ulteriormente le probabilità che un hacker possa accedere alle informazioni riservate;
4. garantendo che tutte le transazioni in rete siano crittografate. Se un hacker intercetta queste transazioni, non sarà in grado di leggerne il contenuto.

A causa del peggioramento delle prestazioni, non è conveniente crittografare l'intero database sul server. Per proteggere queste informazioni è consigliabile adottare tecniche di autenticazione più complesse.

Punti chiave

- La resilienza di un sistema è una stima di come il sistema sia in grado di garantire la continuità dei suoi servizi critici in presenza di eventi avversi, quali il guasto di un dispositivo o un attacco cibernetico.
- La resilienza dovrebbe basarsi sulle quattro caratteristiche dell'ingegneria della resilienza: identificazione, resistenza, riparazione e ripristino.
- La pianificazione della resilienza dovrebbe basarsi sull'ipotesi che i sistemi in rete saranno soggetti ad attacchi cibernetici da parte di utenti interni o esterni e che qualcuno di questi attacchi riuscirà.
- I sistemi dovrebbero essere progettati con un certo numero di strati difensivi di diverso tipo. Se questi strati sono efficaci, i guasti tecnici o gli errori umani potranno essere rilevati e gli attacchi cibernetici potranno essere respinti.
- Per consentire agli operatori e ai manager di risolvere i problemi di un sistema, i processi del sistema dovrebbero essere flessibili e adattabili. Automatizzando i processi, queste persone potrebbero avere maggiori difficoltà a risolvere i problemi.
- I requisiti aziendali della resilienza dovrebbero essere il punto di partenza nella progettazione di un sistema resiliente. Per realizzare un sistema resiliente, occorre concentrarsi sui metodi di identificazione dei problemi, di riparazione delle risorse e dei servizi critici, e di ripristino del normale funzionamento del sistema.
- Una parte importante della progettazione della resilienza è l'identificazione dei servizi critici, ovvero quei servizi che sono essenziali al sistema per svolgere il suo compito primario. I sistemi dovrebbero essere progettati in modo che questi servizi siano protetti e, nel caso di fallimento, possano essere rapidamente ripristinati.

Esercizi

- * 14.1 Spiegate perché le strategie complementari di resistenza, identificazione, riparazione e ripristino possono essere utilizzate per migliorare la resilienza dei sistemi.
- 14.2 Spiegate come la ridondanza e la diversità possano essere utilizzate per migliorare la capacità di un sistema di resistere agli attacchi cibernetici che potrebbero danneggiare le risorse del sistema.
- 14.3 Che cos'è un'organizzazione resiliente? Spiegate come il modello di Hollnagel, illustrato nella Figura 14.4, possa essere una base efficiente per migliorare la resilienza di un'organizzazione.
- * 14.4 La direzione di un ente ospedaliero ha proposto una direttiva secondo la quale qualsiasi membro del personale medico (dottori e infermieri) che svolge o autorizza azioni che procurano danni fisici ai pazienti potrà essere incriminato penalmente. Spiegate perché questa è una cattiva idea, che difficilmente potrà migliorare la sicurezza dei pazienti, e perché molto probabilmente avrà effetti negativi sulla resilienza dell'ente ospedaliero.
- * 14.5 Suggerite tre strati difensivi che potrebbero essere inclusi in un sistema informatico per proteggere i dati contro le modifiche che potrebbero essere apportate da un utente non autorizzato.

- 14.6 Spiegate perché i processi poco flessibili possono inibire la capacità di un sistema sociotecnico di resistere agli eventi avversi, come gli errori del software o gli attacchi cibernetici, e di ripristinare il suo funzionamento normale dopo che si sono verificati simili eventi. Se avete avuto esperienza con processi poco flessibili, illustrate la vostra risposta con esempi collegati alla vostra specifica esperienza.
- * 14.7 Suggerite come l'approccio all'ingegneria della resilienza proposto nella Figura 14.9 possa essere utilizzato in combinazione con un processo di sviluppo agile per il software di un sistema. Quali problemi potrebbero nascere utilizzando lo sviluppo agile per i sistemi nei quali la resilienza è importante?
- * 14.8 Nel sistema di trading descritto nel Paragrafo 13.4.2, (1) un utente non autorizzato immette alcuni ordini per far muovere i prezzi e (2) un hacker danneggia il database delle transazioni che sono state eseguite. Per ciascuno di questi attacchi suggerite le strategie di resistenza, identificazione e riparazione che potrebbero essere adottate.
- 14.9 Nella Figura 14.11 ho indicato un certo numero di eventi avversi che potrebbero influire sul sistema Mentcare. Suggerite un piano di test per valutare la capacità di questo sistema di identificare e resistere a questi eventi e di riparare le risorse danneggiate.
- 14.10 Un manager senior di una società ha il sospetto che alcuni dipendenti insoddisfatti vogliono attaccare il sistema informatico della società per danneggiare il database. Come parte di un programma di miglioramento della resilienza, il manager propone di introdurre un software per registrare e analizzare le attività degli utenti; in questo modo, sarà in grado di catturare ed esaminare le operazioni svolte da tutti i dipendenti. Ovviamente, i dipendenti sospetti non saranno informati di queste nuove misure di protezione. Come giudicate da un punto di vista etico queste misure di protezione che non prevedono di informare gli utenti?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Survivable Network System Analysis: A Case Study.” Un articolo eccellente che introduce il concetto di sopravvivenza dei sistemi e usa il caso di studio di un sistema di cura per malati mentali per illustrare l'applicazione di un metodo di sopravvivenza. (R. J. Ellison, R. C. Linger, T. Longstaff e N. R. Mead, *IEEE Software*, 16 (4), July/August 1999) <http://dx.doi.org/10.1109/52.776952>

Resilience Engineering in Practice: A Guidebook. Una raccolta di articoli e casi di studio sull'ingegneria della resilienza dalla prospettiva più ampia dei sistemi sociotecnici. (E. Hollnagel, J. Paries, D. W. Woods e J. Wreathall, Ashgate Publishing Co., 2011).

“Cyber Risk and Resilience Management.” Un sito web che offre una ricca gamma di risorse sulla protezione cibernetica e sulla resilienza, incluso un modello per la gestione della resilienza. (Software Engineering Institute, 2013) <https://www.cert.org/resilience/>

Ingegneria avanzata del software

Questa parte del libro tratta alcuni argomenti di ingegneria avanzata del software. In questi capitoli si presume che i lettori abbiano studiato i temi fondamentali trattati nei capitoli 1-9.

I Capitoli 15-18 descrivono il paradigma principale dello sviluppo dei sistemi informatici e aziendali basati sul Web – il riutilizzo del software. Il Capitolo 15 introduce questo argomento e spiega i vari tipi possibili di riutilizzo del software; poi spiega l'approccio più comune al riutilizzo, ovvero il riutilizzo dei sistemi applicativi. Questi sistemi vengono configurati e adattati alle specifiche esigenze di ciascuna azienda.

Il Capitolo 16 è dedicato al riutilizzo dei soli componenti software, anziché di tutto il sistema software. Il capitolo spiega che cosa s'intende per componente software e perché sono necessari i modelli di componenti standard per un riutilizzo efficiente dei componenti. Descrive anche il processo generale dell'ingegneria del software basato sui componenti e i problemi della composizione dei componenti.

La maggior parte dei grandi sistemi attuali sono sistemi distribuiti, e il Capitolo 17 tratta i temi e i problemi connessi alla realizzazione dei sistemi distribuiti. Il capitolo descrive l'approccio client-server come paradigma fondamentale dell'ingegneria dei sistemi distribuiti e illustra le tecniche di implementazione di questo stile architettonico. L'ultimo paragrafo spiega il software come servizio – la fruibilità di funzionalità software su Internet, che ha cambiato il mercato dei prodotti software.

Il Capitolo 18 introduce il tema delle architetture orientate ai servizi, che combinano i concetti di distribuzione e riutilizzo. I servizi sono componenti software riutilizzabili le cui funzionalità possono essere fruite su Internet. Sono descritti due approcci molto comuni nello sviluppo dei servizi: SOAP e RESTful. Il capitolo illustra i dettagli dei processi di creazione dei servizi (ingegneria dei servizi) e di composizione dei servizi per realizzare nuovi sistemi software.

CAPITOLO

15

Riutilizzo del software

Questo capitolo ha due obiettivi: presentare il riutilizzo del software e descrivere gli approcci allo sviluppo dei sistemi basati sul riutilizzo del software su larga scala. Dopo aver letto questo capitolo:

- conoscerete i vantaggi e i problemi del riutilizzo del software quando si sviluppano nuovi sistemi;
- conoscerete il concetto di framework applicativo come un insieme di oggetti riutilizzabili, e imparerete a utilizzare i framework nello sviluppo delle applicazioni;
- conoscerete le linee di prodotti software, che sono costituite da un'architettura centrale comune e da componenti riutilizzabili che vengono configurati per ogni versione del prodotto;
- apprenderete come i sistemi possono essere sviluppati configurando e componendo sistemi software off-the-shelf.

15.1 Panoramica sul riutilizzo

15.2 Framework applicativi

15.3 Linee di prodotti software

15.4 Riutilizzo dei sistemi applicativi

L'ingegneria del software basato sul riutilizzo è una strategia di ingegneria del software dove il processo di sviluppo si impenna sul riutilizzo del software esistente. Fino al 2000 circa, il riutilizzo sistematico del software era poco comune, mentre oggi viene largamente adottato nello sviluppo di nuovi sistemi aziendali. La transizione verso lo sviluppo basato sul riutilizzo è iniziata in seguito alla richiesta di ridurre i costi di produzione e manutenzione del software, e di migliorare la qualità del software. Sempre più compagnie considerano il loro software come un investimento importante, e promuovono il riutilizzo di sistemi esistenti per aumentare il ritorno dei loro investimenti sul software.

Software riutilizzabile di vario tipo è adesso largamente disponibile. Il movimento open-source ha fatto sì che ci sia una vasta base di codice riutilizzabile, sotto forma di librerie di programmi o applicazioni complete. Molti sistemi applicativi per domini specifici, come i sistemi ERP, sono disponibili per essere adattati a specifiche esigenze dei clienti. Alcune grandi società offrono ai loro clienti una ricca serie di componenti riutilizzabili. Gli standard, come quelli per i servizi web, hanno semplificato lo sviluppo dei servizi software e del loro riutilizzo attraverso una vasta gamma di applicazioni.

L'ingegneria del software basato sul riutilizzo è un approccio allo sviluppo che tenta di massimizzare il riutilizzo del software esistente. Le unità di software che vengono riutilizzate possono essere di dimensioni notevolmente differenti.

1. *Riutilizzo di sistemi.* Un intero sistema, che può essere formato da un certo numero di programmi applicativi, può essere riutilizzato come parte di un sistema di sistemi.
2. *Riutilizzo di applicazioni.* Un'applicazione può essere riutilizzata incorporandola senza modifiche all'interno di altri sistemi o configurandola per vari clienti. In alternativa, famiglie di applicazioni o linee di prodotti software, che hanno un'architettura comune, ma che sono state adattate per soddisfare le esigenze di specifici clienti, possono essere utilizzate per sviluppare un nuovo sistema.
3. *Riutilizzo di componenti.* Possono essere riutilizzati tutti i componenti di un'applicazione, dai sottosistemi ai singoli oggetti. Per esempio, un sistema di corrispondenza di pattern, sviluppato come parte di un sistema di elaborazione dei testi, può essere riutilizzato in un sistema di gestione di database. I componenti possono essere memorizzati nel cloud o in server privati e possono essere accessibili come servizi tramite un'interfaccia di programmazione delle applicazioni (API).
4. *Riutilizzo di oggetti e funzioni.* Si possono riutilizzare i componenti software che implementano una singola funzione, come una funzione matematica, o una classe di oggetti. Questo metodo di riutilizzo, basato su librerie standard, è stato molto comune negli ultimi 40 anni. Sono disponibili gratuitamente molte librerie di funzioni e classi, che possono essere facilmente riutilizzate collegandole con un nuovo codice applicativo. In aree come la

grafica e gli algoritmi matematici, dove è richiesta un'esperienza specifica per sviluppare oggetti e funzioni, questo approccio è particolarmente efficace ed economico.

Tutti i sistemi e i componenti software che includono funzionalità generiche sono potenzialmente riutilizzabili; tuttavia, questi sistemi o componenti a volte sono così specifici che è molto costoso modificarli per adattarli a una nuova situazione. Anziché riutilizzare il codice, tuttavia, si possono riutilizzare le idee che stanno alla base del software. Questo è detto *riutilizzo concettuale*.

Nel riutilizzo concettuale, anziché riutilizzare un componente software, si riutilizza un'idea, un modo di operare o un algoritmo. L'idea che viene riutilizzata è rappresentata con una notazione astratta, come il modello di un sistema, che non include i dettagli di implementazione; pertanto, può essere configurata e adattata a varie situazioni. Il riutilizzo concettuale è integrato in approcci quali gli schemi di progettazione (Capitolo 7), i prodotti di sistemi configurabili e i generatori di programmi. Quando le idee vengono riutilizzate, il processo di riutilizzo deve includere un'attività in cui i concetti astratti sono istanziati per creare componenti eseguibili.

Un vantaggio evidente del riutilizzo del software è la riduzione dei costi di sviluppo, in quanto deve essere specificata, progettata, implementata e convalidata una quantità inferiore di componenti software. La riduzione di questi costi non è l'unico beneficio del riutilizzo del software. Nella Figura 15.1 sono elencati altri vantaggi del riutilizzo del software.

D'altra parte, ci sono anche costi e problemi associati al riutilizzo del software (Figura 15.2). In particolare, è molto costoso verificare se un componente è adatto a essere riutilizzato in una particolare situazione; è anche costoso testare il componente per controllare la sua fidatezza. Questi costi aggiuntivi potrebbero rendere il risparmio dei costi minore di quello previsto. Occorre comunque valutare gli altri benefici del riutilizzo del software.

Come detto nel Capitolo 2, i processi di sviluppo del software devono essere adattati per poter attuare il riutilizzo. In particolare, ci deve essere una fase di affinamento dei requisiti, nella quale i requisiti del sistema vengono modificati per adattarli al software riutilizzabile che si ha a disposizione. Anche le fasi di progettazione e implementazione del sistema potrebbero includere esplicite attività per cercare e valutare i componenti più appropriati al riutilizzo.

15.1 Panoramica sul riutilizzo

Negli ultimi vent'anni sono state sviluppate molte tecniche per supportare il riutilizzo del software. Queste tecniche sfruttavano il fatto che i sistemi dello stesso dominio applicativo sono simili e, quindi, sono potenzialmente riutilizzabili. Il riutilizzo è possibile a vari livelli, dalle semplici funzioni alle applicazioni complete; gli standard dei componenti riutilizzabili ne facilitano il riutilizzo. La

Beneficio	Spiegazione
Sviluppo accelerato	Inserire un nuovo sistema nel mercato al più presto possibile è spesso più importante dei costi generali di sviluppo. Riutilizzare il software può velocizzare la produzione del sistema, in quanto i tempi di sviluppo e convalida possono essere ridotti.
Uso efficace di specialisti	Invece di ripetere lo stesso lavoro più volte, gli specialisti delle applicazioni possono sviluppare un software riutilizzabile che racchiude le loro conoscenze.
Maggiore fidatezza	Il software riutilizzato, che è stato provato e testato in sistemi funzionanti, dovrebbe essere più affidabile del nuovo software, in quanto i suoi errori di progettazione e implementazione sono già stati trovati e corretti.
Costi di sviluppo ridotti	I costi di sviluppo sono proporzionali alla dimensione del software che si sta sviluppando. Riutilizzare il software significa che bisogna scrivere un minor numero di linee di codice.
Rischi di processo ridotti	Il costo del software esistente è già noto, mentre il costo di sviluppo è sempre una questione di giudizio. Questo è un fattore importante per la gestione dei progetti, perché riduce il margine di errore nella stima dei costi. Questo è vero in particolare per il riutilizzo di componenti software relativamente grandi come i sottosistemi.
Compatibilità con gli standard	Alcuni standard, come quelli per le interfacce utente, possono essere implementati nella forma di componenti riutilizzabili. Per esempio, se vengono implementati i menu di un'interfaccia utente tramite componenti riutilizzabili, tutte le applicazioni presenteranno gli stessi formati di menu agli utenti. L'uso di interfacce utente standard migliora l'affidabilità, perché è meno probabile che un utente commetta errori quando gli si presenta un'interfaccia familiare.

Figura 15.1 Benefici del riutilizzo del software.

Figura 15.3 mostra vari metodi per supportare il riutilizzo del software. Ciascuno di questi metodi è descritto brevemente nella Figura 15.4.

Dato questo insieme di tecniche per il riutilizzo, la domanda chiave è: “Qual è la tecnica più adatta da utilizzare in una particolare situazione?” Ovviamente, la risposta dipende dai requisiti del sistema che si sta sviluppando, dalle tecnologie e dai componenti riutilizzabili di cui si dispone, e dall’esperienza del team di sviluppo. Ci sono diversi fattori chiave da considerare quando si pianifica il riutilizzo del software.

1. *Tempi di sviluppo del software.* Se il software deve essere sviluppato velocemente, conviene riutilizzare sistemi completi, anziché singoli componenti. Anche se i requisiti non vengono perfettamente soddisfatti, questo approccio riduce al minimo i tempi di sviluppo.

Problema	Spiegazione
Creare, manutenere e usare una libreria di componenti	Può essere costoso popolare una libreria di componenti riutilizzabili e assicurarsi che gli sviluppatori del software possano utilizzarla. I processi di sviluppo devono essere adattati per fare in modo che la libreria possa essere utilizzata.
Ricerca, comprensione e adattamento dei componenti riutilizzabili	I componenti software devono essere trovati in una libreria, compresi e a volte adattati per operare in un nuovo ambiente. Gli ingegneri devono avere sufficiente fiducia di trovare un componente nella libreria, prima di includere la ricerca dei componenti nel loro normale processo di sviluppo.
Aumento dei costi di manutenzione	Se non è disponibile il codice sorgente di un sistema o componente riutilizzato, i costi di manutenzione potrebbero crescere, perché gli elementi riutilizzati potrebbero diventare incompatibili con le modifiche apportate al sistema.
Mancanza di supporto degli strumenti	Alcuni strumenti software non supportano lo sviluppo con il riutilizzo. Può essere difficile o impossibile integrare questi strumenti con un sistema di librerie di componenti. Il processo software previsto da questi strumenti potrebbe non tenere conto del riutilizzo. Questo riguarda più spesso gli strumenti che supportano l'ingegneria dei sistemi integrati, anziché gli strumenti di sviluppo orientati agli oggetti.
Sindrome del “non inventato qui”	Alcuni ingegneri informatici preferiscono riscrivere i componenti perché credono di poterli migliorare. Questo ha in parte a che fare con la fiducia e in parte con il fatto che la scrittura di un software originale è considerata una sfida più grande del riutilizzo del software scritto da altri.

Figura 15.2 Problemi associati al riutilizzo del software.

2. *Vita del software*. Se si sta sviluppando un sistema che dovrà durare a lungo, è bene concentrarsi sulla manutenibilità del sistema. Occorre pensare non soltanto ai benefici immediati del riutilizzo del software, ma anche alle implicazioni a lungo termine.

Durante la vita del software, si dovrà adattare il sistema a nuovi requisiti, che probabilmente richiederanno modifiche alle parti del sistema. Se non si ha accesso al codice sorgente dei componenti riutilizzabili, è preferibile evitare di utilizzare sistemi e componenti off-the-shelf di fornitori esterni, perché non si può essere certi che questi fornitori saranno in grado di continuare a prestare assistenza al software riutilizzato. Sarebbe meglio riutilizzare componenti e sistemi open-source (Capitolo 7), perché così sarebbe garantito l'accesso al loro codice sorgente.

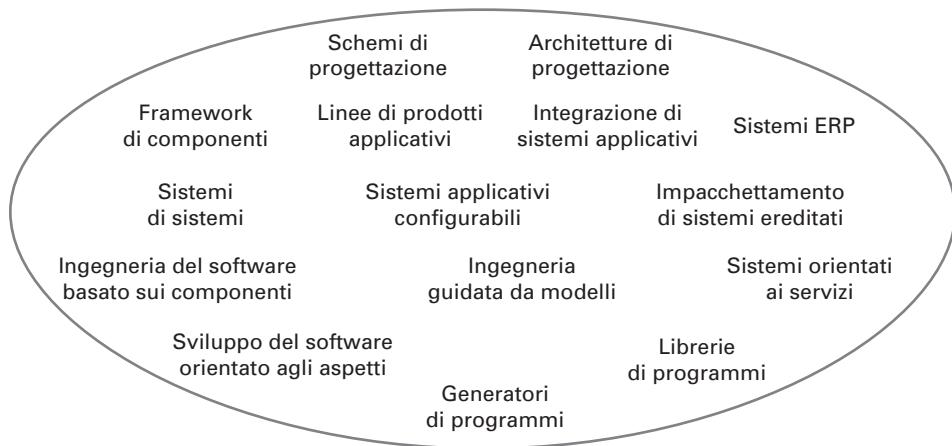


Figura 15.3 Panoramica sul riutilizzo.

3. *Background, capacità ed esperienza del team di sviluppo.* Tutte le tecnologie di riutilizzo del software sono alquanto complesse e occorre molto tempo per capirle e usarle efficacemente. Pertanto, conviene concentrare gli sforzi del riutilizzo in aree dove il team di sviluppo è più esperto.
4. *Criticità del software e suoi requisiti non funzionali.* Per i sistemi critici che devono essere certificati da autorità di controllo esterne potrebbe essere necessario creare un caso di fidatezza o di protezione per il sistema (Capitolo 12). Questo è difficile da fare se non si ha accesso al codice sorgente del software. Se il software ha requisiti di prestazioni molto severi, potrebbe essere impossibile utilizzare strategie come l'ingegneria guidata da modelli (*model-driven engineering* o MDE) (Capitolo 5), che si basa sulla generazione di codice partendo dal modello riutilizzabile di un sistema specifico per un dominio. I generatori di codice MDE spesso generano un codice relativamente inefficiente.
5. *Dominio di applicazione.* In molti domini applicativi, come i sistemi di produzione e di informazione medica, ci sono prodotti generici che possono essere riutilizzati con un'apposita configurazione. Questo è uno degli approcci più efficaci al riutilizzo, ed è quasi sempre più economico acquistare, anziché costruire, un nuovo sistema.
6. *La piattaforma dove sarà eseguito il sistema.* Alcuni modelli di componenti, come .NET, sono specifici per le piattaforme Microsoft. Analogamente, alcuni sistemi applicativi generici possono essere eseguiti su una determinata piattaforma, e possono essere riutilizzati solo se il sistema da sviluppare è progettato per la stessa piattaforma.

Metodo	Descrizione
Framework applicativi	Collezioni di classi astratte e concrete che possono essere adattate ed estese per creare sistemi applicativi.
Integrazione di sistemi applicativi	Due o più sistemi applicativi vengono integrati per fornire funzionalità più estese.
Schemi architetturali	Le architetture software standard che supportano i tipi più comuni di sistemi applicativi sono utilizzate come base delle applicazioni. Consultare i Capitoli 6, 11 e 17.
Sviluppo di software orientato agli aspetti	I componenti condivisi sono inseriti in un'applicazione in diversi punti, durante la compilazione del programma. Consultare il Capitolo 31.
Ingegneria del software basato sui componenti	I sistemi vengono sviluppati integrando componenti (collezioni di oggetti) che sono conformi a modelli standard di componenti, descritti nel Capitolo 16.
Sistemi applicativi configurabili	I sistemi specifici di un dominio vengono progettati in modo che possano essere configurati secondo le esigenze di determinati clienti del sistema.
Schemi di progettazione	Astrazioni generiche che riguardano più applicazioni sono rappresentate come schemi di progettazione che mostrano oggetti astratti e concreti e loro interazioni (Capitolo 7).
Sistemi ERP	Sistemi su larga scala che encapsulano generiche funzionalità e regole aziendali vengono configurati per un'organizzazione.
Impacchettamento di sistemi ereditati	Sistemi ereditati (Capitolo 9) possono essere “impacchettati” definendo un insieme di interfacce attraverso le quali è possibile accedere a tali sistemi.
Ingegneria guidata da modelli	Il software è rappresentato come modelli di domini e modelli di implementazione; il codice è generato da questi modelli (Capitolo 5).
Generatori di programmi	Un sistema generatore integra le conoscenze di un particolare tipo di applicazione per generare sistemi o frammenti di sistema in tale dominio applicativo da un modello di sistema fornito dall'utente.
Librerie di programmi	Librerie di classi e di funzioni, che implementano astrazioni comunemente utilizzate, sono disponibili per il riutilizzo.
Sistemi orientati ai servizi	I sistemi vengono sviluppati collegando servizi condivisi che possono essere forniti esternamente (Capitolo 18).
Linee di prodotti applicativi	Un tipo di applicazione viene generalizzato attorno a un'architettura comune così da poter essere adattata a diversi clienti.
Sistemi di sistemi	Due o più sistemi distribuiti vengono integrati per creare un nuovo sistema.

Figura 15.4 Metodi che supportano il riutilizzo del software.

Riutilizzo basato sui generatori

Il riutilizzo basato sui generatori consiste nell'incorporare alcuni concetti e conoscenze di riutilizzo in appositi strumenti automatici e nell'offrire agli utenti di questi strumenti un modo semplice per integrare un codice specifico con queste conoscenze generiche. Questo approccio di solito è più efficace nelle applicazioni specifiche di un dominio. Le soluzioni note dei problemi in quel dominio vengono incorporate nel generatore e selezionate dall'utente per creare un nuovo sistema.

<http://software-engineering-book.com/web/generator-reuse/>

Il campo delle possibili tecniche di riutilizzo è così vasto che nella maggior parte delle situazioni c'è sempre qualche possibilità di riutilizzare il software. Attuare il riutilizzo spesso è un problema manageriale, anziché tecnico. Alcuni manager non intendono accettare compromessi per i requisiti del loro sistema pur di consentire il riutilizzo di componenti; alcuni manager potrebbero non capire i rischi associati al riutilizzo, mentre conoscono i rischi connessi allo sviluppo di un nuovo sistema. Sebbene i rischi nello sviluppo di un nuovo software di solito siano più elevati, tuttavia alcuni manager potrebbero preferire i rischi noti dello sviluppo a quelli sconosciuti del riutilizzo. Per incentivare il riutilizzo nelle società, potrebbe essere necessario introdurre un programma di riutilizzo basato sulla creazione di sistemi e processi riutilizzabili per facilitare il riutilizzo (Jacobsen, Griss e Jonsson 1997).

15.2 Framework applicativi

I primi sviluppatori entusiasti dello sviluppo orientato agli oggetti dicevano che uno dei vantaggi chiave dei metodi orientati agli oggetti era che gli oggetti potevano essere riutilizzati in sistemi differenti. Tuttavia, l'esperienza ha dimostrato che gli oggetti spesso sono troppo accurati e specifici per una particolare applicazione. Spesso ci vuole più tempo per capire e adattare un oggetto che per implementarlo di nuovo. Oggi sappiamo che il riutilizzo orientato agli oggetti è supportato meglio in un processo di sviluppo orientato agli oggetti tramite astrazioni più generiche dette framework.

Un *framework* è una struttura generica che viene estesa per creare un sottosistema (o applicazione) più specifico. Schmidt e altri (Schmidt et al. 2004) definiscono il framework in questo modo:

un insieme integrato di artefatti software (come classi, oggetti e componenti) che contribuiscono a realizzare un'architettura riutilizzabile per una famiglia di applicazioni correlate.¹

¹ Schmidt D. C., A. Gokhale e B. Natarajan 2004. “Leveraging Application Frameworks” *ACM Queue* 2 (5 (July/August)): 66-75. doi:10.1145/1016998.1017005.

I framework forniscono il supporto a generiche funzionalità che molto probabilmente saranno utilizzate in tutte le applicazioni di un certo tipo. Per esempio, il framework di un'interfaccia utente fornirà il supporto alla gestione degli eventi dell'interfaccia e includerà una serie di strumenti per costruire le videate. Viene lasciata allo sviluppatore la facoltà di personalizzare un'interfaccia aggiungendo funzionalità specifiche per una particolare applicazione. Per esempio, in un framework di interfacce utente, lo sviluppatore definisce i layout che sono appropriati all'applicazione che sta implementando.

I framework supportano il riutilizzo dei progetti in quanto forniscono lo scheletro dell'architettura di un'applicazione, e il riutilizzo di specifiche classi di un sistema. L'architettura è implementata dalle classi di oggetti e dalle loro interazioni. Le classi sono riutilizzate direttamente e possono essere estese utilizzando funzionalità quali l'ereditarietà e il polimorfismo.

I framework vengono implementati come una collezione di classi di oggetti concrete e astratte in un linguaggio di programmazione orientato agli oggetti. I framework sono, quindi, specifici di un linguaggio. I framework sono disponibili nei linguaggi di programmazione orientati agli oggetti, come Java, C# e C++, e nei linguaggi dinamici come Ruby e Python. In effetti, un framework può incorporare altri framework, dove ciascun framework è progettato per supportare lo sviluppo di una parte dell'applicazione. È possibile utilizzare un framework per creare un'applicazione completa o per implementare una parte dell'applicazione, come l'interfaccia grafica utente.

I framework più diffusi sono quelli utilizzati per realizzare applicazioni web (*Web Application Framework* o WAF), che supportano la costruzione di siti web dinamici. L'architettura di un WAF di solito si basa su uno schema composito MVC (Model-View-Controller) illustrato nella Figura 15.5. Lo schema MVC fu proposto originariamente negli anni '80 come un approccio alla progettazione delle interfacce grafiche utente (GUI), che accettavano più rappresentazioni di un oggetto e stili di interazione distinti in ciascuna di queste rappresentazioni. In

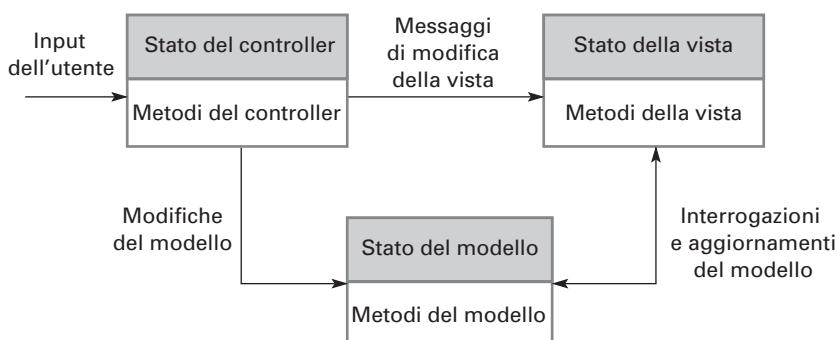


Figura 15.5 Lo schema Model-View-Controller.

sostanza, lo stato era separato della sua rappresentazione, in modo tale che lo stato poteva essere aggiornato da ciascuna rappresentazione.

Un framework MVC supporta la rappresentazione dei dati in vari modi e consente l’interazione con ciascuna di queste rappresentazioni. Quando i dati vengono modificati tramite una di queste rappresentazioni, il modello del sistema viene modificato e i controller associati a ciascuna vista (view) aggiornano le loro rappresentazioni.

I framework sono spesso implementazioni di schemi di progettazione (descritti nel Capitolo 7). Per esempio, un framework MVC include gli schemi Observer, Strategy e Composite e altri schemi descritti da Gamma (Gamma et al. 1995). La natura generale degli schemi e il loro utilizzo di classi astratte e concrete permettono l’estensibilità. Senza schemi, i framework sarebbero quasi certamente inutilizzabili.

Mentre ciascun framework include una funzionalità leggermente differente, i framework delle applicazioni web di solito forniscono componenti e classi che supportano le seguenti funzioni.

1. *Protezione.* I WAF possono includere classi che agevolano l’implementazione dell’autenticazione degli utenti (login) e del controllo degli accessi per garantire che gli utenti possano accedere soltanto ad alcune funzionalità consentite nel sistema.
2. *Pagine web dinamiche.* Sono fornite alcune classi che aiutano a definire i modelli delle pagine web e a riempire queste pagine dinamicamente con i dati provenienti dal database del sistema.
3. *Integrazione del database.* I framework di solito non includono un database, ma suppongono che sarà utilizzato un database esterno, come MySQL. Un framework potrebbe includere classi che forniscono un’interfaccia astratta con diversi database.
4. *Gestione delle sessioni.* Fanno parte di un WAF apposite classi per creare e gestire le sessioni (un numero di interazioni con il sistema da parte di un utente).
5. *Interazioni degli utenti.* I framework del Web forniscono il supporto all’AJAX (Holdener 2008) e all’HTML5 (Sarris 2013), che consente di creare pagine web interattive. Sono incluse classi che permettono di creare interfacce indipendenti dai dispositivi, che possono essere adattate automaticamente ai cellulari e ai tablet.

Per implementare un sistema tramite un framework, occorre aggiungere delle classi concrete che ereditano le operazioni dalle classi astratte nel framework. Inoltre, occorre definire i “callback” – metodi che vengono chiamati in risposta a eventi riconosciuti dal framework. Gli oggetti del framework, non quelli specifici dell’applicazione, sono responsabili del controllo nel sistema. Schmidt e altri (Schmidt, Gokhale e Natarajan 2004) chiamano questo “inversione del controllo”.

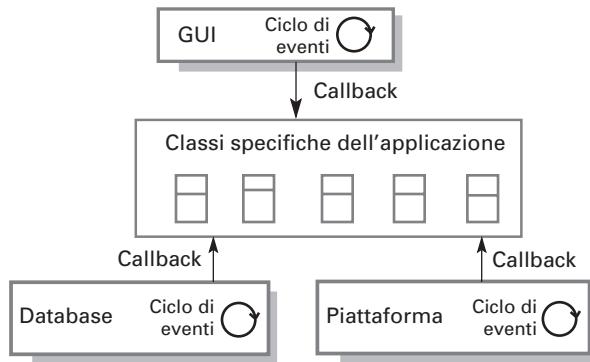


Figura 15.6 Inversione del controllo nei framework.

In seguito al verificarsi di eventi nell’interfaccia utente e nel framework del database, gli oggetti chiamano i “metodi hook” (gancio), che sono collegati alle funzionalità create per l’utente. Queste funzionalità definiscono come l’applicazione deve rispondere agli eventi (Figura 15.6). Per esempio, un framework avrà un metodo che gestisce i clic del mouse dall’ambiente operativo. Questo metodo è detto metodo hook, che può essere configurato per chiamare i metodi appropriati dell’applicazione che gestiscono i clic del mouse.

Fayad e Schmidt (Fayad e Schmidt 1997) hanno analizzato tre classi di framework.

1. *Framework delle infrastrutture del sistema.* Questi framework supportano lo sviluppo di infrastrutture come le comunicazioni, le interfacce utente e i compilatori.
2. *Framework per l’integrazione di middleware.* Sono composti da un insieme di standard e di classi di oggetti associate che supportano la comunicazione e lo scambio di informazioni tra i componenti. Esempi di questo tipo di framework includono .NET di Microsoft ed Enterprise Java Beans (EJB). Questi framework forniscono il supporto ai modelli di componenti standardizzati (Capitolo 16).
3. *Framework per applicazioni aziendali.* Riguardano specifici domini applicativi, come le telecomunicazioni o i sistemi finanziari (Baumer et al. 1997). Questi framework incorporano le conoscenze dei domini delle applicazioni e supportano lo sviluppo di applicazioni per utenti finali. Attualmente non sono molto utilizzati e sono stati in gran parte sostituiti dalle linee di prodotti software.²

² Fayad, M. E. e D. C. Schmidt. 1997. “Object-Oriented Application Frameworks.” *Comm. ACM* 40 (10): 32-38. doi:10.1145/262793.262798.

Le applicazioni costruite tramite framework possono costituire la base per altri riutilizzi sfruttando il concetto di linee di prodotti software o famiglie di applicazioni. Poiché queste applicazioni sono costruite utilizzando un framework, modificare i membri di una famiglia per creare istanze del sistema, di solito, è un processo molto semplice; basta semplicemente riscrivere le classi concrete e i metodi che sono stati aggiunti al framework.

I framework sono un approccio molto efficace al riutilizzo del software. Tuttavia, essi sono costosi da introdurre nei processi di sviluppo del software, in quanto sono intrinsecamente complessi; inoltre, potrebbero essere necessari parecchi mesi per imparare a utilizzarli. Potrebbe essere difficile e costoso valutare i framework disponibili per scegliere quello più appropriato alle proprie esigenze. Il debugging delle applicazioni basate sui framework è più difficile del debugging del codice originale, in quanto gli sviluppatori potrebbero non capire come interagiscono i metodi di un framework. Gli strumenti di debugging potrebbero fornire informazioni sui componenti riutilizzati del framework che gli sviluppatori potrebbero non capire.

15.3 Linee di prodotti software

Quando una società deve supportare un certo numero di sistemi simili, non identici, uno degli approcci più efficaci al riutilizzo è creare una linea di prodotti software. I sistemi di controllo dell'hardware sono spesso utilizzati adottando questo approccio al riutilizzo, in quanto sono applicazioni specifiche di domini, in aree quali i sistemi logistici o medicali. Per esempio, un costruttore di stampanti deve sviluppare il software di controllo delle stampanti, con una versione specifica del prodotto per ogni tipo di stampante. Queste versioni del software hanno molto in comune, quindi ha senso creare un prodotto principale (*nucleo* o *core*), che potrà essere adattato a ogni tipo di stampante, generando una linea di prodotti.

Una linea di prodotti software è un insieme di applicazioni con un'architettura comune e componenti condivisi, dove ciascuna applicazione è specializzata a soddisfare determinate richieste dei clienti. Il sistema principale è progettato in modo che possa essere configurato e adattato per soddisfare le esigenze di vari clienti o dispositivi. Questo potrebbe richiedere la configurazione di alcuni componenti, implementando componenti aggiuntivi e modificando alcuni dei componenti per soddisfare i nuovi requisiti.

Sviluppare applicazioni adattando una generica versione di un'applicazione principale significa che un'alta percentuale del codice dell'applicazione viene riutilizzata in ciascun sistema. Il test è semplificato, in quanto anche i test di gran parte dell'applicazione possono essere riutilizzati, riducendo così il tempo di sviluppo complessivo dell'applicazione. Gli ingegneri conoscono il dominio dell'applicazione tramite la linea dei prodotti software e, quindi, diventano specialisti che possono operare più rapidamente per sviluppare nuove applicazioni.

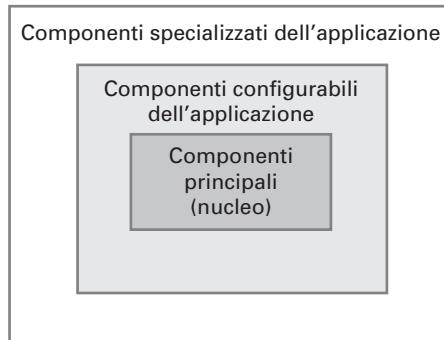


Figura 15.7 Organizzazione di un sistema base per una linea di prodotti software.

Le linee di prodotti software di solito scaturiscono da applicazioni esistenti. In altre parole, una società sviluppa un'applicazione e, poi, quando le viene richiesto un sistema simile, riutilizza informalmente il codice di questa applicazione nella nuova. Lo stesso processo viene utilizzato per sviluppare altre applicazioni simili. Tuttavia, poiché le modifiche tendono a corrompere la struttura delle applicazioni, man mano che vengono sviluppate nuove istanze, diventa sempre più difficile creare una nuova versione. Di conseguenza, potrebbe essere opportuno decidere di progettare una generica linea di prodotti. Questo richiede l'identificazione delle funzionalità comuni nelle istanze dei prodotti e lo sviluppo di un'applicazione di base, che sarà poi utilizzata per i successivi sviluppi.

Questa applicazione di base (Figura 15.7) viene progettata per semplificare il riutilizzo e la riconfigurazione. In generale, un'applicazione di base include i seguenti elementi:

1. i componenti principali che forniscono il supporto alle infrastrutture. Questi componenti di solito non vengono modificati quando si sviluppa una nuova istanza della linea dei prodotti software;
2. i componenti configurabili che possono essere modificati e configurati per specializzarli in una nuova applicazione. A volte, è possibile riconfigurare questi componenti senza modificare il loro codice utilizzando un apposito linguaggio di configurazione dei componenti;
3. i componenti specializzati in determinati domini, che possono essere rimpiazzati quando viene creata una nuova istanza della linea dei prodotti.

I framework applicativi e le linee di prodotti software hanno molto in comune. Entrambi supportano un'architettura e componenti comuni, e richiedono un nuovo processo di sviluppo per creare una versione specifica di un sistema. Le differenze principali tra questi approcci sono le seguenti:

1. un framework applicativo si basa su caratteristiche orientate agli oggetti, quali l'ereditarietà e il polimorfismo, per implementare le estensioni del framework. In generale, il codice del framework non viene modificato, e le

possibili modifiche sono limitate a quanto è supportato dal framework. Le linee di prodotti software non sono necessariamente create adottando un approccio orientato agli oggetti. I componenti delle applicazioni vengono modificati, cancellati o riscritti. Non ci sono limiti, almeno in teoria, alle modifiche che possono essere apportate;

2. quasi tutti i framework applicativi offrono un supporto generico, non specifico a un particolare dominio. Per esempio, ci sono framework applicativi per creare applicazioni web. Una linea di prodotti software di solito incorpora informazioni dettagliate su una piattaforma o su un dominio. Per esempio, ci potrebbe essere una linea di prodotti software per applicazioni web che gestiscono cartelle cliniche;
3. le linee di prodotti software spesso sono applicazioni per il controllo di unità hardware. Per esempio, ci potrebbe essere una linea di prodotti software per una famiglia di stampanti. Questo significa che la linea di prodotti deve fornire il supporto all’interfaccia con le unità hardware. I framework applicativi di solito sono orientati al software e non includono componenti che interagiscono con l’hardware;
4. le linee di prodotti software sono composte da una famiglia di applicazioni correlate, di proprietà di un’unica società. Quando viene creata una nuova applicazione, il punto di partenza spesso è il membro più vicino alla famiglia delle applicazioni, non una generica applicazione principale.

Se si sta sviluppando una linea di prodotti software tramite un linguaggio di programmazione orientato agli oggetti, allora si potrebbe utilizzare un framework applicativo come base per il sistema. Si crea il nucleo della linea di prodotti estendendo il framework con componenti specifici del dominio utilizzando i suoi meccanismi interni. Segue una seconda fase dello sviluppo nella quale vengono create le versioni del sistema per clienti differenti. Per esempio, è possibile utilizzare un framework basato sul Web per costruire il nucleo di una linea di prodotti software che supporta un servizio di informazioni e assistenza sul Web (help desk). Questa “linea di prodotti help desk” potrebbe essere ulteriormente specializzata per fornire particolari tipi di servizi di help desk.

L’architettura di una linea di prodotti software spesso rispecchia un generico stile o schema architettonicale specifico di alcune applicazioni. Per esempio, consideriamo un sistema di una linea di prodotti che è progettato per gestire l’invio di veicoli per servizi di emergenza. Gli operatori di questo sistema ricevono le telefonate di segnalazione degli incidenti, trovano il veicolo appropriato al tipo di incidente e inviano il veicolo sul luogo dell’incidente. Gli sviluppatori di tale sistema potrebbero creare versioni di questo sistema per la polizia, i vigili del fuoco e gli ospedali.

Il sistema di invio dei veicoli di emergenza è un esempio di una generica architettura di gestione e allocazione di risorse (Figura 15.8). I sistemi di gestione delle risorse usano un database di risorse disponibili e includono i componenti per

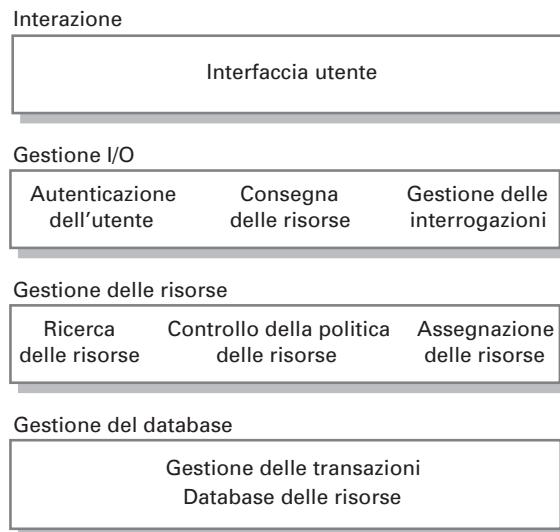


Figura 15.8 Architettura di un sistema di gestione di risorse.

implementare le politiche di allocazione delle risorse che sono state decise dalla società che usa il sistema. Gli utenti interagiscono con il sistema di gestione delle risorse per richiedere e ottenere le risorse e per porre domande sulle risorse e la loro disponibilità. Nella Figura 15.9 si può vedere come questa struttura a quattro strati viene istanziata: sono rappresentati i moduli, che possono essere inclusi in una linea di prodotti di sistemi per l'invio dei veicoli, e i componenti per ogni livello del sistema.

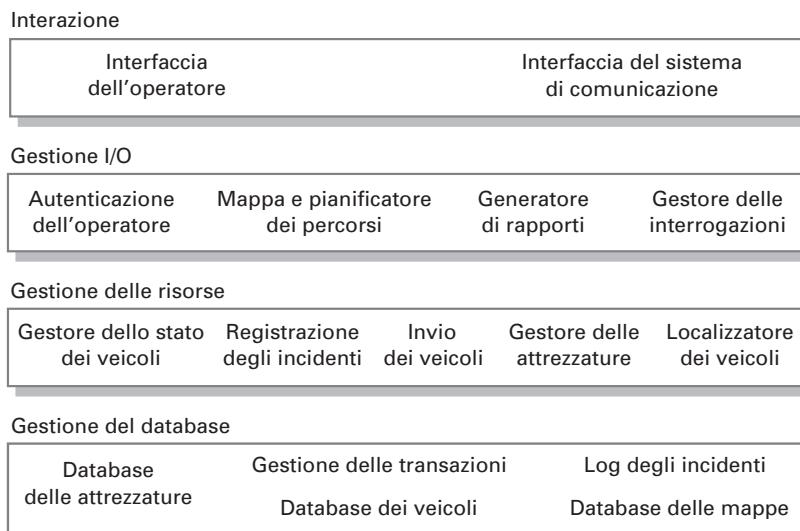


Figura 15.9 Architettura della linea di prodotti per un sistema di invio di veicoli.

1. A livello di interazione, i componenti forniscono un’interfaccia di visualizzazione con l’operatore e un’interfaccia con il sistema di comunicazione utilizzato.
2. A livello di gestione I/O (livello 2), i componenti gestiscono l’autenticazione dell’operatore, generano i rapporti degli incidenti e dei veicoli inviati, supportano la visualizzazione delle mappe e la pianificazione dei percorsi e forniscono agli operatori un meccanismo per interrogare i database del sistema.
3. Al livello di gestione delle risorse (livello 3), i componenti permettono di localizzare e inviare i veicoli, aggiornano lo stato dei veicoli e delle attrezzature e registrano i dettagli degli incidenti.
4. Al livello del database, oltre al solito supporto alla gestione delle transazioni, ci sono database separati per i veicoli, le attrezzature e le mappe.

Per creare una nuova istanza di questo sistema si devono modificare i singoli componenti. Per esempio, la polizia ha un gran numero di veicoli, ma un numero minore di tipi di veicoli, mentre i vigili del fuoco hanno molti tipi di veicoli specializzati, ma un numero relativamente limitato di veicoli. Quindi, per implementare un sistema per questi differenti servizi, potrebbe essere necessario definire una struttura diversa per il database dei veicoli.

Si possono sviluppare vari tipi di specializzazione per una linea di prodotti software.

1. *Specializzazione della piattaforma.* Si sviluppano versioni dell’applicazione per diverse piattaforme. Per esempio, possono esistere versioni per piattaforme Windows, Mac OS e Linux. In tal caso, le funzionalità dell’applicazione restano inalterate; si modificano soltanto quei componenti che si interfacciano con l’hardware e il sistema operativo.
2. *Specializzazione dell’ambiente.* Si creano versioni dell’applicazione per gestire particolari ambienti operativi e unità periferiche. Per esempio, un sistema per servizi di emergenza può esistere in diverse versioni, a seconda del tipo di hardware di comunicazione utilizzato da ciascun servizio. Per esempio, le radio della polizia potrebbero avere un sistema di cifratura che deve essere utilizzato. I componenti della linea di prodotti vengono modificati per riflettere le funzionalità e le caratteristiche delle attrezzature utilizzate.
3. *Specializzazione funzionale.* Si creano versioni dell’applicazione per clienti specifici con requisiti diversi. Per esempio, un sistema di automazione per biblioteche potrebbe essere modificato a seconda che sia utilizzato in una biblioteca pubblica, in una biblioteca di consultazione o in una biblioteca universitaria. In tal caso, si modificano i componenti che implementano le funzionalità e se ne aggiungono di nuovi.

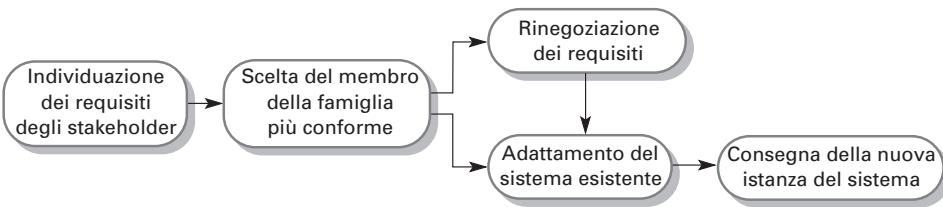


Figura 15.10 Sviluppo di un'istanza del prodotto.

4. *Specializzazione dei processi.* Il sistema viene adattato a specifici processi aziendali. Per esempio, un sistema di gestione degli ordini può essere adattato per far fronte a un processo centralizzato di elaborazione degli ordini in una società e a un processo distribuito in un'altra società.

La Figura 15.10 mostra il processo di estensione di una linea di prodotti software per creare una nuova applicazione. Le attività in questo processo sono le seguenti:

1. *individuazione dei requisiti degli stakeholder.* Si può partire con un normale processo di ingegneria dei requisiti. Tuttavia, poiché esiste già un sistema, gli stakeholder possono provarlo ed eventualmente suggerire le loro modifiche alle funzioni fornite;
2. *scelta del sistema esistente che è più conforme ai requisiti.* Quando si crea un nuovo membro della linea di prodotti, si può iniziare dall'istanza del prodotto più vicino ai requisiti richiesti. Si analizzano i requisiti e si sceglie per le modifiche il membro della famiglia che più li soddisfa;
3. *rinegoziazione dei requisiti.* Quando emergono maggiori dettagli sulle modifiche richieste e il progetto è pianificato, alcuni requisiti potrebbero essere rinegoziati con il cliente per minimizzare le modifiche da apportare all'applicazione di base;
4. *adattamento del sistema esistente.* Vengono sviluppati nuovi moduli per il sistema esistente; i moduli esistenti vengono adattati per soddisfare i nuovi requisiti;
5. *consegna del nuovo membro della famiglia di prodotti.* Viene consegnata al cliente la nuova istanza della linea di prodotti. Potrebbe essere richiesta una configurazione alla consegna per rispecchiare i particolari ambienti in cui il sistema sarà utilizzato. In questa fase si dovrebbero documentare le sue funzionalità chiave, in modo che il sistema possa essere utilizzato come base per futuri sviluppi.

Quando si crea un nuovo membro di una linea di prodotti, occorre trovare un compromesso tra il maggior riutilizzo possibile dell'applicazione generica e il soddisfacimento delle richieste specifiche degli stakeholder. Quanto più i requisiti del sistema sono dettagliati, tanto meno è probabile che i componenti esistenti

li soddisfino. Tuttavia, se gli stakeholder dimostrano una maggiore flessibilità e limitano le richieste di modifiche da apportare al sistema, si può consegnare il sistema più velocemente e a costi minori.

Le linee di prodotti software sono progettate per essere configurabili. Questa riconfigurazione può richiedere l'aggiunta o la rimozione di componenti dal sistema, la definizione dei parametri e dei vincoli per i componenti del sistema e l'inclusione di conoscenze dei processi aziendali. Tale configurazione può avvenire in due fasi diverse del processo di sviluppo.

1. *Configurazione durante la progettazione.* La società che sta sviluppando il software modifica il nucleo comune di una linea di prodotti sviluppando, selezionando o adattando i componenti per creare un nuovo sistema per un cliente.
2. *Configurazione alla consegna.* Viene progettato un sistema generico che può essere configurato dal cliente o dai consulenti che lavorano con lui. La conoscenza dei requisiti specifici del cliente e l'ambiente operativo del sistema vengono integrati in un insieme di dati di configurazione utilizzati dal sistema generico.

Quando un sistema è configurato durante la progettazione, il fornitore inizia da un sistema generico o da un'istanza di un prodotto software esistente. Modificando ed estendendo i moduli di questo sistema, il fornitore crea un sistema specifico che fornisce le funzionalità richieste dal cliente. Questo di solito richiede la modifica e l'estensione del codice sorgente del sistema, quindi è possibile una maggiore flessibilità rispetto alla configurazione alla consegna.

La configurazione durante la progettazione è utilizzata quando non è possibile applicare l'attuale configurazione di un sistema per sviluppare una nuova versione del sistema. Tuttavia, col tempo, dopo aver creato diversi membri della famiglia con funzionalità simili, si potrebbe decidere di rifactorizzare il nucleo della linea dei prodotti software per includere le funzionalità che sono state implementate in vari membri della famiglia delle applicazioni. Successivamente, queste nuove funzionalità possono essere configurate quando il sistema viene consegnato.

La configurazione alla consegna richiede l'utilizzo di uno strumento di configurazione per creare una configurazione specifica per un sistema, che viene memorizzata in un database o in alcuni file di configurazione (Figura 15.11). Il sistema, che può essere eseguito su un server o su un PC indipendente, consulta questo database in modo che, durante l'esecuzione, le sue funzionalità possano essere adattate al particolare contesto di esecuzione.

Sono possibili diversi livelli di configurazione alla consegna per un sistema.

1. *Scelta dei componenti.* Si scelgono i moduli di un sistema che forniscono le funzionalità richieste. Per esempio, in un sistema per la gestione dei pazienti di un ospedale, si potrebbe scegliere un componente che consente di colle-

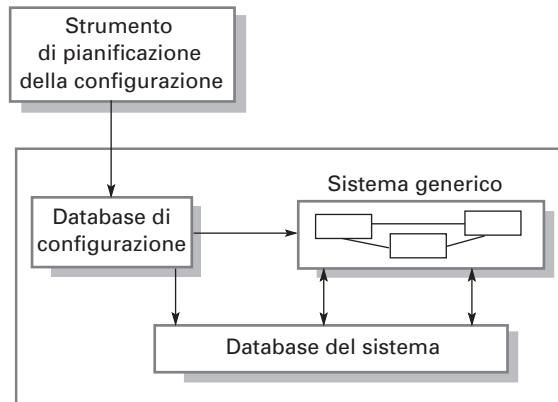


Figura 15.11 Configurazione alla consegna.

gare le immagini medicali (raggi X, TAC ecc.) alle cartelle cliniche dei pazienti.

2. *Definizione dei workflow e delle regole.* Si definiscono i workflow (come le informazioni devono essere elaborate, stadio per stadio) e le regole di validità che devono essere applicate alle informazioni immesse dagli utenti o generate dal sistema.
3. *Definizione dei parametri.* Si specificano i valori dei parametri di un determinato sistema che rispecchiano l'istanza dell'applicazione che si sta creando. Per esempio, potreste specificare la lunghezza massima dei campi di input dell'utente o le caratteristiche dell'hardware collegato al sistema.

La configurazione alla consegna può essere molto complessa e, per i grandi sistemi, potrebbero essere necessari parecchi mesi per configurare e testare il sistema per un cliente. I grandi sistemi potrebbero supportare il processo di configurazione tramite appositi strumenti software, come gli strumenti di pianificazione. Il Paragrafo 15.4.1 fornisce ulteriori dettagli sulla configurazione alla consegna, in quanto tratta il riutilizzo dei sistemi applicativi che devono essere configurati per poter operare in ambienti differenti.

15.4 Riutilizzo dei sistemi applicativi

Un sistema applicativo è un prodotto software che può essere adattato alle esigenze di vari clienti senza modificare il codice sorgente del sistema. I sistemi applicativi sono sviluppati per un mercato generale; non sono sviluppati specificatamente per un singolo cliente. Questi prodotti software sono anche detti COTS (Commercial Off-The-Shelf System). Tuttavia, poiché il termine “COTS” è utilizzato principalmente nei sistemi militari, preferisco chiamare questi prodotti *sistemi applicativi*.

Virtualmente tutto il software per desktop aziendali e molti sistemi basati sui server sono sistemi applicativi. Questo software è progettato per un uso generico, quindi include molte funzionalità che lo rendono potenzialmente riutilizzabile in diversi ambienti operativi e come parti di applicazioni differenti. Torchiano e Morisio (Torchiano e Morisio 2004) hanno scoperto che anche i prodotti open-source sono stati utilizzati spesso senza modifiche e senza guardare il codice sorgente.

I sistemi applicativi vengono adattati utilizzando meccanismi interni di configurazione che consentono di personalizzare le funzionalità dei sistemi per soddisfare particolari esigenze dei clienti. Per esempio, in un sistema di gestione dei pazienti di un ospedale, potrebbero essere definiti i moduli di input e i report di output per vari tipi di pazienti. Altre caratteristiche di configurazione potrebbero consentire al sistema di accettare alcuni plug-in che estendono le sue funzionalità o controllano gli input degli utenti per garantirne la validità.

Questo approccio al riutilizzo del software è stato largamente adottato da grandi società a partire dagli ultimi anni '90, in quanto offre significativi vantaggi rispetto allo sviluppo del software personalizzato:

1. analogamente ad altri tipi di riutilizzo del software, è possibile una consegna più rapida di sistemi affidabili;
2. è possibile verificare quali funzionalità sono fornite dalle applicazioni e, quindi, è più facile valutare se esse sono appropriate. Poiché altre società potrebbero utilizzare già le applicazioni, si dispone anche di una certa esperienza dei sistemi applicativi;
3. alcuni rischi connessi allo sviluppo sono evitati, in quanto si usa un software esistente. Questo approccio, tuttavia, ha i suoi rischi, come dirò in seguito;
4. le aziende possono concentrare le loro risorse sull'attività principale, senza sprecare preziose energie per sviluppare sistemi IT;
5. poiché le piattaforme operative si evolvono, gli aggiornamenti tecnologici sono semplificati, in quanto questi dipendono direttamente dal venditore del sistema applicativo, anziché dal cliente.

Ovviamente, questo approccio all'ingegneria del software ha i suoi problemi:

1. i requisiti di solito devono essere adattati per rispecchiare le funzionalità e le modalità operative del sistema applicativo off-the-shelf. Questo può portare a modifiche devastanti per i processi aziendali esistenti;
2. il sistema applicativo potrebbe basarsi su ipotesi che sono praticamente impossibili da modificare. Il cliente deve quindi adattare le sue attività a tali ipotesi;
3. scegliere il sistema applicativo più appropriato per una società potrebbe essere un'operazione difficile, in quanto molti di questi sistemi non sono

Sistemi applicativi configurabili	Sistemi applicativi integrati
Prodotto singolo che fornisce le funzionalità richieste da un cliente	Più sistemi applicativi vengono integrati per fornire funzionalità personalizzate
Sistemi basati su una generica soluzione e processi standardizzati	Possono essere sviluppate soluzioni flessibili per i processi del cliente
Lo sviluppo è incentrato sulla configurazione del sistema	Lo sviluppo è incentrato sull'integrazione dei sistemi
Il venditore del sistema è responsabile della manutenzione	Il proprietario del sistema è responsabile della manutenzione
Il venditore del sistema fornisce la piattaforma per il sistema	Il proprietario del sistema fornisce la piattaforma per il sistema

Figura 15.12 Sistemi applicativi singoli e integrati.

ben documentati. Una scelta sbagliata potrebbe rendere impossibile il corretto funzionamento del nuovo sistema;

4. potrebbe mancare l'esperienza locale per supportare lo sviluppo dei sistemi. Di conseguenza, il cliente dovrà affidarsi al venditore o a consulenti esterni per avere consigli sullo sviluppo. Questi consigli potrebbero essere indirizzati più alla vendita di prodotti e servizi, dedicando un tempo insufficiente alle reali esigenze del cliente;
5. il venditore del sistema controlla il supporto e l'evoluzione del sistema. Potrebbe uscire dal mercato, rientrare, o apportare modifiche che potrebbero causare problemi ai clienti.

I sistemi applicativi possono essere utilizzati come sistemi singoli o integrati con altri sistemi. I sistemi singoli sono formati da una generica applicazione di un singolo venditore che è configurata per soddisfare i requisiti di un cliente. I sistemi integrati richiedono l'integrazione delle funzionalità dei singoli sistemi, spesso forniti da vendori differenti. La Figura 15.12 riassume le differenze tra questi due approcci. Il Paragrafo 15.4.2 descrive l'integrazione dei sistemi applicativi.

15.4.1 Sistemi applicativi configurabili

I sistemi applicativi configurabili sono sistemi generici che possono essere progettati per supportare un particolare tipo di azienda, attività aziendale o, a volte, un'intera organizzazione aziendale. Per esempio, un sistema prodotto per i dentisti potrebbe gestire appuntamenti, promemoria, cartelle cliniche, telefonate ai pazienti e fatturazioni. Su scala più larga, un sistema ERP (*Enterprise Resource Planning*) può supportare la produzione, le ordinazioni e i processi di gestione delle relazioni con i clienti di una grande società.

I sistemi applicativi specifici di un dominio, come quelli che supportano una funzione aziendale (per esempio, la gestione dei documenti), forniscono le funzionalità che con molta probabilità saranno richieste da una vasta gamma di potenziali utenti. Essi incorporano anche delle ipotesi su come operano gli utenti, e queste ipotesi potrebbero causare problemi in determinate situazioni. Per esempio, un sistema che gestisce le iscrizioni degli studenti universitari potrebbe ipotizzare che gli studenti possano iscriversi a un solo corso di laurea. Tuttavia, se più università collaborano per offrire corsi di laurea congiunti, potrebbe essere impossibile rappresentare questo caso nel sistema.

I sistemi ERP, come quelli prodotti da SAP e Oracle, sono sistemi integrati su larga scala, progettati per supportare pratiche aziendali, quali le ordinazioni, la fatturazione, la gestione del magazzino e la programmazione della produzione (Monk e Wagner 2013). Il processo di configurazione di questi sistemi prevede l’acquisizione di informazioni dettagliate sulle attività e sui processi aziendali, e l’inserimento di tali informazioni in un database di configurazione. Questo processo spesso richiede conoscenze dettagliate delle notazioni e degli strumenti di configurazione e, di solito, è svolto da consulenti che operano a fianco dei clienti del sistema.

Un generico sistema ERP include un numero di moduli che possono essere composti in vari modi per creare un sistema per un particolare cliente. Il processo di configurazione prevede la scelta dei moduli da includere, la configurazione di ciascuno di essi, la definizione dei processi e delle regole aziendali, la definizione della struttura e dell’organizzazione del database del sistema. Un modello dell’architettura di un sistema ERP che supporta una serie di funzioni aziendali è illustrato nella Figura 15.13.

Le caratteristiche chiave di questa architettura sono elencate qui di seguito.

1. Un certo numero di moduli supportano le funzioni aziendali. Sono moduli generici che possono supportare interi dipartimenti o divisioni di una società. Nell’esempio illustrato nella Figura 15.13, i moduli che sono stati selezionati per il sistema sono: un modulo per la gestione degli acquisti; un modulo per la gestione delle forniture; un modulo per la logistica di supporto alle consegne dei prodotti e un modulo per la gestione delle informazioni sui clienti.
2. Una serie di modelli di processi aziendali che sono in relazione con le attività previste nei singoli moduli. Per esempio, il modello del processo delle ordinazioni può definire il modo in cui gli ordini devono essere creati e approvati; vengono specificati i ruoli e le attività che riguardano l’emissione di un ordine.
3. Un database comune che mantiene le informazioni su tutte le funzioni aziendali correlate. Quindi, non dovrebbe essere necessario replicare le informazioni, come quelle sui clienti, in parti differenti del sistema.

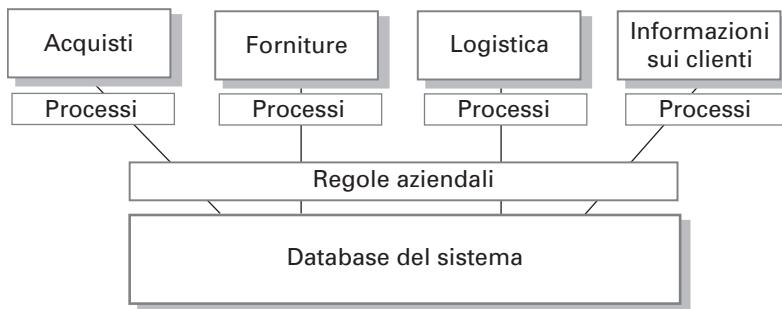


Figura 15.13 Architettura di un sistema ERP.

4. Una serie di regole aziendali che si applicano a tutti i dati del database. Di conseguenza, quando i dati vengono immessi da una funzione, queste regole dovrebbero garantire che tali dati siano coerenti con quelli richiesti da altre funzioni. Per esempio, una regola aziendale potrebbe richiedere che tutti i rimborsi spese siano autorizzati dal direttivo responsabile di chi effettua la richiesta di un rimborso.

I sistemi ERP sono utilizzati in quasi tutte le grandi aziende per supportare alcune o tutte le loro funzioni; quindi, sono forme di riutilizzo del software largamente diffuse. Il limite ovvio di questo approccio al riutilizzo è che la funzionalità dell'applicazione del cliente è ristretta alle funzionalità dei moduli del sistema ERP. Se un'azienda richiede funzionalità aggiuntive, potrebbe essere necessario sviluppare un apposito sistema add-on per fornire questa nuova funzionalità.

In aggiunta, i processi e le operazioni della società cliente devono essere definiti nel linguaggio di configurazione del sistema ERP. Questo linguaggio incorpora le conoscenze dei processi aziendali così come sono visti dal venditore del sistema; potrebbero esserci delle incompatibilità tra queste ipotesi e i concetti e i processi utilizzati nelle attività del cliente. Se c'è una grave incompatibilità tra il modello delle attività del cliente e il modello utilizzato dal sistema ERP, aumentano notevolmente le probabilità che il sistema ERP non sarà in grado di soddisfare le esigenze reali del cliente (Scott 1999).

Per esempio, in un sistema ERP che era stato venduto a una università, un concetto fondamentale del sistema era quello di cliente. In questo sistema, un cliente era un agente esterno che acquistava beni e servizi da un fornitore. Questo concetto causò grandi difficoltà durante la configurazione del sistema. Le università in effetti non hanno clienti, ma hanno relazioni simili a quelle dei clienti con una vasta gamma di persone e organizzazioni, quali studenti, agenzie di finanziamento della ricerca ed enti di beneficenza. Nessuna di queste relazioni è compatibile con quella di cliente, dove una persona o un'azienda acquista un prodotto o un servizio da un altro soggetto. In questo caso particolare, ci sono voluti parecchi mesi per risolvere questo problema di incompatibilità, e la soluzione finale riuscì solo parzialmente a soddisfare le richieste dell'università.

I sistemi ERP di solito richiedono una configurazione dettagliata per essere adattati ai requisiti delle organizzazioni che vogliono utilizzarli. La configurazione consiste nei seguenti punti.

1. Selezionare la funzionalità richiesta dal sistema, per esempio, decidendo quali moduli includere.
2. Stabilire un modello di dati che definisce come i dati dell’organizzazione dovranno essere strutturati nel database del sistema.
3. Definire le regole aziendali da applicare ai dati.
4. Definire le interazioni previste con i sistemi esterni.
5. Progettare i moduli di input e i report di output generati dal sistema.
6. Progettare i nuovi processi aziendali conformemente al modello dei processi supportato del sistema.
7. Impostare i parametri che definiscono come il sistema viene installato nella sua piattaforma.

Una volta completata la configurazione, il nuovo sistema è pronto per essere testato. Il test è un problema importante quando un sistema è configurato, anziché programmato, utilizzando un linguaggio convenzionale, per due motivi:

1. automatizzare il test potrebbe essere difficile o impossibile. Potrebbe non esserci un facile accesso a un’API che può essere utilizzata testando i framework, come JUnit, e quindi il sistema deve essere testato manualmente immettendo dei dati di prova nel sistema. Inoltre, i sistemi spesso sono specificati in modo informale, quindi definire i test case potrebbe essere difficile senza un consistente aiuto da parte degli utenti finali;
2. gli errori dei sistemi sono spesso subdoli e specifici di ogni processo aziendale. Il sistema applicativo o il sistema ERP sono una piattaforma affidabile, quindi i guasti tecnici sono rari. I problemi che si verificano spesso sono dovuti a incomprensioni tra quelli che configurano il sistema e gli stakeholder. Chi prova un sistema senza avere conoscenze dettagliate dei processi degli utenti finali non è in grado di rilevare questi errori.

15.4.2 Sistemi applicativi integrati

I sistemi applicativi integrati includono due o più sistemi applicativi o, a volte, ereditati. Questo approccio può essere utilizzato quando un singolo sistema applicativo non riesce a soddisfare tutti i requisiti o quando si vuole integrare un nuovo sistema applicativo con sistemi che sono già in uso. I sistemi dei componenti possono interagire tramite le loro API o le interfacce dei servizi. In alternativa, essi possono essere composti collegando gli output di un sistema con l’input di un altro o aggiornando i database utilizzati dalle applicazioni.

Per sviluppare i sistemi applicativi integrati, occorre fare alcune scelte di progettazione.

1. *Quali sistemi applicativi singoli offrono le funzionalità più appropriate?* Di solito, sono disponibili molti prodotti software, che possono essere combinati in vari modi. Se non avete esperienza con un particolare sistema applicativo, potrebbe essere difficile decidere quale prodotto è più appropriato.
2. *Come saranno scambiati i dati?* Sistemi differenti di solito usano strutture e formati di dati differenti. Sarà opportuno scrivere degli adattatori che convertono i dati da una rappresentazione all'altra. Questi adattatori sono sistemi a runtime che operano a fianco dei sistemi applicativi.
3. *Quali caratteristiche di un prodotto saranno effettivamente utilizzate?* I singoli sistemi applicativi possono includere più funzionalità di quelle richieste, e alcune funzionalità potrebbero essere duplicate nei vari prodotti. Dovrete decidere quali caratteristiche di un prodotto sono più appropriate alle vostre esigenze. Se possibile, dovreste anche negare l'accesso a funzionalità inutilizzate, perché queste potrebbero interferire con il funzionamento normale del sistema.

Consideriamo il seguente scenario come rappresentazione dell'integrazione di sistemi applicativi. Una grande società vuole sviluppare un sistema di acquisti per permettere ai propri dipendenti di emettere ordini dai propri PC. Con l'introduzione di questo sistema, la società stima un risparmio di 5 milioni di euro all'anno. Centralizzando gli acquisti, il nuovo sistema assicura che gli ordini siano sempre inviati ai fornitori che offrono i prezzi migliori e riduce i costi della documentazione cartacea associata agli ordini. Come nei sistemi manuali, il nuovo sistema di acquisti prevede la scelta dei beni offerti da un fornitore, la creazione di un ordine e la sua approvazione, l'invio dell'ordine al fornitore, il ritiro della merce e la conferma del pagamento.

La società ha già un sistema software ereditato che è utilizzato da un ufficio acquisti centrale. Questo sistema software è già integrato con un sistema di fatturazione e consegna. Per creare il nuovo sistema di acquisti, il sistema ereditato viene integrato con una piattaforma di commercio elettronico basata su Web e con un sistema di posta elettronica che gestisce le comunicazioni con gli utenti. La struttura del sistema finale risultante è illustrata nella Figura 15.14.

Il nuovo sistema di acquisti è un sistema client-server, con un browser standard e un sistema di posta elettronica sul client. Sul server la piattaforma di commercio elettronico deve essere integrata con il sistema di ordinazioni esistente tramite un adattatore. Il sistema e-commerce ha un suo formato per gli ordini, per le conferme delle consegne e così via; tutto questo deve essere convertito nel formato utilizzato dal nuovo sistema di acquisti. Il sistema e-commerce usa il sistema di posta elettronica per inviare notifiche agli utenti, ma il sistema di ordinazioni non è mai stato progettato per questo compito. Quindi, è necessario scrivere un altro adattatore per convertire le notifiche del sistema di ordinazioni in messaggi di posta elettronica.

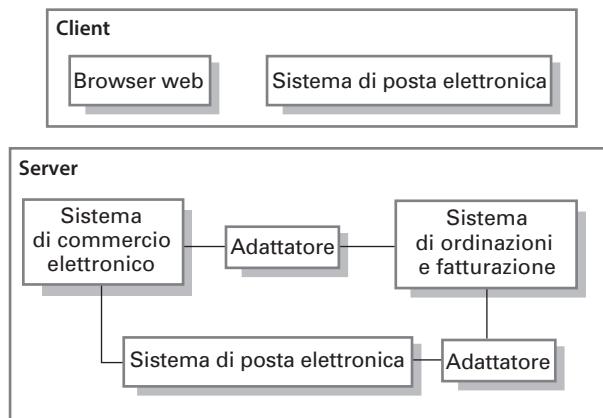


Figura 15.14 Un sistema di acquisti integrato.

Mesi e a volte anni di lavoro di implementazione possono essere salvati, e il tempo per sviluppare e installare un sistema può essere drasticamente ridotto, integrando i sistemi applicativi esistenti. Il sistema di acquisti prima descritto è stato implementato e installato in una grande azienda in nove mesi. Originariamente, si era stimato che sarebbero stati necessari tre anni per sviluppare un sistema di acquisti in Java integrandolo con un sistema di ordinazioni ereditato.

L'integrazione dei sistemi applicativi può essere semplificata se si adotta un approccio orientato ai servizi. Essenzialmente, per approccio orientato ai servizi s'intende consentire l'accesso alle funzionalità del sistema applicativo tramite un'interfaccia di servizi standard, con un servizio per ciascuna unità discreta di funzionalità. Alcune applicazioni possono offrire un'interfaccia di servizi, ma a volte questa interfaccia deve essere implementata dall'integratore dei sistemi. Sostanzialmente, occorre programmare un “wrapper” che nasconde l'applicazione e fornisce i servizi visibili esternamente (Figura 15.15). Questo approccio è particolarmente valido per i sistemi ereditati che devono essere integrati con i nuovi sistemi applicativi.

In teoria, integrare i sistemi applicativi è come utilizzare un qualsiasi altro componente. Si devono capire le interfacce del sistema e utilizzarle esclusivamente per comunicare con il software; occorre trovare un compromesso tra requisiti specifici e processi rapidi di sviluppo e riutilizzo; si deve progettare un'architettura del sistema che permetta ai sistemi applicativi di operare insieme.

Tuttavia, il fatto che questi prodotti in genere siano grandi sistemi, spesso venduti come sistemi indipendenti, introduce ulteriori problemi. Boehm e Abts (Boehm e Abts 1999) individuano quattro problemi importanti dell'integrazione dei sistemi.

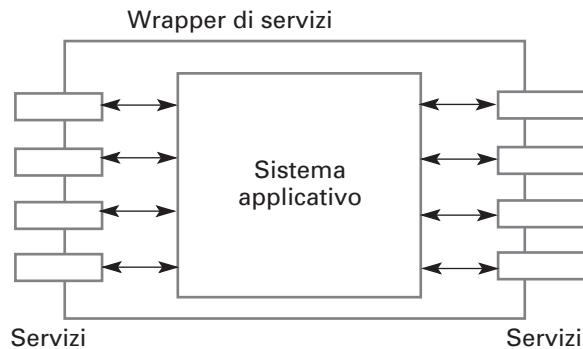


Figura 15.15 Wrapper delle applicazioni.

1. *Mancanza di controllo sulle funzionalità e le prestazioni.* Anche se l'interfaccia pubblica di un prodotto sembra offrire le funzionalità richieste, il sistema potrebbe non essere stato implementato adeguatamente o avere scarse prestazioni. Il prodotto può avere operazioni nascoste che interferiscono con il suo utilizzo in una situazione specifica. Correggere questi problemi potrebbe essere una priorità per l'integratore dei sistemi, ma potrebbe non essere un problema reale per il produttore. Gli utenti possono trovare il modo di aggirare i problemi, se vogliono riutilizzare il sistema applicativo.
2. *Problemi con l'interoperabilità dei sistemi.* A volte è difficile far lavorare insieme singoli sistemi applicativi, poiché ogni sistema incorpora le proprie ipotesi su come sarà utilizzato. Garlan e altri (Garlan et al. 1994) hanno documentato la loro esperienza sul tentativo di integrare quattro sistemi applicativi: tre di questi sistemi erano basati su eventi, ma ciascuno di essi utilizzava un diverso modello di eventi e supponeva di avere l'accesso esclusivo alla coda degli eventi. Di conseguenza, il processo di integrazione era molto difficile. Il progetto richiese cinque volte lo sforzo previsto inizialmente e neanche la tempistica fu rispettata poiché furono necessari due anni, anziché i sei mesi previsti.

Dieci anni dopo, in un'analisi retrospettiva del loro lavoro, Garlan e altri (Garlan, Allen e Ockerbloom 2009) conclusero che i problemi di integrazione che avevano scoperto non erano stati risolti. Torchiano e Morisio (Torchiano e Morisio 2004) hanno scoperto che la mancanza di conformità agli standard in molti sistemi applicativi rende più difficile del previsto il processo di integrazione.

3. *Nessun controllo sull'evoluzione del sistema.* I distributori di sistemi applicativi decidono le modifiche del sistema in risposta a pressioni del mercato. Per i prodotti per PC in particolare, vengono realizzate spesso nuove versio-

ni, che possono non essere compatibili con tutte le versioni precedenti. Le versioni più nuove possono avere funzionalità aggiuntive non richieste, e le versioni precedenti potrebbero diventare indisponibili o perdere il supporto tecnico.

4. *Supporto dai distributori dei sistemi applicativi.* Il livello di supporto fornito dai distributori di sistemi applicativi varia molto. Il supporto da parte dei distributori è particolarmente importante quando sorgono problemi, poiché gli sviluppatori non hanno accesso al codice sorgente e alla documentazione dettagliata del sistema. Sebbene i distributori possano impegnarsi a fornire il supporto, tuttavia il mercato in continua evoluzione e le circostanze economiche potrebbero rendere difficile il rispetto dell'impegno assunto dai distributori. Per esempio, un distributore di sistemi potrebbe decidere di abbandonare un prodotto a causa della riduzione della domanda, oppure al suo posto potrebbe subentrare un'azienda che non vuole più supportare i prodotti che hanno acquisito.

Boehm e Abts sostengono che, in molti casi, il costo della manutenzione e dell'evoluzione di un sistema può essere maggiore per i sistemi applicativi integrati. Tutte le difficoltà sopraindicate sono problemi che interessano tutto il ciclo di vita, non soltanto lo sviluppo iniziale del sistema. Quanto più le persone coinvolte nella manutenzione del sistema si allontanano dagli sviluppatori del sistema originale, tanto più è probabile che sorgano difficoltà con il sistema integrato.

Punti chiave

- Ci sono tanti modi di riutilizzare il software, dal semplice riutilizzo delle classi e dei metodi delle librerie al riutilizzo di interi sistemi applicativi.
- I vantaggi del riutilizzo del software sono: costi minori, sviluppo del software più rapido e rischi minori. La fidatezza del sistema è superiore. Gli specialisti operano in modo più efficace, perché possono focalizzare la loro esperienza sulla progettazione dei componenti riutilizzabili.
- I framework applicativi sono collezioni di oggetti concreti e astratti che sono progettati per essere riutilizzati tramite la specializzazione e l'aggiunta di nuovi oggetti. Di solito, incorporano buone pratiche di progettazione utilizzando schemi di progettazione.
- Le linee di prodotti software sono correlate alle applicazioni che vengono sviluppate da una o più applicazioni di base. Un generico sistema viene adattato e specializzato per soddisfare determinati requisiti di funzionalità, piattaforme o configurazioni operative.
- Il riutilizzo di un sistema applicativo implica il riutilizzo di sistemi off-the-shelf su larga scala. Questi sistemi offrono varie funzionalità, e il loro riutilizzo può ridurre

notevolmente i costi e i tempi di sviluppo. I sistemi possono essere sviluppati configurando un singolo sistema applicativo oppure integrando due o più sistemi applicativi.

- I problemi potenziali connessi al riutilizzo dei sistemi applicativi includono: la mancanza di controllo su funzionalità, prestazioni ed evoluzione dei sistemi; la necessità di supporto da parte dei distributori esterni; difficoltà nel garantire che i sistemi possano operare congiuntamente.

Esercizi

- 15.1 Quali sono i principali fattori tecnici e non tecnici che ostacolano il riutilizzo del software? Siete soliti riutilizzare il software e, se no, perché no?
- * 15.2 Spiegate perché il risparmio dei costi derivante dal riutilizzo del software esistente non è semplicemente proporzionale alla dimensione dei componenti che vengono riutilizzati.
- * 15.3 Indicate quattro casi in cui non consigliereste il riutilizzo del software.
- 15.4 Spiegate che cosa s'intende con "inversione di controllo" nei framework applicativi. Spiegate perché questo approccio potrebbe causare problemi se si integrano due sistemi distinti che erano stati creati originariamente utilizzando lo stesso framework applicativo.
- * 15.5 Utilizzando l'esempio del sistema delle stazioni meteorologiche descritto nei Capitoli 1 e 7, suggerite l'architettura di una linea di prodotti software per una famiglia di applicazioni che riguardano il monitoraggio e la raccolta di dati a distanza. Dovreste presentare la vostra architettura come un modello a strati, che rappresenta i componenti che potrebbero essere inclusi in ciascun livello.
- 15.6 Il software per desktop, come quello per l'elaborazione dei testi, può essere configurato in vari modi. Esaminate il software che utilizzate regolarmente ed elencate le sue opzioni di configurazione. Indicate le difficoltà che potrebbero incontrare gli utenti nel configurare questo software. Microsoft Office (o una delle sue alternative open-source) è un buon esempio da utilizzare per questo esercizio.
- 15.7 Perché molte grandi società hanno scelto i sistemi ERP come base per i loro sistemi informatici? Quali problemi potrebbero sorgere durante lo sviluppo di un sistema ERP su larga scala in una organizzazione?
- * 15.8 Identificate sei possibili rischi che possono presentarsi quando i sistemi vengono costruiti utilizzando sistemi applicativi esistenti. Quali rimedi può adottare una società per ridurre questi rischi?
- * 15.9 Spiegate perché di solito è necessario utilizzare degli adattatori quando i sistemi vengono costruiti integrando i sistemi applicativi. Indicate tre problemi pratici che potrebbero presentarsi durante la scrittura del software degli adattatori per collegare due sistemi applicativi.

- 15.10 Il riutilizzo del software genera alcuni problemi di diritti di proprietà intellettuale e copyright. Se un cliente paga una società per sviluppare un sistema software, chi ha il diritto di riutilizzare il codice sviluppato? La società che ha sviluppato il sistema ha il diritto di usare questo codice come base di un generico componente? Quali meccanismi di pagamento potrebbero essere utilizzati per rimborcare i fornitori di componenti riutilizzabili? Analizzate questi problemi e altri temi etici associati al riutilizzo del software.

* *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

“Overlooked Aspects of COTS-Based Development.” Un interessante articolo che illustra i risultati di una ricerca di alcuni sviluppatori che hanno adottato un approccio basato sui prodotti COTS, e i problemi che hanno incontrato. (M. Torchiano e M. Morisio, *IEEE Software*, 21 (2), March-April 2004) <http://dx.doi.org/10.1109/MS.2004.1270770>

CRUISE – Component Reuse in Software Engineering. Questo e-book tratta numerosi problemi del riutilizzo, inclusi i casi di studio, il riutilizzo basato sui componenti e i processi del riutilizzo. Il riutilizzo dei sistemi applicativi è trattato in modo limitato. (L. Nascimento et al. 2007) http://www.academia.edu/179616/C.R.U.I.S.E._-_Component_Reuse_in_Software_Engineering

“Construction by Configuration: A New Challenge for Software Engineering.” In questo documento, descrivo i problemi e le difficoltà nel costruire una nuova applicazione configurando i sistemi esistenti. (I. Sommerville, *Proc. 19th Australian Software Engineering Conference*, 2008) <http://dx.doi.org/10.1109/ASWEC.2008.75>

“Architectural Mismatch: Why Reuse Is Still So Hard.” Questo articolo analizza un documento precedente che trattava i problemi del riutilizzo e dell'integrazione di un certo numero di sistemi applicativi . Gli autori concludono che, sebbene siano stati fatti alcuni progressi, c'erano sono ancora problemi con le ipotesi in conflitto fatte dai progettisti dei singoli sistemi. (D. Garlan et al., *IEEE Software*, 26 (4), July-August 2009) <http://dx.doi.org/10.1109/MS.2009.86>

CAPITOLO

16

Ingegneria del software basato sui componenti

Questo capitolo si propone di descrivere un approccio al riutilizzo del software basato sulla composizione di componenti standardizzati riutilizzabili. Dopo aver letto questo capitolo:

- saprete che cos'è un componente software che può essere incluso in un programma come elemento eseguibile;
- conoscerete gli elementi chiave dei modelli dei componenti software e il supporto fornito dal middleware per questi modelli;
- apprenderete le attività chiave nel processo di ingegneria del software basato sui componenti (CBSE, component-based software engineering) per il riutilizzo del software;
- conoscerete i tre tipi di composizione dei componenti e alcuni dei problemi che devono essere risolti quando i componenti vengono composti per creare nuovi componenti o sistemi.

16.1 Componenti e modelli di componenti

16.2 Processi CBSE

16.3 Composizione dei componenti

L’ingegneria del software basato sui componenti (CBSE, component-based software engineering) emerse alla fine degli anni ’90 come un approccio allo sviluppo dei sistemi software basato sul riutilizzo dei componenti software. La sua nascita fu motivata dalla frustrazione che lo sviluppo orientato agli oggetti non aveva portato al riutilizzo estensivo del software, come originariamente previsto. Le classi di singoli oggetti erano troppo dettagliate e specifiche, e spesso dovevano essere legate a un’applicazione durante la compilazione. Occorreva una conoscenza dettagliata delle classi per poterle utilizzare, e questo di solito significava che era necessario disporre del codice sorgente dei componenti. La vendita o la distribuzione degli oggetti come singoli componenti riutilizzabili era quindi impossibile.

I componenti sono astrazioni di livello più elevato degli oggetti e sono definiti dalle loro interfacce. Di solito, sono più grandi dei singoli oggetti, e tutti i dettagli di implementazione sono nascosti agli altri componenti. L’ingegneria del software basato sui componenti è il processo che definisce, implementa e integra questi componenti indipendenti, debolmente accoppiati.

CBSE è diventato un importante approccio allo sviluppo del software per sistemi aziendali su larga scala, in seguito alle crescenti richieste di prestazioni e sicurezza del software. I clienti richiedono un software protetto e affidabile con tempi di consegna e installazione sempre più rapidi. L’unico modo per soddisfare queste richieste consiste nel realizzare il software riutilizzando componenti esistenti.

Gli elementi essenziali dell’ingegneria del software basato sui componenti sono diversi.

1. *Componenti indipendenti*: sono specificati completamente dalle loro interfacce. Dovrebbe esserci una separazione netta tra l’interfaccia dei componenti e la sua implementazione, in modo che l’implementazione di un componente possa essere sostituita da un’altra, senza dover modificare altre parti del sistema.
2. *Standard dei componenti*: definiscono le interfacce e facilitano l’integrazione dei componenti. Questi standard sono incorporati in un modello di componenti; definiscono come dovrebbero essere specificate le interfacce dei componenti e come i componenti comunicano tra loro. Alcuni modelli vanno oltre, definendo le interfacce che dovrebbero essere implementate da tutti i componenti. Se i componenti sono conformi agli standard, allora la loro operatività è indipendente dal linguaggio di programmazione. I componenti scritti in linguaggi differenti possono essere integrati nello stesso sistema.
3. *Middleware*: fornisce un supporto software all’integrazione dei componenti. Per far sì che componenti indipendenti e distribuiti lavorino assieme, occorre un supporto middleware che gestisca la comunicazione tra i com-

Problemi con CBSE

CBSE è oggi un approccio predominante nell'ingegneria del software ed è largamente utilizzato per creare nuovi sistemi. Tuttavia, se utilizzato come approccio al riutilizzo del software, si presentano alcuni problemi, quali l'affidabilità dei componenti, la loro certificazione, compromessi sui requisiti e la previsione delle proprietà dei componenti, specialmente quando questi vengono integrati con altri componenti.

<http://software-engineering-book.com/web/cbse-problems/>

ponenti. Il middleware per il supporto dei componenti gestisce efficacemente i problemi di basso livello e consente agli sviluppatori di concentrarsi sui problemi relativi all'applicazione; questo middleware, inoltre, può fornire un supporto all'allocazione delle risorse, alla gestione delle transazioni, alla protezione e alla simultaneità.

4. *Un processo di sviluppo* che sia adattato all'ingegneria del software basato sui componenti. Occorre un processo di sviluppo che consente l'evoluzione dei requisiti, in base alle funzionalità dei componenti disponibili.

Lo sviluppo basato sui componenti incorpora le buone pratiche di ingegneria del software. Spesso ha senso progettare un sistema utilizzando componenti, anche se si devono sviluppare, anziché riutilizzare, questi componenti. L'ingegneria del software basato sui componenti include i sani principi di progettazione che supportano la realizzazione di software comprensibile e manutenibile.

1. Ogni componente è indipendente, quindi non interferisce con le operazioni di un altro componente. I dettagli dell'implementazione sono nascosti. L'implementazione di un componente può essere modificata senza influire sulle parti restanti del sistema.
2. I componenti comunicano attraverso interfacce ben definite. Se queste interfacce sono mantenute, un componente può essere sostituito da un altro che fornisce nuove o migliori funzionalità.
3. Le infrastrutture dei componenti forniscono una vasta gamma di servizi standard che possono essere utilizzati nei sistemi applicativi; questo riduce la quantità di nuovo codice da sviluppare.

La motivazione iniziale dell'ingegneria del software basato sui componenti era la necessità di fornire un supporto sia al riutilizzo del software sia all'ingegneria del software distribuito. Un componente era visto come un elemento di un sistema software cui potevano accedere, tramite un meccanismo remoto di chiamate di procedure, altri componenti che erano eseguiti su computer separati. Ogni sistema che riutilizzava un componente doveva incorporare la sua copia di questo componente. Questa idea di componente estendeva il concetto di oggetti distribuiti, così come erano stati definiti nei modelli dei sistemi distribuiti, come ad esempio la

specifica CORBA (Pope 1997). Furono introdotti molti protocolli e “standard” specifici per tecnologie per supportare questo concetto di componente, inclusi Enterprise Java Beans (EJB) di Sun, COM e .NET di Microsoft, e CCM di CORBA (Lau e Wang 2007).

Sfortunatamente, le società coinvolte nel proporre standard non si accordarono su un singolo standard per i componenti, limitando così l’impatto di questo approccio sul riutilizzo del software. È impossibile per i componenti sviluppati che usano approcci differenti operare insieme. I componenti che sono stati sviluppati per piattaforme differenti, come .NET o J2EE, non possono cooperare. Inoltre, gli standard e i protocolli proposti erano complessi e difficili da capire. Questo fu un altro ostacolo alla loro adozione.

Come risposta a questi problemi, fu sviluppato il concetto di componente come servizio, e furono proposti nuovi standard a supporto dell’ingegneria del software orientata ai servizi. La differenza più significativa tra un componente come servizio e il concetto originale di componente è che i servizi sono entità autonome esterne al programma che li usa. Quando si costruisce un sistema orientato ai servizi, si fa riferimento al servizio esterno, anziché includere una copia del servizio nel sistema.

L’ingegneria del software orientata ai servizi è un tipo di ingegneria del software basato sui componenti. Essa usa un concetto di componente più semplice di quello originariamente proposto da CBSE, dove i componenti erano routine eseguibili che erano incluse in sistemi più grandi. Ogni sistema che utilizzava un componente incorporava la sua versione di questo componente. Gli approcci orientati ai servizi stanno gradualmente rimpiazzando CBSE con i componenti incorporati come un approccio allo sviluppo dei sistemi. Questo capitolo descrive l’uso di CBSE con i componenti incorporati; il Capitolo 18 tratta l’ingegneria del software orientata ai servizi.

16.1 Componenti e modelli di componenti

La comunità di riutilizzo del software concorda nel definire un componente come un’unità software indipendente che può essere composta con altri componenti per creare un sistema software. Oltre a questa, però, sono state proposte altre definizioni. Councill e Heineman (Councill e Heineman 2001) definiscono un componente software come:

Un elemento software è conforme a un modello di componenti standard, può essere consegnato singolarmente e composto senza modifiche secondo uno standard di composizione.¹

¹ Councill W. T. e G. T. Heineman 2001. “Definition of a Software Component and Its Elements.” In Component-Based Software Engineering, a cura di G. T. Heineman e W. T. Councill, 5-20. Boston: Addison Wesley.

Questa definizione si basa essenzialmente sugli standard: un'unità software che è conforme a tali standard è un componente. Szyperski (Szyperski 2002), tuttavia, non cita gli standard nella sua definizione di componente, ma si concentra invece sulle caratteristiche chiave dei componenti:

Un componente software è un'unità di composizione soltanto con interfacce definite per contratto e dipendenze esplicite di contesto. Un componente software può essere consegnato singolarmente ed è soggetto a composizione da terze parti.²

Entrambe queste definizioni furono sviluppate attorno all'idea che un componente fosse un elemento incorporato in un sistema, anziché un servizio che viene richiamato dal sistema. Tuttavia, queste definizioni sono ugualmente applicabili ai componenti di servizi.

Szyperski dichiara anche che un componente non ha uno stato osservabile esternamente, ovvero che le copie dei componenti sono indistinguibili. Tuttavia, alcuni modelli di componenti, come il modello Enterprise Java Beans, ammettono i componenti *stateful* (con stato), che quindi non corrispondono alla definizione di Szyperski. Per quanto i componenti *stateless* (senza stato) siano più semplici da utilizzare, tuttavia in alcuni sistemi i componenti stateful sono più convenienti e riducono la complessità dei sistemi.

Queste definizioni concordano su un punto: i componenti sono indipendenti e sono l'unità fondamentale di composizione di un sistema. Se combiniamo queste due proposte, otteniamo una descrizione più completa di componente riutilizzabile. La Figura 16.1 mostra quelle che io considero le caratteristiche essenziali di un componente come viene utilizzato nell'ingegneria del software basato sui componenti.

Un componente può essere immaginato come un fornitore di uno o più servizi, anche se il componente è incorporato nel sistema, anziché essere implementato come servizio. Quando un sistema necessita di alcuni servizi, chiama il componente che li fornisce, senza bisogno di sapere dove il componente viene eseguito o quale linguaggio di programmazione è stato utilizzato per svilupperlo. Per esempio un componente in un sistema bibliotecario può fornire un servizio di ricerca che permette agli utenti di cercare un libro in diversi cataloghi della biblioteca. Un componente che converte un'immagine da un formato grafico a un altro (per esempio, da TIFF a JPEG) può fornire un servizio di conversione dati e così via.

Immaginare un componente come fornitore di servizi permette di enfatizzare due caratteristiche critiche dei componenti riutilizzabili.

1. Il componente è un'entità eseguibile indipendente, che è definita dalle sue interfacce. Non occorre conoscere il suo codice sorgente per utilizzarlo. Il

² Szyperski C. 2002. *Component Software: Beyond Object-Oriented Programming*, 2nd Ed. Harlow, UK: Addison Wesley.

Caratteristica del componente	Descrizione
Componibile	Perché un componente sia componibile, tutte le interazioni esterne devono avvenire attraverso interfacce pubbliche. Inoltre, il componente deve fornire accesso esterno alle sue informazioni, come i suoi metodi e i suoi attributi.
Consegnabile	Per essere consegnabile, un componente deve essere autonomo e in grado di operare come entità indipendente su una piattaforma che implementa il suo modello. Questo solitamente significa che il componente è binario e non deve essere compilato prima della consegna. Se un componente è implementato come servizio, non occorre che sia consegnato da un utente di quel componente, ma viene consegnato dal fornitore dei servizi.
Documentato	I componenti devono essere documentati interamente in modo che i potenziali utenti possano decidere se il componente soddisfa o no le loro necessità. Deve essere specificata la sintassi e, in teoria, la semantica di tutte le interfacce dei componenti.
Indipendente	Un componente dovrebbe essere indipendente – dovrebbe essere possibile comporlo e consegnarlo senza dover utilizzare altri componenti specifici. Qualora i componenti avessero bisogno di servizi forniti esternamente, questi dovrebbero essere esplicitamente indicati in una specifica delle interfacce.
Standardizzato	La standardizzazione dei componenti implica che un componente utilizzato in un processo CBSE deve essere conforme a un modello di componenti standardizzato. Questo modello può definire interfacce, metadata, documentazione, composizione e consegna dei componenti.

Figura 16.1 Caratteristiche dei componenti.

componente può essere chiamato come servizio esterno o incluso direttamente in un programma.

2. I servizi offerti da un componente sono resi disponibili tramite un’interfaccia, attraverso la quale passano tutte le interazioni. L’interfaccia del componente è espressa in termini di operazioni parametrizzate e il suo stato interno non viene mai mostrato.

In teoria, tutti i componenti hanno due interfacce correlate, come mostra la Figura 16.2. Queste interfacce riflettono i servizi forniti dal componente e i servizi richiesti dal componente per operare correttamente.

1. L’interfaccia dei “servizi forniti” definisce i servizi forniti dal componente. L’interfaccia è l’API del componente; definisce i metodi che possono essere chiamati da un utente del componente. In diagramma UML, questa interfaccia per un componente è indicata da un cerchio alla fine di una linea che parte dall’icona del componente.

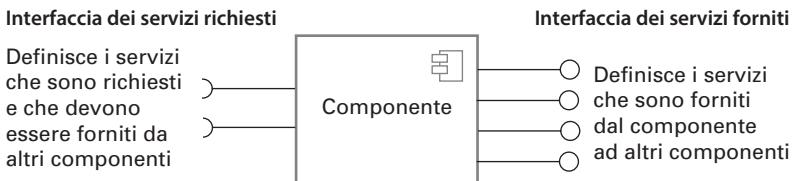


Figura 16.2 Interfacce di un componente.

2. L'interfaccia dei "servizi richiesti" specifica i servizi che altri componenti del sistema devono fornire per far sì che il componente operi correttamente. Se questi servizi non sono disponibili, il componente non potrà funzionare. Questo non compromette l'indipendenza o la consegna del componente, perché l'interfaccia dei servizi richiesti non definisce come questi servizi devono essere forniti. In un diagramma UML, questa interfaccia è indicata da un semicerchio alla fine di una linea che parte dall'icona del componente. Si noti che le icone delle interfacce dei servizi forniti e richiesti si possono collegare insieme come una presa e una spina elettrica.

Per illustrare queste interfacce, la Figura 16.3 mostra un modello di un componente che è stato progettato per raccogliere e ordinare le informazioni provenienti da un insieme di sensori. Il componente opera autonomamente per raccogliere i dati in un periodo di tempo e, se richiesto, fornisce i dati raccolti al componente che li richiede. L'interfaccia dei "servizi forniti" include i metodi per aggiungere, rimuovere, avviare, fermare e testare i sensori; include anche il metodo `report` che fornisce i dati raccolti dai sensori e il metodo `listAll` che fornisce informazioni sui sensori collegati. Sebbene non siano qui mostrati, questi metodi hanno parametri associati che specificano gli identificatori, le posizioni dei sensori e così via.

L'interfaccia dei "servizi richiesti" è utilizzata per collegare i componenti ai sensori. Si presume che i sensori abbiano un'interfaccia dati, accessibile tramite `sensorData`, e un'interfaccia di gestione, accessibile tramite `sensorManagement`. Questa interfaccia è stata progettata per collegare vari tipi di sensori, quindi non

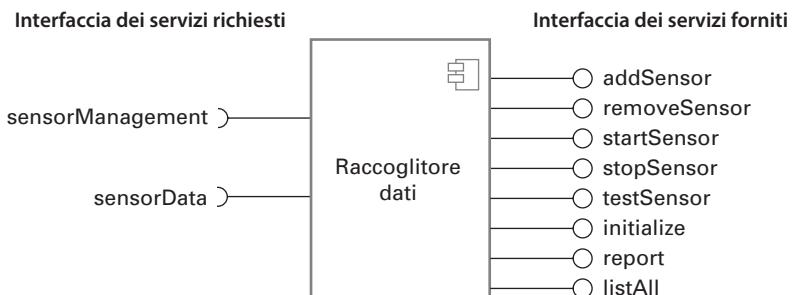


Figura 16.3 Modello di un componente per la raccolta di dati.

Componenti e oggetti

I componenti sono spesso implementati nei linguaggi orientati agli oggetti e, in alcuni casi, l'accesso all'interfaccia dei “servizi forniti” di un componente avviene tramite chiamate di metodi. Tuttavia, componenti e classi di oggetti non sono la stessa cosa. Diversamente dalle classi di oggetti, i componenti sono installabili in modo indipendente, non definiscono tipi, sono indipendenti dai linguaggi e si basano su un modello di componenti standard.

<http://software-engineering-book.com/web/components-and-objects/>

include operazioni specifiche dei sensori, come `Test` e `provideReading`. Piuttosto, i comandi utilizzati da un particolare tipo di sensore vengono incorporati in una stringa, che è un parametro per le operazioni nell'interfaccia dei “servizi richiesti”. I componenti adattatori analizzano questa stringa di parametri e traducono i comandi incorporati nell'interfaccia specifica di controllo di ciascun tipo di sensore. Descriverò più avanti nel capitolo l'uso degli adattatori, quando mostrerò come il componente di raccolta dei dati può essere collegato a un sensore (Figura 16.12).

L'accesso ai componenti avviene utilizzando delle chiamate di procedure remote (CPR). Ciascun componente ha un identificatore unico e, tramite questo nome, può essere chiamato da un altro computer. Il componente chiamato usa lo stesso meccanismo per accedere ai componenti dei “servizi richiesti” che sono definiti nella sua interfaccia.

Una differenza importante tra un componente come servizio esterno e un componente come elemento di programma utilizzato tramite una chiamata di procedura remota è che i servizi sono entità completamente indipendenti. Essi non hanno un'esplicita interfaccia di “servizi richiesti”. Ovviamente, non richiedono altri componenti a supporto delle loro operazioni, ma questi sono forniti internamente. Altri programmi possono utilizzare i servizi, senza bisogno di implementare un supporto addizionale richiesto dal servizio.

16.1.1 Modelli di componenti

Un modello di componenti è una definizione di standard per l'implementazione, la documentazione e la consegna dei componenti. Questi standard servono agli sviluppatori per garantire che i componenti possano operare insieme; sono utilizzati anche dai fornitori di infrastrutture per l'esecuzione di componenti che forniscono il middleware a supporto del funzionamento dei componenti. Per i componenti di servizi, il modello di componenti più importante è Web Service; per i componenti incorporati, i modelli più diffusi sono Enterprise Java Beans (EJB) di Sun e il modello .NET di Microsoft (Lau e Wang 2007).

Gli elementi fondamentali di un modello di componenti ideale sono stati descritti da Weinreich e Sametinger (Weinreich e Sametinger 2001) e sono rias-

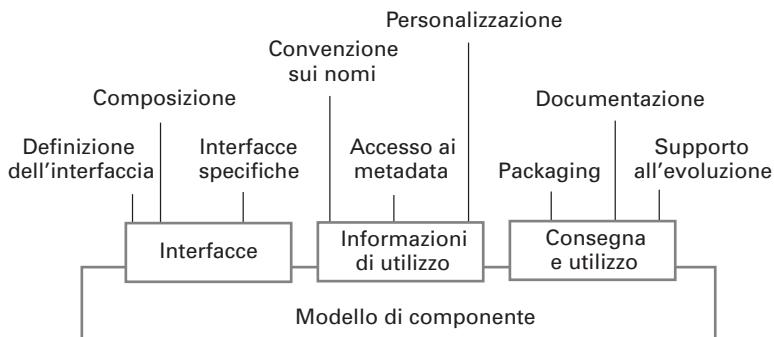


Figura 16.4 Elementi fondamentali di un modello di componenti.

sunti nella Figura 16.4. Il diagramma mostra che gli elementi di un modello definiscono le interfacce del componente, le informazioni che servono per utilizzare il componente in un programma e come un componente deve essere consegnato.

1. *Interfacce.* I componenti sono definiti specificando le loro interfacce. Il modello dei componenti specifica come devono essere definite le interfacce e gli elementi, quali i nomi delle operazioni, i parametri e le eccezioni, che devono essere inclusi nella definizione dell'interfaccia. Il modello dovrebbe anche specificare il linguaggio utilizzato per definire le interfacce dei componenti.

Per i servizi web, la specifica delle interfacce usa i linguaggi basati su XML, di cui si dirà nel Capitolo 18; EJB è un modello specifico di Java, quindi Java è utilizzato come linguaggio di definizione delle interfacce; nei modelli .NET, le interfacce sono definite utilizzando il linguaggio CIL (Common Intermediate Language) di Microsoft. Alcuni modelli di componenti richiedono interfacce specifiche che devono essere definite da un componente; sono utilizzati per comporre il componente con l'infrastruttura del modello, che fornisce i servizi standard, quali la protezione e la gestione delle transazioni.

2. *Utilizzo.* Affinché i componenti possano essere distribuiti e utilizzati in modo remoto tramite le chiamate di procedure remote, devono avere un nome unico o un handle associato; questo deve essere globalmente unico. Per esempio, nel modello EJB, un nome gerarchico viene generato con la radice che si basa sul nome di un dominio di Internet. I servizi hanno un URI (Uniform Resource Identifier) unico.

I metadati di un componente sono informazioni sul componente stesso, come quelle sulle sue interfacce e sui suoi attributi. I metadati sono importanti perché consentono agli utenti del componente di trovare i servizi for-

niti e quelli richiesti dal componente. Le implementazioni dei modelli di componenti di solito includono metodi specifici (come l’uso dell’interfaccia di riflessione in Java) per accedere ai metadati dei componenti.

I componenti sono entità generiche e, quando consegnati, devono essere personalizzati per il loro particolare sistema applicativo. Per esempio, il componente RaccoglitoreDati, mostrato nella Figura 16.3, può essere personalizzato definendo il numero massimo di sensori. Il modello del componente può dunque specificare come i componenti binari possono essere configurati per un particolare ambiente operativo.

3. *Consegna.* Il modello di componenti include una specifica su come i componenti dovrebbero essere impacchettati (packaging) per la consegna come routine eseguibili indipendenti. Poiché i componenti sono entità indipendenti, devono essere impacchettati con tutto il software di supporto che non è fornito dall’infrastruttura dei componenti o che non è definito nell’interfaccia dei servizi “richiesti”. I dati di consegna includono le informazioni sul contenuto di un package e la sua organizzazione binaria.

Inevitabilmente, in seguito alla richiesta di nuovi requisiti, i componenti devono essere modificati o sostituiti. Il modello di componenti deve dunque includere le regole che stabiliscono quando e come è possibile sostituire i componenti. Infine, il modello potrebbe definire la documentazione del componente da produrre, che viene utilizzata per trovare il componente e decidere se è appropriato.

Per i componenti che sono routine eseguibili, anziché servizi esterni, il modello definisce i servizi forniti dal middleware che supporta l’esecuzione dei componenti. Weinreich e Sametinger utilizzano l’analogia di un sistema operativo per spiegare i modelli di componenti. Un sistema operativo fornisce un insieme di servizi generici che possono essere utilizzati dalle applicazioni; l’implementazione di un modello di componenti fornisce ai componenti analoghi servizi condivisi. La Figura 16.5 mostra alcuni dei servizi che possono essere forniti dall’implementazione di un modello di componenti.

I servizi forniti dall’implementazione di un modello di componenti possono essere classificati in due categorie.

1. *Servizi di piattaforma.* Permettono ai componenti di comunicare e interagire in un ambiente distribuito. Sono servizi fondamentali che devono essere disponibili in un tutti i sistemi basati sui componenti.
2. *Servizi di supporto.* Sono servizi comuni che potrebbero essere richiesti da più componenti. Per esempio, molti componenti richiedono l’autenticazione per garantire che gli utenti siano autorizzati a usufruire dei servizi dei componenti. Ha senso mettere a disposizione di tutti i componenti un insieme di servizi middleware standard. In questo modo, si riducono i costi di sviluppo dei componenti e si evita il rischio di incompatibilità tra i componenti.

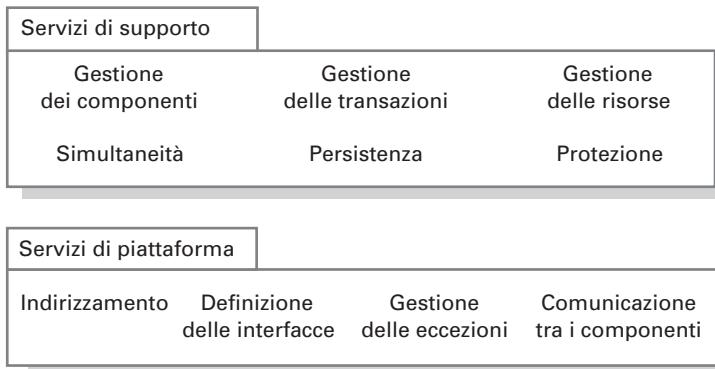


Figura 16.5 I servizi middleware definiti in un modello di componenti.

Il middleware implementa i servizi comuni dei componenti e fornisce le interfacce per tali servizi. Per utilizzare i servizi forniti dall’infrastruttura di un modello di componenti, si può pensare che i componenti siano consegnati in un “contenitore”. Un contenitore è un’implementazione dei servizi di supporto più una definizione delle interfacce che un componente deve fornire per integrarsi con il contenitore. Quando sono utilizzate, le interfacce del componente non sono accessibili direttamente da altri componenti. L’accesso a queste interfacce avviene tramite un’apposita interfaccia del contenitore che chiama il codice per accedere all’interfaccia del componente incorporato.

I contenitori sono grandi e complessi e, quando viene consegnato un componente in un contenitore, si ottiene l’accesso a tutti i servizi middleware. Tuttavia, i componenti semplici potrebbero non aver bisogno di tutte le funzionalità offerte dal middleware di supporto. L’approccio adottato nei servizi web riguardo alla fornitura dei servizi comuni è quindi molto diverso. Per i servizi web, gli standard sono stati definiti per i servizi comuni, quali la protezione e la gestione delle transazioni, e questi standard sono stati implementati come librerie di programmi. Se state implementando un componente di servizi, utilizzate soltanto i servizi che servono.

I servizi associati a un modello di componenti hanno molto in comune con le funzionalità offerte dai framework orientati agli oggetti, descritti nel Capitolo 15. Sebbene non siano completi, tuttavia i servizi dei framework sono spesso più efficienti di quelli basati sui contenitori. Di conseguenza, alcune persone ritengono che sia meglio utilizzare un framework come SPRING (Wheeler e White 2013) per lo sviluppo Java, anziché un modello di componenti ricco di funzionalità come EJB.

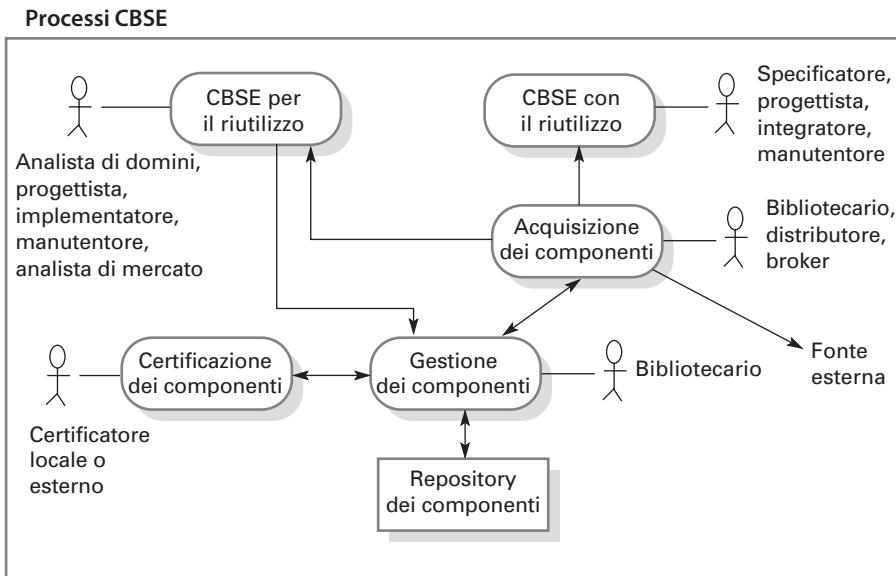


Figura 16.6 Processi CBSE.

16.2 Processi CBSE

I processi CBSE sono processi software che supportano l'ingegneria del software basato sui componenti; tengono conto della possibilità di riutilizzo e delle diverse attività dei processi coinvolti nello sviluppo e nell'uso di componenti riutilizzabili. La Figura 16.6 (Kotonya 2003) presenta una panoramica dei processi CBSE. Al livello più alto, ci sono due tipi di processi CBSE.

1. *Sviluppo per il riutilizzo.* Questo processo riguarda lo sviluppo di componenti o servizi che saranno riutilizzati in altre applicazioni; di solito consiste nella generalizzazione di componenti esistenti.
2. *Sviluppo con il riutilizzo.* È il processo che sviluppa nuove applicazioni utilizzando componenti e servizi esistenti.

Questi processi hanno obiettivi differenti e quindi includono attività diverse. Nello sviluppo per il riutilizzo, l'obiettivo è produrre uno o più componenti riutilizzabili. Conosciamo i componenti con i quali stiamo operando, e abbiamo accesso al loro codice sorgente per generalizzarli. Nello sviluppo con il riutilizzo, non conosciamo quali componenti sono disponibili, quindi dobbiamo scoprire questi componenti e progettare il nostro sistema per utilizzarli nel modo più efficiente; il codice sorgente dei componenti potrebbe non essere disponibile.

Come si può notare dalla Figura 16.6, i processi CBSE di base con e per il riutilizzo hanno processi di supporto che riguardano l'acquisizione, la gestione e la certificazione dei componenti.

1. *Acquisizione dei componenti.* È il processo di acquisizione dei componenti per il riutilizzo o lo sviluppo in un componente riutilizzabile. Può riguardare l'accesso a componenti o servizi sviluppati localmente o la ricerca di questi componenti in una fonte esterna.
2. *Gestione dei componenti.* Riguarda la gestione di componenti riutilizzabili di una società, per garantire che essi siano appropriatamente catalogati, immagazzinati e disponibili al riutilizzo.
3. *Certificazione dei componenti.* È il processo che controlla i componenti e certifica che essi soddisfano le loro specifiche.

I componenti gestiti da una società possono essere immagazzinati in un repository che include sia i componenti sia le informazioni sul loro utilizzo.

16.2.1 CBSE per il riutilizzo

CBSE per il riutilizzo è il processo che sviluppa componenti riutilizzabili e li rende disponibili al riutilizzo attraverso un sistema di gestione dei componenti. I primi sostenitori dell'ingegneria del software basato sui componenti (Szyperski 2002) prevedevano che si sarebbe sviluppato un mercato fiorente di componenti, che ci sarebbero stati fornitori e distributori specializzati di componenti che avrebbero organizzato la vendita di componenti prodotti da vari sviluppatori. Gli sviluppatori di software avrebbero comprato i componenti da includere in un sistema o avrebbero pagato l'uso dei servizi offerti dai componenti. Questa previsione non si è mai avverata. Ci sono pochi fornitori di componenti, ed è poco diffuso l'acquisto di componenti off-the-shelf.

Di conseguenza, il processo CBSE per il riutilizzo è adottato principalmente all'interno di quelle organizzazioni che hanno scelto di sviluppare il software secondo i principi dell'ingegneria del software guidata dal riutilizzo. Queste società hanno una base di componenti sviluppati al loro interno che possono essere riutilizzati. Il riutilizzo di questi componenti, tuttavia, potrebbe richiedere qualche modifica. Spesso essi includono caratteristiche e interfacce specifiche per determinate applicazioni che difficilmente potrebbero essere richieste da altri programmi dove i componenti dovrebbero essere riutilizzati.

Per rendere riutilizzabili i componenti, occorre adattare ed estendere i componenti specifici per determinate applicazioni in modo da ottenere versioni più generiche e quindi più facilmente riutilizzabili. Ovviamente, questo adattamento ha dei costi associati. Occorre stabilire, in primo luogo, se un componente avrà buone probabilità di essere riutilizzato e, in secondo luogo, se la riduzione dei costi derivante dal futuro riutilizzo giustifica i costi da sostenere per rendere il componente riutilizzabile.

Per rispondere alla prima di queste domande, si deve decidere se un componente implementa una o più astrazioni stabili di dominio. Le astrazioni stabili di dominio sono elementi fondamentali di un dominio di applicazioni che cambiano lentamente. Per esempio, in un sistema bancario, le astrazioni di dominio possono

includere i conti correnti, gli intestatari dei conti e gli estratti conto; in un sistema di gestione di un ospedale, le astrazioni di dominio possono includere i pazienti, le cure e gli infermieri. Queste astrazioni di dominio sono a volte chiamate “oggetti aziendali”. Se un componente è un’implementazione di un’astrazione di dominio comunemente utilizzata o di un gruppo di oggetti aziendali correlati, è molto probabile che esso venga riutilizzato.

Per rispondere alla domanda sulla convenienza economica, si devono valutare i costi delle modifiche richieste per rendere il componente riutilizzabile. Questi sono i costi per la documentazione e la convalida del componente, e quelli per rendere il componente più generico. Le modifiche per rendere un componente più riutilizzabile includono:

- eliminazione di metodi specifici per un’applicazione;
- modifica dei nomi per renderli più generici;
- aggiunta di metodi per fornire una gamma di funzioni più completa;
- rendere la gestione delle eccezioni compatibile con tutti i metodi;
- aggiunta di un’interfaccia di “configurazione” per permettere al componente di essere adattato a diverse condizioni d’uso;
- integrazione dei componenti richiesti per aumentare l’indipendenza.

Il problema della gestione delle eccezioni è particolarmente difficile. In teoria, i componenti non dovrebbero gestire le eccezioni autonomamente, in quanto ogni applicazione avrà i propri requisiti per la gestione delle eccezioni. Il componente, invece, dovrà definire quali eccezioni possono sorgere e pubblicare queste eccezioni come parte dell’interfaccia. Per esempio, un semplice componente che implementa una struttura a stack di dati, dovrebbe rilevare e pubblicare le eccezioni di overflow e underflow dello stack. In pratica, però, ci sono due problemi in questo processo:

1. pubblicare tutte le eccezioni significa gonfiare le interfacce, che diventerebbero più difficili da capire. Questo potrebbe scoraggiare l’uso del componente da parte di potenziali utenti;
2. il funzionamento del componente potrebbe dipendere dalla gestione locale delle eccezioni; la sua modifica potrebbe avere gravi implicazioni sulle funzionalità del componente.

Pertanto, occorre adottare un approccio pragmatico alla gestione delle eccezioni di un componente. Le tipiche eccezioni tecniche, dove il ripristino è importante per il funzionamento del componente, dovrebbero essere gestite localmente. Queste eccezioni e la loro modalità di gestione dovrebbero essere documentate con il componente. Le altre eccezioni, che sono correlate alla funzione aziendale del componente, dovrebbero essere passate al componente chiamante per essere gestite.

Mili e altri (Mili et al. 2002) hanno descritto i metodi di stima dei costi per rendere un componente riutilizzabile e per calcolare il ritorno di tale investimento. I benefici derivanti dal riutilizzo rispetto allo sviluppo di un nuovo componente non sono soltanto guadagni in termini di produttività. Ci sono anche miglioramenti della qualità, in quanto un componente riutilizzato è più affidabile, e risparmi nei tempi di commercializzazione del componente. Questi sono ritorni aggiuntivi che derivano dalla consegna più veloce del software.

Mili e altri presentano diverse formule per la stima di tali guadagni, come fa il modello COCOMO che è descritto nel Capitolo 20. Tuttavia, i parametri di queste formule sono difficili da stimare con esattezza e le formule devono essere adattate alle situazioni locali, e questo le rende difficili da utilizzare. Credo che pochi project manager utilizzino questi modelli per stimare il ritorno degli investimenti dal riutilizzo dei componenti.

La riutilizzabilità di un componente dipende dal suo dominio di applicazione, dalle sue funzionalità e genericità. Se il dominio è generico e il componente implementa una funzionalità standard in tale dominio, è più probabile che esso sia riutilizzabile. Se miglioriamo la genericità di un componente, lo rendiamo più riutilizzabile, perché può essere applicato a una gamma più vasta di ambienti operativi. Purtroppo, questo di solito significa che il componente possiede più operazioni e diventa più complesso, e quindi più difficile da capire e utilizzare.

C'è un compromesso tra riutilizzabilità e comprensibilità di un componente. Per rendere un componente riutilizzabile, occorre fornire un insieme di interfacce generiche con operazioni che soddisfano tutti i modi in cui il componente può essere utilizzato. La riutilizzabilità aumenta la complessità e quindi riduce la comprensibilità di un componente. Questo rende più difficile e lungo il processo per stabilire se un componente è adatto al riutilizzo. A causa del tempo impiegato per capire se un componente è riutilizzabile, a volte è più conveniente reimplementare un componente più semplice con la funzionalità specifica che è stata richiesta.

Una fonte potenziale di componenti sono i sistemi ereditati. Come detto nel Capitolo 9, i sistemi ereditati sono sistemi che svolgono importanti funzionalità aziendali, ma sono scritti usando tecnologie software obsolete. A causa di ciò, potrebbe essere difficile utilizzarli con i nuovi sistemi. Tuttavia, se si riesce a convertire questi vecchi sistemi in componenti, le loro funzionalità possono essere riutilizzate in nuove applicazioni.

Ovviamente, i sistemi ereditati non hanno interfacce chiaramente definite di requisiti "richiesti" e "forniti". Per rendere riutilizzabili questi componenti, bisogna creare un wrapper che definisce le interfacce dei componenti. Il wrapper nasconde la complessità del codice sottostante e fornisce un'interfaccia ai componenti esterni per accedere ai servizi forniti. Sebbene questo wrapper sia un blocco di software abbastanza complesso, il costo del suo sviluppo potrebbe essere significativamente minore del costo di reimplementazione del sistema ereditato.

Una volta sviluppato e testato un componente o un servizio riutilizzabile, esso deve essere gestito per il riutilizzo futuro. La gestione richiede una scelta della modalità di classificazione del componente, in modo che esso possa essere scoperto, mantenendo le informazioni sul suo utilizzo e tenendo traccia delle sue differenti versioni. Se il componente è open-source, è possibile renderlo disponibile in un repository pubblico, come GitHub o Sourceforge. Se il componente è destinato a essere utilizzato in una società, si può utilizzare un repository interno.

Una società con un programma di riutilizzo potrebbe eseguire alcune forme di certificazione del componente, prima che questo sia reso disponibile al riutilizzo. La certificazione implica che qualcuno, eccetto lo sviluppatore, controlli la qualità del componente. Il collaudatore testa il componente e certifica che esso ha raggiunto un livello di qualità accettabile, prima che sia reso disponibile al riutilizzo. Poiché questo processo potrebbe essere costoso, molte società lasciano agli sviluppatori il test e il controllo della qualità dei componenti.

16.2.2 CBSE con il riutilizzo

Affinché il riutilizzo dei componenti abbia successo, è necessario che il processo di sviluppo sia personalizzato in modo da includere i componenti nel software da sviluppare. Il CBSE con il processo di riutilizzo deve includere le attività che trovano e integrano i componenti riutilizzabili. La struttura di tale processo è stata descritta nel Capitolo 2, e la Figura 16.7 mostra le principali attività all'interno di questo processo. Alcune delle attività, come quella iniziale di identificazione dei requisiti dell'utente, si svolgono nello stesso modo di altri processi software. Tuttavia, le differenze essenziali tra CBSE con il riutilizzo e processi software per lo sviluppo di software originale sono le seguenti.

1. I requisiti dell'utente inizialmente vengono sviluppati a grandi linee, anziché nei dettagli, e gli stakeholder sono incoraggiati a essere più flessibili nella definizione dei loro requisiti. I requisiti che sono troppo specifici limitano il numero di componenti che possono soddisfarli. Diversamente dallo sviluppo incrementale, però, occorre una descrizione completa dei requisiti in modo che si possano identificare più componenti possibili per il riutilizzo.
2. I requisiti vengono perfezionati e modificati all'inizio del processo a seconda dei componenti disponibili. Se i requisiti dell'utente non possono essere soddisfatti dai componenti disponibili, si dovrebbero discutere i requisiti correlati che possono essere supportati dai componenti riutilizzabili. Gli utenti potrebbero cambiare idea se questo portasse a una consegna più rapida ed economica del sistema.
3. C'è un'ulteriore attività di ricerca dei componenti e di perfezionamento del progetto dopo che l'architettura del sistema è stata definita. Alcuni componenti utilizzabili potrebbero rivelarsi inadatti o non funzionare appropriata-

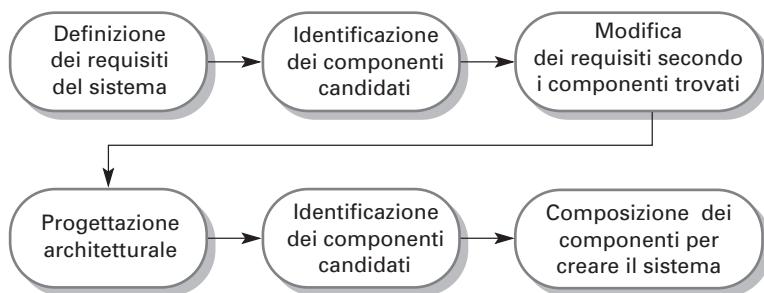


Figura 16.7 CBSE con il riutilizzo.

mente con gli altri componenti scelti. Di conseguenza, potrebbero essere necessarie ulteriori modifiche dei requisiti, a seconda delle funzionalità di questi componenti.

4. Lo sviluppo è un processo di composizione dove i componenti individuati vengono integrati nell'infrastruttura del modello; spesso questo richiede lo sviluppo di adattatori che riconciliano le interfacce dei componenti incompatibili. Ovviamente, potrebbero essere richieste funzionalità aggiuntive, oltre a quelle fornite dai componenti riutilizzati.

Lo stadio di progettazione architettonica è particolarmente importante. Jacobsen e altri (Jacobsen, Griss e Jonsson 1997) hanno scoperto che la definizione di un'architettura robusta è un fattore cruciale per il successo del processo di riutilizzo. Durante la progettazione architettonica, è possibile scegliere un modello di componenti e la piattaforma di implementazione. Tuttavia, molte società hanno una piattaforma di sviluppo standard (per esempio, .NET), quindi il modello dei componenti è predeterminato. Come detto nel Capitolo 6, è possibile definire anche l'architettura di alto livello del sistema in questo stadio e decidere le modalità di distribuzione e controllo del sistema.

Un'attività che è unica del processo CBSE è l'identificazione dei componenti o servizi che possono essere riutilizzati. Questo richiede un certo numero di sottotestività, come illustrato nella Figura 16.8. Inizialmente, ci si focalizza sulla ricerca e sulla selezione dei componenti. Occorre convincersi che ci sono componenti disponibili che possono soddisfare i requisiti dell'utente. Ovviamente, è necessario un controllo iniziale per verificare se il componente è appropriato, anche senza eseguire un test dettagliato. Nello stadio successivo, dopo che è stata progettata l'architettura del sistema, si dovrebbe spendere più tempo nella convallida dei componenti. Si deve essere certi che i componenti identificati siano realmente adatti alla propria applicazione; se non lo sono, bisogna ripetere i processi di ricerca e selezione.

Il primo passo nell'identificazione dei componenti è la ricerca di componenti che siano disponibili all'interno della società o presso fornitori fidati. Ci sono pochi fornitori di componenti, quindi è probabile che cercherete i componenti che



Figura 16.8 Il processo di identificazione dei componenti.

sono stati sviluppati nella vostra società o in repository di software open-source. Le società di sviluppo del software possono costruire i loro database di componenti riutilizzabili, evitando i rischi intrinseci derivanti dall'uso di componenti di fornitori esterni. In alternativa, potreste cercare nelle librerie disponibili sul web, come Sourceforge, GitHub o Google Code, il codice sorgente per il componente che vi occorre.

Una volta che il processo di ricerca ha identificato una lista di possibili componenti, occorre selezionare quelli più appropriati. In alcuni casi, questo è un compito semplice. Alcuni componenti potrebbero implementare direttamente i requisiti dell'utente, e potrebbero non esserci altri componenti che soddisfano tali requisiti. In altri casi, invece, il processo di selezione è più complesso. Potrebbe non esserci una corrispondenza chiara tra requisiti e componenti. Potrebbe essere necessario utilizzare diversi componenti per soddisfare uno specifico requisito o un gruppo di requisiti. Occorre quindi decidere quali di queste composizioni di componenti offrono la migliore copertura dei requisiti.

Una volta selezionati i componenti per una possibile inclusione nel sistema, si dovrebbero convalidare per verificare che si comportano come previsto. L'entità del processo di convalida dipende dalla fonte dei componenti. Se si stanno utilizzando componenti che sono stati sviluppati da una fonte conosciuta e fidata, si può decidere di non effettuare i test. I componenti vengono testati solo quando sono integrati con altri componenti. D'altra parte, se la sorgente è sconosciuta, i componenti dovrebbero essere sempre controllati e testati, prima di essere inclusi nel sistema.

La convalida dei componenti richiede lo sviluppo di un insieme di test case per i componenti (o l'estensione dei test case forniti con i componenti) e di un apposito ambiente per eseguire i test. Il problema principale della convalida dei componenti è che le loro specifiche possono non essere sufficientemente dettagliate per sviluppare un insieme completo di test. I componenti di solito sono specificati informalmente, e la loro unica documentazione formale è la specifica della loro interfaccia. Questa potrebbe non includere informazioni sufficienti per sviluppare un insieme completo di test tale da convincersi che l'interfaccia annunciata per i componenti sia quella di cui si ha effettivamente bisogno.

Oltre a testare che il componente da riutilizzare svolga i compiti richiesti, occorre controllare che il componente non includa codice dannoso o funzionalità non richieste. Gli sviluppatori professionisti raramente usano componenti provenienti da fonti inaffidabili, specialmente se queste fonti non forniscono il codice

L'errore del dispositivo di lancio dell'Ariane 5

Mentre stavano sviluppando il sistema di lancio dell'Ariane 5, i progettisti decisero di riutilizzare il software di riferimento inerziale che aveva funzionato con successo nel dispositivo di lancio dell'Ariane 4, ovvero il software che mantiene la stabilità del razzo. Decisero di riutilizzarlo senza modifiche (come si farebbe per i componenti), sebbene includesse funzionalità aggiuntive, che non erano richieste dall'Ariane 5.

Nel primo lancio dell'Ariane 5, il software di navigazione inerziale fallì, e il razzo non poté essere controllato. Il razzo e tutto il suo carico si distrussero. Si scoprì che la causa del problema era stata un'eccezione non gestita quando la conversione di un numero a virgola fissa in un numero intero provocò un overflow numerico. Questo causò lo spegnimento del sistema inerziale di riferimento da parte del sistema, che impedì il mantenimento della stabilità del dispositivo di lancio. L'errore non si era mai verificato nell'Ariane 4, perché aveva motori meno potenti e il valore che veniva convertito non poteva esser così grande da mandare in overflow la conversione.

Questo esempio dimostra un importante problema del riutilizzo del software. Il software si può basare su alcune ipotesi relative al contesto in cui il sistema sarà utilizzato, e queste ipotesi potrebbero non essere valide in situazioni differenti. Per ulteriori informazioni, consultate il sito <http://software-engineering-book.com/case-studies/ariane5/>

Figura 16.9 Esempio di errore di convalida con un software riutilizzato.

sorgente. In questo modo si evitano i problemi del codice dannoso. Tuttavia, i componenti riutilizzati spesso contengono funzionalità che non sono richieste; quindi, occorre verificare che queste funzionalità non interferiscano con l'utilizzo del componente.

Il problema delle funzionalità non richieste è che esse possono essere attivate dal componente stesso. Sebbene questo possa non avere effetto sull'applicazione che riutilizza il componente, tuttavia può rallentare il componente, causare risultati imprevisti o, in casi eccezionali, provocare gravi malfunzionamenti del sistema. La Figura 16.9 riassume una situazione nella quale il malfunzionamento di un sistema software riutilizzato, che aveva funzionalità superflue, ha provocato un errore catastrofico del sistema.

Il problema nel sistema di lancio dell'Ariane 5 sorse perché le ipotesi fatte riguardo al software dell'Ariane 4 non erano valide per l'Ariane 5. Si tratta di un problema generale dei componenti riutilizzabili. I componenti originariamente vengono implementati per uno specifico ambiente applicativo e, naturalmente, includono le ipotesi su tale ambiente. Queste ipotesi raramente sono documentate, perciò, quando il componente viene riutilizzato, è impossibile sviluppare i test per verificare se tali ipotesi sono ancora valide. Se state riutilizzando un componente in un nuovo ambiente, potreste non scoprire le ipotesi sottostanti finché non utilizzerete il componente in un sistema operativo.

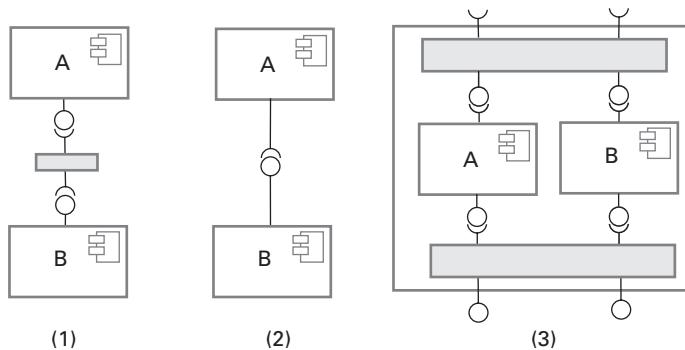


Figura 16.10 Tipi di composizione dei componenti.

16.3 Composizione dei componenti

La composizione dei componenti è il processo che permette di integrare più componenti per ottenere, tramite uno speciale “codice colla”, un nuovo sistema o un altro componente. I componenti possono essere composti in vari modi, come mostra la Figura 16.10. Questi diagrammi, da sinistra a destra, illustrano i vari tipi di composizione: sequenziale, gerarchica e additiva. Nella seguente discussione, si suppone che si stiano componendo due soli componenti (A e B) per creare un nuovo componente.

1. *Composizione sequenziale.* In questo tipo di composizione, si crea un nuovo componente da due componenti esistenti, chiamandoli in sequenza. Questa composizione può essere immaginata come la composizione delle interfacce dei “servizi forniti”, nel senso che vengono chiamati i servizi offerti dal componente A e i risultati restituiti da A vengono poi utilizzati nella chiamata dei servizi offerti dal componente B. I componenti non si chiamano a vicenda in una composizione sequenziale, ma sono chiamati da un’applicazione esterna. Questo tipo di composizione può essere utilizzato con i componenti incorporati o con i componenti di servizi.

Potrebbe essere richiesto un codice colla aggiuntivo per chiamare i servizi dei componenti nell’ordine corretto e per garantire che i risultati generati dal componente A siano compatibili con gli input attesi dal componente B. Il “codice colla” trasforma questi output nella forma appropriata per il componente B.

2. *Composizione gerarchica.* Questo tipo di composizione si verifica quando un componente chiama direttamente i servizi forniti da un altro componente, ovvero il componente A chiama il componente B. Il componente chiamato fornisce i servizi che servono al componente chiamante. Quindi, l’interfaccia dei “servizi forniti” del componente chiamato deve essere

compatibile con l’interfaccia dei “servizi richiesti” del componente chiamante.

Il componente A chiama direttamente il componente B e, se le loro interfacce sono compatibili, potrebbe non essere necessario scrivere un codice addizionale. Se, invece, l’interfaccia dei servizi richiesti da A non è compatibile con l’interfaccia dei servizi offerti da B, allora occorre un codice per consentire il dialogo tra le due interfacce. Poiché i servizi non hanno un’interfaccia di “servizi richiesti”, questa modalità di composizione non viene utilizzata quando i componenti sono implementati come servizi accessibili sul Web.

3. *Composizione additiva.* Questo tipo di composizione si verifica quando due o più componenti vengono assemblati insieme (sommati) per creare un nuovo componente, che combina le loro funzionalità. L’interfaccia dei “servizi offerti” e quella dei “servizi richiesti” del nuovo componente sono una combinazione delle corrispondenti interfacce nei componenti A e B. I componenti vengono chiamati separatamente tramite l’interfaccia esterna del componente composto e possono essere chiamati in qualsiasi ordine. I componenti A e B non sono dipendenti e non si chiamano fra loro. Questo tipo di composizione può essere utilizzato con i componenti incorporati o con i componenti di servizi.

Per creare un sistema, si possono usare tutte le forme di composizione dei componenti. In tutti i casi, potrebbe essere necessario scrivere un “codice colla” per collegare i componenti. Per esempio, nella composizione sequenziale, l’output del componente A diventa l’input del componente B; potrebbero servire le istruzioni intermedie per chiamare il componente A, raccogliere i risultati e chiamare il componente B con tali risultati come parametri. Quando un componente chiama un altro componente, potrebbe essere necessario introdurre un componente intermedio per garantire che l’interfaccia dei servizi richiesti e quella dei servizi forniti siano compatibili.

Quando si scrivono nuovi componenti specificamente per la composizione, bisogna progettare le interfacce di questi componenti in modo che siano compatibili con gli altri componenti del sistema. Questi componenti si possono facilmente comporre in una singola unità. Quando, invece, i componenti sono sviluppati in modo indipendente per il riutilizzo, spesso si devono risolvere alcune incompatibilità tra le interfacce; questo significa che le interfacce dei componenti che si vogliono comporre non sono le stesse. Possono verificarsi tre tipi di incompatibilità.

1. *Incompatibilità dei parametri.* Le operazioni hanno lo stesso nome in entrambe le parti, ma il tipo o il numero di parametri sono diversi. Nella Figura 16.11, il parametro di posizione restituito da addressFinder è incompatibile con i parametri richiesti dai metodi displayMap e printMap di mapDB.

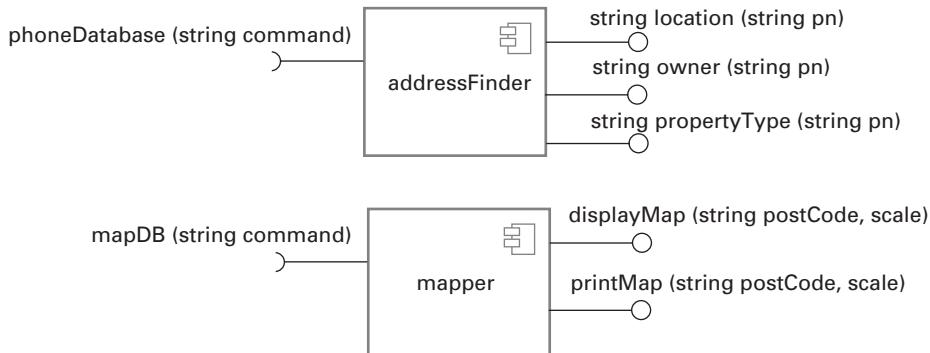


Figura 16.11 Componenti con interfacce incompatibili.

2. *Incompatibilità delle operazioni.* I nomi delle operazioni nelle interfacce dei servizi richiesti e forniti sono differenti. Questa è un’ulteriore incompatibilità tra i componenti illustrati nella Figura 16.11.
3. *Incompletezza delle operazioni.* L’interfaccia dei servizi forniti da un componente è un sottoinsieme dell’interfaccia dei servizi richiesti da un altro componente o viceversa.

In ogni caso, i problemi di incompatibilità possono essere risolti scrivendo un componente adattatore che rende compatibili le interfacce dei due componenti da riutilizzare. Un adattatore converte un’interfaccia nell’altra interfaccia.

La forma precisa dell’adattatore dipende dal tipo di composizione. A volte, come nel prossimo esempio, l’adattatore prende semplicemente il risultato di un componente e lo converte in una forma che possa essere utilizzata come input di un altro componente. In altri casi, l’adattatore può essere chiamato come proxy per il componente B. Questa situazione si verifica se A vuole chiamare B, ma i dettagli dell’interfaccia dei servizi richiesti da A non sono compatibili con quelli dell’interfaccia dei servizi forniti da B. L’adattatore riconcilia queste differenze convertendo i suoi parametri di input provenienti da A nei parametri di input richiesti per B; poi chiama B per consegnare i servizi richiesti da A.

Per illustrare gli adattatori, si considerino i due semplici componenti illustrati nella Figura 16.11, le cui interfacce sono incompatibili. Questi componenti potrebbero essere parti di un sistema utilizzato dai servizi di emergenza. Quando un operatore riceve una telefonata, il numero di telefono viene passato come input del componente **addressFinder** per localizzare l’indirizzo; poi, tramite il componente **mapper**, l’operatore stampa una mappa da trasmettere al veicolo incaricato di eseguire l’emergenza.

Il primo componente, **addressFinder**, trova l’indirizzo che corrisponde al numero di telefono; può anche restituire l’intestatario e il tipo di utenza telefonica. Il componente **mapper** riceve un codice postale e visualizza o stampa una mappa stradale dell’area associata al codice postale in una determinata scala.

In teoria, questi componenti sono componibili, in quanto l'indirizzo dell'intestatario dell'utenza telefonica include il codice postale. Tuttavia, è necessario scrivere un componente adattatore, chiamato `postCodeStripper`, che prende i dati dell'indirizzo forniti da `addressFinder` per estrarne il codice postale. Questo codice viene utilizzato come input del `mapper` per visualizzare una mappa stradale in scala 1:10000. Il codice seguente, che è un esempio di composizione sequenziale, illustra la sequenza di chiamate che sono necessarie per implementare il processo descritto:

```
address = addressFinder.location (phonenumbers);
postCode = postCodeStripper.getPostCode (address);
mapper.displayMap(postCode, 10000);
```

Un altro caso in cui può essere utilizzato un adattatore si ha nella composizione gerarchica, dove un componente vuole utilizzare un altro componente, ma c'è un'incompatibilità tra l'interfaccia dei servizi forniti e quella dei servizi richiesti dei componenti da comporre. La Figura 16.12 illustra l'uso di un adattatore per collegare il componente di raccolta dati con il componente sensore. Questi componenti potrebbero essere utilizzati nell'implementazione del sistema di stazioni meteorologiche, descritto nel Capitolo 7.

I due componenti, sensore e raccoglitore dati, vengono composti utilizzando un adattatore che rende compatibili le interfacce dei servizi richiesti dal componente che raccoglie i dati con l'interfaccia dei servizi forniti dal sensore. Il componente per la raccolta dei dati è stato progettato con una generica interfaccia di servizi richiesti che supporta la gestione della raccolta dei dati e dei sensori. Per ciascuna di queste operazioni, il parametro è una stringa di testo che rappresenta i comandi specifici del sensore. Per esempio, l'istruzione `sensorData("collect")` esegue un comando di raccolta dei dati. Come mostra la Figura 16.12, il sensore stesso può eseguire operazioni distinte, come `start`, `stop` e `getdata`.

L'adattatore analizza la stringa di input, identifica il comando (per esempio, `collect`) e poi chiama `Sensor.getData` per acquisire il valore rilevato dal sensore. Questo stile di interfaccia indica che il collettore dei dati può interagire con vari tipi di sensori. Per ogni tipo di sensore viene implementato un apposito adattatore, che converte i comandi del sensore da `Data collector` all'interfaccia dei sensori.

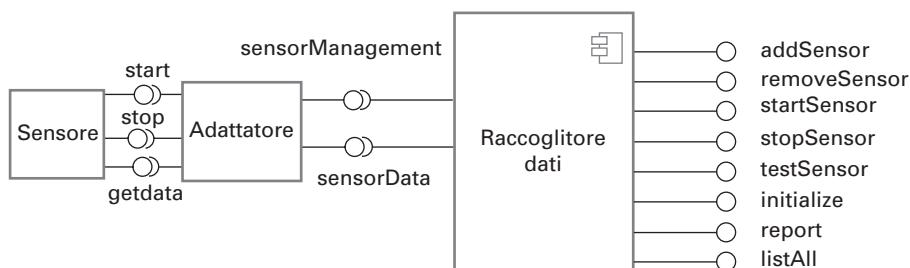


Figura 16.12 Un adattatore collega un raccoglitore dati con un sensore.

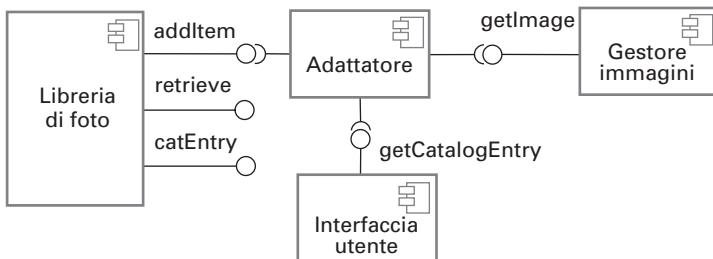


Figura 16.13 Composizione di una libreria di fotografie.

La precedente discussione sulla composizione dei componenti presume che si possa capire dalla documentazione del componente se le interfacce sono compatibili oppure no. Ovviamente, la definizione dell'interfaccia include il nome dell'operazione e i tipi di parametri, in modo che da queste informazioni si possano fare alcune valutazioni della compatibilità. Si noti, però, che la decisione sulla compatibilità semantica delle interfacce dipende dalla documentazione dei componenti.

Per spiegare questo problema, si consideri la composizione mostrata nella Figura 16.13. Questi componenti sono utilizzati per implementare un sistema che scarica immagini da una fotocamera digitale e le memorizza in una libreria. L'utente del sistema può fornire informazioni aggiuntive per descrivere e catalogare le fotografie. Per semplificare, non sono stati mostrati tutti i metodi delle interfacce, ma soltanto quelli necessari a illustrare il problema della documentazione dei componenti. I metodi nell'interfaccia della libreria delle foto sono:

```

public void addItem (Identifier pid; Photograph p; CatalogEntry photodesc);
public Photograph retrieve (Identifier pid);
public CatalogEntry catEntry (Identifier pid);
  
```

Si supponga che la documentazione del metodo `addItem` nella libreria delle foto contenga la seguente descrizione:

Questo metodo aggiunge una fotografia alla libreria e associa a essa un identificatore e un descrittore del catalogo.

Questa descrizione sembra spiegare che cosa fa il componente, ma consideriamo le seguenti domande:

- Che cosa succede se l'identificatore della fotografia è già associato a una fotografia nella libreria?
- Il descrittore della fotografia è associato a una voce del catalogo e anche alla fotografia? Ovvero, se si cancella la fotografia, si cancellano anche le informazioni nel catalogo?

- La parola chiave context indica il componente al quale applicare le condizioni

context addItem

- Le precondizioni specificano che cosa deve essere vero prima di eseguire addItem

pre: PhotoLibrary.libSize() > 0
 PhotoLibrary.retrieve(pid) = null

- Le postcondizioni specificano che cosa deve essere vero dopo l'esecuzione

post: libSize () = libSize()@pre + 1
 PhotoLibrary.retrieve(pid) = p
 PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid) <> null;
 post: PhotoLibrary.retrieve(pid) = null
 PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
 PhotoLibrary.libSize() = libSize()@pre-1

Figura 16.14 Descrizione OCL dell'interfaccia della libreria delle foto.

Non ci sono informazioni sufficienti nella descrizione informale di addItem per rispondere a queste domande. Ovviamente, è possibile aggiungere altre informazioni alla descrizione nel linguaggio naturale del metodo, ma in generale il modo migliore per chiarire le ambiguità è utilizzare un linguaggio formale per descrivere l'interfaccia. La specifica mostrata nella Figura 16.14 fa parte della descrizione dell'interfaccia della libreria delle foto che aggiunge informazioni alla descrizione informale.

La Figura 16.14 mostra le precondizioni e le postcondizioni definite in una notazione basata sul linguaggio dei vincoli degli oggetti (Object Constraint Language, OCL), che è parte di UML (Warmer e Kleppe 2003). OCL è progettato per descrivere i vincoli nei modelli di oggetti UML; permette di esprimere predicati che devono essere veri sempre, prima o dopo l'esecuzione di un metodo; sono le invarianti, le precondizioni e le postcondizioni. Per accedere al valore di una variabile prima di un'operazione, si aggiunge @pre al suo nome; per esempio

age = age@pre + 1

significa che il valore di age dopo un'operazione è aperto a quello che valeva prima più uno.

Gli approcci basati su OCL sono utilizzati principalmente per aggiungere informazioni semantiche ai modelli UML nell'ingegneria del software basato sui modelli. L'approccio generale è stato derivato dal metodo "Progettazione su contratto" di Meyer (Meyer 1992), in cui le interfacce e i vincoli degli oggetti di comunicazione sono specificati formalmente e rafforzati dal sistema a runtime. Meyer sostiene che l'uso della progettazione su contratto è essenziale per sviluppare componenti fidati (Meyer 2003).

La Figura 16.14 mostra la specifica dei metodi `addItem` e `delete` della libreria delle foto. Il metodo che si sta specificando è indicato dalla parola chiave `context`, mentre le precondizioni e le postcondizioni sono indicate dalle parole chiave `pre` e `post`. Le precondizioni per `addItem` sono qui di seguito elencate.

1. La libreria non può contenere una fotografia con lo stesso identificatore della fotografia da inserire.
2. La libreria deve esistere: si supponga che la creazione di una libreria implichi l'aggiunta di un singolo elemento, in modo che la dimensione della libreria sia sempre maggiore di zero.
3. Le postcondizioni per `addItem` sono le seguenti:
 - la dimensione della libreria è aumentata di uno (quindi è stato inserito un singolo elemento);
 - se si esegue il comando `retrieve` (ritrovamento) utilizzando lo stesso identificatore, si ottiene la fotografia che è stata aggiunta;
 - se si effettuano delle ricerche nel catalogo utilizzando tale identificatore, si ottiene l'elemento che è stato creato.

La specifica di `delete` fornisce ulteriori informazioni. Le precondizioni indicano che per cancellare un oggetto, questo deve essere nella libreria e che, dopo la cancellazione, la foto non può più essere recuperata e la dimensione della libreria si è ridotta di 1. Tuttavia, il comando `delete` non cancella l'elemento dal catalogo, e quindi la foto cancellata può essere ancora recuperata. La ragione di questo è che potrebbe essere richiesto di conservare nel catalogo le informazioni sul perché la foto è stata eliminata, sulla sua nuova posizione e così via.

Quando si crea un sistema componendo più componenti, possono emergere potenziali conflitti tra requisiti funzionali e non funzionali, la necessità di conseguire un sistema il più velocemente possibile e la necessità di creare un sistema che possa evolversi quando i requisiti vengono modificati. Per questo potrebbe essere necessario accettare determinati compromessi sui componenti.

1. Quale composizione di componenti è più efficace nella consegna dei requisiti funzionali del sistema?
2. Quale composizione di componenti permette l'adattamento di future modifiche dei requisiti?
3. Quali saranno le proprietà emergenti del sistema composto? Queste proprietà includono le prestazioni e la fidatezza. Queste proprietà possono essere valutate soltanto dopo che il sistema completo è stato implementato.

In molti casi, purtroppo, le soluzioni dei problemi di composizione possono entrare in conflitto. Per esempio, si consideri il caso illustrato nella Figura 16.15, in cui un sistema può essere creato tramite due composizioni alternative. È un sistema per raccogliere dati e generare report: i dati provengono da fonti differenti, vengono immagazzinati in un database e, infine, riassunti in report diversi.

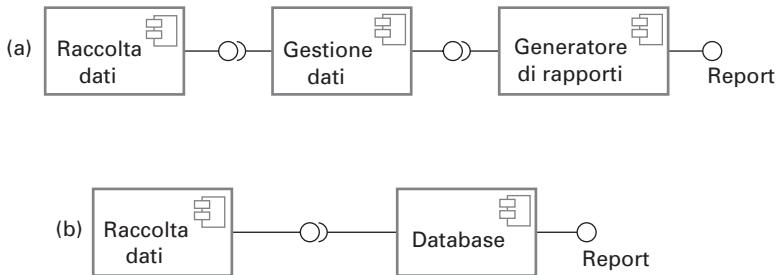


Figura 16.15 Componenti di raccolta dati e produzione di report.

Qui c’è un potenziale conflitto tra adattabilità e prestazioni. La composizione (a) è più adattabile, mentre la composizione (b) è più rapida e affidabile. Il vantaggio della composizione (a) è che la gestione dei dati e la generazione dei report sono separate; questo permette una maggiore flessibilità per le modifiche future. Il sistema di gestione dei dati può essere sostituito; inoltre, se sono richiesti report che il componente attuale non può generare, anche questo componente può essere sostituito, senza bisogno di modificare il componente che gestisce i dati.

Nella composizione (b) viene utilizzato un componente per database con la funzionalità di produrre report (Microsoft Access, per esempio). Il vantaggio chiave di questa composizione consiste nel minor numero di componenti, che rende più veloce l’implementazione, in quanto non ci sono overhead per la comunicazione tra i componenti. Inoltre, le regole di integrità dei dati che si applicano al database valgono anche per i report, che non potranno combinare i dati in modo errato. Nella composizione (a) non ci sono questi vincoli, quindi le probabilità di errore sono più alte.

In generale, una buona regola di composizione da seguire è il principio di separazione dei ruoli, ovvero si deve provare a progettare il sistema in modo che ogni componente abbia un ruolo ben definito. In teoria, i ruoli dei componenti non dovrebbero sovrapporsi. Tuttavia, potrebbe essere più conveniente economicamente comprare un singolo componente multifunzionale, anziché due o tre componenti distinti. Inoltre, la fidatezza e le prestazioni potrebbero essere compromesse utilizzando più componenti.

Punti chiave

- L’ingegneria del software basato sui componenti (CBSE) è un approccio utilizzato per definire, implementare e comporre componenti indipendenti debolmente accoppiati al fine di formare sistemi.

- Un componente è un’unità software le cui funzionalità e dipendenze sono definite completamente da un insieme di interfacce pubbliche. I componenti possono essere combinati tra loro senza conoscere le loro implementazioni e possono essere consegnati come un’unità eseguibile.
- Un modello di componenti definisce un insieme di standard per i componenti, che comprende gli standard di interfaccia, di utilizzo e di consegna. L’implementazione del modello di componenti fornisce un insieme di servizi comuni che possono essere utilizzati da tutti i componenti.
- Durante il processo CBSE si devono intrecciare i processi di ingegneria dei requisiti e di progettazione del sistema. Si devono trovare compromessi appropriati tra i requisiti desiderabili e i servizi che sono disponibili tramite i componenti riutilizzabili esistenti.
- La composizione dei componenti è il processo che “collega” vari componenti per ottenere un sistema. La composizione può essere sequenziale, gerarchica e additiva.
- Quando si compongono componenti riutilizzabili che non sono stati scritti per la propria applicazione, potrebbe essere necessario scrivere degli adattatori o un “codice colla” per rendere compatibili le diverse interfacce dei componenti.
- Quando si scelgono le composizioni, si devono considerare le funzionalità richieste dal sistema, i requisiti non funzionali e la facilità con cui un componente può essere sostituito da un altro quando il sistema viene modificato.

Esercizi

- * 16.1 Perché è importante che tutte le interazioni tra i componenti siano definite attraverso interfacce di servizi richiesti e forniti?
- 16.2 Per il principio di indipendenza dei componenti dovrebbe essere possibile sostituire un componente con un altro implementato in modo completamente diverso. Usando un esempio, esponete come tale sostituzione di componenti possa avere conseguenze indesiderate e causare errori del sistema.
- * 16.3 Quali sono le differenze fondamentali tra i componenti usati come elementi di programma e i componenti usati come servizi?
- 16.4 Perché è importante che i componenti si basino su un modello di componenti standard?
- * 16.5 Usando un esempio di un componente che implementa un tipo di dati astratto, come uno stack o una lista, spiegate perché di solito è necessario estendere e adattare i componenti per il riutilizzo.
- * 16.6 Spiegate perché è difficile convalidare un componente riutilizzabile senza il suo codice sorgente. In quali modi una specifica formale del componente può semplificare i problemi di convalida?
- 16.7 Progettate l’interfaccia di “servizi forniti” e l’interfaccia di “servizi richiesti” per un componente riutilizzabile che può essere utilizzato per rappresentare un paziente nel sistema Mentcare descritto nel Capitolo 1.
- * 16.8 Tramite esempi spiegate i diversi tipi di adattatori necessari per supportare le composizioni sequenziale, gerarchica e additiva.

-
- 16.9 Progettate le interfacce dei componenti che possono essere utilizzati in un sistema per una sala di controllo delle emergenze. Dovreste progettare le interfacce per un componente che registra le telefonate effettuate e per un componente che, noti il codice postale e il tipo di incidente, trova il veicolo adatto più vicino da inviare sul luogo dell'incidente.
 - 16.10 È stata proposta la creazione di un'authority indipendente di certificazione alla quale i produttori possono inviare i propri componenti per la convalida della loro fidatezza. Quali sarebbero i vantaggi e gli svantaggi di tale authority di certificazione?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Component Software: Beyond Object-Oriented Programming, 2nd ed. Quest'edizione aggiornata del primo libro su CBSE tratta problemi tecnici e non tecnici. Rispetto al libro di Heineman e Councill, include nuovi dettagli sulle tecnologie specifiche oltre a un'analisi completa delle problematiche commerciali (C. Szyperski, Addison-Wesley, 2002).

“Specification, Implementation and Deployment of Components.” Una buona introduzione ai concetti fondamentali di CBSE. Lo stesso numero di *CACM* include articoli sui componenti e sullo sviluppo basato sui componenti (I. Crnkovic, B. Hnich, T. Jonsson e Z. Kiziltan, *Comm. ACM*, 45(10), October 2002) <http://dx.doi.org/10.1145/570907.570928>

“Software Component Models.” Una trattazione completa dei modelli di componenti commerciali e di ricerca. I modelli vengono prima classificati, poi ne vengono spiegate le differenze. (K-K. Lau e Z. Wang, *IEEE Transactions on Software Engineering*, 33 (10), October 2007) <http://dx.doi.org/10.1109/TSE.2007.70726>

“Software Components Beyond Programming: From Routines to Services.” È il primo articolo in un numero speciale della rivista che include vari articoli sui componenti software. Questo articolo descrive l'evoluzione dei componenti e come i componenti orientati ai servizi stanno sostituendo le routine dei programmi. (I. Crnkovic, J. Stafford e C. Szyperski, *IEEE Software*, 28 (3), May/June 2011) <http://dx.doi.org/10.1109/MS.2011.62>

Object Constraint Language (OCL) Tutorial. Una buona introduzione all'uso del linguaggio dei vincoli degli oggetti. (J. Cabot 2012) <http://modeling-languages.com/ocl-tutorial/>

CAPITOLO

17

Ingegneria del software distribuito

Questo capitolo si propone di descrivere l'ingegneria dei sistemi distribuiti e i loro schemi architetturali. Dopo aver letto questo capitolo:

- conoscerete i problemi chiave da considerare per progettare e implementare i sistemi software distribuiti;
- conoscerete il modello di calcolo client-server e l'architettura a strati dei sistemi client-server;
- conoscerete gli schemi comunemente utilizzati nelle architetture dei sistemi distribuiti e i vari tipi di sistemi ai quali tali schemi possono essere applicati;
- apprenderete il concetto di software come servizio, che fornisce l'accesso tramite Web ai sistemi applicativi remoti.

- 17.1 Sistemi distribuiti
- 17.2 Calcolo client-server
- 17.3 Schemi architetturali per sistemi distribuiti
- 17.4 Software come servizio

Tutti i grandi sistemi informatici sono ora sistemi distribuiti. Un sistema distribuito è quello in cui un'applicazione viene eseguita su più computer, anziché su una singola macchina. Anche quelle applicazioni apparentemente indipendenti (come gli editor di immagini), che vengono eseguite su un PC o un laptop, sono sistemi distribuiti. Esse vengono eseguite su un singolo computer, ma spesso utilizzano sistemi cloud remoti per l'aggiornamento, la memorizzazione di dati e altri servizi. Tanenbaum e Van Steen (Tanenbaum e Van Steen 2007) definiscono un “sistema distribuito” come “un gruppo di computer indipendenti che appaiono all’utente come un unico sistema coerente”.¹

Quando si progetta un sistema distribuito, ci sono problemi specifici da affrontare, semplicemente perché il sistema è distribuito. Questi problemi derivano dal fatto che varie parti del sistema vengono eseguite su computer gestiti in modo indipendente; inoltre, durante la progettazione, bisogna tenere in considerazione le caratteristiche della rete, quali la latenza e l'affidabilità.

Coulouris e altri (Coulouris et al. 2001) hanno identificato cinque vantaggi offerti dai sistemi distribuiti.

1. *Condivisione delle risorse.* Un sistema distribuito permette di condividere le risorse hardware e software (dischi, stampanti, file e compilatori) associate ai computer di una rete.
2. *Apertura.* I sistemi distribuiti solitamente sono sistemi aperti, ovvero progettati su protocolli Internet standard che permettono di combinare più unità hardware e software di diversi fornitori.
3. *Simultaneità.* In un sistema distribuito possono operare contemporaneamente più processi su vari computer della rete. Questi processi possono comunicare tra loro durante il loro normale funzionamento.
4. *Scalabilità.* Teoricamente, i sistemi distribuiti sono scalabili, nel senso che le capacità di un sistema possono essere potenziate, aggiungendo nuove risorse, per soddisfare un incremento delle richieste del sistema. In pratica, la rete che collega i diversi computer può limitare la scalabilità di un sistema.
5. *Tolleranza ai guasti.* La disponibilità di diversi computer e la possibilità di replicare le informazioni permettono ai sistemi distribuiti di tollerare alcuni guasti hardware e software (Capitolo 11). In molti sistemi distribuiti potrebbe essere fornito un servizio deteriorato quando si verifica un guasto; la perdita completa del servizio si verifica soltanto se c’è un guasto nella rete.²

¹ Tanenbaum A. S. e M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms*, 2nd Ed. Upper Saddle River, NJ: Prentice-Hall.

² Coulouris G., J. Dollimore, T. Kindberg e G. Blair. 2011. *Distributed Systems: Concepts and Design*, 5th Edition. Harlow, UK.: Addison Wesley.

I sistemi distribuiti sono intrinsecamente più complessi di quelli centralizzati. Questo complica la progettazione, l'implementazione e i test. È più difficile capire le proprietà emergenti dei sistemi distribuiti, a causa della complessità delle interazioni tra i componenti e l'infrastruttura del sistema. Per esempio, anziché essere dipendenti dalla velocità di esecuzione di un processore, le prestazioni di un sistema distribuito dipendono dalla larghezza di banda della rete, dal carico della rete e dalla velocità degli altri computer che fanno parte del sistema. Lo spostamento delle risorse da una parte all'altra del sistema può influire significativamente sulle prestazioni del sistema.

Inoltre, come noto a tutti gli utenti del Web, i sistemi distribuiti sono imprevedibili nelle loro risposte. Il tempo di risposta dipende dal carico complessivo del sistema, dalla sua architettura e dal carico della rete. Poiché tutti questi fattori possono cambiare in un breve periodo di tempo, il tempo necessario per rispondere alla richiesta di un utente può variare notevolmente da una richiesta all'altra.

Gli sviluppi più importanti che hanno influito sui sistemi software distribuiti negli ultimi anni sono i sistemi orientati ai servizi e il calcolo cloud, che offrono infrastrutture, piattaforme e software come servizi. In questo capitolo tratterò i problemi generali dei sistemi distribuiti; nel Paragrafo 17.4 spiegherò il concetto di software come servizio. Nel Capitolo 18 tratterò gli aspetti dell'ingegneria del software orientato ai servizi.

17.1 Sistemi distribuiti

Come detto nell'introduzione di questo capitolo, i sistemi distribuiti sono più complessi di quelli che vengono eseguiti su un singolo processore. Questa complessità nasce perché è praticamente impossibile avere un modello top-down per il controllo di questi sistemi. I nodi del sistema che forniscono le funzionalità spesso sono sistemi indipendenti che sono gestiti e controllati dai loro proprietari. Non c'è un'unica autorità che è responsabile di tutto il sistema distribuito. Anche la rete che collega questi nodi è un sistema gestito separatamente. Si tratta di un sistema intrinsecamente complesso, che non può essere controllato dai proprietari dei sistemi utilizzando la rete. C'è, quindi, una imprevedibilità intrinseca nel funzionamento dei sistemi distribuiti che deve essere presa in considerazione quando si progetta un sistema.

Alcuni dei più importanti problemi di progettazione che devono essere considerati nei sistemi distribuiti sono qui di seguito elencati.

1. *Trasparenza.* In che misura un sistema distribuito dovrà apparire come singolo sistema all'utente? Quando è utile per gli utenti capire che il sistema è distribuito?

2. *Apertura.* Un sistema dovrà essere progettato utilizzando protocolli standard che supportano l’interoperabilità o protocolli più specializzati? Sebbene oggi si utilizzino universalmente protocolli di rete standard, tuttavia questo non è vero per livelli di interazione più elevati, come la comunicazione dei servizi.
3. *Scalabilità.* Come si costruisce un sistema per renderlo scalabile? Ovvero, come può essere progettato un intero sistema in modo che le sue capacità possano essere potenziate in seguito alle crescenti richieste fatte al sistema?
4. *Protezione.* Come possono essere definite e implementate utili strategie di protezione da applicare a un gruppo di sistemi che sono gestiti autonomamente?
5. *Qualità dei servizi.* Come dovrebbe essere specificata la qualità dei servizi forniti agli utenti del sistema? Come dovrebbe essere implementato il sistema per fornire a tutti gli utenti una qualità di servizi accettabile?
6. *Gestione dei guasti.* Come possono essere rilevati, contenuti (in modo che abbiano effetti minimi sui componenti del sistema) e riparati i guasti dei sistemi?

In un mondo ideale, il fatto che un sistema sia distribuito dovrebbe essere trasparente agli utenti, che lo vedrebbero come un singolo sistema il cui comportamento non è influenzato dal modo in cui esso è distribuito. In pratica, questo è impossibile, in quanto non c’è un controllo centrale sul sistema nel suo complesso. Ne consegue che i singoli computer di un sistema possono comportarsi in modo diverso in periodi differenti. In aggiunta, poiché occorre sempre un periodo di tempo finito affinché i segnali attraversino la rete, i ritardi della rete sono inevitabili. La durata di questi ritardi dipende dalla posizione delle risorse nel sistema, dalla qualità della connessione di rete dell’utente e dal carico della rete.

Per rendere trasparente un sistema distribuito (ovvero celare la sua natura distribuita), dovete nascondere la distribuzione sottostante. Create astrazioni che nascondono le risorse del sistema in modo che la posizione e l’implementazione di queste risorse possano essere modificate senza dover cambiare l’applicazione distribuita. Il middleware (descritto nel Paragrafo 17.1.2) è utilizzato per mappare le risorse logiche cui fa riferimento un programma alle risorse fisiche e per gestire le interazioni delle risorse.

In pratica, è impossibile rendere un sistema completamente trasparente e, in generale, gli utenti sono consapevoli che stanno utilizzando un sistema distribuito. Pertanto, potreste decidere di svelare la distribuzione agli utenti, i quali potranno essere pronti ad accettare qualcuna delle conseguenze della distribuzione, quali i ritardi della rete e i guasti dei nodi remoti.

I sistemi distribuiti aperti sono costruiti secondo standard universalmente accettati. I componenti di un fornitore possono quindi essere integrati nel sistema e interagire con altri componenti del sistema. Al livello della rete, l’apertura dei

CORBA (Common Object Request Broker Architecture)

CORBA fu proposto come una specifica per un sistema middleware negli anni '90 dall'OMG (Object Management Group). Fu ideato come uno standard aperto che avrebbe consentito lo sviluppo di middleware a supporto della comunicazione e dell'esecuzione di componenti distribuiti, fornendo anche una serie di servizi standard utilizzabili da questi componenti. Furono prodotte diverse implementazioni di CORBA, ma il sistema non ebbe una grande diffusione. Gli utenti preferirono i sistemi proprietari, come quelli di Oracle e Microsoft, oppure passarono alle architetture orientate ai servizi.

<http://software-engineering-book.com/web/corba/>

sistemi distribuiti è ormai data per scontata, con i sistemi che sono conformi ai protocolli Internet, ma al livello dei componenti, l'apertura non è ancora accettata universalmente. L'apertura implica che i componenti del sistema possano essere sviluppati in modo indipendente con qualsiasi linguaggio di programmazione e, se essi sono conformi agli standard, potranno interagire con altri componenti.

Lo standard CORBA (Pope 1997), sviluppato negli anni '90, fu ideato come standard universale per i sistemi distribuiti; tuttavia, esso non raggiunse mai una massa critica di utilizzatori; piuttosto, molte società preferirono sviluppare sistemi utilizzando standard proprietari per i componenti realizzati da società come Sun (adesso Oracle) e Microsoft. Queste società fornivano implementazioni e software di supporto migliori e garantivano un supporto a lungo termine per i protocolli industriali.

Furono sviluppati standard di servizi web (descritti nel Capitolo 18) per le architetture orientate ai servizi come standard aperti. Tuttavia, questi standard incontrarono una significativa resistenza a causa della loro inefficienza percepita. Molti sviluppatori di sistemi basati sui servizi optarono invece per i cosiddetti protocolli RESTful, che hanno overhead intrinsecamente più bassi dei protocolli dei servizi web. L'uso dei protocolli RESTful non è standardizzato.

La scalabilità di un sistema rispecchia la sua capacità di fornire servizi di alta qualità al crescere delle richieste ricevute dal sistema. Le tre dimensioni della scalabilità sono la taglia, la distribuzione e la gestibilità.

1. *Taglia.* È possibile aggiungere altre risorse a un sistema per soddisfare le richieste di un numero crescente di utenti. Teoricamente, all'aumentare del numero di utenti, il sistema dovrebbe aumentare la sua taglia automaticamente per gestire il maggior numero di utenti.
2. *Distribuzione.* Dovrebbe essere possibile disperdere geograficamente i componenti di un sistema ignorando le sue prestazioni. Quando vengono aggiunti nuovi componenti, non è importante il luogo in cui essi si trovano. Le grandi società spesso fanno uso di risorse di calcolo presso le loro sedi distribuite in tutto il mondo.

3. *Gestibilità.* Dovrebbe essere possibile gestire un sistema al crescere delle sue dimensioni, anche se alcune parti del sistema si trovano in organizzazioni indipendenti. Questa è una delle sfide più difficili della scalabilità, in quanto richiede la comunicazione tra più manager e l'accordo sulle politiche di gestione. In pratica, la gestibilità di un sistema è spesso il fattore che limita l'entità della sua scalabilità.

Quando un sistema cambia taglia, potrebbe essere necessario potenziare le risorse esistenti (*scaling up*) o aggiungere nuove risorse (*scaling out*). Nel primo caso, le risorse esistenti vengono sostituite con risorse più potenti; per esempio, si potrebbe aumentare la memoria del sistema da 16 Gb a 64 Gb. Nel secondo caso, si aggiungono nuove risorse al sistema (per esempio, un server web aggiuntivo viene affiancato al server esistente). Il processo di scaling out spesso è economicamente più efficiente dello scaling up, specialmente adesso che il calcolo cloud semplifica l'aggiunta o l'eliminazione di server da un sistema. Tuttavia, questo processo permette di migliorare le prestazioni soltanto quando è possibile eseguire più processi contemporaneamente.

Nella Parte II di questo libro ho trattato i problemi generali della protezione dei sistemi. Quando un sistema è distribuito, i pirati informatici potrebbero attaccare singoli componenti del sistema o la rete stessa. Se l'attacco a una parte del sistema ha successo, gli hacker potrebbero essere in grado di usare questa parte come “porta posteriore” di accesso ad altre parti del sistema.

Un sistema distribuito deve difendersi dai seguenti tipi di attacchi:

1. *Intercettazione.* Un hacker intercetta le comunicazioni fra le parti del sistema, con conseguente perdita di riservatezza.
2. *Interruzione.* I servizi del sistema vengono attaccati e non possono essere offerti come previsto. Gli attacchi denial-of-service includono il bombardamento di un nodo con richieste illegittime di servizi in modo da rendere impossibile la gestione delle richieste valide.
3. *Modifica.* Un hacker riesce ad accedere al sistema e modifica i dati o i servizi del sistema.
4. *Falsificazione.* Un hacker crea informazioni false che non dovrebbero esistere; poi le usa per ottenere determinati privilegi. Per esempio, potrebbe generare una falsa password per accedere al sistema.

La principale difficoltà nei sistemi distribuiti è definire una politica di protezione che può essere applicata in modo affidabile a tutti i componenti di un sistema. Come detto nel Capitolo 13, una politica di protezione stabilisce il livello di protezione che deve essere assicurato a un sistema. I meccanismi di protezione, come l'autenticazione e la crittografia, sono utilizzati per rafforzare tale politica. Le difficoltà in un sistema distribuito nascono perché più organizzazioni possono essere proprietarie di parti del sistema. Queste organizzazioni possono avere politiche e meccanismi di protezione tra loro incompatibili. Per poter garantire il

funzionamento delle varie parti di un sistema distribuito, potrebbe essere necessario raggiungere determinati compromessi sulla protezione.

La qualità del servizio (QoS, Quality of Service) offerta da un sistema distribuito riflette la capacità del sistema di fornire i suoi servizi in modo affidabile e con tempi di risposta e throughput accettabili per gli utenti. In teoria, i requisiti QoS dovrebbero essere specificati in anticipo e il sistema dovrebbe essere progettato e configurato per garantire tali requisiti. Purtroppo, questo non è sempre possibile per due ragioni.

1. Potrebbe non essere economicamente conveniente progettare e configurare il sistema per fornire un'alta qualità di servizi in condizioni di un picco di carico. Per soddisfare le richieste di picco e mantenere i tempi di risposta potrebbero essere necessari molti server aggiuntivi. Questo problema si è un po' ridotto con l'avvento del calcolo cloud, in quanto è possibile affittare alcuni server cloud da un provider per il tempo necessario. Quando le richieste di servizi aumentano, è possibile aggiungere automaticamente dei server extra.
2. I parametri della qualità del servizio potrebbero essere in conflitto tra loro. Per esempio, una maggiore affidabilità potrebbe significare un throughput minore, in quanto vengono introdotte le procedure di controllo per garantire che tutti gli input del sistema siano validi.

La qualità del servizio è particolarmente importante quando il sistema elabora dati critici nel tempo, come suoni o video. In questi casi, se la qualità del servizio scende sotto un certo valore di soglia, il suono o il video potrebbero degradarsi a tal punto da comprometterne la comprensibilità. I sistemi che elaborano suoni o video dovrebbero includere appositi componenti per la gestione e la negoziazione della qualità del servizio. Questi componenti dovrebbero valutare i requisiti QoS tenendo conto delle risorse disponibili e, se queste sono insufficienti, negoziare l'aggiunta di altre risorse o la riduzione del livello QoS.

In un sistema distribuito, è inevitabile che si verifichino dei guasti, quindi il sistema deve essere progettato in modo da essere resiliente ai guasti. I guasti sono così onnipresenti che hanno indotto Leslie Lamport, un eminente ricercatore di sistemi distribuiti, a dare la seguente definizione impertinente di sistema distribuito:

Vi accorgerete di avere un sistema distribuito quando il crash di un sistema, di cui non avete mai sentito parlare, vi impedirà di completare il vostro lavoro.³

Ciò è tanto più vero oggi che sempre più sistemi vengono eseguiti nel cloud. La gestione dei guasti richiede l'applicazione di tecniche tolleranti ai guasti, che sono descritte nel Capitolo 11. I sistemi distribuiti dovrebbero quindi includere

³ Leslie Lamport, in Ross J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems* (2nd ed.), Wiley (April 14, 2008).

dei meccanismi per stabilire se un componente del sistema ha fallito, se continuare a fornire il maggior numero possibile di servizi nonostante tale guasto e per risolvere automaticamente il guasto. Un vantaggio importante del calcolo cloud è che sono stati ridotti significativamente i costi per fornire i componenti ridondanti di un sistema.

17.1.1 Modelli di interazione

Due tipi fondamentali di interazioni possono avvenire tra i computer in un sistema di calcolo distribuito: l’interazione procedurale e l’interazione basata sui messaggi. Nell’interazione procedurale, un computer chiama un servizio offerto da qualche altro computer e resta in attesa del servizio. Nell’interazione basata sui messaggi, il computer invia a un altro computer un messaggio che contiene le informazioni su ciò di cui ha bisogno. I messaggi di solito trasmettono in una sola interazione più informazioni di una chiamata di procedura.

Per illustrare la differenza tra un’interazione procedurale e un’interazione basata sui messaggi, supponete di essere al ristorante e di voler ordinare da mangiare. Parlando con il cameriere, avrete una serie di interazioni procedurali sincrone che definiscono la vostra ordinazione. Voi fate una richiesta e il cameriere accetta la richiesta, fate un’altra richiesta, che il cameriere accetta e così via. Questa situazione è paragonabile a quella in cui i componenti interagiscono in un sistema software, quando un componente chiama i metodi di altri componenti. Il cameriere scrive l’ordinazione per tutti i vostri commensali; poi, passa alla cucina questa ordinazione, che include i dettagli di tutto ciò che avete ordinato. Sostanzialmente, il cameriere passa al personale della cucina un messaggio che definisce il cibo da preparare. Questa è un’interazione basata su messaggi.

Ho illustrato questo tipo di interazione nella Figura 17.1, che mostra il processo di ordinazione sincrono come una serie di chiamate, e nella Figura 17.2, che riporta un ipotetico messaggio XML che definisce un’ordinazione per un tavolo di tre clienti. La differenza tra queste forme di scambio di informazioni è chiara. Il cameriere riceve l’ordinazione come una serie di interazioni, ciascuna delle quali definisce una parte dell’ordinazione. Il cameriere, però, ha una sola interazione con la cucina, in quanto il messaggio definisce tutti i dettagli dell’ordinazione.

La comunicazione procedurale in un sistema distribuito di solito è implementata utilizzando delle chiamate di procedure remote (CPR). In una CPR, ogni componente ha un nome unico (come un URL). Utilizzando tali nomi, un componente può chiamare i servizi offerti da un altro componente, come se fosse una procedura o un metodo locale. Il middleware del sistema intercetta questa chiamata e la passa a un componente remoto. Questo esegue i calcoli richiesti e, tramite il middleware, restituisce il risultato al componente chiamante. In Java, le invocazioni dei metodi remoti (RMI) sono chiamate di procedure remote (CPR).

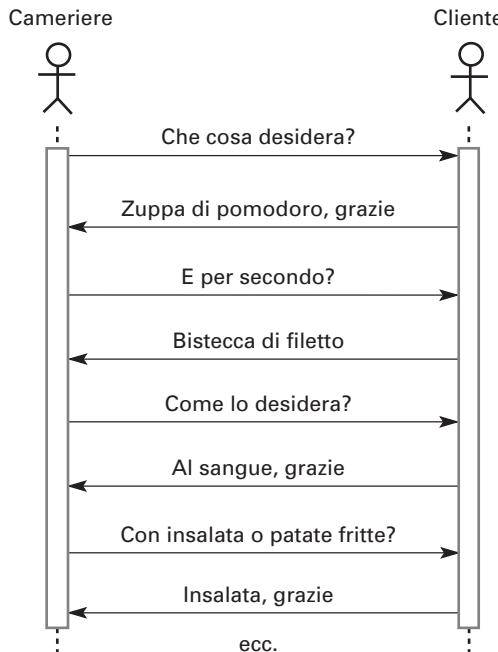


Figura 17.1 Interazione procedurale tra un cliente e un cameriere.

Le chiamate di procedure remote richiedono uno “stub” (una matrice di riconoscimento) affinché la procedura chiamata sia accessibile sul computer che ha effettuato la chiamata. Questo stub definisce l’interfaccia della procedura remota. Lo stub viene chiamato, poi traduce i parametri della procedura in una rappresentazione standard per la trasmissione alla procedura remota. Tramite il middleware invia la richiesta di esecuzione alla procedura remota; questa procedura usa le funzioni di libreria per convertire i parametri nel formato richiesto, esegue i calcoli e, poi, restituisce i risultati tramite lo stub che rappresenta il componente chiamante.

L’interazione basata sui messaggi di solito richiede un componente che crea un messaggio con i dettagli sui servizi richiesti da un altro componente. Questo messaggio viene inviato al componente ricevente tramite il middleware del sistema. Il ricevente analizza il messaggio, esegue i calcoli e crea un messaggio per il componente trasmittente con i risultati richiesti. Questo messaggio viene poi passato al middleware, che lo invia al componente trasmittente.

Un problema dell’approccio CPR all’interazione è che chiamante e chiamato devono essere disponibili durante la comunicazione; inoltre il chiamante deve sapere come fare riferimento al chiamato e viceversa. In sostanza, l’approccio CPR ha gli stessi requisiti delle chiamate di procedure o metodi locali. In un approccio basato sui messaggi, invece, non è tollerata l’indisponibilità. Se il com-

```

<primo>
    <nome piatto = "zuppa" tipo = "pomodoro" / >
    <nome piatto = "zuppa" tipo = "pesce" / >
    <nome piatto = "insalata di pollo" / >
</primo>
<secondo>
    <nome piatto = "bistecca" tipo = "braciola" cottura = "media" / >
    <nome piatto = "bistecca" tipo = "filetto" cottura = "al sangue" / >
    <nome piatto = "branzino" / >
</secondo>
<contorno>
    <piatto nome = "patatine fritte" porzioni = "2" / >
    <piatto nome = "insalata" porzioni = "1" / >
</contorno>

```

Figura 17.2 Interazione basata sui messaggi tra un cameriere e il personale della cucina.

ponente del sistema che sta elaborando il messaggio non è disponibile, il messaggio resta in coda finché il componente ricevente non ritornerà online. Inoltre, non è necessario che il trasmittente conosca il nome del ricevente per garantire che i messaggi siano trasmessi al sistema appropriato.

17.1.2 Middleware

I componenti di un sistema distribuito possono essere implementati in diversi linguaggi di programmazione e possono essere eseguiti su vari tipi di processori. I modelli di dati, la rappresentazione delle informazioni e i protocolli per la comunicazione possono essere tutti diversi. Un sistema distribuito, dunque, richiede un software in grado di gestire queste parti diverse e assicurare che queste comunichino e si scambino dati correttamente.

Per fare riferimento a questo software si utilizza il termine *middleware*, in quanto il software si trova *in mezzo* ai componenti distribuiti del sistema. Questo concetto è illustrato nella Figura 17.3, che mostra che il middleware è uno strato tra il sistema operativo e i programmi applicativi. Il middleware di solito viene implementato come una serie di librerie, che sono installate su ciascun computer distribuito, più un sistema a runtime che gestisce queste comunicazioni.

Bernstein (Bernstein 1996) descrive i tipi di middleware disponibili per il supporto al calcolo distribuito. Il middleware è un software universale che di solito viene acquistato come prodotto off-the-shelf, anziché essere scritto espressamente da sviluppatori di applicazioni. Esempi di middleware sono i programmi che gestiscono le comunicazioni con i database, i convertitori di dati e i controllori delle comunicazioni.

In un sistema distribuito, il middleware fornisce due tipi distinti di supporto.

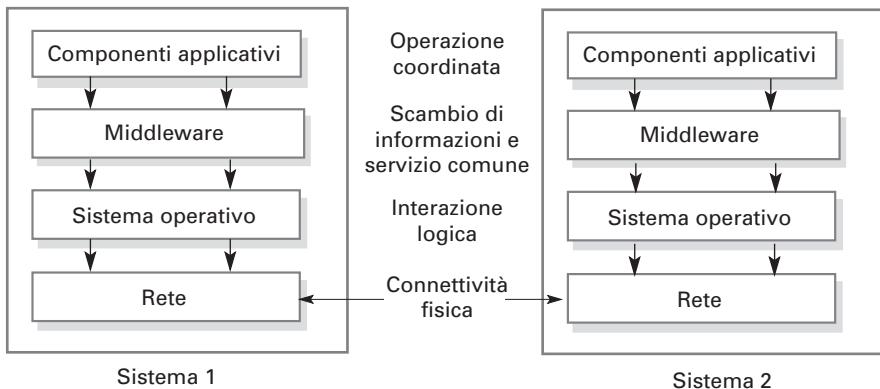


Figura 17.3 Il middleware in un sistema distribuito.

1. *Supporto alle interazioni.* Il middleware coordina le interazioni tra i vari componenti del sistema; fornisce la trasparenza delle posizioni, nel senso che non è necessario che ogni componente conosca le posizioni fisiche degli altri componenti. Può anche supportare la conversione dei parametri, se si usano linguaggi di programmazione differenti per implementare i componenti, la gestione degli eventi, la comunicazione e così via.
2. *Supporto ai servizi comuni.* Il middleware fornisce implementazioni riutilizzabili di servizi che possono essere richiesti da più componenti nel sistema distribuito. Utilizzando questi servizi comuni, i componenti possono interagire facilmente e offrire i servizi coerenti.

Nel Paragrafo 17.1.1 ho già fatto alcuni esempi di supporto alle interazioni che il middleware può fornire. Il middleware può essere utilizzato per supportare le chiamate di procedure o metodi remoti, lo scambio di messaggi e così via.

I servizi comuni sono quei servizi che possono essere richiesti da vari componenti indipendentemente delle funzioni svolte da questi componenti. Come detto nel Capitolo 16, questi servizi possono includere i servizi di protezione (autenticazione e autorizzazione), i servizi di notifica e gestione dei nomi, e i servizi di gestione delle transazioni. Per i componenti distribuiti, potete pensare che questi servizi siano forniti da un contenitore middleware; i servizi sono forniti tramite librerie condivise. Un componente, una volta installato, può accedere e utilizzare questi servizi comuni.

17.2 Calcolo client-server

I sistemi distribuiti, accessibili su Internet, sono organizzati come sistemi client-server. In un sistema client-server l'utente interagisce con un programma che viene eseguito sul suo computer locale, come un browser o un'app su un'unità

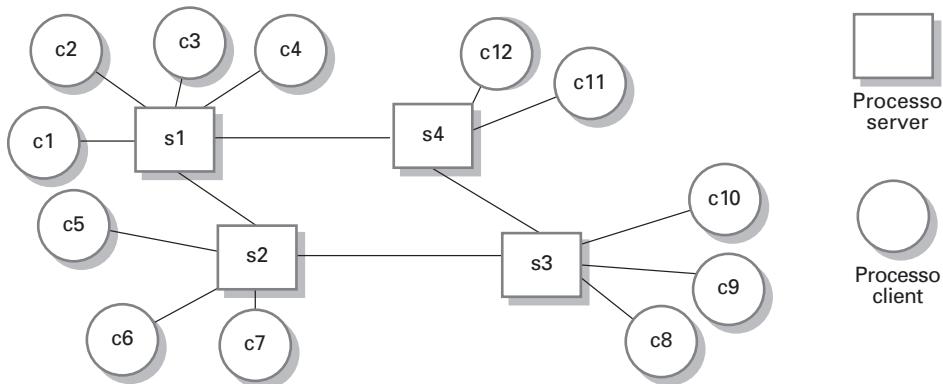


Figura 17.4 Interazione client-server.

mobile. Il programma interagisce con un altro programma eseguito su un computer remoto, come un server web. Il computer remoto fornisce i servizi, come l'accesso alle pagine web, che sono disponibili ai client esterni. Il modello client-server, descritto nel Capitolo 6, è un generico modello architettonale di un'applicazione. Non è limitato soltanto alle applicazioni distribuite su più macchine. È possibile utilizzarlo come un modello logico di interazioni, dove client e server sono eseguiti sullo stesso computer.

In un'architettura client-server un'applicazione è modellata come un insieme di servizi che sono forniti dai server. I client accedono a questi servizi e presentano i risultati agli utenti finali. I client devono conoscere i server che sono disponibili, ma non hanno bisogno di sapere nulla su altri client. Client e server sono processi separati, come mostra la Figura 17.4. Questa figura illustra una situazione in cui ci sono quattro server (s1-s4) che offrono servizi differenti. Ciascun servizio ha un insieme di client associati che accedono a tali servizi.

La Figura 17.4 mostra i processi client e server, anziché i processori. È normale che più processi client siano eseguiti su un singolo processore. Per esempio, sul vostro PC potete eseguire un client di posta elettronica che scarica le e-mail da un server remoto di posta elettronica. Potrete anche eseguire un browser che interagisce con un server remoto e un client di stampa che invia documenti a una stampante remota. La Figura 17.5 mostra una possibile configurazione, dove i 12 client logici illustrati nella Figura 17.4 vengono eseguiti su sei computer. I quattro processi server sono mappati su due computer fisici.

Diversi processi server possono essere eseguiti sullo stesso processore, ma spesso i server sono implementati come sistemi a più processori, nei quali un'istanza separata di un processo server viene eseguita su ciascuna macchina. Il software di bilanciamento del carico distribuisce le richieste di servizi fatte dai client ai vari server, in modo che ciascun server svolga la stessa quantità di lavoro. Questo permette di gestire un volume più elevato di transazioni con i client, senza ridurre la velocità di risposta ai singoli client.

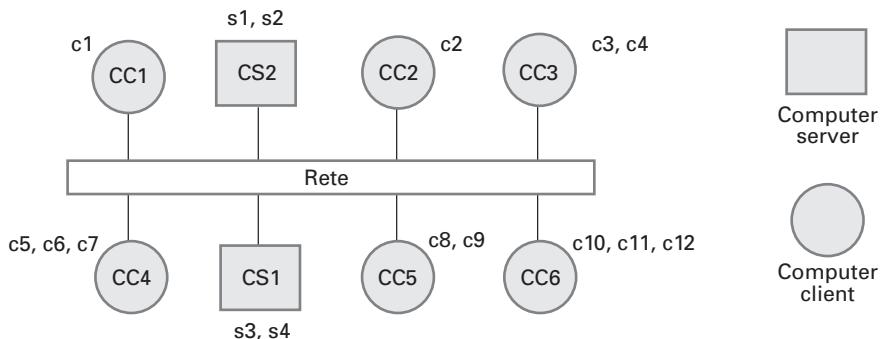


Figura 17.5 Mappatura di client e server su computer in rete.

I sistemi client-server dipendono dall'esistenza di una netta separazione tra la presentazione delle informazioni e i calcoli che creano ed elaborano tali informazioni. Di conseguenza, l'architettura dei sistemi client-server distribuiti dovrebbe essere progettata in modo che i sistemi siano strutturati in più strati logici, con interfacce chiare tra questi strati. Questo consente di distribuire ciascuno strato su un computer differente. La Figura 17.6 illustra questo modello, mostrando un'applicazione strutturata in quattro strati.

1. *Strato di presentazione.* Presenta le informazioni all'utente e gestisce tutte le interazioni dell'utente.
2. *Strato di gestione dei dati.* Gestisce i dati che vengono scambiati con il client. Questo strato può implementare i controlli sui dati, generare pagine web e così via.
3. *Strato di elaborazione delle applicazioni.* Implementa la logica delle applicazioni, fornendo le funzionalità richieste agli utenti finali.
4. *Strato del database.* Memorizza i dati, gestisce le transazioni e fornisce i servizi per le interrogazioni del database.

Il successivo paragrafo spiega come varie architetture client-server distribuiscono questi strati logici in modi differenti. Il modello client-server include anche il concetto di software come servizio (SaaS, Software as a Service), un modo importante di sviluppare il software e di accedere ad esso tramite Internet. Questo argomento è trattato nel Paragrafo 17.4.

17.3 Schemi architetturali per sistemi distribuiti

Come detto all'inizio di questo capitolo, i progettisti dei sistemi distribuiti devono organizzare i loro progetti in modo da trovare un bilanciamento tra prestazioni, fidatezza, protezione e gestibilità dei sistemi. Poiché non esiste un modello uni-



Figura 17.6 Modello architetturale a strati per un'applicazione client-server.

versale di organizzazione per tutti i sistemi, si sono diffusi vari stili architetturali per i sistemi distribuiti. Quando si progetta un'applicazione distribuita, si deve scegliere uno stile architettonico che supporta i requisiti non funzionali critici del sistema da sviluppare.

In questo paragrafo, ho discusso cinque stili architetturali.

1. *Architettura master-slave.* È utilizzata nei sistemi real-time in cui devono essere garantiti i tempi di risposta delle interazioni.
2. *Architettura client-server a due livelli* (two-tier). È utilizzata per sistemi client-server semplici e nei casi in cui è importante centralizzare il sistema per ragioni di protezione.
3. *Architettura client-server a più livelli* (multi-tier). È utilizzata quando il server deve elaborare un grande volume di transazioni.
4. *Architettura di componenti distribuiti.* È utilizzata quando è necessario combinare insieme le risorse di vari sistemi e database, o come modello di implementazione per i sistemi client-server multi-tier.
5. *Architettura peer-to-peer.* È utilizzata quando i client si scambiano localmente le informazioni memorizzate e il ruolo del server consiste nel presentare un client a un altro client. Può essere utilizzata anche quando devono essere eseguiti numerosi calcoli indipendenti.

17.3.1 Architetture master-slave

Le architetture master-slave per i sistemi distribuiti sono utilizzate di solito nei sistemi real-time. In questi sistemi ci possono essere appositi processori che sono preposti all'acquisizione dei dati dall'ambiente operativo del sistema, all'elaborazione dei dati e alla gestione degli attuatori. Gli attuatori sono dispositivi controllati dal software del sistema che agiscono per cambiare l'ambiente del sistema. Per esempio, un attuatore potrebbe controllare una valvola e modificare il suo stato da “aperta” a “chiusa”. Il processo “master” (principale) di solito è respon-

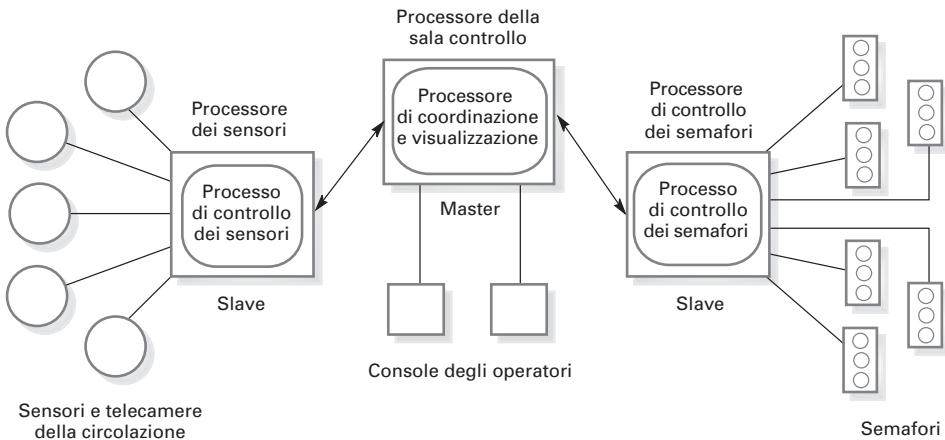


Figura 17.7 Un sistema di gestione del traffico con un'architettura master-slave.

sabile dei calcoli, della coordinazione e delle comunicazioni, e controlla i processi “slave” (secondari). I processi “slave” svolgono azioni specifiche, come l’acquisizione di dati da un gruppo di sensori.

La Figura 17.7 illustra un esempio di questo modello architettonico. Un sistema per il controllo del traffico in una città ha tre processi logici che vengono eseguiti su processori separati. Il processo master è quello della sala controllo, che comunica con processi slave distinti, che sono responsabili dell’acquisizione dei dati e della gestione dei semafori.

Un gruppo di sensori distribuiti raccoglie informazioni sul flusso del traffico. Il processo di controllo dei sensori scansiona periodicamente i sensori per acquisire informazioni sul traffico; memorizza queste informazioni per analizzarle successivamente. Anche il processore dei sensori viene scansionato periodicamente dal processo master, che ha il compito di visualizzare lo stato del traffico agli operatori, calcolare le sequenze dei semafori e accettare i comandi degli operatori per modificare queste sequenze. Il sistema della sala controllo invia i comandi a un processo di controllo dei semafori, che converte questi comandi in segnali che controllano l’hardware dei semafori. Anche il sistema master della sala controllo è organizzato come sistema client-server, con i processi client che vengono eseguiti sulle console degli operatori.

Il modello master-slave di un sistema distribuito viene utilizzato nei casi in cui è possibile prevedere il calcolo distribuito richiesto e quando l’elaborazione può essere facilmente localizzata nei processori slave. Queste situazioni sono tipiche dei sistemi real-time, dove è importante rispettare con esattezza i tempi di elaborazione. I processori slave possono essere utilizzati per svolgere intense operazioni di calcolo, come l’elaborazione dei segnali e la gestione di dispositivi controllati dal sistema.

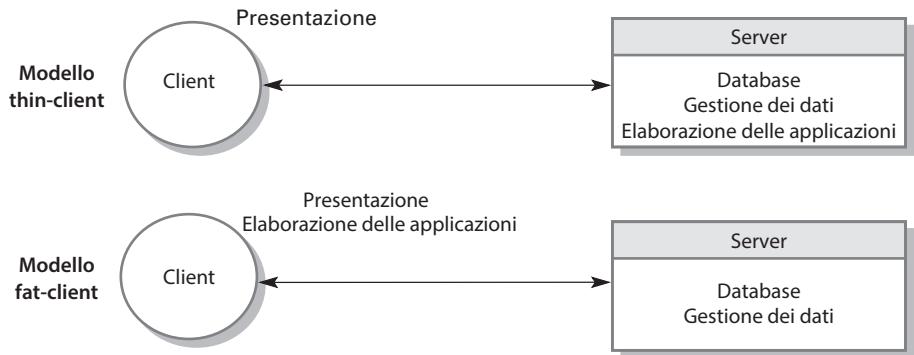


Figura 17.8 Modelli architettonici thin-client e fat-client.

17.3.2 Architetture client-server a due livelli

Nel Paragrafo 17.2 ho spiegato l'organizzazione generale dei sistemi client-server, in cui una parte del sistema viene eseguita sul computer dell'utente (il client) e un'altra parte viene eseguita su un computer remoto (il server). Ho anche presentato un modello di applicazioni a strati (Figura 17.6), dove i vari strati di un sistema possono essere eseguiti su computer differenti.

L'architettura client-server a due livelli è la forma più semplice di architettura client-server. Il sistema è implementato come un singolo server logico più un numero indefinito di client che usano tale server. Questo è illustrato nella Figura 17.8, che mostra due forme di questo modulo architettonico.

1. **Modello thin-client** (client leggero). Lo strato di presentazione è implementato sul client e tutti gli altri strati (gestione dei dati, elaborazione dei processi e database) sono implementati sul server. Il software di presentazione del client di solito è un browser web, ma sono disponibili anche le app per le unità mobili.
2. **Modello fat-client** (client pesante). Alcune o tutte le elaborazioni delle applicazioni vengono svolte sul client. La gestione dei dati e le funzioni del database sono implementate sul server. In questo caso, il software del client può essere un programma appositamente scritto per essere strettamente integrato nell'applicazione del server.

Il vantaggio del modello thin-client è che è semplice gestire i client. Questo diventa un problema importante quando ci sono numerosi client, in quanto potrebbe essere difficile e costoso installare un nuovo software su tutti i client. Se un browser web viene utilizzato come client, non occorre installare alcun software.

Lo svantaggio del metodo thin-client è che esso pone un pesante carico di elaborazione sia sul server sia sulla rete. Il server è responsabile di tutti i calcoli, e questo può generare un significativo traffico nella rete tra il client e il server.

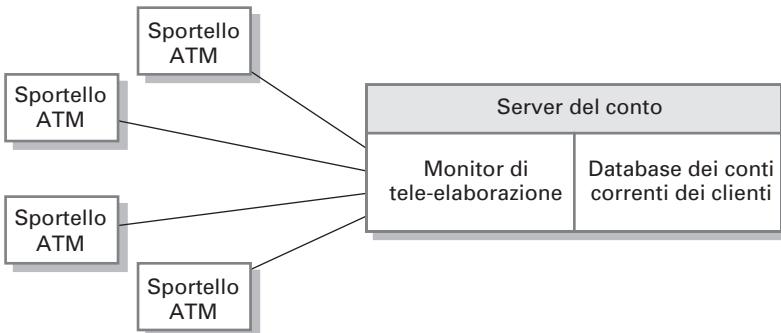


Figura 17.9 Un'architettura fat-client per un sistema ATM.

Implementare un sistema che usa questo modello potrebbe quindi richiedere ulteriori investimenti nella capacità del server e della rete.

Il modello fat-client fa uso della potenza di elaborazione disponibile sul computer che esegue il software del client e distribuisce alcune o tutte le elaborazioni delle applicazioni e la presentazione al client. Il server è essenzialmente un server di transazioni che gestisce tutte le transazioni del database. La gestione dei dati è semplice, in quanto non è necessario gestire le interazioni tra il client e il sistema che elabora le applicazioni. Il modello fat-client richiede che chi gestisce il sistema installi e mantenga il software sul computer client.

Un esempio di architettura fat-client è il sistema che gestisce le unità ATM (Automated Teller Machine, meglio note come bancomat), che offre agli utenti la possibilità di prelevare contanti e altri servizi. Lo sportello bancomat è il computer client, mentre il server tipicamente è un mainframe che gestisce il database dei conti correnti dei clienti. Un mainframe è una macchina potente che è progettata per elaborare le transazioni; può quindi gestire grandi volumi di transazioni generate dagli sportelli bancomat e da altre unità online. Il software in una macchina ATM svolge varie elaborazioni associate a una transazione.

La Figura 17.9 mostra una versione semplificata dell'organizzazione di un sistema ATM. Si noti che gli sportelli bancomat non sono collegati direttamente al database dei clienti, ma a un monitor di tele-elaborazione (TP). Un monitor TP è un sistema middleware che organizza le comunicazioni con i client remoti e serializza le transazioni dei client affinché siano elaborate dal database. Questo garantisce che le transazioni siano indipendenti, senza interferenze fra di loro. L'uso delle transazioni seriali consente al sistema di riprendersi da un guasto senza danneggiare i dati.

Sebbene il modello fat-client distribuisca le elaborazioni in modo più efficace del modello thin-client, tuttavia la gestione del sistema è più complessa se viene utilizzato un client specializzato, anziché un browser. Le funzionalità dell'applicazione sono distribuite su molti computer. Se il software dell'applicazione deve essere modificato, è necessario reinstallare il software su tutti i computer client.

Questo può implicare costi notevoli se ci sono centinaia di client nel sistema. L'aggiornamento automatico del software dei client può ridurre questi costi, ma introduce i suoi problemi se le funzionalità dei client devono essere modificate. Le nuove funzionalità potrebbero obbligare le aziende a cambiare il modo in cui usano il sistema.

L'uso sempre più diffuso di unità mobili significa che è importante minimizzare il traffico della rete quando possibile. Questi dispositivi adesso includono potenti computer in grado di svolgere elaborazioni locali. Di conseguenza, la distinzione tra architetture thin-client e fat-client non è più così netta. Le app hanno funzionalità incorporate che svolgono elaborazioni locali, e le pagine web possono includere componenti Javascript che vengono eseguiti sul computer locale dell'utente. Resta il problema dell'aggiornamento delle app, anche se è stato risolto in qualche misura aggiornando automaticamente le app senza l'intervento esplicito dell'utente. Ne consegue che, sebbene a volte sia conveniente utilizzare questi modelli come base generica per l'architettura di un sistema distribuito, in pratica però le applicazioni basate sul web implementano tutte le elaborazioni sul server remoto.

17.3.3 Architetture client-server a più livelli

Il problema fondamentale dell'approccio client-server a due livelli è che gli strati logici del sistema (presentazione, elaborazione delle applicazioni, gestione dei dati e database) devono essere mappati su due computer: il client e il server. Potrebbero sorgere problemi di scalabilità e di prestazioni se si sceglie il modello thin-client, o problemi di gestione del sistema se si sceglie il modello fat-client. Per evitare qualcuno di questi problemi, si può utilizzare l'architettura “client-server a più livelli”. In questa architettura i vari strati del sistema (presentazione, gestione dei dati, elaborazione delle applicazioni e database) sono processi separati che possono essere eseguiti su processori differenti.

Un sistema di Internet banking (Figura 17.10) è un esempio di architettura client-server a tre livelli. Il database dei clienti della banca (solitamente ospitato su un mainframe) fornisce i servizi di database. Un server web fornisce i servizi di gestione dei dati, quali la generazione delle pagine web e qualche servizio applicativo. I servizi applicativi, come il trasferimento del denaro, la generazione degli estratti conto, il pagamento delle fatture e così via, sono implementati nel server web e vengono eseguiti come script dal client. Il computer dell'utente dotato di un browser per Internet è il client. Questo sistema è scalabile, perché è relativamente facile aggiungere nuovi server web (scaling out) se aumenta il numero di clienti.

In questo caso, l'uso di un'architettura a tre livelli permette di ottimizzare lo scambio delle informazioni tra il server web e il server del database. Per gestire le informazioni del database si usa un middleware efficiente che supporta le interrogazioni dei database in SQL (Structured Query Language, linguaggio di interrogazione strutturato).

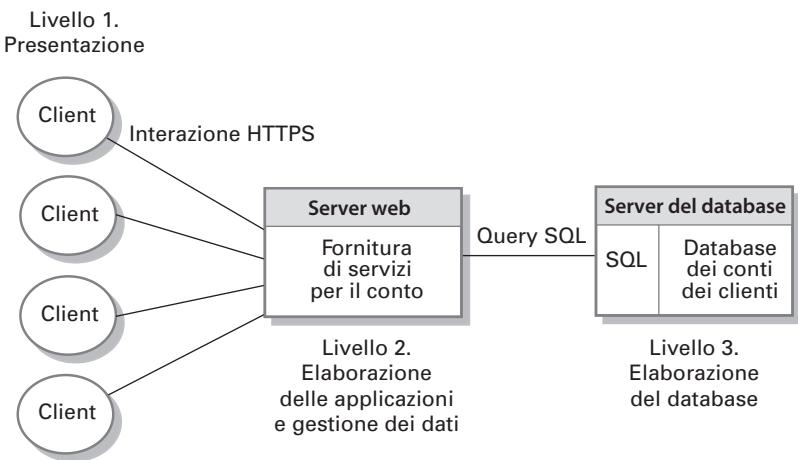


Figura 17.10 Architettura client-server a tre livelli per un sistema di Internet banking.

Il modello client-server a tre livelli può essere esteso a un modello a più livelli, aggiungendo nuovi server al sistema. Per farlo, potrebbe essere necessario utilizzare un server web per la gestione dei dati e altri server per l'elaborazione delle applicazioni e i servizi del database. I sistemi a più livelli possono essere utilizzati anche quando le applicazioni richiedono l'accesso e l'utilizzo di dati di più database. In questo caso, occorre aggiungere al sistema un server di integrazione, che ha il compito di raccogliere i dati distribuiti e presentarli al server dell'applicazione come se provenissero da un unico database. Come dirò nel prossimo paragrafo, le architetture di componenti distribuiti possono essere utilizzate per implementare sistemi client-server a più livelli.

I sistemi client-server a più livelli che distribuiscono l'elaborazione delle applicazioni su più server sono intrinsecamente più scalabili delle architetture a due livelli. I livelli nel sistema possono essere gestiti in maniera indipendente, aggiungendo nuovi server quando il carico aumenta. L'elaborazione può essere distribuita fra il server dell'applicazione logica e il server per la gestione dei dati, ottenendo risposte più rapide alle richieste degli utenti.

I progettisti delle architetture client-server devono valutare una serie di fattori quando scelgono l'architettura di distribuzione più adatta. I casi in cui le architetture client-server qui descritte possono essere utilizzate sono riportati nella Figura 17.11.

17.3.4 Architetture di componenti distribuiti

Organizzando i processi in strati, come mostrato nella Figura 17.6, ogni strato di un sistema può essere implementato come un server logico separato. Questo modello si adatta bene a molti tipi di applicazioni; tuttavia, limita la flessibilità dei progettisti del sistema, in quanto questi devono decidere quali servizi devono

Architettura	Applicazioni
Architettura client-server a due livelli con thin-client	<p>Applicazioni di sistemi ereditati, dove non è possibile separare l'elaborazione delle applicazioni dalla gestione dei dati. I client possono accedere a queste applicazioni come servizi, secondo le modalità descritte nel Paragrafo 17.4.</p> <p>Applicazioni di calcolo intenso, come i compilatori con gestione di dati minima o inesistente.</p> <p>Applicazioni con grandi volumi di dati (ricerca e interrogazione), ma con elaborazione minima. La semplice navigazione nel Web è il tipico esempio di utilizzazione di questa architettura.</p>
Architettura client-server a due livelli con fat-client	<p>Applicazioni dove l'elaborazione fornita da software off-the-shell (per esempio, Microsoft Excel) avviene sul client.</p> <p>Applicazioni dove è richiesta un'elaborazione con calcoli intensivi sui dati (per esempio, la visualizzazione di dati).</p> <p>Applicazioni mobili dove la connessione a Internet non può essere garantita. È quindi possibile un'elaborazione locale che usa le informazioni del database memorizzate nella cache.</p>
Architettura client-server a più livelli	<p>Applicazioni su vasta scala con centinaia o migliaia di client.</p> <p>Quando i dati e le applicazioni sono entrambi volatili.</p> <p>Applicazioni dove i dati vengono integrati da più sorgenti.</p>

Figura 17.11 Uso degli schemi architettonici client-server.

essere inclusi in ciascuno strato. In pratica, però, non è sempre chiaro se un servizio è un servizio per la gestione dei dati, per le applicazioni o per i database. I progettisti devono anche pianificare la scalabilità e quindi prevedere in quale modo i server dovranno essere duplicati quando nuovi client vengono aggiunti al sistema.

Un approccio più generale alla progettazione dei sistemi distribuiti consiste nel progettare un sistema come un insieme di servizi, evitando di allocare questi servizi agli strati del sistema. Ciascun servizio, o gruppo di servizi correlati, può essere implementato utilizzando un oggetto o componente separato. In un'architettura di componenti distribuiti (Figura 17.12), il sistema è organizzato come un insieme di componenti che interagiscono tra loro, come descritto nel Capitolo 16. Questi componenti forniscono un'interfaccia all'insieme dei servizi che essi offrono. Altri componenti chiamano questi servizi attraverso il middleware, effettuando chiamate di procedure o metodi remoti.

I sistemi di componenti distribuiti dipendono dal middleware, il cui ruolo è gestire le interazioni dei componenti, riconciliare le differenze tra i vari tipi di parametri scambiati tra i componenti e fornire una serie di servizi comuni che possono essere utilizzati dai componenti. Lo standard CORBA (Orfali, Harkey e

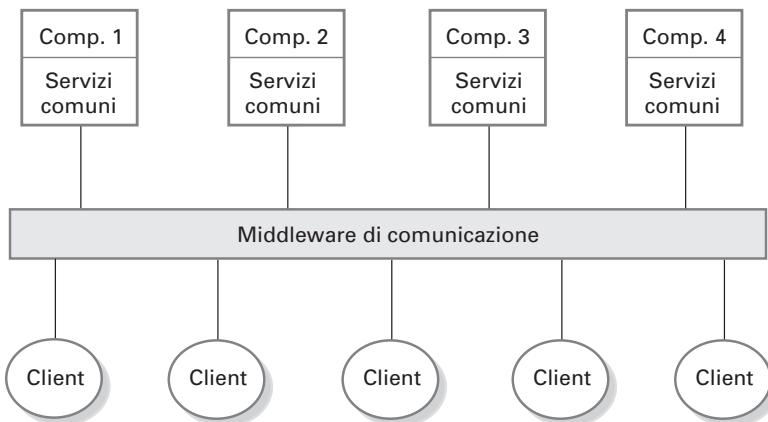


Figura 17.12 Architettura di componenti distribuiti.

Edwards 1997) ha definito il middleware per i sistemi di componenti distribuiti, anche se le implementazioni CORBA non sono largamente adottate. Le aziende preferiscono utilizzare il software proprietario, come Enterprise Java Beans (EJB) o .NET.

Utilizzando il modello di componenti distribuiti per implementare i sistemi distribuiti si ottengono diversi vantaggi.

1. I progettisti del sistema possono ritardare le decisioni su dove e come i servizi devono essere forniti. I componenti che forniscono i servizi possono essere eseguiti su un nodo qualsiasi della rete. Non è necessario decidere in anticipo se un servizio debba far parte dello strato di gestione dei dati, dello strato delle applicazioni o dello strato dell'interfaccia utente.
2. L'architettura del sistema è molto aperta, in quanto permette di aggiungere nuove risorse quando servono. È facile aggiungere nuovi servizi al sistema, senza grandi modifiche del sistema esistente.
3. Il sistema è flessibile e scalabile. È possibile aggiungere nuovi oggetti o duplicare quelli esistenti quando il carico del sistema aumenta, senza danneggiare altre parti del sistema.
4. È possibile riconfigurare il sistema dinamicamente con i componenti che migrano attraverso la rete quando è necessario. Questo potrebbe essere importante quando ci sono richieste di servizi fluttuanti. Un componente che fornisce servizi può migrare sullo stesso processore come oggetto che richiede servizi, migliorando quindi le prestazioni del sistema.

Un'architettura a componenti distribuiti può essere usata come modello logico che permette di strutturare e organizzare il sistema. In questo caso, occorre stabilire come fornire le funzionalità dell'applicazione soltanto in termini di servizi e

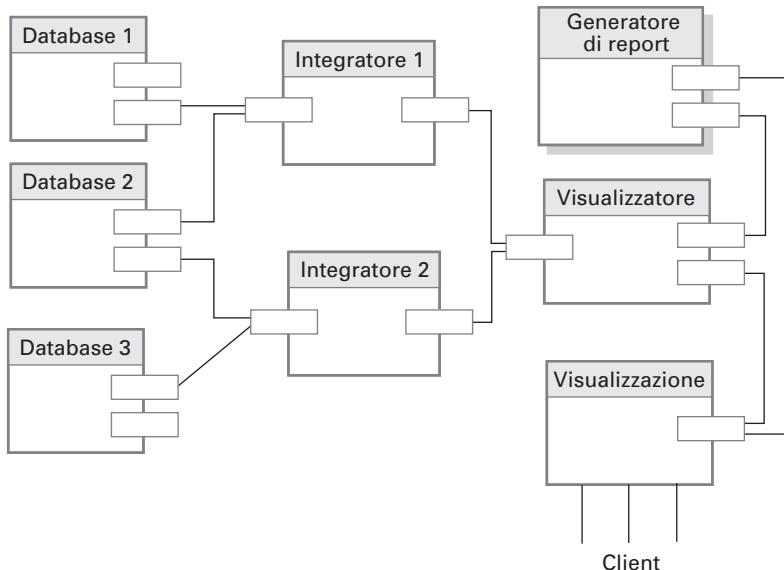


Figura 17.13 Architettura di componenti distribuiti per un sistema di data mining.

combinazioni di servizi; poi, si provvede a implementare tali servizi. Per esempio, in un'applicazione di vendite al dettaglio ci possono essere componenti che si occupano del controllo delle scorte, delle comunicazioni con i clienti, dell'ordinazione delle merci e così via.

I sistemi di data mining sono un buon esempio di un tipo di sistema che può essere implementato utilizzando un'architettura di componenti distribuiti. I sistemi di data mining cercano le relazioni tra i dati che possono essere distribuiti in vari database (Figura 17.13). Questi sistemi estraggono le informazioni da vari database, eseguono numerose elaborazioni e presentano visualizzazioni, facili da capire, delle relazioni che hanno scoperto.

Un esempio di sistema applicativo di data mining potrebbe essere un sistema di vendite al dettaglio di generi alimentari e libri. Le aziende hanno creato database distinti per memorizzare informazioni dettagliate sui prodotti alimentari e sui libri. Utilizzano un sistema di carte di fedeltà per tenere traccia degli acquisti dei clienti; c'è un grosso database che collega i codici a barre dei prodotti con le informazioni sui clienti. Il dipartimento di marketing vuole trovare le relazioni tra gli acquisti dei generi alimentari e quelle dei libri. Per esempio, potrebbe esserci un percentuale relativamente elevata di persone che acquistano la pizza e che comprano anche romanzi polizieschi. Sulla base di queste conoscenze, le aziende potrebbero informare i clienti che acquistano determinati cibi sui nuovi libri che vengono pubblicati.

In questo esempio, ogni database con i dati delle vendite può essere incapsulato come componente distribuito con un'interfaccia che fornisce accesso di sola

lettura ai propri dati. I componenti integratori si occupano di specifici tipi di relazioni; raccolgono informazioni da tutti i database per tentare di dedurre qualche relazione. Potrebbe esserci un componente integratore che si occupa delle variazioni stagionali dei prodotti venduti e un altro che si occupa delle relazioni tra diversi tipi di prodotti.

I componenti visualizzatori interagiscono con gli integratori e generano una visualizzazione o un report sulle relazioni che sono state trovate. A causa della grande quantità di dati da gestire, i visualizzatori di solito rappresentano graficamente i loro risultati. Infine, un componente di visualizzazione potrebbe avere il compito di fornire i modelli grafici ai client per la rappresentazione finale nei loro browser web.

Un’architettura di componenti distribuiti, diversamente da un’architettura a strati, è appropriata per questo tipo di applicazioni, in quanto è possibile aggiungere nuovi database al sistema senza apportare modifiche significative. L’accesso a ogni nuovo database avviene semplicemente aggiungendo un altro componente distribuito. I componenti per l’accesso ai database forniscono un’interfaccia semplificata che controlla l’accesso ai dati. Questi database possono trovarsi su macchine differenti. L’architettura semplifica anche la scoperta (mining) di nuovi tipi di relazioni aggiungendo nuovi oggetti integratori.

Le architetture di componenti distribuiti presentano essenzialmente due svantaggi.

1. Sono più complesse da progettare rispetto ai sistemi client-server. I sistemi client-server a più strati sembrano essere un modo quasi intuitivo di pensare ai sistemi; rispecchiano molte transazioni umane, quando le persone chiedono e ricevono servizi da altre persone che sono specializzate nel fornire tali servizi. La complessità delle architetture di componenti distribuiti aumenta i costi di implementazione.
2. Non esistono standard universali per il middleware o i modelli dei componenti distribuiti; anzi, vari produttori, come Microsoft e Sun, hanno sviluppato middleware incompatibili. Il middleware è complesso, e la dipendenza da esso aumenta significativamente la complessità dei sistemi di componenti distribuiti.

A causa di questi problemi, le architetture di componenti distribuiti sono state sostituite dai sistemi orientati ai servizi (descritti nel Capitolo 18). Tuttavia, i sistemi di componenti distribuiti presentano migliori performance rispetto ai sistemi orientati ai servizi. Le comunicazioni CPR di solito sono più veloci delle interazioni basate sui messaggi che sono utilizzate nei sistemi orientati ai servizi. Le architetture di componenti distribuiti sono quindi ancora utilizzate nei sistemi ad alto throughput, dove occorre elaborare rapidamente un gran numero di transazioni.

17.3.5 Architetture peer-to-peer

Il modello di calcolo client-server, che ho descritto nei precedenti paragrafi di questo capitolo, fa una distinzione chiara tra i server, che sono fornitori di servizi, e i client, che sono fruitori di servizi. Questo modello di solito porta a una distribuzione sbilanciata del carico sul sistema, dove i server svolgono più lavoro dei client. Questo potrebbe richiedere alle aziende un notevole lavoro di organizzazione delle capacità dei server, mentre resta inutilizzata la capacità di elaborazione di centinaia o migliaia di personal computer e unità mobili che vengono utilizzati per accedere ai server del sistema.

I sistemi peer-to-peer (p2p) sono sistemi decentralizzati (Oram 2001), nei quali i calcoli possono essere eseguiti da qualsiasi nodo della rete e, almeno in teoria, non ci sono distinzioni tra client e server. Nelle applicazioni peer-to-peer, il sistema generale viene progettato per sfruttare la potenza di calcolo e la capacità di memoria che sono disponibili su una vasta rete di computer. Gli standard e i protocolli che consentono la comunicazione tra i nodi sono integrati nell'applicazione stessa, e ogni nodo deve eseguire una copia di tale applicazione.

Le tecnologie peer-to-peer sono state utilizzate principalmente per i sistemi personali, anziché per i sistemi aziendali. Il fatto che non ci siano server centrali significa che questi sistemi sono più difficili da monitorare; quindi, è possibile raggiungere livelli più elevati nella privacy delle comunicazioni.

Per esempio, i sistemi di condivisione dei file basati sul protocollo BitTorrent sono largamente utilizzati per scambiare i file sui PC degli utenti. I sistemi di messaggistica istantanea, come ICQ e Jabber, forniscono una comunicazione diretta tra gli utenti senza un server intermedio. Bitcoin è un sistema di pagamento peer-to-peer che usa la valuta elettronica Bitcoin. Freenet è un database centralizzato che è stato progettato per semplificare la pubblicazione anonima di informazioni e per rendere difficile alle autorità la loro soppressione.

Altri sistemi p2p sono stati sviluppati nei casi in cui la privacy non è un requisito importante. I servizi telefonici VoIP (Voice over IP, ovvero “voce tramite protocollo Internet”), come Viber, si basano sulle comunicazioni peer-to-peer tra le parti coinvolte in una conferenza o chiamata telefonica. SETI@home è un progetto che elabora i dati provenienti dai radiotelescopi sui PC di casa con lo scopo di cercare segnali di vita extraterrestre. In questi sistemi, il vantaggio del modello p2p è che un server centrale non costituisce un collo di bottiglia delle elaborazioni.

I sistemi peer-to-peer sono stati utilizzati anche dalle aziende per sfruttare la potenza delle loro reti di PC (McDougall 2000). Intel e Boeing hanno implementato sistemi p2p per applicazioni con calcoli intensivi. Tali sistemi sfruttano la capacità di elaborazione inutilizzata sui computer locali. Anziché comprare costose unità hardware di prestazioni elevate, i calcoli scientifici possono essere svolti di notte quando i personal computer non sono utilizzati. Le aziende hanno fatto anche un uso estensivo di sistemi p2p commerciali, come i sistemi VoIP e di messaggistica.

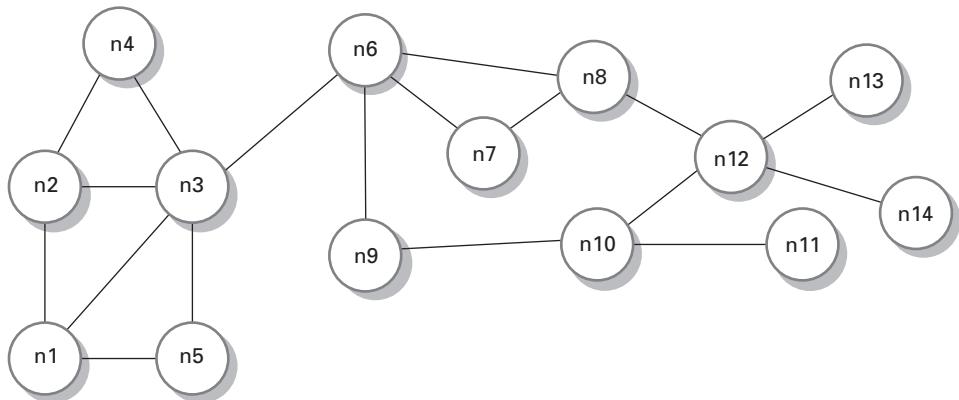


Figura 17.14 Architettura p2p decentralizzata.

In teoria, ogni nodo in una rete p2p può essere a conoscenza di tutti gli altri nodi. Un nodo può connettersi e scambiare dati direttamente con qualsiasi altro nodo della rete. In pratica, questo è impossibile, a meno che la rete non abbia solo pochi membri. Di conseguenza, i nodi vengono organizzati in “località”, con alcuni nodi che fungono da ponte con altre località. La Figura 17.14 mostra questa architettura p2p decentralizzata.

In un’architettura decentralizzata, i nodi della rete non sono semplici elementi funzionali, ma anche commutatori di comunicazione che indirizzano i dati e i segnali di controllo da un nodo all’altro. Per esempio, si supponga che la Figura 17.14 rappresenti un sistema decentralizzato per la gestione di documenti. Un gruppo di ricercatori utilizza questo sistema per condividere i documenti. Ciascun ricercatore mantiene la propria cartella di documenti. Quando un documento viene richiesto, il nodo che lo carica, lo rende disponibile a tutti gli altri nodi.

Se qualcuno ha bisogno di un documento che è memorizzato in qualche altra parte della rete, emette un comando di ricerca, che viene indirizzato ai nodi nelle loro “località”. Questi nodi controllano se hanno il documento e, in caso affermativo, lo restituiscono al richiedente. Se non hanno il documento, indirizzano la ricerca su altri nodi. Per esempio, se il nodo n1 emette un comando di ricerca per un documento salvato nel nodo n10, la ricerca passerà attraverso i nodi n3, n6 e n9 fino a n10. Quando il documento viene trovato, il nodo che lo contiene lo indirizza direttamente al nodo che lo ha richiesto, effettuando una connessione peer-to-peer.

Questa architettura decentralizzata ha il vantaggio di essere molto ridondante e, quindi, tollerante agli errori e ai nodi che si disconnettono dalla rete. Tuttavia, ci sono alcuni svantaggi: il primo è che più nodi possono elaborare lo stesso ordine di ricerca; il secondo è un aumento significativo degli overhead dovuto alle comunicazioni replicate tra i peer.

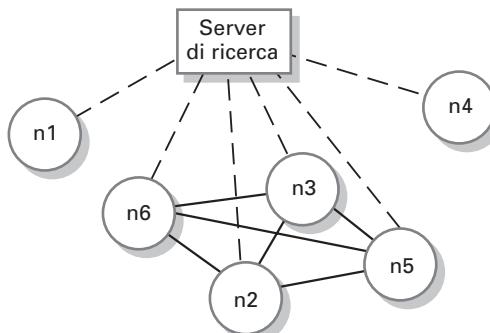


Figura 17.15 Architettura p2p semicentralizzata.

Un modello architettonale p2p alternativo, che si discosta dalla pura architettura p2p, è l’architettura semicentralizzata, dove uno o più nodi nella rete fungono da server per semplificare le comunicazioni tra i nodi. In questo modo si riduce l’intensità del traffico nei nodi. La Figura 17.15 illustra come questo modello architettonale semicentralizzato differisce dal modello completamente decentralizzato illustrato nella Figura 17.14.

In un’architettura semicentralizzata il ruolo del server (detto *superpeer*) consiste nell’agevolare la connessione tra i peer nella rete o nel coordinare i risultati dei calcoli. Per esempio, se la Figura 17.15 rappresenta un sistema di messaggistica istantanea, allora i nodi della rete comunicano con il server (secondo le linee tratteggiate) per trovare quali altri nodi sono disponibili. Una volta trovati questi nodi, può essere stabilita una comunicazione diretta tra i nodi e la connessione con il server diventa superflua. Per questo motivo i nodi n2, n3, n5 e n6 sono in comunicazione diretta.

Nei sistemi di calcolo p2p, dove un calcolo intensivo viene distribuito su un gran numero di nodi, è normale che alcuni nodi siano superpeer. Il loro ruolo è distribuire il lavoro agli altri nodi, e poi raccogliere e verificare i risultati dei calcoli.

Il modello architettonale peer-to-peer può essere il miglior modello per un sistema distribuito in due casi.

1. Quando il sistema è ad alta intensità di calcolo ed è possibile separare l’elaborazione richiesta in un gran numero di calcoli indipendenti. Per esempio, un sistema peer-to-peer che supporta la ricerca di farmaci distribuisce i calcoli che ricercano potenziali trattamenti dei tumori analizzando un gran numero di molecole per vedere se hanno le caratteristiche richieste per arrestare lo sviluppo dei tumori. Ciascuna molecola può essere considerata in modo separato, quindi non è necessario che i peer del sistema comunichino tra loro.

2. Quando il sistema richiede principalmente lo scambio di informazioni tra computer indipendenti in una rete e non è necessario che queste informazioni siano memorizzate o gestite centralmente. Esempi di queste applicazioni includono i sistemi di condivisione dei file e i sistemi telefonici che supportano le comunicazioni audio e video tra i computer.

Le architetture peer-to-peer consentono di utilizzare in modo efficiente le capacità di una rete. I problemi di protezione sono il motivo principale per cui questi sistemi non si sono ampiamente diffusi, specialmente nelle aziende (Wahlach 2003). La mancanza di una gestione centralizzata significa che i pirati informatici possono attaccare alcuni nodi per diffondere spam e malware. Le comunicazioni peer-to-peer richiedono il consenso dei vostri computer alle interazioni dirette con altri peer, e questo significa che questi sistemi potrebbero accedere alle vostre risorse. Per contrastare questa evenienza, dovete organizzare i vostri sistemi in modo da proteggere le loro risorse. Se questo è fatto male, i vostri sistemi diventano vulnerabili agli attacchi esterni.

17.4 Software come servizio

Nei precedenti paragrafi ho descritto i modelli client-server e come sia possibile distribuire le funzionalità tra client e server. Per implementare un sistema client-server, occorre installare un programma o un'app nel computer client, che comunica con il server, implementa le funzionalità lato client, e gestisce l'interfaccia utente. Per esempio, un client di posta elettronica, come Outlook o Mac Mail, offre le funzioni per la gestione della posta sul vostro computer. Questo evita il problema di sovraccaricare il server nei sistemi thin-client, dove tutte le elaborazioni sono svolte dal server.

I problemi del sovraccarico del server possono essere ridotti significativamente utilizzando le tecnologie del Web, come AJAX (Holdener, 2008) e HTML5 (Sarris 2013). Queste tecnologie supportano una gestione efficiente della presentazione delle pagine web e del calcolo locale, eseguendo script che sono parte delle pagine web. Questo significa che un browser può essere configurato e usato come client, con una significativa elaborazione locale. Il software applicativo può essere pensato come un servizio remoto, che può essere utilizzato da qualsiasi unità dove può essere eseguito un browser standard. Esempi di software come servizio (SaaS, Software as a Service) sono i sistemi di posta elettronica basata sul Web, come Yahoo e Gmail, e le applicazioni per gli uffici, come Google Docs e Office 365.

Questa idea di software come servizio richiede che il software sia ospitato in una unità remota, accessibile tramite Internet. Gli elementi chiave del software come servizio sono i seguenti.

1. Il software è installato in un server (o più comunemente nel cloud) che è accessibile tramite un browser web. Non viene installato in un PC locale;
2. Il software è gestito da un fornitore di software (che è anche proprietario del software), anziché dalle organizzazioni che lo usano;
3. Gli utenti pagano il software in funzione dell'uso che ne fanno oppure tramite una sottoscrizione mensile o annuale. A volte, il software è gratuito per tutti, ma gli utenti devono accettare i messaggi pubblicitari, che sono la fonte di finanziamento del servizio.

Lo sviluppo del software come servizio ha subito un'accelerazione negli ultimi anni in seguito al diffondersi del calcolo cloud. Quando un servizio viene installato nel cloud, il numero di server può cambiare rapidamente per adattarsi alle richieste di tale servizio da parte degli utenti. Non c'è bisogno che i fornitori di servizi si preoccupino dei picchi di carico; di conseguenza, i costi per questi fornitori si sono ridotti significativamente.

Per gli acquirenti del software, il vantaggio del SaaS è che i costi di gestione del software si trasferiscono al fornitore, il quale ha la responsabilità di correggere i bug, aggiornare il software, adattare le modifiche alla piattaforma del sistema operativo e garantire che la capacità dell'hardware possa soddisfare le richieste degli utenti. I costi di gestione della licenza del software sono nulli. Se qualcuno ha più computer, non occorre una licenza per tutti questi computer. Se un'applicazione software è utilizzata soltanto occasionalmente, il modello pay-per-use può essere più economico dell'acquisto dell'applicazione. Le unità mobili, come gli smartphone, possono accedere al software da qualsiasi parte del mondo.

Il problema principale che impedisce il diffondersi del SaaS è il trasferimento dei dati con il servizio remoto. Il trasferimento avviene alla velocità della rete, quindi trasferire grandi quantità di dati, come immagini o video di alta qualità, richiede molto tempo. Il fornitore dei servizi potrebbe anche richiedere un pagamento in base alla quantità dei dati trasferiti. Altri problemi sono la mancanza di un controllo sull'evoluzione del software (il fornitore può cambiare il software quando vuole), le leggi e i regolamenti. Molte nazioni hanno leggi che controllano la memorizzazione, la gestione, la conservazione e l'accesso ai dati; quindi passare i dati a un servizio remoto potrebbe violare queste leggi.

Il software come servizio e le architetture orientate ai servizi (SOA, Service-Oriented Architectures), descritte nel Capitolo 18, sono correlati, ma non sono la stessa cosa.

1. Il software come servizio è un modo di offrire funzionalità su un server remoto, alle quali il cliente può accedere tramite un browser web. Il server mantiene i dati e lo stato dell'utente durante una sessione di interazioni. Le transazioni di solito sono operazioni lunghe, per esempio, la correzione di un documento.

2. Le architetture orientate ai servizi sono approcci per strutturare un sistema software come una serie di servizi stateless distinti. Questi servizi possono essere forniti da più fornitori e possono essere distribuiti. Tipicamente, le transazioni sono operazioni brevi, dove viene chiamato un servizio, che svolge un determinato compito e restituisce un risultato.

SaaS è un modo di offrire agli utenti le funzionalità delle applicazioni, mentre SOA è una tecnologia di implementazione dei sistemi applicativi. Gli utenti non devono accedere ai sistemi implementati tramite SOA come se fossero servizi web. Le applicazioni SaaS per le aziende possono essere implementate utilizzando componenti, anziché servizi. Tuttavia, se un software viene implementato come servizio tramite SOA, le applicazioni possono utilizzare le API di servizio per accedere alle funzionalità di altre applicazioni. Esse possono poi essere integrate in sistemi più complessi. Questi sistemi, chiamati *mashup*, rappresentano un ulteriore approccio al riutilizzo del software e allo sviluppo rapido del software.

Dal punto di vista dello sviluppo del software, il processo di sviluppo dei servizi ha molto in comune con altri tipi di sviluppo del software. Tuttavia, la costruzione dei servizi di solito non è guidata dai requisiti degli utenti, ma dalle ipotesi che il fornitore dei servizi ha fatto sulle esigenze degli utenti. Di conseguenza, il software deve essere in grado di evolversi rapidamente dopo che il fornitore dei servizi ha ricevuto un feedback dagli utenti sulle loro reali esigenze. Lo sviluppo agile con consegna incrementale è quindi un approccio efficiente per il software che deve essere sviluppato come servizio.

Alcune applicazioni software che sono implementate come servizi, per esempio Google Docs per gli utenti del Web, offrono funzionalità generiche per tutti gli utenti. Tuttavia, le aziende potrebbero preferire servizi più specifici, appositamente progettati per le loro esigenze. Se state implementando un software come servizio per un'azienda, potreste basare il vostro servizio su un servizio generico che è stato adattato alle esigenze di un particolare cliente. Tre fattori importanti devono essere considerati.

1. *Configurabilità*. Come configurate il software per soddisfare i requisiti specifici di un'organizzazione?
2. *Multi-tenancy*. Come fate a dare a ciascun utente del software l'impressione che sta lavorando con la propria copia del sistema mentre, contemporaneamente, sta utilizzando efficientemente le risorse comuni del sistema?
3. *Scalabilità*. Come progettate il sistema in modo che possa essere scalato per soddisfare le richieste di un numero imprevedibilmente grande di utenti?

Il concetto di architettura di una linea di prodotti software, descritto nel Capitolo 15, è un modo di configurare il software per gli utenti che hanno requisiti simili, non identici. Si parte da un generico sistema e lo si adatta alle specifiche esigenze di ciascun utente.

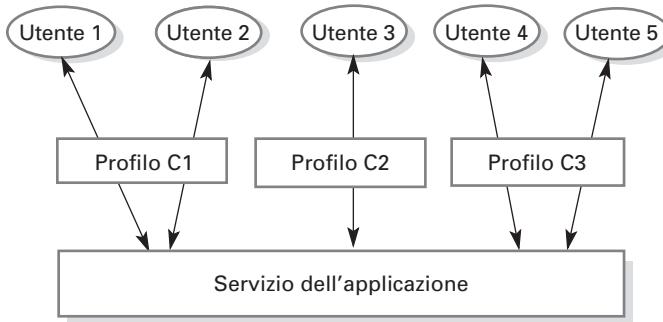


Figura 17.16 Configurazione di un sistema software offerto come servizio.

Questo non funziona con il software progettato come servizio, in quanto si dovrebbe installare una copia diversa del servizio per ciascuna organizzazione che usa il software. Piuttosto, bisogna progettare un sistema configurabile, realizzando un'interfaccia di configurazione che consenta agli utenti di specificare le loro preferenze. Queste preferenze vengono poi utilizzate per adattare dinamicamente il comportamento del software, mentre viene utilizzato. Le funzioni di configurazione dovrebbero consentire le seguenti caratteristiche:

1. *marchio*: gli utenti vedono un'interfaccia che rispecchia le caratteristiche tipiche dell'azienda di appartenenza;
2. *workflow e regole aziendali*: ogni azienda definisce le regole che governano l'utilizzo dei servizi e dei dati;
3. *estensioni del database*: ogni azienda definisce come estendere il generico modello dei servizi del database in modo da soddisfare le sue specifiche esigenze;
4. *controllo dell'accesso*: i clienti dei servizi creano account individuali per il loro personale e definiscono le risorse e le funzioni alle quali può accedere ciascuno dei loro utenti.

La Figura 17.16 illustra questa situazione. Il diagramma mostra cinque utenti del servizio che lavorano per tre clienti differenti del fornitore di servizi. Gli utenti interagiscono con il servizio tramite un profilo utente che definisce la configurazione del servizio per il loro datore di lavoro.

La multi-tenancy è la situazione nella quale più utenti accedono allo stesso sistema e l'architettura del sistema è definita in modo da consentire la condivisione efficiente delle risorse del sistema. Ogni utente deve avere la sensazione di essere l'unico a utilizzare il sistema. La multi-tenancy richiede che il sistema sia progettato in modo che ci sia una separazione netta tra funzioni e dati del sistema. Quindi, tutte le operazioni devono essere stateless, in modo che possano essere condivise. I dati dovrebbero essere forniti dal client oppure essere disponibili in

Tenant	Chiave	Nome	Indirizzo
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Figura 17.17 Un database multi-tenant.

un sistema di memorizzazione o in un database; qualsiasi istanza del sistema può avere accesso a questi dati.

Un problema particolare nei sistemi multi-tenancy è la gestione dei dati. Il modo più semplice per offrire la gestione dei dati è che tutti i clienti abbiano il loro database, che possono utilizzare e configurare come preferiscono. Questo, però, richiede che il fornitore dei servizi mantenga più istanze del database (una per ogni cliente) e renda disponibili questi database su richiesta.

In alternativa, il fornitore dei servizi può utilizzare un singolo database, con più utenti virtualmente isolati all'interno del database. Questo è illustrato nella Figura 17.17, dove si può notare che le voci del database hanno anche un “identificatore del tenant” che collega tali voci a specifici utenti. Utilizzando le funzioni view del database, è possibile estrarre le voci per ogni cliente del servizio e così presentare gli utenti di quel cliente con un proprio database virtuale. Questo processo può essere esteso per soddisfare specifiche esigenze dei clienti, utilizzando le caratteristiche di configurazione descritte in precedenza.

La scalabilità è la capacità del sistema di soddisfare le richieste di un numero crescente di utenti, senza ridurre la qualità complessiva del servizio che viene fornito a ciascun utente. In generale, quando si considera la scalabilità nel contesto SaaS, si sta valutando la possibilità di aggiungere nuove risorse (scaling out), anziché potenziare quelle esistenti (scaling up). Ricordiamo che scaling out significa aggiungere nuovi server e, quindi, incrementare il numero di transazioni che possono essere eseguite in parallelo. La scalabilità è un argomento complesso che non posso trattare qui in modo dettagliato; mi limito a indicare alcune linee guida generali per implementare un software scalabile.

1. Sviluppare applicazioni dove ciascun componente è implementato come un semplice servizio stateless che può essere eseguito su qualsiasi server. Nel corso di una singola transazione, un utente può quindi interagire con le istanze dello stesso servizio che sono eseguite su diversi server.
2. Progettare il sistema utilizzando interazioni asincrone, in modo che l'applicazione non debba attendere il risultato di un'interazione (come una richiesta di lettura). Questo consente all'applicazione di svolgere un lavoro utile mentre attende che l'interazione finisca.

3. Gestire le risorse, quali le connessioni con la rete e i database, come un insieme comune in modo che nessun server rischi di esaurire le risorse.
4. Progettare i database per consentire un blocco accurato delle risorse, nel senso di non bloccare tutti i record di un database quando viene utilizzata una sola parte del record.
5. Utilizzare una piattaforma PaaS cloud, come Google App Engine (Sanderson 2012) o altra piattaforma PaaS per implementare i sistemi. Queste includono i meccanismi che incrementano (scaling out) automaticamente il vostro sistema quando il carico aumenta.

Il concetto di software come servizio è un importante cambiamento di paradigma per il calcolo distribuito. Abbiamo visto già come il software per i consumatori e le applicazioni professionali, come Photoshop, siano passati a questo modello di consegna. Sempre più spesso, le aziende stanno sostituendo i loro sistemi, come i CRM e i sistemi per gli inventari, con sistemi SaaS basati sul cloud offerti da fornitori esterni, come Salesforce. Le società di software specializzate che implementano applicazioni aziendali preferiscono i sistemi SaaS in quanto semplificano l'aggiornamento e la gestione del software.

SaaS rappresenta un nuovo modo di pensare l'ingegneria dei sistemi aziendali. È stato sempre utile immaginare sistemi che potessero offrire servizi agli utenti, ma, prima di SaaS, tale funzione richiedeva varie astrazioni, come gli oggetti, per implementare i sistemi. Dove c'è un corrispondenza più stretta tra utente e astrazioni, il sistema risultante è più facile da capire, mantenere e migliorare.

Punti chiave

- I sistemi distribuiti offrono alcuni vantaggi: possono essere scalati per soddisfare la crescente domanda degli utenti, possono continuare a fornire servizi agli utenti (anche quando alcune parti del sistema sono guaste) e consentono la condivisione delle risorse.
- I problemi da considerare quando si progetta un sistema distribuito includono la trasparenza, l'apertura, la scalabilità, la protezione, la qualità dei servizi e la gestione dei guasti.
- I sistemi client-server sono sistemi distribuiti che hanno una struttura a strati, con lo strato della presentazione implementato su un computer client. I server forniscono la gestione dei dati, le applicazioni e i servizi di database.
- I sistemi client-server possono avere più livelli, con i vari strati distribuiti su diversi computer.
- Gli schemi architetturali per i sistemi distribuiti includono le architetture master-slave, le architetture client-server a due e più livelli, le architetture di componenti distribuiti e le architetture peer-to-peer.

- I sistemi di componenti distribuiti richiedono il middleware per gestire le comunicazioni dei componenti e per consentire agli oggetti di essere aggiunti e rimossi dal sistema.
- Le architetture peer-to-peer sono architetture decentralizzate nelle quali client e server non sono distinti. I calcoli possono essere distribuiti su più sistemi di varie società.
- Il software come servizio è un modo di installare le applicazioni come sistemi client-server, dove il client è un browser web.

Esercizi

- 17.1 Che cosa s'intende per "scalabilità"? Descrivete le differenze tra "scaling out" e "scaling up" e spiegate quando questi due approcci alla scalabilità possono essere utilizzati.
- 17.2 Spiegate perché i sistemi distribuiti sono più complessi di quelli centralizzati, dove tutte le funzionalità di un sistema sono implementate in un solo computer.
- * 17.3 Utilizzando un esempio di chiamata di procedura remota, spiegate come il middleware coordina l'interazione dei computer in un sistema distribuito.
- * 17.4 Qual è la differenza fondamentale tra gli approcci fat-client e thin-client alle architetture dei sistemi client-server?
- 17.5 Vi è stato chiesto di progettare un sistema protetto che richiede rigide procedure di autenticazione e autorizzazione. Il sistema deve essere progettato in modo che le comunicazioni tra le parti del sistema non possano essere intercettate e lette da un pirata informatico. Indicate l'architettura client-server più appropriata per questo sistema e, spiegando le ragioni della vostra risposta, proponete come le funzionalità dovrebbero essere distribuite tra i sistemi client e server.
- * 17.6 Un vostro cliente vuole sviluppare un sistema per gestire un magazzino di merci; i commercianti possono accedere alle informazioni sulle società e valutare vari scenari di investimenti utilizzando un sistema di simulazione. Ogni commerciante usa questa simulazione in modo diverso, a seconda della sua esperienza e del tipo di merce che tratta. Suggerite un'architettura client-server per questo sistema, indicando dove si trova la funzionalità del sistema. Spiegate perché avete scelto questo modello client-server.
- 17.7 Utilizzando un modello di componenti distribuiti, proponete un'architettura per un sistema di prenotazioni di biglietti per teatri a livello nazionale. Gli utenti possono verificare la disponibilità delle poltrone e prenotare i biglietti per un gruppo di teatri. Il sistema deve supportare la restituzione dei biglietti in modo che questi possano essere rivenduti nella biglietteria last minute ad altri clienti.
- * 17.8 Indicate due vantaggi e due svantaggi delle architetture peer-to-peer decentralizzate e semicentralizzate.
- * 17.9 Spiegate perché distribuendo il software come servizio è possibile ridurre i costi per il supporto informatico di un'azienda. Quali altri costi potrebbero essere necessari se si adottasse questo modello di sviluppo?

- 17.10 La vostra società desidera passare dall’uso di applicazioni su computer locali all’uso delle stesse funzionalità come servizi remoti. Identificate tre potenziali rischi di questa scelta e suggerite il modo in cui è possibile ridurre tali rischi.

* *Gli esercizi contrassegnati con l’asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Sebbene questo libro non fornisca moltissime informazioni sulle architetture p2p, tuttavia è un’eccellente introduzione al calcolo p2p e descrive l’organizzazione e i metodi che sono utilizzati in vari sistemi p2p. (A. Oram (ed.), O'Reilly and Associates Inc., 2001).

“Turning Software into a Service.” Un buon documento di sintesi che descrive i principi del calcolo orientato ai servizi. Diversamente da altri documenti su questo argomento, non nasconde questi principi dietro una discussione degli standard. (M. Turner, D. Budgen e P. Brereton, *IEEE Computer*, 36 (10), October 2003) <http://dx.doi.org/10.1109/MC.2003.1236470>

Distributed Systems, 5th ed. Un testo completo che tratta tutti gli aspetti della progettazione e dell’implementazione dei sistemi distribuiti. Esamina anche i sistemi peer-to-peer e i sistemi mobili. (G. Coulouris, J. Dollimore, T. Kindberg e G. Blair. Addison-Wesley, 2011).

Engineering Software as a Service: An Agile Approach Using Cloud Computing. Questo libro accompagna il corso online dell’autore sull’argomento. Un buon libro pratico, consigliato alle persone che non conoscono questo tipo di sviluppo. (A. Fox e D. Patterson, Strawberry Canyon LLC, 2014) <http://www.saasbook.info>

CAPITOLO

18

Ingegneria del software orientato ai servizi

Questo capitolo si propone di presentare l'ingegneria del software orientato ai servizi come un modo di costruire applicazioni distribuite utilizzando i servizi web. Dopo aver letto questo capitolo:

- conoscerete i concetti fondamentali dei servizi web, degli standard dei servizi web e dell'architettura orientata ai servizi;
- apprenderete il concetto di servizi RESTful e le differenze principali tra servizi RESTful e SOAP;
- conoscerete il processo di ingegneria dei servizi che ha lo scopo di produrre servizi web riutilizzabili;
- capirete come la composizione dei servizi basati sul workflow possa essere utilizzata per creare software orientato ai servizi a supporto dei processi aziendali.

- 18.1 Architettura orientata ai servizi
- 18.2 Servizi RESTful
- 18.3 Ingegneria dei servizi
- 18.4 Composizione dei servizi

Lo sviluppo del Web negli anni '90 ha rivoluzionato lo scambio delle informazioni fra le società. I computer client possono avere accesso alle informazioni su server remoti esterni alle loro società. Tuttavia, l'accesso avveniva soltanto attraverso un browser web, e l'accesso diretto alle informazioni tramite altri programmi non era praticabile. Questo significava che non era possibile stabilire connessioni tra i server, dove, per esempio, un programma avrebbe potuto interrogare un certo numero di cataloghi di diversi fornitori.

Per aggirare questo problema furono sviluppati i *servizi web*, che consentivano ai programmi di accedere e aggiornare le risorse disponibili sul Web. Utilizzando un servizio web, le organizzazioni che vogliono rendere accessibili le proprie informazioni ad altri programmi possono farlo, definendo e pubblicando un'interfaccia programmatica di servizi web. Questa interfaccia definisce i dati disponibili e le modalità di accesso e di utilizzo di tali dati.

Più in generale, un servizio web è una rappresentazione standard di risorse che possono essere utilizzate da altri programmi; possono essere risorse di informazioni, come le parti di un catalogo, risorse di calcolo, come un processore specializzato, o risorse di memoria. Per esempio, potrebbe essere implementato un servizio di archiviazione per memorizzare in modo affidabile e permanente i dati di un'azienda che, per legge, devono essere conservati per molti anni.

Un servizio web è un'istanza di un più generico concetto di servizio, definito da Lovelock (Lovelock et al. 1996) come:

Un atto o una prestazione offerta da una parte a un'altra. Per quanto il processo possa essere legato a un prodotto fisico, la prestazione è essenzialmente intangibile e non risulta solitamente in possesso di nessuno dei fattori di produzione.¹

I servizi sono un naturale sviluppo dei componenti software, dove il modello dei componenti è, in essenza, un insieme di standard associati ai servizi web. Un servizio web può quindi essere definito come segue:

Un componente software riutilizzabile, debolmente accoppiato, che incapsula funzionalità discrete, che può essere distribuito e il cui accesso avviene tramite programmi. Un servizio web è un servizio a cui si accede per mezzo di protocolli standard di Internet e protocolli basati su XML.

Una differenza fondamentale tra servizi e componenti software, definiti nell'ingegneria del software basato sui componenti (CBSE), è che i servizi devono essere indipendenti e debolmente accoppiati. In altre parole, essi devono operare sempre nello stesso modo, indipendentemente dall'ambiente di esecuzione; non devono affidarsi a componenti esterni che potrebbero avere comportamenti differenti e non funzionali. Pertanto, i servizi web non hanno un'interfaccia di "servizi richie-

¹ Lovelock C., Vandermerwe S. e Lewis B. (1996). *Services Marketing*. Englewood Cliffs, NJ: Prentice Hall.

sti” che, in CBSE, definisce gli altri componenti di sistema che devono essere presenti. Un’interfaccia di servizi web è semplicemente un’interfaccia di “servizi offerti” che definisce le funzionalità e i parametri dei vari servizi.

I sistemi orientati ai servizi sono un modo di sviluppare sistemi distribuiti, dove i componenti del sistema sono servizi indipendenti, che vengono eseguiti su computer distribuiti in tutto il mondo. I servizi sono indipendenti dalle piattaforme e dai linguaggi di implementazione. I sistemi software possono essere realizzati componendo servizi locali e servizi esterni offerti da fornitori differenti, con un’interazione continua tra i servizi nel sistema.

Come detto nel Capitolo 17, i concetti di “software come servizio” e “sistemi orientati ai servizi” non sono uguali. Il software come servizio significa offrire agli utenti funzionalità software in modo remoto sul web, anziché tramite applicazioni installate sul computer di un utente. I sistemi orientati ai servizi sono sistemi implementati utilizzando componenti di servizi riutilizzabili e accessibili da altri programmi, anziché direttamente dagli utenti. Il software che è offerto come servizio può essere implementato utilizzando un sistema orientato ai servizi; tuttavia, non bisogna implementare il software in questo modo per offrirlo come servizio agli utenti.

Adottando un approccio orientato ai servizi nell’ingegneria del software, si ottengono alcuni importanti benefici:

1. I servizi possono essere offerti da qualsiasi fornitore di servizi dentro o fuori un’organizzazione. Supponendo che questi servizi siano conformi a determinati standard, le organizzazioni possono creare applicazioni integrando servizi da altri fornitori. Per esempio, un’azienda di produzione può collegarsi direttamente ai servizi offerti dai propri fornitori di materiali.
2. I fornitori di servizi rendono pubbliche le informazioni sul servizio in modo che qualsiasi utente autorizzato possa utilizzarlo. Il fornitore e l’utente non devono negoziare le funzionalità offerte da un servizio, prima che questo sia incorporato in un programma applicativo.
3. Le applicazioni possono ritardare il collegamento ai servizi finché questi non saranno consegnati o eseguiti. Per esempio, un’applicazione che utilizza un servizio per avere informazioni sui prezzi azionari può cambiare il fornitore di servizi dinamicamente mentre il sistema è in esecuzione. Questo significa che le applicazioni possono essere reattive e adattare le loro operazioni per soddisfare questi cambiamenti del loro ambiente operativo.
4. È possibile costruire opportunisticamente nuovi servizi. Un fornitore di servizi può riconoscere nuovi servizi che possono essere creati collegando i servizi esistenti in modo innovativo.
5. Gli utenti possono pagare i servizi in base all’utilizzo effettivo, anziché alla loro fornitura. Quindi anziché acquistare un componente costoso che viene utilizzato raramente, l’autore di un’applicazione può utilizzare un servizio esterno che pagherà solo quando ne avrà bisogno.

6. Le applicazioni possono essere più piccole (caratteristica importante se devono essere integrate in dispositivi mobili che hanno capacità limitate di calcolo e di memoria). Alcuni calcoli intensivi e la gestione delle eccezioni possono essere affidate a servizi esterni.

I sistemi orientati ai servizi sono architetture debolmente accoppiate, in quanto i collegamenti ai servizi possono cambiare durante l'esecuzione. Una versione diversa, ma equivalente, del servizio potrebbe quindi essere eseguita in momenti diversi. Alcuni sistemi possono essere costruiti utilizzando solamente servizi web, mentre altri potrebbero unire servizi web e componenti sviluppati localmente. Per illustrare come le applicazioni che usano un mix di servizi e componenti possono essere organizzate, si consideri il seguente scenario:

Un sistema informativo per automobili fornisce informazioni sulle condizioni meteorologiche, sul traffico, notizie locali e così via. Il sistema si collega all'autoradio in modo da fornire tali informazioni su uno specifico canale radio. L'automobile è dotata di un ricevitore GPS per individuare la propria posizione e, in base a questa, il sistema accede a una serie di servizi informativi. Le informazioni possono essere fornite nella lingua selezionata dall'automobilista.

La Figura 18.1 illustra una possibile organizzazione di questo sistema. Il software a bordo dell'automobile include cinque moduli, che gestiscono le comunicazioni con l'automobilista, con il ricevitore GPS che riporta la posizione dell'auto, e con l'autoradio; i moduli *Trasmettitore* e *Ricevitore* gestiscono tutte le comunicazioni con i servizi esterni.

L'automobile comunica con un servizio informativo mobile esterno che raccoglie le informazioni da una serie di altri servizi, che forniscono informazioni sul tempo, sul traffico e sui fatti locali. Questi servizi sono offerti da vari fornitori in luoghi differenti; il sistema a bordo dell'auto accede a un servizio di ricerca esterno per localizzare i servizi disponibili nell'area locale. Il servizio informativo mobile usa anche il servizio di ricerca per collegare i servizi appropriati sul tempo, sul traffico e sulle notizie locali. Le informazioni aggregate sono inviate all'automobile attraverso un servizio che le traduce nella lingua scelta dall'automobilista.

Questo esempio illustra uno dei vantaggi chiave dell'approccio orientato ai servizi. Non è necessario decidere quale fornitore di servizi deve essere utilizzato durante la programmazione o la configurazione del sistema. Mentre l'auto si sposta, il software a bordo usa il servizio di ricerca per trovare il servizio informativo locale più appropriato. Grazie al servizio di traduzione, l'auto può attraversare i confini di uno stato e rendere accessibili le informazioni locali alle persone che non parlano la lingua locale.

Credo che l'approccio orientato ai servizi nell'ingegneria del software sia un importante passo in avanti come l'ingegneria del software orientato agli oggetti. I sistemi orientati ai servizi sono essenziali per i dispositivi mobili e i sistemi

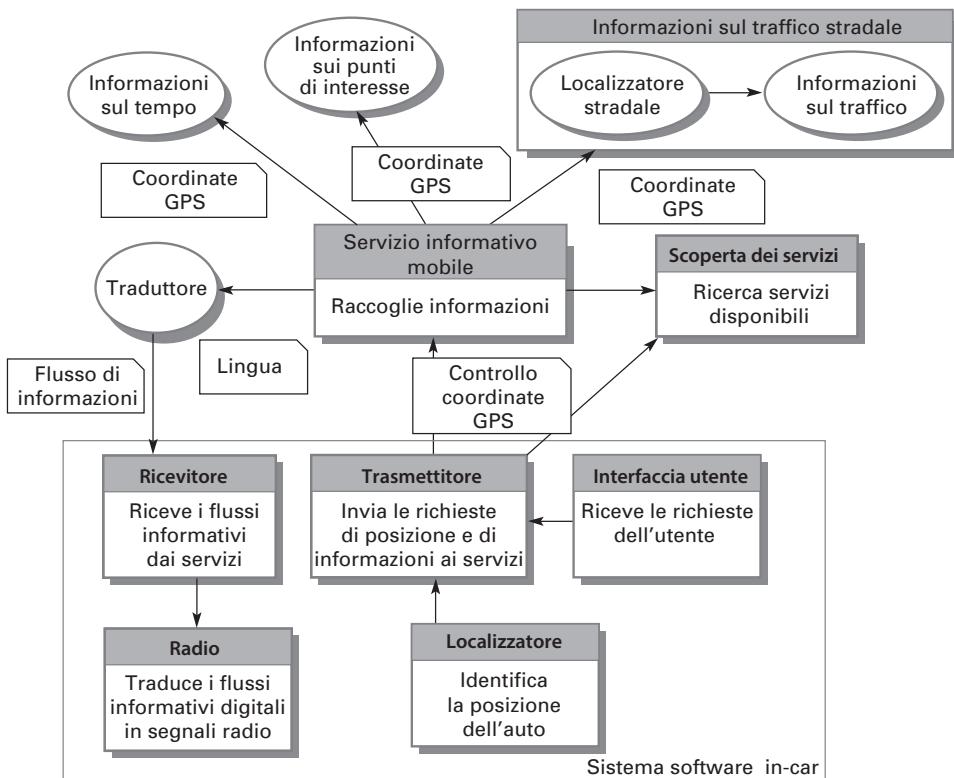


Figura 18.1 Un sistema informativo per automobili basato sui servizi.

cloud. Newcomer e Lomow (Newcomer e Lomow 2005), nel loro libro sull'architettura orientata ai servizi (SOA), sintetizzano così le potenzialità dell'approccio orientato ai servizi:

Grazie alla convergenza di diverse tecnologie chiave e all'adozione universale dei servizi web, l'impresa orientata ai servizi promette di migliorare significativamente l'agilità dell'organizzazione, ridurre il time-to-market di nuovi prodotti e servizi, diminuire i costi legati all'Information Technology e aumentare l'efficienza delle operazioni.²

Creare applicazioni basate sui servizi consente alle aziende e altre organizzazioni di cooperare e condividere le proprie funzionalità. In questo modo diventa più facile automatizzare i sistemi che richiedono un intenso interscambio di informazioni, come nelle catene di approvvigionamento in cui un'azienda ordina beni da un'altra azienda. Le applicazioni basate sui servizi possono essere realizzate collegando i servizi di vari fornitori, utilizzando un linguaggio di programma-

² Newcomer E. e Lomow G. (2005). *Understanding SOA with Web Services*. Boston: Addison-Wesley.

zione standard o un linguaggio di workflow specializzato, come descritto nel Paragrafo 18.4.

Il lavoro iniziale di fornitura e implementazione dei servizi fu notevolmente influenzato dal fallito accordo dell'industria del software sulla standardizzazione dei componenti. Tutte le principali compagnie svilupparono i loro standard; questo portò a una serie di standard differenti (standard WS*) e al concetto di architetture orientate ai servizi. Queste furono proposte come architetture per sistemi basati sui servizi, con tutte le comunicazioni dei servizi basate su standard. Tuttavia, gli standard proposti erano complessi e avevano significativi overhead di esecuzione. Questo problema indusse molte compagnie ad adottare un approccio architettonico alternativo, basato sui cosiddetti servizi RESTful. Un approccio RESTful è un metodo più semplice dell'architettura orientata ai servizi, ma è meno appropriato ai servizi che offrono funzionalità complesse. In questo capitolo tratterò entrambi questi approcci architettonici.

18.1 Architettura orientata ai servizi

L'architettura orientata ai servizi (Service-Oriented Architecture, SOA) è uno stile architettonico basato sul concetto che i servizi eseguibili possano essere inclusi nelle applicazioni. I servizi hanno interfacce pubbliche ben definite, e le applicazioni possono scegliere se tali servizi siano appropriati oppure no. Un concetto importante che sta alla base della SOA è che lo stesso servizio può essere reso disponibile da più fornitori e che le applicazioni possono decidere durante l'esecuzione quale fornitore scegliere.

La Figura 18.2 illustra la struttura di un'architettura orientata ai servizi. I fornitori dei servizi progettano e implementano i servizi e specificano l'interfaccia per questi servizi; inoltre, pubblicano le informazioni sui servizi in un registro accessibile. Coloro che richiedono un servizio, detti anche *client del servizio*, accedono alla specifica del servizio e ne localizzano il fornitore; possono collegare la loro applicazione a quel servizio specifico e comunicare con esso tramite protocolli standard per i servizi.

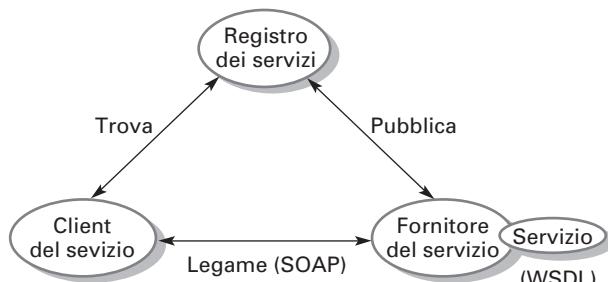


Figura 18.2 Schema di un'architettura orientata ai servizi.

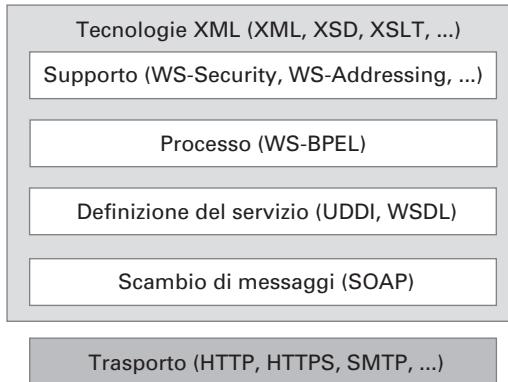


Figura 18.3 Standard per i servizi web.

Lo sviluppo e l'uso di standard internazionali approvati è fondamentale nella SOA. Di conseguenza, le architetture orientate ai servizi non sono state ostacolate dalle incompatibilità che di solito sorgono con le innovazioni tecniche, dove vari fornitori mantengono la versione della tecnologia di loro proprietà. La Figura 18.3 mostra la pila dei più importanti standard per il supporto dei servizi web.

I protocolli per i servizi web coprono tutti gli aspetti delle architetture orientate ai servizi, dai meccanismi base per lo scambio di informazioni (SOAP) agli standard per i linguaggi di programmazione (WS-BPEL). Questi standard sono tutti basati su XML, una notazione leggibile dalle macchine e dagli esseri umani che permette di definire dati strutturati, dove i testi sono marcati con tag e identificatori appropriati. XML è supportato da una vasta gamma di tecnologie, tra cui XSD per la definizione degli schemi utilizzati per estendere e manipolare le descrizioni XML. Erl (Erl 2004) fornisce una sintesi delle tecnologie XML e del loro ruolo nei servizi web.

Gli standard fondamentali per le architetture orientate ai servizi sono qui di seguito elencati.

1. *SOAP* è uno standard per lo scambio di messaggi che supporta la comunicazione tra i servizi e definisce i componenti essenziali e opzionali dei messaggi. I servizi in un'architettura orientata ai servizi sono anche detti “servizi basati sul SOAP”.
2. *WSDL* (Web Service Definition Language) è uno standard per definire l’interfaccia dei servizi. Specifica come le operazioni dei servizi (nomi delle operazioni, parametri e loro tipi) e i loro legami devono essere definiti.
3. *WS-BPEL* è uno standard per un linguaggio di workflow che viene utilizzato per definire processi che coinvolgono più servizi. Questo standard sarà trattato nel Paragrafo 18.3.

Lo standard UDDI (Universal Description, Discovery and Integration) definisce i componenti della specifica di un servizio che possono essere utilizzati per scoprire l'esistenza del servizio. Questo standard nacque con lo scopo di consentire alle compagnie di impostare i registri, con le descrizioni UDDI che definiscono i servizi offerti. Alcune compagnie impostarono i registri UDDI nei primi anni del XXI secolo, ma gli utenti preferirono i motori di ricerca standard per trovare i servizi. Tutti i registri UDDI pubblici adesso sono chiusi.

I principali standard SOA sono supportati da diversi standard che si concentrano sugli aspetti più specialistici della SOA. Il loro numero è elevato, perché l'obiettivo è supportare l'architettura in vari tipi di applicazioni aziendali; ecco alcuni esempi di questi standard.

1. *WS-Reliable Messaging* è uno standard per lo scambio di messaggi che assicura che ogni messaggio sarà consegnato una volta soltanto.
2. *WS-Security* è un'insieme di standard che supportano la protezione dei servizi web, tra cui gli standard che specificano la definizione delle politiche di protezione e l'uso delle firme digitali.
3. *WS-Addressing* definisce il modo in cui si devono rappresentare le informazioni di indirizzamento in un messaggio SOAP.
4. *WS-Transactions* definisce il coordinamento delle transazioni all'interno dei servizi distribuiti.

Gli standard per i servizi web sono un argomento vastissimo, che non è possibile trattare dettagliatamente in questo libro. Per una buona panoramica su questi standard consiglio i libri di Erl (Erl 2004; Erl 2005). Le descrizioni dettagliate degli standard sono disponibili anche sul Web come documenti pubblici (W3C 2013).

18.1.1 Componenti dei servizi nella SOA

Lo scambio dei messaggi, come detto nel Paragrafo 17.1, è un importante meccanismo per coordinare le azioni in un sistema di calcolo distribuito. I servizi in una SOA comunicano scambiandosi messaggi, espressi in XML, e questi messaggi sono distribuiti utilizzando protocolli di trasporto standard di Internet, quali HTTP e TCP/IP.

Un servizio definisce che cosa ha bisogno da un altro servizio, impostando le sue richieste in un messaggio, che viene spedito a tale servizio. Il servizio che riceve il messaggio, esegue i calcoli e, dopo aver finito, invia una risposta, come messaggio, al servizio richiedente. Questo servizio esamina il messaggio per estrarre le informazioni richieste. Diversamente dai componenti software, i servizi non usano chiamate di metodi o procedure remote per accedere alle funzionalità associate ad altri servizi.

Per utilizzare un servizio web, bisogna conoscere dove si trova il servizio – il suo identificatore URI (Uniform Resource Identifier) – e i dettagli della sua inter-

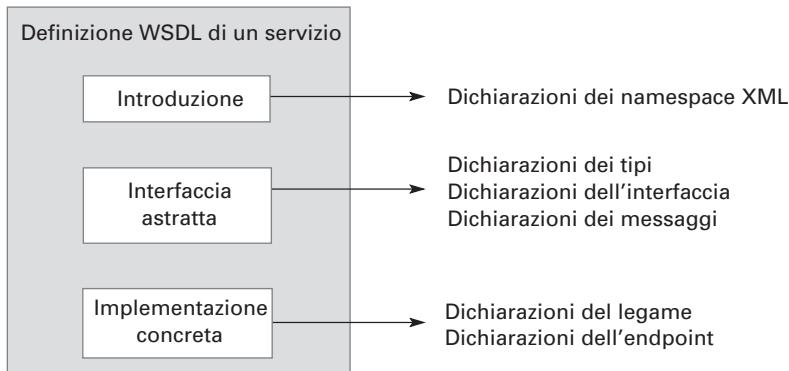


Figura 18.4 Organizzazione di una specifica WSDL.

faccia. Questi dettagli sono forniti in una descrizione di servizi che è scritta in un linguaggio basato su XML, detto WSDL (Web Service Description Language). La specifica WSDL definisce tre aspetti di un servizio web: *cosa* fa il servizio, *come* comunica e *dove* si trova.

1. La parte del documento WSDL che si occupa del “cosa”, chiamata interfaccia, specifica le operazioni supportate dal servizio e definisce il formato dei messaggi inviati e ricevuti.
2. La parte del documento WSDL che tratta il “come”, chiamata *legame* o *binding*, fa corrispondere l’interfaccia astratta a un insieme specifico di protocolli. Il legame descrive gli aspetti tecnici della comunicazione con un servizio web.
3. La parte del documento WSDL che riguarda il “dove” descrive la posizione dell’implementazione di uno specifico servizio web (il suo *endpoint*).

Il modello concettuale WSDL (Figura 18.4) mostra gli elementi della descrizione di un servizio. Ognuno elemento è espresso in XML e può occupare un file separato; questi elementi sono elencati qui di seguito.

1. Una parte introduttiva che solitamente definisce i namespace XML utilizzati; può includere una sezione con informazioni aggiuntive sul servizio.
2. Una descrizione facoltativa dei tipi utilizzati nei messaggi scambiati dal servizio.
3. Una descrizione dell’interfaccia del servizio, ovvero delle operazioni che il servizio offre ad altri servizi o agli utenti.
4. Una descrizione dei messaggi di input e output elaborati dal servizio.
5. Una descrizione del legame usato dal servizio, cioè del protocollo usato per inviare e ricevere messaggi. Per default, tale protocollo è SOAP, ma è possibile sceglierne altri. Il legame precisa come devono essere costruiti i mes-

Definizione di alcuni tipi utilizzati. Si presume che ‘ws’ faccia riferimento all’URI del namespace per gli schemi XML e che il prefisso del namespace associato a questa definizione sia weathns.

```
<types>
  <xs: schema targetNameSpace = “http://. . ./weathns”
    xmlns: weathns = “http://. . ./weathns” >
    <xs:element name = “LuogoAndData” type = “pdrec” />
    <xs:element name = “MaxMinTemp” type = “mmtrec” />
    <xs:element name = “InDataFault” type = “errmess” />

    <xs:complexType name = “pdrec”
      <xs:sequence>
        <xs:element name = “town” type = “xs:string”/>
        <xs:element name = “country” type = “xs:string”/>
        <xs:element name = “day” type = “xs:date” />
      </xs:sequence>
    </xs:complexType>
    Qui vanno inserite le definizioni di MaxMinType e InDataFault ...
  </schema>
</types>
```

Segue la definizione dell’interfaccia e delle sue operazioni. In questo caso c’è una sola interfaccia, che restituisce le temperature massime e minime.

```
<interface name = “informazioniMeteo” >
  <operation name = “getMaxMinTemps” pattern = “wsdlIns: in-out”>
    <input messageLabel = “In” element = “weathns:LuogoAndData” />
    <output messageLabel = “Out” element = “weathns:MaxMinTemp” />
    <outfault messageLabel = “Out” element = “weathns:InDataFault” />
  </operation>
</interface>
```

Figura 18.5 Parte di una descrizione WSDL per un servizio web.

saggi di input e output associati al servizio e specifica i protocolli di comunicazione utilizzati. Opzionalmente, il legame può anche specificare come includere informazioni aggiuntive, come le credenziali di protezione o gli identificatori di transazione.

6. Una specifica dell’endpoint, ovvero la posizione fisica del servizio espressa sotto forma di Uniform Resource Identifier (URI) – l’indirizzo di una risorsa accessibile tramite Internet.

La Figura 18.5 mostra una parte dell’interfaccia di un semplice servizio che, in base a una data e una località di input, fornisce le temperature massime e minime registrate nella località e nel giorno specificati. Il messaggio di input indica se le temperature dovranno espresse in gradi Celsius o Fahrenheit.

Le descrizioni dei servizi basate su XML includono le definizioni dei namespace XML. Un identificatore di namespace può precedere qualsiasi identificatore utilizzato nella descrizione XML, rendendo così possibile distinguere gli identificatori con lo stesso nome che sono stati definiti in parti differenti di una descrizione XML. Non è necessario conoscere i dettagli dei namespace per capire gli esempi qui descritti. È sufficiente sapere che i nomi possono avere come prefisso un identificatore del namespace e che la coppia *namespace:nome* deve essere unica.

Nella Figura 18.5 la prima parte della descrizione mostra alcune definizioni degli elementi e dei tipi usati nella specifica del servizio. Sono definiti gli elementi *LuogoAndData*, *MaxMinTemp* e *InDataFault*. Successivamente è riportata solo la specifica di *LuogoAndData*, che si può considerare un record composto da tre campi: città, regione e giorno. Per definire *MaxMinTemp* e *InDataFault* si userà un approccio simile.

La seconda parte della descrizione mostra un esempio di definizione dell’interfaccia: in questo caso, il servizio *informazioniMeteo* ha una singola operazione, sebbene non ci siano restrizioni al numero di operazioni che si possono definire. L’operazione *informazioniMeteo* è associata a uno schema (*pattern*) in-out; questo significa che l’operazione riceve un messaggio in input e ne genera uno in output. La specifica WSDL 2.0 comprende svariati schemi di scambio di messaggi, come *in-only*, *in-out*, *out-only*, *in-optional-out*, *out-in* ecc. Successivamente, sono definiti i messaggi di input e output, che fanno riferimento alle definizioni incluse precedentemente nella sezione dedicata ai tipi.

L’interfaccia di un servizio che è definita in WSDL è semplicemente una descrizione delle operazioni del servizio e dei loro parametri (*signature*). Essa non include alcuna informazione sulla semantica del servizio o su caratteristiche quali le prestazioni o la fidatezza. Se decidete di usare il servizio, dovrete determinare che cosa fa essenzialmente il servizio e il significato dei messaggi di input e output. Dovrete anche scoprire le prestazioni e la fidatezza del servizio. Sebbene l’utilizzo di nomi significativi e di una documentazione accurata possano aiutare a conoscere meglio il servizio, tuttavia c’è sempre il rischio che si sbagli a interpretare correttamente le funzioni svolte dal servizio.

Le descrizioni dei servizi basate su XML sono lunghe, dettagliate e noiose da leggere. Le specifiche WSDL di solito non sono scritte a mano; molte delle sue informazioni sono generate automaticamente.

18.2 Servizi RESTful

Lo sviluppo iniziale dei servizi web e dell’ingegneria del software orientato ai servizi si basava sugli standard, con messaggi XML scambiati tra i servizi. Si trattava di un approccio generale che consentiva lo sviluppo di servizi complessi, binding dinamico tra i servizi e controllo della qualità dei servizi e della loro fidatezza. Tuttavia, durante lo sviluppo dei servizi, emerse che molti di essi erano

servizi a singola funzione con interfacce di input e output relativamente semplici. Gli utenti dei servizi non erano effettivamente interessati al binding dinamico e all’uso di più fornitori di servizi. Raramente utilizzavano servizi web standard per la qualità dei servizi, l’affidabilità e così via.

Il problema è che gli standard dei servizi web sono molto “pesanti” e, a volte, eccessivamente generici e inefficienti. L’implementazione di questi standard richiede una notevole attività di elaborazione per creare, trasmettere e interpretare i messaggi XML associati. Questo rallenta le comunicazioni tra i servizi e, per i sistemi ad alto throughput, potrebbe essere necessario un hardware aggiuntivo per ottenere la qualità del servizio richiesto.

Per superare questi problemi, è stato sviluppato un approccio più “leggero” all’architettura dei servizi web. Questo approccio si basa sullo stile architettonicale REST, dove REST sta per Representational State Transfer (Fielding 2000). REST è uno stile architettonicale che si basa sul trasferimento delle rappresentazioni delle risorse da un server a un client. È lo stile che sta alla base dell’intero Web e che è stato utilizzato come un metodo molto più semplice di SOAP/WSDL per implementare le interfacce dei servizi web.

L’elemento fondamentale di un’architettura RESTful è la risorsa. Una risorsa è essenzialmente un elemento di dati, come un catalogo o il record di una cartella clinica, o un documento, come un capitolo di questo libro. In generale, le risorse possono avere più rappresentazioni; ovvero possono esistere in formati differenti. Per esempio, un capitolo di questo libro ha tre rappresentazioni: una rappresentazione in Microsoft Word, utilizzata per l’editing, una rappresentazione PDF, utilizzata per visualizzare il capitolo nel Web, e una rappresentazione in InDesign, utilizzata per le pubblicazioni. La risorsa logica sottostante, fatta di testi e immagini, è la stessa in tutte e tre le rappresentazioni.

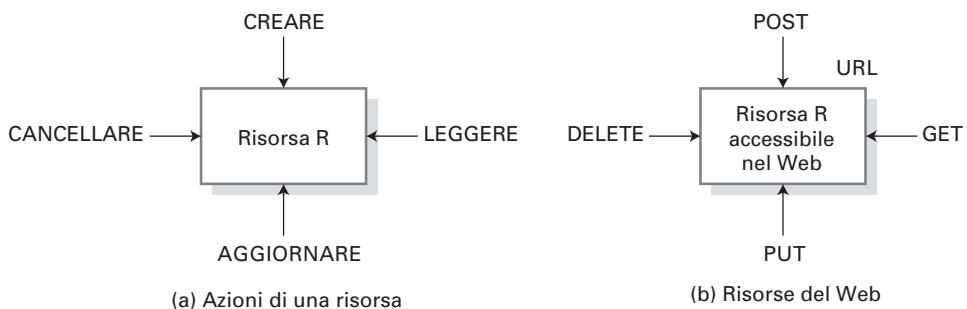
In un’architettura RESTful tutto è rappresentato come una risorsa. Le risorse hanno un identificatore unico, che è il loro URL. Le risorse sono simili agli oggetti e sono associate a quattro operazioni polimorfiche fondamentali, come illustra la Figura 18.6(a):

1. *creare* – genera la risorsa;
2. *leggere* – restituisce una rappresentazione della risorsa;
3. *aggiornare* – modifica il valore di una risorsa;
4. *cancellare* – rende inaccessibile una risorsa.

Il Web è un esempio di sistema che ha un’architettura RESTful. Le pagine web sono risorse, e l’identificatore unico di una pagina web è il suo URL.

I protocolli web *http* e *https* si basano su quattro azioni: POST, GET, PUT e DELETE. Queste azioni corrispondono alle quattro operazioni fondamentali di una risorsa, come illustra la Figura 18.6(b):

1. POST è utilizzata per creare una risorsa, associando i dati che la definiscono;

**Figura 18.6** Risorse e azioni.

2. GET è utilizzata per leggere il valore di una risorsa e restituirlo al richiedente nella rappresentazione specificata, come XHTML, che può essere visualizzata in un browser del Web;
3. PUT è utilizzata per aggiornare il valore di una risorsa;
4. DELETE è utilizzata per cancellare una risorsa.

Tutti i servizi, in qualche misura, operano sui dati. Per esempio, il servizio descritto nel Paragrafo 18.2, che restituisce le temperature massime e minime di una località, utilizza un database che contiene dati meteorologici. I servizi basati su SOAP svolgono azioni su questo database per restituire particolari valori contenuti nel database. I servizi RESTful (Richardson e Ruby 2007) accedono direttamente ai dati.

Quando viene utilizzato un approccio RESTful, i dati sono identificati e letti tramite il loro URL. I servizi RESTful usano i protocolli http o https, con le uniche azioni consentite POST, GET, PUT e DELETE. È possibile accedere ai dati meteorologici di ciascuna località memorizzati nel database utilizzando gli URL in questo modo:

```
http://weather-info-example.net/temperatures/boston
http://weather-info-example.net/temperatures/edinburgh
```

In questo modo, viene chiamata l'operazione GET, che restituisce una lista di temperature massime e minime. Per richiedere le temperature in una specifica data, si può utilizzare una query con l'identificatore URL:

```
http://weather-info-example.net/temperatures/edinburgh?date=20140226
```

Le query con URL possono essere utilizzate per eliminare eventuali ambiguità dalla richiesta, dato che ci possono essere più località nel mondo con lo stesso nome:

```
http://weather-info-example.net/temperatures/boston?date=20140226&country=USA&state="Mass"
```

Una differenza importante tra i servizi RESTful e i servizi SOAP è che i primi non sono servizi che si basano esclusivamente su XML. Pertanto, quando una risorsa viene richiesta, creata o modificata, è possibile specificare la sua rappresentazione. Questo aspetto è importante per i servizi RESTful, in quanto è possibile utilizzare rappresentazioni come JSON (Javascript Object Notation) e XML. Queste rappresentazioni possono essere elaborate più efficientemente delle notazioni basate su XML, riducendo gli overhead in una chiamata di servizio. Quindi, la precedente richiesta di temperature massime e minime per Boston potrebbe restituire le seguenti informazioni:

```
{  
  "place": "Boston",  
  "country": "USA",  
  "state": "Mass",  
  "date": "26 Feb 2014",  
  "units": "Fahrenheit",  
  "max temp": 41,  
  "min temp": 29  
}
```

La risposta a una richiesta GET in un servizio RESTful può includere gli identificatori URL. Pertanto, se la risposta a una richiesta è un insieme di risorse, allora potrebbe essere incluso l'URL di ciascuno di questi servizi. Il servizio richiedente potrebbe elaborare le richieste in un suo particolare modo. Quindi, una richiesta di informazioni meteorologiche con il nome di una località che non è unico potrebbe restituire gli URL di tutte le località che soddisfano la richiesta. Per esempio:

```
http://weather-info-example.net/temperatures/edinburgh-scotland  
http://weather-info-example.net/temperatures/edinburgh-australia  
http://weather-info-example.net/temperatures/edinburgh-maryland
```

Un principio fondamentale della progettazione dei servizi RESTful è che questi dovrebbero essere stateless; ovvero, in una sessione interattiva, la risorsa stessa non dovrebbe includere alcuna informazione sullo stato, come l'ora dell'ultima richiesta; piuttosto, tutte le informazioni necessarie sullo stato dovrebbero essere restituite al richiedente. Se le informazioni sullo stato sono necessarie in successive richieste, esse dovrebbero essere restituite al server dal richiedente.

I servizi RESTful si sono diffusi molto di più negli ultimi anni a causa della vasta diffusione dei dispositivi mobili. Questi dispositivi hanno capacità di elaborazione limitate, quindi i minori overhead dei servizi RESTful consentono migliori prestazioni dei sistemi. Essi sono anche più facili da utilizzare con i siti web esistenti – implementare un'API RESTful per un sito web, di solito, è un'operazione abbastanza semplice.

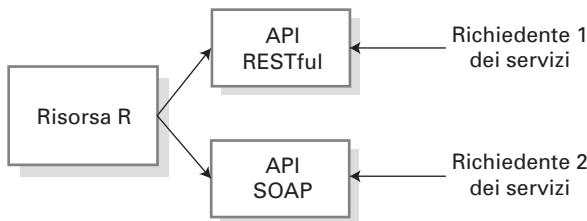


Figura 18.7 API RESTful e SOAP.

Tuttavia, l’approccio RESTful presenta alcuni problemi.

1. Quando un servizio ha un’interfaccia complessa e non è una risorsa semplice, può essere difficile progettare un insieme di servizi RESTful per rappresentare la sua interfaccia.
2. Non ci sono standard per la descrizione delle interfacce RESTful, quindi gli utenti dei servizi devono affidarsi alla documentazione informale per capire l’interfaccia.
3. Quando utilizzate i servizi RESTful, dovete implementare la vostra infrastruttura per monitorare e gestire la qualità dei servizi e la loro affidabilità. I servizi SOAP hanno standard aggiuntivi a supporto delle infrastrutture, come WS-Reliability e WS-Transactions.

Pautasso e altri (Pautasso, Zimmermann e Leymann 2008) hanno spiegato quando utilizzare i servizi RESTful e SOAP. Tuttavia, spesso è possibile fornire interfacce SOAP e RESTful allo stesso servizio o alla stessa risorsa (Figura 18.7). Questo approccio duale adesso è comune ai servizi cloud forniti da provider come Microsoft, Google e Amazon. I client dei servizi possono quindi scegliere il metodo di accesso ai servizi che meglio si adatta alle loro applicazioni.

18.3 Ingegneria dei servizi

L’ingegneria dei servizi è il processo che permette di sviluppare servizi riutilizzabili nelle applicazioni orientate ai servizi. Ha molto in comune con l’ingegneria dei componenti software. Gli ingegneri devono garantire che un servizio rappresenti un’astrazione che può essere riutilizzata in sistemi differenti. Devono progettare e sviluppare funzionalità genericamente utili associate a tale astrazione e assicurare che il servizio sia robusto e affidabile. Il servizio deve anche essere ben documentato, in modo che possa essere facilmente identificato e compreso dai potenziali utenti.

Come illustra la Figura 18.8, ci sono tre stadi logici nel processo di ingegneria dei servizi.

1. *Identificazione dei servizi candidati* – si identificano i servizi che potrebbero essere implementati e si definiscono i loro requisiti.

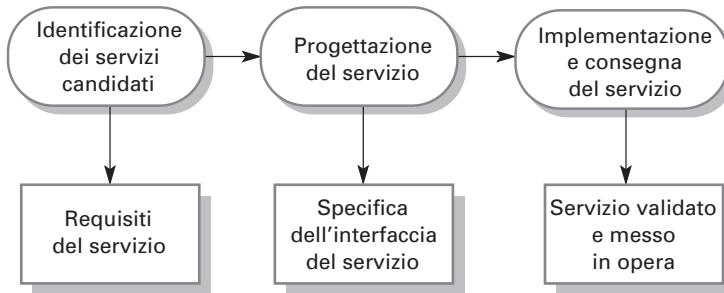


Figura 18.8 Il processo di ingegneria dei servizi.

2. *Progettazione dei servizi* – si sviluppano le interfacce logiche dei servizi e le relative interfacce di implementazione (SOAP e/o RESTful).
3. *Implementazione e consegna dei servizi* – si implementano e si provano i servizi e li si rende disponibili al pubblico.

Come detto nel Capitolo 16, lo sviluppo di un componente riutilizzabile può iniziare da un componente esistente che è già stato implementato e utilizzato in un'applicazione. Lo stesso vale per i servizi – il punto di partenza di questo processo di solito è un servizio esistente o un componente che deve essere convertito in un servizio. In questa situazione, il processo di progettazione richiede una generalizzazione del componente esistente in modo che siano rimosse le caratteristiche specifiche dell'applicazione. Implementare significa adattare il componente aggiungendo le interfacce dei servizi e poi implementare le generalizzazioni richieste.

18.3.1 Identificazione dei servizi candidati

Il concetto che sta alla base dell'elaborazione orientata ai servizi è che questi devono supportare i processi aziendali. Dato che ogni organizzazione dispone di una vasta gamma di processi, potrebbero essere implementati numerosi servizi. L'identificazione dei servizi candidati richiede quindi un'analisi dei processi aziendali di un'organizzazione per stabilire quali servizi riutilizzabili potrebbero essere implementati per supportare tali processi.

Erl (Erl 2005) identifica tre tipi fondamentali di servizi.

1. *Servizi di utilità*. Sono servizi che implementano una funzionalità generale, che può essere sfruttata da diversi processi aziendali. Un esempio potrebbe essere un servizio di conversione delle valute che calcola il cambio da una valuta (per esempio, il dollaro) a un'altra (per esempio, l'euro).
2. *Servizi aziendali*. Sono associati a una specifica funzione aziendale. Un esempio di funzione aziendale in una università potrebbe essere l'iscrizione degli studenti a un particolare corso di laurea.

	Servizio di utilità	Servizio aziendale	Coordinamento
Compito	Convertitore di valuta Localizzatore di impiegati	Convalida richiesta di rimborso Verifica stato del credito	Gestione rimborsi Pagamenti fornitori
Entità	Programma per la verifica dello stile di un documento Convertitore da form web a XML	Modulo spese Modulo di richiesta degli studenti	

Figura 18.9 Classificazione dei servizi.

3. *Servizi di coordinamento o di processo.* Sono servizi che supportano un processo aziendale più generale, che di solito coinvolge più attori e attività. Un esempio di servizio di coordinamento in un'azienda è il servizio di ordinazioni che consente di inviare ordini ai fornitori, di verificare il ricevimento delle merci ordinate e di effettuare i relativi pagamenti.

Erl suggerisce anche che i servizi possano essere suddivisi in servizi orientati ai compiti e servizi orientati alle entità. I primi riguardano qualche particolare attività, mentre i secondi sono associati a qualche risorsa del sistema. La risorsa è un'entità aziendale, per esempio un modulo per le richieste di assunzioni. La Figura 18.9 mostra alcuni esempi di servizi orientati ai compiti o alle entità. I servizi di utilità o aziendali possono essere orientati alle entità o ai compiti. I servizi di coordinamento sono sempre orientati ai compiti.

Scopo dell'identificazione dei servizi candidati è selezionare quei servizi che sono logicamente coerenti, indipendenti e riutilizzabili. In questo senso la classificazione di Erl è utile, perché indica come si possono scoprire i servizi riutilizzabili esaminando le entità aziendali come risorse e attività. Tuttavia, tale identificazione non è semplice, in quanto bisogna capire come i servizi potrebbero essere utilizzati. Bisogna compilare innanzitutto una lista di possibili candidati e poi porsi una serie di domande per appurare se tali servizi possono essere effettivamente utili. Ecco alcune possibili domande che potrebbero servire a identificare i servizi potenzialmente riutilizzabili.

1. Nel caso di un servizio orientato alle entità, il servizio è associato a una singola entità logica che è utilizzata in diversi processi aziendali? Quali operazioni normalmente eseguite su tale entità devono essere supportate? Queste operazioni soddisfano le operazioni PUT, CREATE, POST e DELETE dei servizi RESTful?
2. Nel caso di un servizio orientato ai compiti, l'attività in questione è svolta normalmente da persone diverse all'interno dell'organizzazione? Queste persone saranno disposte ad accettare l'inevitabile standardizzazione che si verifica quando è fornito un solo servizio di supporto? Tutto questo è conforme al modello RESTful o dovrà essere riprogettato come servizio orientato alle entità?

3. Il servizio è indipendente? In altre parole, fino a che punto si affida alla disponibilità di altri servizi?
4. È necessario che il servizio mantenga aggiornato uno stato interno? Se sono richieste informazioni sullo stato, queste dovranno essere memorizzate in un database o dovranno essere passate come parametro al servizio. L'uso di un database influisce sulla riutilizzabilità del servizio in quanto si crea una dipendenza del servizio dal database. In generale, i sistemi in cui lo stato è passato al servizio sono più facili da riutilizzare, in quanto non richiedono un database.
5. Il servizio potrà essere utilizzato da clienti esterni all'organizzazione? Per esempio, un servizio orientato alle entità associate a un catalogo potrebbe consentire l'accesso a utenti interni ed esterni.
6. Per quanto riguarda i requisiti non funzionali, è possibile che utenti diversi possano avere requisiti diversi? In caso affermativo, potrebbero essere implementate più versioni del servizio.

Le risposte a queste domande possono aiutare gli ingegneri a selezionare e affinare le astrazioni che possono essere implementate come servizi. Tuttavia, non esiste un metodo sperimentato per decidere quali sono i servizi migliori. L'identificazione dei servizi è un processo che si affida all'intuito e all'esperienza.

L'output di questo processo di selezione è un insieme di servizi con i relativi requisiti. I requisiti funzionali definiscono che cosa deve fare ogni servizio, mentre quelli non funzionali riguardano argomenti come la protezione, le prestazioni e la disponibilità.

Per aiutarvi a capire il processo di identificazione e implementazione dei servizi candidati, considerate il seguente esempio.

Un'azienda, che vende computer, ha previsto un piano di sconti per alcune configurazioni di computer da riservare ai clienti più importanti. Per facilitare le ordinazioni automatiche, l'azienda vuole rendere disponibile un servizio di catalogo che permetta agli utenti di scegliere le apparecchiature informatiche desiderate. A differenza del catalogo destinato ai consumatori finali, gli ordini non sono immessi direttamente tramite l'interfaccia del catalogo. Piuttosto, le apparecchiature vengono ordinate tramite un sistema di acquisizione basato sul Web di ogni singola azienda che accede al catalogo come un servizio web. La ragione di questo è che le grandi aziende di solito hanno procedure specifiche per l'approvazione dei budget e degli acquisti, che definiscono un processo che deve essere rispettato ogni volta che si effettua un'ordinazione.

Il servizio di catalogo è un esempio di servizio orientato alle entità, dove la risorsa sottostante è il catalogo. I requisiti funzionali di questo servizio sono elencati qui di seguito.

1. A ogni azienda utente sarà fornita una versione personalizzata del catalogo, che includerà le configurazioni e le attrezzature elettroniche che possono essere ordinate dai suoi dipendenti e i prezzi concordati.
2. Il catalogo permetterà a ogni cliente di scaricare una versione del catalogo consultabile off-line.
3. Il catalogo consentirà ai clienti di confrontare le specifiche e i prezzi di più articoli, fino a un massimo di sei.
4. Il catalogo fornirà agli utenti le funzioni di navigazione e di ricerca degli articoli.
5. Gli utenti potranno richiedere la data prevista di consegna per un determinato numero di articoli.
6. Gli utenti saranno in grado di effettuare “ordini virtuali” i cui articoli resteranno bloccati per 48 ore. Un ordine virtuale dovrà essere confermato da un ordine reale, da effettuare tramite il sistema entro 48 ore dalla creazione dell’ordine virtuale.

Oltre a questi requisiti funzionali, il catalogo ha anche alcuni requisiti non funzionali.

1. L’accesso al catalogo sarà limitato ai dipendenti delle aziende accreditate.
2. I prezzi e le configurazioni offerte a un cliente saranno confidenziali e nessun dipendente di altre aziende potrà accedervi.
3. Il catalogo sarà sempre disponibile, senza interruzioni di servizio, almeno dalle ore 0700 GMT alle 1100 GMT.
4. Il servizio sarà in grado di gestire come carico massimo di lavoro fino a 100 richieste al secondo.

Non c’è un requisito non funzionale riguardante il tempo di risposta del servizio. Questa caratteristica dipende dalle dimensioni del catalogo e dal numero di utenti presenti simultaneamente. Dato che il servizio non è critico rispetto al tempo, in questa fase non occorre specificare questo fattore delle prestazioni.

18.3.2 Progettazione dell’interfaccia dei servizi

Una volta identificati i siti candidati, il passo successivo nel processo di ingegneria dei servizi è progettare le interfacce dei servizi. Questo richiede la definizione delle operazioni associate al servizio e dei loro parametri. Se si usano servizi SOAP, bisogna progettare i messaggi di input e output. Se si usano i servizi RESTful, bisogna considerare quali risorse sono richieste e come devono essere utilizzate le operazioni standard per implementare quelle dei servizi.

Il punto di partenza della progettazione di un’interfaccia dei servizi è progettare un’interfaccia astratta, dove vengono identificate le entità e le operazioni associate al servizio, i loro input e output, e le eccezioni associate a queste ope-

Operazione	Descrizione
CreaCatalogo	Crea una versione personalizzata del catalogo per uno specifico cliente. Include un parametro facoltativo per consentire la creazione di una versione scaricabile del catalogo in formato PDF.
Visualizza	Visualizza tutti i dati associati a uno specifico elemento del catalogo.
Ricerca	Prende un'espressione logica e la utilizza per effettuare una ricerca nel catalogo; poi visualizza una lista di tutti gli articoli che corrispondono all'espressione di ricerca.
Confronta	Permette di confrontare fino a sei caratteristiche (ad esempio, prezzo, dimensioni, velocità del processore ecc.) degli articoli del catalogo, fino a un massimo di quattro articoli.
DataConsegna	Restituisce la data stimata della consegna se l'articolo in questione fosse ordinato nella data odierna.
CreaOrdineVirtuale	Prenota gli articoli specificati e fornisce le informazioni relative all'ordine al particolare sistema di acquisizioni ordini utilizzato dal cliente.

Figura 18.10 Le operazioni del servizio di catalogo.

razioni. Successivamente, occorre considerare come questa interfaccia astratta sia realizzata come servizi SOAP o RESTful.

Se scegliete un approccio SOAP, dovrete progettare la struttura dei messaggi XML che vengono inviati e ricevuti dal servizio. Le operazioni e i messaggi sono la base della descrizione di un'interfaccia scritta in WSDL. Se scegliete un approccio RESTful, dovete stabilire come le operazioni di un servizio possano essere realizzate con le operazioni RESTful.

La progettazione di un'interfaccia astratta inizia dai requisiti dei servizi e definisce i nomi delle operazioni e i parametri. In questa fase, occorre definire anche le eccezioni che possono sollevarsi quando viene invocata un'operazione di un servizio. La Figura 18.10 mostra le operazioni del catalogo che implementano i requisiti. In questa fase non è necessario che la specifica sia dettagliata: di questo ci si occuperà nella fase successiva del processo.

Una volta definita una descrizione logica informale di quello che il servizio dovrebbe fare, il passo successivo è aggiungere nuovi dettagli agli input e output del servizio. Questo è mostrato nella Figura 18.11, che estende la descrizione funzionale rappresentata nella Figura 18.10.

È particolarmente importante definire le eccezioni e come queste devono essere comunicate agli utenti di un servizio. Gli ingegneri dei servizi non sanno come i loro servizi saranno utilizzati. Di solito non è prudente supporre che gli utenti dei servizi abbiano capito perfettamente la specifica dei servizi. I messaggi di input potrebbero essere sbagliati, quindi occorre definire le eccezioni che segnalano gli input errati all'utente di un servizio. È buona norma durante lo sviluppo

Operazione	Input	Output	Eccezioni
CreaCatalogo	<i>ccln</i> ID dell'azienda Flag PDF	<i>ccOut</i> URL del catalogo per quell'azienda	<i>ccFault</i> ID azienda non valido
Visualizza	<i>visIn</i> URL del catalogo Numero di catalogo	<i>visOut</i> URL della pagina con le informazioni sull'articolo	<i>visFault</i> Numero di catalogo non valido
Ricerca	<i>ricIn</i> URL del catalogo Stringa di ricerca	<i>ricOut</i> URL della pagina web con i risultati della ricerca	<i>ricFault</i> Stringa di ricerca errata
Confronta	<i>confIn</i> URL del catalogo Caratteristiche (fino a 6) Numero di catalogo (fino a 4)	<i>confOut</i> URL della pagina con la tabella dei confronti	<i>confFault</i> ID azienda non valido Numero di catalogo non valido Caratteristica sconosciuta
DataConsegna	<i>dataln</i> ID dell'azienda Numero di catalogo Numero di articoli ordinati	<i>dataOut</i> Data di consegna stimata	<i>dataFault</i> ID azienda non valido Nessuna disponibilità Articoli ordinati zero
CreaOrdineVirtuale	<i>cordIn</i> ID dell'azienda Numero di catalogo Numero di articoli ordinati	<i>cordOut</i> Numero di catalogo Numero di articoli ordinati Data di consegna stimata Prezzo unitario stimato Prezzo totale stimato	<i>cordFault</i> ID azienda non valido Numero di catalogo non valido Articoli ordinati zero

Figura 18.11 Progetto dell'interfaccia del catalogo.

dei componenti riutilizzabili lasciare tutta la gestione delle eccezioni all'utente del componente. Gli sviluppatori dei servizi non dovrebbero imporre le loro idee sulle modalità di gestione delle eccezioni.

In alcuni casi, basta una descrizione testuale delle operazioni e dei loro input e output. La realizzazione dettagliata del servizio è lasciata agli implementatori. A volte, tuttavia, occorre un progetto più dettagliato, e una descrizione dettagliata dell'interfaccia può essere specificata in una notazione grafica, come nei diagrammi UML o in un formato leggibile come JSON. La Figura 18.12, che descrive gli input e gli output dell'operazione *DataConsegna*, mostra come utilizzare UML per descrivere l'interfaccia in dettaglio.

Si noti come siano stati aggiunti alcuni dettagli alla descrizione annotando il diagramma UML con i vincoli. Questi dettagli definiscono la lunghezza delle stringhe che rappresentano l'azienda e l'articolo del catalogo; specificano che il numero di articoli deve essere maggiore di zero e che la consegna deve avvenire

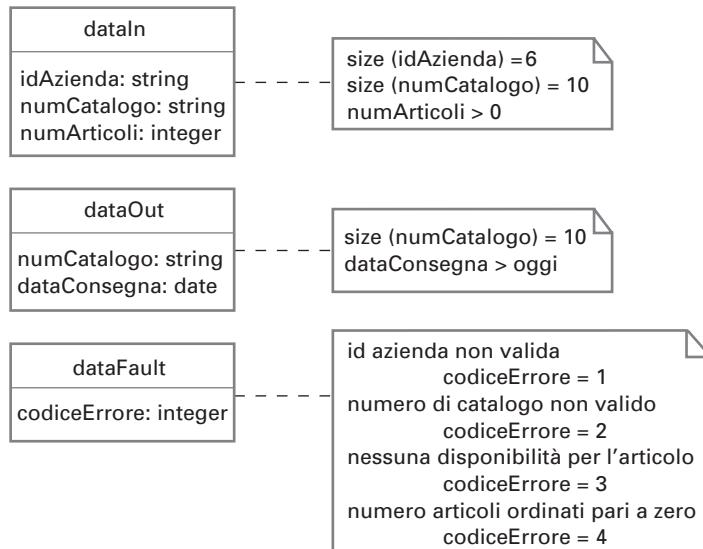


Figura 18.12 Definizione UML dei messaggi di input e output.

dopo quella corrente. Le annotazioni mostrano anche quali codici di errore sono associati a ogni possibile guasto.

Il servizio di catalogo è un esempio di servizio pratico, che dimostra che per implementare un servizio non è sempre facile scegliere tra approccio RESTful o approccio SOAP. Come servizio basato sulle entità, il catalogo può essere rappresentato come una risorsa, e questo suggerisce di utilizzare un modello RESTful. Tuttavia, le operazioni sul catalogo non sono semplici operazioni GET, e quindi occorre mantenere qualche stato in una sessione di interazione con il catalogo; questo suggerisce di utilizzare un approccio SOAP. Alcuni dilemmi sono comuni nell'ingegneria dei servizi, e di solito le circostanze locali (per esempio, la disponibilità di competenze) sono un fattore primario nella scelta dell'approccio da utilizzare.

Per implementare un insieme di servizi RESTful, occorre scegliere le risorse da utilizzare per rappresentare il catalogo e scegliere in che modo dovranno agire le operazioni GET, POST e PUT su queste risorse. Alcune delle decisioni di progettazione sono semplici:

1. occorre una risorsa che rappresenta un catalogo specifico di un'azienda, con un URL della forma `<catalogo base>/<nome dell'azienda>`, che dovrebbe essere creato utilizzando un'operazione POST;
2. ogni articolo del catalogo deve avere il suo URL con la forma `<catalogo base>/<nome dell'azienda>/<identificatore dell'articolo>`;

3. utilizzare l'operazione GET per identificare gli articoli. L'operazione *Visualizza* è implementata utilizzando l'URL di un articolo del catalogo come parametro GET. L'operazione *Ricerca* è implementata utilizzando GET con il catalogo dell'azienda come URL e la stringa di ricerca come parametro della query. L'operazione GET restituisce una lista di URL degli articoli che corrispondono alla stringa di ricerca.

Le operazioni *Confronta*, *DataConsegna* e *CreaOrdineVirtuale* sono invece più complesse:

1. l'operazione *Confronta* può essere implementata come una sequenza di operazioni GET per identificare i singoli articoli, seguita da un'operazione POST per creare la tabella di confronto e, infine, da un'operazione GET per restituire il risultato all'utente;
2. le operazioni *DataConsegna* e *CreaOrdineVirtuale* richiedono una risorsa addizionale che rappresenta un ordine virtuale. Un'operazione POST è utilizzata per creare questa risorsa con il numero di articoli richiesti. L'identificatore dell'azienda è utilizzato per completare automaticamente il modulo di ordinazione; poi viene calcolata la data di consegna. La risorsa può essere quindi identificata utilizzando un'operazione GET.

Occorre riflettere attentamente su come le eccezioni sono mappate ai codici di risposta standard http, come il codice 404 per indicare che un URL non può essere identificato. Non c'è spazio per spiegare qui questo argomento, ma esso aumenta ulteriormente il livello di complessità della progettazione dell'interfaccia dei servizi.

Per i servizi SOAP, il processo di realizzazione in questo caso è più semplice in quanto la progettazione dell'interfaccia logica può essere automaticamente tradotta in WSDL. Molti ambienti di programmazione che supportano lo sviluppo orientato ai servizi (per esempio, l'ambiente ECLIPSE) includono strumenti che possono tradurre la descrizione di un'interfaccia logica nella corrispondente rappresentazione WSDL.

18.3.3 Implementazione e consegna dei servizi

Una volta identificati i servizi candidati e progettate le loro interfacce, la fase finale del processo di ingegneria dei servizi consiste nell'implementare i servizi. Questa implementazione può essere realizzata programmando i servizi per mezzo di un linguaggio come Java o C#. Entrambi questi linguaggi includono librerie che supportano lo sviluppo di servizi SOAP e RESTful.

In alternativa, i servizi possono essere implementati creando interfacce con componenti esistenti o sistemi ereditati. Questo significa che il software che si è già dimostrato utile può essere riutilizzato. Nel caso dei sistemi ereditati, questo significa che le loro funzionalità possono essere accessibili alle nuove applicazioni. È anche possibile sviluppare nuovi servizi definendo delle composizioni di servizi esistenti, come spiegato nel Paragrafo 18.4.

Servizi basati su sistemi ereditati

I sistemi ereditati sono vecchi sistemi software che sono ancora in uso presso le aziende. Potrebbe non essere conveniente riscrivere o sostituire questi sistemi, e molte aziende preferirebbero utilizzarli con i sistemi più moderni. Uno degli utilizzi più importanti dei servizi è quello di implementare dei “wrapper” che forniscono l'accesso alle funzioni e ai dati di un sistema ereditato. I wrapper permettono l'accesso ai sistemi ereditati attraverso il Web e consentono di integrare questi sistemi in altre applicazioni.

<http://software-engineering-book.com/web/legacy-services/>

Una volta che un servizio è stato implementato, deve essere sottoposto a test prima di essere consegnato agli utenti. Per far questo, occorre innanzitutto esaminare e partizionare i suoi input (Capitolo 8), scrivere messaggi che riflettano queste combinazioni di input e poi verificare che gli output siano quelli desiderati. Durante i test è opportuno cercare di generare tutte le eccezioni, per controllare che il servizio sia in grado di gestire input non validi. Per i servizi SOAP, sono disponibili diversi strumenti di prova che permettono di esaminare un servizio e che generano test partendo da una specifica WSDL. Questi strumenti, tuttavia, possono solo dimostrare che l'interfaccia del servizio sia conforme alla specifica WSDL; non possono verificare il comportamento funzionale di un servizio.

Nell'ultima fase del processo, la consegna del servizio, il servizio viene reso disponibile per essere utilizzato in un server web. La maggior parte del software installato sui server rende questa operazione molto semplice. Basta installare il file che contiene il servizio eseguibile in una determinata directory; il servizio diventa automaticamente disponibile.

Se un servizio deve essere messo a disposizione di una grande azienda o è destinato a un pubblico molto vasto, è necessario fornire una documentazione agli utenti esterni del servizio. I potenziali utenti possono così valutare se il servizio è in grado di soddisfare le loro esigenze e se ha le caratteristiche di affidabilità e sicurezza. Le informazioni che voi, in qualità di provider del servizio, dovreste includere nella documentazione potrebbero essere le seguenti.

1. Informazioni sulla vostra azienda, sulle persone da contattare e così via. Queste informazioni sono importanti per gli utenti perché permettono loro di acquisire fiducia nel provider. Gli utenti esterni di un servizio devono avere la certezza che il servizio si comporterà in modo corretto. Tali informazioni permettono agli utenti di verificare le credenziali del provider presso le agenzie di informazioni aziendali.
2. Una descrizione informale delle funzionalità offerte dal servizio. Ciò aiuta i potenziali utenti a decidere se il servizio è adeguato alle loro esigenze.
3. Una descrizione delle modalità di utilizzo del servizio. Per servizi semplici, basta una descrizione informale che spiega i parametri di input e output. Per servizi SOAP più complessi, può essere pubblicata la descrizione WSDL.

4. Informazioni che permettono agli utenti di registrarsi per ottenere notifiche sugli aggiornamenti del servizio.

Un problema generale delle specifiche dei servizi sta nel fatto che il comportamento funzionale di un servizio è descritto solo informalmente nel linguaggio naturale. Le descrizioni nel linguaggio naturale sono facili da leggere, ma sono soggette a interpretazioni errate. Per risolvere questo problema, alcuni ricercatori hanno studiato l'uso di linguaggi ontologici per specificare la semantica dei servizi, associando al servizio informazioni ontologiche (W3C 2012). Purtroppo le specifiche ontologiche sono complesse e non facilmente comprensibili; per questo, non hanno avuto una vasta diffusione.

18.4 Composizione dei servizi

L'ingegneria del software orientato ai servizi si basa sulla composizione e sulla configurazione di servizi esistenti per crearne di nuovi. Questi nuovi servizi possono essere integrati in un'interfaccia utenti implementata in un browser per creare un'applicazione web, oppure possono essere usati come componenti di altri servizi. I servizi coinvolti nella composizione possono essere servizi sviluppati appositamente per un'applicazione, servizi sviluppati all'interno di un'azienda o forniti da un provider esterno. I servizi SOAP o RESTful possono essere composti per creare servizi con funzionalità estese.

Molte aziende stanno convertendo le loro applicazioni in sistemi orientati ai servizi, dove l'elemento costruttivo di base è un servizio, anziché un componente. In questo modo, si favorisce un riutilizzo più ampio del software all'interno di una stessa azienda. Oggi stiamo assistendo allo sviluppo di applicazioni interaziendali, dove un'azienda utilizza i servizi di altre aziende. L'ingegneria dei sistemi orientati ai servizi ha come obiettivo finale, a lungo termine, la creazione di un "mercato di servizi", nel quale i servizi sono forniti da provider esterni affidabili.

La composizione dei servizi può essere utilizzata per integrare processi aziendali distinti e ottenere un processo unificato con funzionalità più ricche. Per esempio, supponiamo che una compagnia aerea voglia offrire ai passeggeri un pacchetto vacanze completo. Oltre ai biglietti aerei, i passeggeri potrebbero prenotare anche gli alberghi nelle località preferite, noleggiare un'autovettura, prenotare un'auto per gli spostamenti, consultare guide turistiche e prenotare i biglietti per spettacoli locali. Per creare quest'applicazione la compagnia aerea dovrà comporre i suoi servizi di prenotazione con quelli degli alberghi, delle compagnie di autonoleggio e dei gestori degli spettacoli locali. Il risultato finale sarà un servizio unico che integra i servizi di tutti i differenti provider.

Questo processo può essere immaginato come una serie di fasi distinte, come illustra la Figura 18.13. Le informazioni passano da una fase all'altra. Per esempio, la compagnia di autonoleggio riceve come input l'orario di arrivo del volo. La sequenza delle fasi, detta *workflow*, rappresenta un insieme di attività ordinate

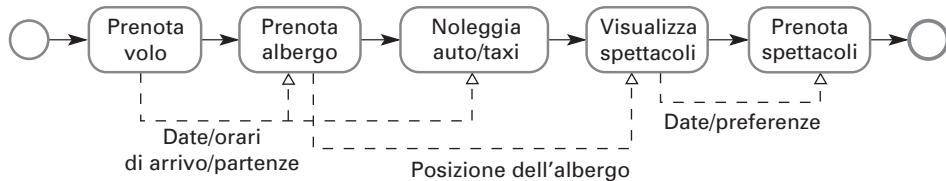


Figura 18.13 Workflow di un pacchetto vacanze.

nel tempo, in ciascuna delle quali viene svolta una parte del lavoro. Un workflow può essere considerato come il modello di un processo aziendale, nel quale ogni fase consente di ottenere un obiettivo parziale. In questo caso, il processo aziendale è il servizio di prenotazione di pacchetti vacanza, offerto dalla compagnia aerea.

Il workflow è un concetto semplice, come lo scenario appena descritto. In realtà, la composizione dei servizi è molto più complessa di quanto possa apparire esaminando questo semplice modello. Bisogna considerare la possibilità che un servizio possa fallire, e includere meccanismi appositi per gestire questi fallimenti. Inoltre qualche utente potrebbe porre al sistema richieste non standard: si pensi al caso di un utente disabile, che ha bisogno di speciali modalità di trasporto. Questi casi potrebbero richiedere servizi extra da implementare e comporre, con ulteriori fasi da aggiungere al workflow.

Quando si progetta un servizio turistico aggregato, occorre gestire anche le situazioni in cui l'esecuzione normale di uno dei servizi sia incompatibile con l'esecuzione di un altro servizio. Supponiamo per esempio che un utente selezioni un volo con partenza 1 giugno e ritorno 7 giugno. Il workflow a questo punto passa alla fase di prenotazione dell'albergo. Tuttavia, la località turistica prescelta ospita un importante congresso e negli hotel non ci sono stanze libere fino al 2 giugno. Il servizio di prenotazioni alberghiere segnala questo fatto. Non si tratta di un fallimento; l'indisponibilità di stanze è un evento comune.

Occorre quindi “annullare” la prenotazione del volo e passare all'utente le informazioni riguardanti la mancanza di stanze libere negli hotel. L'utente dovrà decidere se modificare la data di partenza o la località turistica. Nella terminologia del workflow, questa è chiamata “azione di compensazione”. Le azioni di compensazione servono ad annullare azioni che erano state precedentemente completate, ma che devono essere modificate a causa di successive attività del workflow.

Progettare nuovi servizi riutilizzando servizi preesistenti è un processo di progettazione del software con riutilizzo (Figura 18.13). La progettazione con riutilizzo inevitabilmente richiede di accettare compromessi sui requisiti. I requisiti “ideali” del sistema devono essere modificati in base ai servizi che sono effettivamente disponibili, i cui costi rientrano nel budget preventivato e la cui qualità è accettabile.

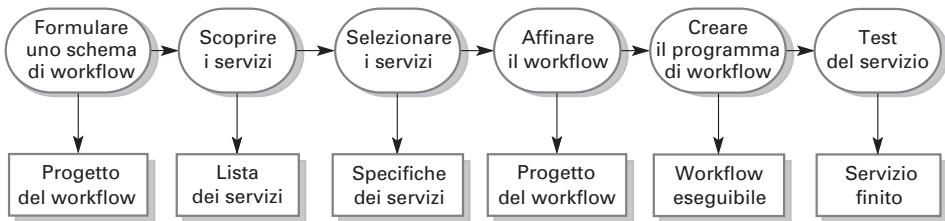


Figura 18.14 Costruzione di un servizio tramite composizione.

La Figura 18.14 mostra le sei fasi principali del processo di costruzione di un servizio mediante composizione.

1. *Formulare uno schema di workflow* – In questa fase iniziale della progettazione si usano i requisiti del servizio composto come base per progettare un servizio “ideale”. Il progetto iniziale deve essere astratto, prevedendo di aggiungere successivamente i dettagli, quando saranno disponibili maggiori informazioni sui servizi disponibili.
2. *Scoprire i servizi* – Durante questa fase del processo si effettua una ricerca per scoprire quali servizi esistenti possono essere inclusi nella composizione. Molti servizi possono essere trovati all’interno di una stessa azienda, quindi basta effettuare la ricerca nei cataloghi dei servizi locali. In alternativa, la ricerca può essere effettuata tra i servizi offerti da provider affidabili, come Oracle e Microsoft.
3. *Selezionare i possibili servizi* – Partendo dall’insieme dei servizi candidati che sono stati scoperti, si selezionano quelli che possono implementare le attività del workflow. I criteri di selezione naturalmente includeranno le funzionalità offerte dai servizi, ma è possibile includere anche il costo dei servizi e le loro qualità (tempo di risposta, disponibilità ecc.).
4. *Affinare il workflow* – Sulla base delle informazioni raccolte si può affinare il workflow, aggiungendo dettagli alla descrizione astratta e, talvolta, aggiungendo o rimuovendo alcune attività del workflow. A questo punto si devono ripetere le fasi di scoperta e selezione dei servizi. Una volta scelti i servizi e definito il progetto del workflow, si può passare alla fase successiva del processo.
5. *Creare il programma di workflow* – In questa fase il progetto astratto del workflow viene trasformato in un programma eseguibile e viene definita l’interfaccia del servizio. Per implementare il programma si può usare un linguaggio di programmazione come Java o C#, o un linguaggio specializzato nei workflow, come BPMN (descritto in seguito). Questa fase può includere la creazione di una o più interfacce web che consentono di accedere al nuovo servizio tramite un browser.

6. *Test del servizio o dell'applicazione* – Il processo di prova di un servizio composto è più complesso di quello di un sistema a più componenti quando sono utilizzati servizi esterni. I problemi relativi al test dei servizi sono descritti nel Paragrafo 18.4.2.

Il processo suppone che i servizi esistenti siano disponibili per la composizione. Se si è fatto affidamento su informazioni esterne che non sono disponibili tramite un'interfaccia di servizi, potrebbe essere necessario implementare questi servizi. Questo di solito richiede un processo di “screen scraping”, nel quale il vostro programma estraie le informazioni dal testo HTML delle pagine web che vengono inviate a un browser.

18.4.1 Progettazione e implementazione di un workflow

La progettazione di un workflow comincia con l'analisi dei processi aziendali esistenti o pianificati, allo scopo di comprenderne le fasi per rappresentarle con la notazione tipica dei workflow. In questo modo si pongono anche in evidenza le informazioni scambiate tra le diverse fasi. Tuttavia, i processi preesistenti potrebbero essere definiti informalmente o dipendere dall'abilità e dall'esperienza delle persone coinvolte. Potrebbe non esistere un modo “normale” di operare o di definire un processo. In questi casi occorre sfruttare le proprie conoscenze del processo per progettare un workflow che sia in grado di raggiungere gli stessi obiettivi.

I workflow rappresentano modelli di processi aziendali. Sono modelli grafici scritti tramite diagrammi di attività UML o notazioni BPMN (White e Miers 2008; OMG 2011). Negli esempi di questo capitolo ho utilizzato le notazioni BPMN (Business Process Modeling Notation). Se usate i servizi SOAP, è possibile convertire automaticamente i workflow BPMN in WS-BPEL, un linguaggio per workflow basato su XML. Ciò è conforme con altri standard per servizi web, quali SOAP e WSDL. I servizi RESTful possono essere composti all'interno di un programma in un linguaggio di programmazione standard come Java. In alternativa, si può utilizzare un linguaggio per la composizione dei servizi (Rosenberg et al. 2008).

La Figura 18.15 mostra un esempio di un semplice modello BPMN, relativo a una parte dello scenario di prenotazione dei pacchetti vacanze descritto nella Figura 18.14. Il modello mostra un workflow semplificato per la prenotazione di camere d'albergo e presume che esista un servizio *Alberghi* con le operazioni associate *GetRequisiti*, *VerificaDisponibilità*, *PrenotaCamere*, *NessunaDisponibilità*, *ConfermaPrenotazione* e *CancellaPrenotazione*. Il processo acquisisce informazioni dal cliente, verifica la disponibilità delle camere e, se ci sono camere libere, effettua la prenotazione nelle date richieste.

Questo modello introduce alcuni dei concetti principali di BPMN che sono utilizzati per creare modelli di workflow.

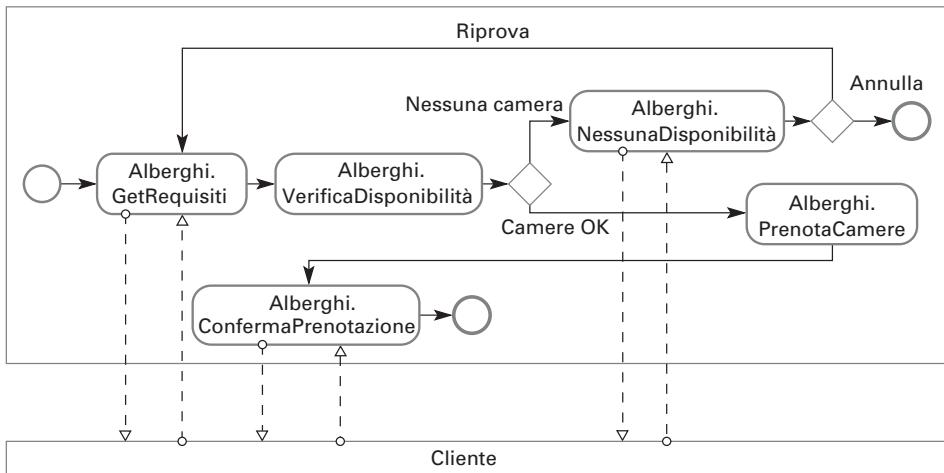


Figura 18.15 Parte del workflow per la prenotazione di una camera d'albergo.

- Le attività sono rappresentate da un rettangolo con gli angoli smussati. Un'attività può essere eseguita da un essere umano o da un servizio automatico.
- Gli eventi sono rappresentati da cerchi. Un evento è qualcosa che accade durante un processo aziendale. Per indicare un evento iniziale si utilizza un cerchio sottile; un evento finale è rappresentato da un cerchio dai bordi più spessi. Un cerchio doppio (non mostrato nella figura) rappresenta un evento intermedio. Gli eventi possono essere temporizzati, permettendo ai workflow di essere eseguiti periodicamente o sospesi temporaneamente (timeout).
- Un rombo rappresenta un gateway, ovvero una fase del processo in cui si deve fare una scelta. Per esempio, nella Figura 18.15, la scelta si basa sulla disponibilità delle camere dell'albergo.
- Una freccia con linea continua mostra la sequenza delle attività; una freccia con linea tratteggiata rappresenta il flusso dei messaggi tra le attività. Nella Figura 18.15 questi messaggi sono scambiati tra il servizio di prenotazione e il cliente.

Queste caratteristiche chiave sono sufficienti a descrivere la maggior parte dei workflow. BPMN include molte altre caratteristiche che non possono essere tratte qui per ragioni di spazio. Tra le altre cose è possibile aggiungere informazioni alla descrizione di un processo aziendale; ciò consente di tradurre automaticamente tale descrizione in un servizio eseguibile.

La Figura 18.15 mostra un processo eseguito all'interno di una singola azienda, quella che offre il servizio di prenotazione. Tuttavia, il beneficio principale dell'approccio orientato ai servizi è il supporto al calcolo interaziendale; ciò significa che un calcolo viene svolto da processi e servizi di varie aziende. Questo

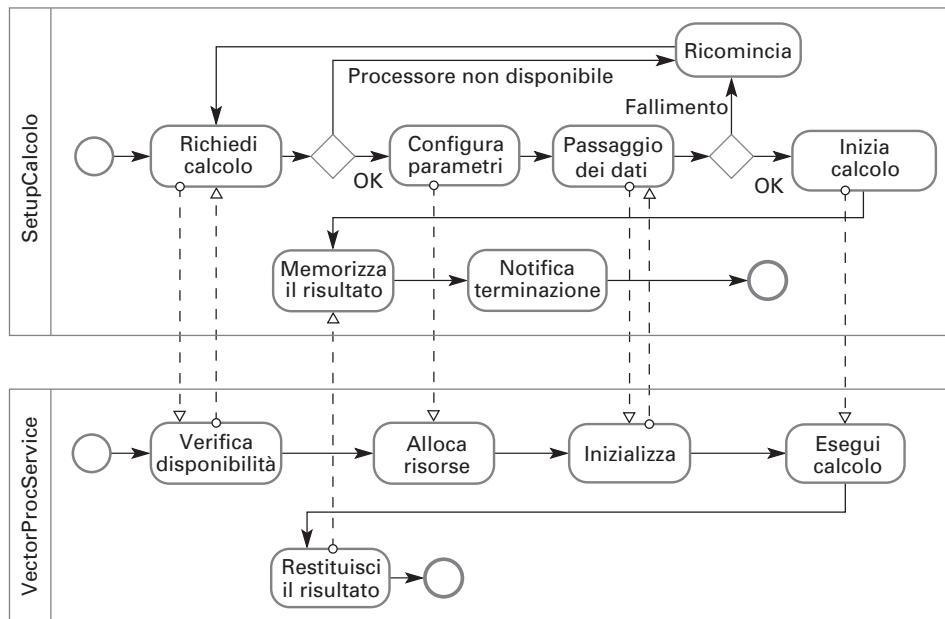


Figura 18.16 Interazioni tra più workflow.

processo è rappresentato nella notazione BPMN per mezzo di workflow separati, uno per ogni azienda coinvolta, con l'indicazione delle loro interazioni.

Per illustrare più processi di workflow utilizzerò un altro esempio, riferito al calcolo ad alte prestazioni, dove l'hardware è offerto come servizio. I servizi sono creati per fornire l'accesso a calcolatori ad alte prestazioni a una vasta comunità di utenti geograficamente distribuita. In questo esempio, un calcolatore per l'elaborazione vettoriale (una macchina che può svolgere calcoli in parallelo su array di valori) viene offerto come servizio (*VectorProcService*) da un laboratorio di ricerca. L'accesso a questo calcolatore avviene tramite un altro servizio detto *SetupCalcolo*. Questi servizi e le loro interazioni sono illustrati nella Figura 18.16.

In questo esempio, il workflow del servizio *SetupCalcolo* chiede innanzitutto l'accesso a un processore vettoriale e, se un processore è disponibile, specifica i parametri del calcolo e trasferisce i dati al servizio di calcolo. Una volta terminata l'elaborazione, i risultati vengono memorizzati sul computer locale. Il workflow di *VectorProcService* include le seguenti fasi:

- verifica che il processore sia disponibile;
- alloca le risorse per il calcolo;
- inizializza il sistema;
- esegue i calcoli richiesti;
- restituisce il risultato al servizio client.

Nella notazione BPMN, il workflow di ogni azienda è rappresentato in un gruppo separato; questo si rappresenta graficamente racchiudendo il processo in un rettangolo con il nome del servizio scritto verticalmente sul lato sinistro. Il coordinamento tra i workflow di ciascun gruppo si ottiene mediante lo scambio di messaggi. Nei casi in cui diverse parti della stessa azienda siano coinvolte in un workflow, i gruppi vengono suddivisi in “corsie”, ciascuna delle quali mostra le attività in quella parte dell’azienda.

Una volta progettato il modello del processo aziendale, occorre affinarlo in base ai servizi che sono stati scoperti. Come suggerito durante la discussione della Figura 18.14, può darsi che il modello debba subire diverse revisioni prima che possa essere creato un progetto che permetta di riutilizzare il maggior numero possibile di servizi disponibili.

Una volta completato il progetto, è possibile sviluppare il sistema finale orientato ai servizi. Per far ciò, occorre implementare quei servizi che non sono disponibili per il riutilizzo e convertire il modello del workflow in un programma eseguibile. Dato che i servizi sono indipendenti dal linguaggio di implementazione, i nuovi servizi possono essere scritti in un linguaggio qualsiasi. Il modello del workflow può essere automaticamente elaborato per creare un modello WS-BPEL eseguibile, se si usano servizi SOAP. In alternativa, se si usano servizi RESTful, il workflow può essere programmato manualmente, con il modello che agisce come specifica del programma.

18.4.2 Test delle composizioni dei servizi

Il test è una fase importante in tutti i processi di sviluppo, in quanto dimostra che un sistema soddisfa i suoi requisiti (funzionali e non) e rileva errori e difetti che potrebbero essere stati introdotti durante il processo di sviluppo. Molte tecniche di testing, come l’ispezione dei programmi e il coverage testing, si basano sull’analisi del codice sorgente del software. Il codice sorgente, tuttavia, non è disponibile nel caso in cui i servizi siano offerti da un provider esterno; quindi, non è possibile utilizzare tecniche di testing “a scatola bianca” che si basano sul codice sorgente del sistema.

Oltre ai problemi nel capire l’implementazione dei servizi, coloro che eseguono i test devono affrontare anche altre difficoltà durante i test delle composizioni dei servizi.

1. I servizi esterni sono controllati dal provider, non dall’utente. Il provider potrebbe ritirare o modificare in qualsiasi momento questi servizi, invalidando tutti i test svolti in precedenza. Questi problemi si affrontano nei componenti del software mantenendo più versioni dello stesso componente, ma di solito non è adottata la pratica di avere più versioni di un stesso servizio.
2. Se i servizi sono collegati dinamicamente, un’applicazione non sempre può utilizzare lo stesso servizio ogni volta che viene eseguito. Di conseguenza,

un test potrebbe avere successo quando l'applicazione è collegata a un particolare servizio, ma non potrà essere garantito il legame allo stesso servizio durante le successive esecuzioni. Questo problema è stato uno dei motivi per cui il collegamento dinamico non ha avuto una vasta diffusione.

3. Il comportamento non funzionale di un servizio non dipende semplicemente dal modo in cui è utilizzato dall'applicazione che si sta provando. Le prestazioni di un servizio potrebbero essere buone durante i test, perché il carico di lavoro è minimo. Nella pratica, il comportamento del servizio potrebbe variare a seconda delle richieste effettuate dagli altri utenti del servizio.
4. Il modello di pagamento adottato da molti servizi potrebbe rendere i testi molto costosi. Ci sono vari modelli di pagamento: alcuni servizi sono utilizzabili gratuitamente, altri richiedono un abbonamento e altri ancora richiedono un prezzo per ogni singolo utilizzo. Nel caso di servizi gratuiti, il provider non gradirà che vengano caricati da applicazioni in fase di test; se è richiesto un abbonamento, il cliente potrebbe essere riluttante a sottoscrivere l'abbonamento prima di aver concluso i test; analogamente, se il servizio richiede un pagamento per ogni utilizzo, il costo dei test potrebbe diventare proibitivo.
5. In precedenza ho accennato al concetto di “azione di compensazione”, che viene invocata quando si verifica un’eccezione e occorre annullare un’azione precedente (ad esempio la prenotazione di un biglietto aereo). Sottoporre a test queste azioni è problematico, perché possono dipendere dal fallimento di altri servizi; simulare il fallimento di questi servizi durante il test di solito è difficile.

Tutti questi problemi si accentuano quando si usano servizi esterni; sono meno gravi quando i servizi appartengono alla stessa azienda o ad aziende che cooperano e si fidano l’una dell’altra. In questi casi, il codice sorgente potrebbe essere disponibile per guidare il processo di test, e il pagamento dei servizi non sarà un problema. La soluzione di questi problemi di prova e la produzione di linee guida, strumenti e tecniche per testare le applicazioni orientate ai servizi sono temi di ricerca molto importanti.

Punti chiave

- L’architettura orientata ai servizi è un approccio all’ingegneria del software dove i servizi standard riutilizzabili sono i mattoni elementari per costruire nuovi sistemi applicativi.
- I servizi possono essere implementati all’interno di un’architettura orientata ai servizi utilizzando una serie di standard sui servizi web basati su XML, tra i quali figurano

gli standard per la comunicazione dei servizi, la definizione delle interfacce e la messa in opera dei servizi nei workflow.

- In alternativa, può essere utilizzata un'architettura RESTful, che si basa su risorse e operazioni standard su queste risorse. Un approccio RESTful usa i protocolli http e https nelle comunicazioni dei servizi e associa le operazioni ai comandi standard POST, GET, PUT e DELETE.
- I servizi possono essere classificati in servizi di utilità che forniscono una funzionalità generale, servizi aziendali che implementano parte di un processo aziendale e servizi di coordinamento che coordinano l'esecuzione di altri servizi.
- Il processo di ingegneria dei servizi richiede l'identificazione dei servizi candidati da implementare, la definizione dell'interfaccia dei servizi, e l'implementazione, i test e la consegna dei servizi.
- Lo sviluppo del software che usa i servizi si basa sul concetto che i programmi possono essere creati componendo e configurando servizi esistenti per creare nuovi sistemi e servizi composti.
- I linguaggi per workflow grafici, come BPMN, possono essere utilizzati per descrivere un processo aziendale e i servizi utilizzati in tale processo. Questi linguaggi possono descrivere le interazioni tra le aziende coinvolte nel processo.

Esercizi

- 18.1 Quali sono le differenze più importanti tra servizi e componenti software?
- 18.2 Gli standard sono fondamentali per le architetture orientate ai servizi; si ritiene che la conformità agli standard sia essenziale per il successo di un approccio basato sui servizi. Tuttavia, i servizi RESTful, la cui diffusione è in continua crescita, non si basano su standard. Spiegate perché si è verificato questo cambiamento e perché la mancanza di standard potrà inibire oppure no lo sviluppo e l'adozione dei servizi RESTful.
- * 18.3 Estendete la Figura 18.5 per includere le definizioni WSDL per *MaxMinType* e *InDataFault*. Le temperature dovrebbero essere rappresentate da numeri interi, con un campo aggiuntivo che indica se la temperatura è espressa in gradi Celsius o Fahrenheit. *InDataFault* dovrebbe essere un tipo di dati semplice formato da un codice di errore.
- 18.4 Definite la specifica di un'interfaccia per i servizi di conversione delle valute e di verifica dello stato del credito illustrati nella Figura 18.9.
- * 18.5 Suggerite come potrebbero essere implementati i servizi di conversione delle valute (*CurrencyConverter*) e di verifica dello stato del credito (*CheckCreditRating*) come servizi RESTful.
- 18.6 Definite i messaggi di input e output per i servizi illustrati nella Figura 18.13; potete specificarli nel linguaggio XML o UML.
- * 18.7 Spiegando la vostra risposta, suggerite due importanti tipi di applicazioni dove *non* sarebbe raccomandato l'utilizzo di un'architettura orientata ai servizi.

- * 18.8 Spiegate che cosa s'intende per “azione di compensazione” e, mediante un esempio, dimostrate perché queste azioni dovrebbero essere incluse nei workflow.
 - * 18.9 Per l'esempio del servizio di prenotazione del pacchetto vacanze, progettate un workflow che permetta di prenotare un mezzo di trasporto a terra per un gruppo di passeggeri che arrivano in un aeroporto. Gli utenti dovranno avere la possibilità di prenotare un'automobile o un taxi. Supponete che la società di noleggio dei taxi e delle automobili offra un servizio web per effettuare le prenotazioni.
- 18.10 Spiegate con un esempio perché è difficile eseguire un test completo dei servizi che includono azioni di compensazione.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Sul Web è disponibile un'enorme quantità di materiale didattico che tratta tutti gli aspetti dei servizi web. Tuttavia, credo che il libro di Thomas Erl fornisca la migliore panoramica e descrizione dei servizi e dei loro standard. Erl analizza anche i problemi dell'ingegneria del software nel calcolo orientato ai servizi. Inoltre ha scritto altri libri più recenti sui servizi RESTful.

Service-Oriented Architecture: Concepts, Technology and Design. Erl ha scritto alcuni libri sui sistemi orientati ai servizi che trattano le architetture SOA e RESTful. In questo libro, Erl esamina l'architettura orientata ai servizi e gli standard dei servizi web, concentrandosi principalmente su come un approccio orientato ai servizi possa essere utilizzato in tutte le fasi del processo software (T. Erl, Prentice-Hall, 2005).

“Service-oriented architecture.” Una buona e chiara introduzione all’architettura orientata ai servizi (autori vari 2008) <http://msdn.microsoft.com/en-us/library/bb833022.aspx>

“RESTful Web Services: The Basics.” Un buon tutorial introduttivo sul metodo RESTful e sui servizi RESTful (A. Rodriguez 2008). <https://www.ibm.com/developerworks/webservices/library/ws-restful/>

Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL, and RESTful Web Services. È un testo più avanzato per sviluppatori che desiderano utilizzare i servizi web in applicazioni aziendali. Descrive un certo numero di problemi tipici dei servizi web e le relative soluzioni (R. Daigneau, Addison-Wesley, 2012).

“Web Services Tutorial.” Questo è un ampio tutorial che tratta tutti gli aspetti dell’architettura orientata ai servizi, dei servizi web e dei loro standard; è stato scritto da persone coinvolte nello sviluppo di questi standard. È molto utile se avete bisogno di conoscere dettagliatamente gli standard (W3C schools, 1999-2014) <http://www.w3schools.com/webservices/default.asp>

Gestione del software

Qualcuno sostiene che la differenza principale tra l'ingegneria del software e altri tipi di programmazione stia nel fatto che l'ingegneria del software sia un processo gestito, ovvero che lo sviluppo del software avvenga all'interno di un'azienda e sia soggetto a una serie di vincoli organizzativi, di tempistica e di budget. Questa parte tratta diversi argomenti relativi alla gestione, focalizzandosi principalmente sui problemi della gestione tecnica, e secondariamente su problemi "più lievi" come la gestione del personale o la gestione più strategica dei sistemi aziendali.

I Capitoli 19 e 20 si occupano delle attività fondamentali della gestione della progettazione, della pianificazione, della gestione dei rischi e della gestione del personale. Il Capitolo 19 tratta la gestione della progettazione del software, con il primo paragrafo dedicato alla gestione dei rischi, dove i manager identificano gli elementi potenzialmente critici e i corrispondenti rimedi. Questo capitolo tratta anche la gestione del personale e il lavoro di squadra.

Il Capitolo 20 illustra le tecniche di stima e pianificazione dei progetti. Descriverò i grafici a barre come strumenti fondamentali di pianificazione e spiegherò perché lo sviluppo guidato da piani resterà un importante metodo di sviluppo, nonostante il successo dei metodi agili. Tratterò anche i fattori che influiscono sul prezzo di un sistema e le tecniche di stima dei costi del software. Utilizzerò la famiglia COCOMO dei modelli di costo per descrivere la modellazione algoritmica dei costi e presentare vantaggi e svantaggi degli approcci algoritmici.

Il Capitolo 21 descrive i temi fondamentali della gestione della qualità del software, come viene applicata nei grandi progetti. La gestione della qualità riguarda i processi e le tecniche che consentono di garantire e migliorare la qualità del software. Spiegherò la particolare importanza che gli standard rivestono nella gestione della qualità, l'uso delle ispezioni e delle revisioni nel processo di garanzia della qualità. L'ultimo paragrafo di questo capitolo, dedicato alle tecniche di misurazione del software, illustra i vantaggi e i problemi nell'uso di dati analitici e metriche del software nella gestione della qualità.

Infine, il Capitolo 22 tratta la gestione della configurazione, un argomento critico per tutti i grandi sistemi. La necessità della gestione della configurazione non è sempre chiara agli studenti che si sono occupati solo di sviluppo di software personale. Per questo saranno presentati i vari aspetti di questo

argomento, come la gestione delle versioni, la costruzione dei sistemi, la gestione delle modifiche e delle release. Spiegherò perché è importante l'integrazione continua o costruzione giornaliera di un sistema. Una significativa modifica in questa edizione è l'inclusione di nuovo materiale sui sistemi di gestione distribuita delle versioni, come Git, che si stanno diffondendo sempre più a supporto delle squadre distribuite di ingegneri del software.

CAPITOLO

19

Gestione della progettazione

L'obiettivo di questo capitolo è descrivere la gestione della progettazione del software e due importanti attività di gestione, ovvero la gestione dei rischi e la gestione del personale. Dopo aver letto questo capitolo:

- conoscerete i principali compiti dei manager della progettazione del software;
- apprenderete il concetto di gestione dei rischi e conoscerete alcuni dei rischi che possono sorgere durante la progettazione del software;
- conoscerete i fattori che influiscono sulle motivazioni del personale e che cosa potrebbero significare per i manager della progettazione del software;
- conoscerete i fattori chiave che influiscono sul lavoro di squadra, come la composizione e l'organizzazione di una squadra e la comunicazione tra i suoi membri.

- 19.1 Gestione dei rischi
- 19.2 Gestione del personale
- 19.3 Lavoro di squadra

La gestione dei progetti software è una parte essenziale dell'ingegneria del software. I progetti devono essere gestiti perché l'ingegneria del software professionale è sempre soggetta a vincoli aziendali di budget e di tempi. Il compito di un project manager è garantire che un progetto software soddisfi questi vincoli e abbia un livello elevato di qualità. La buona gestione non può garantire il successo di un progetto, ma la cattiva gestione di solito ne determina il fallimento: il software viene consegnato in ritardo, costa più del previsto e non soddisfa le aspettative dei clienti.

I criteri per il successo della gestione dei progetti ovviamente variano da progetto a progetto, ma per la maggior parte dei progetti, gli obiettivi più importanti sono:

- consegnare il software al cliente entro il termine concordato;
- mantenere i costi complessivi entro il budget;
- consegnare un software che soddisfa le richieste dell'utente;
- mantenere una squadra di sviluppo affiatata e collaudata.

Questi obiettivi non appartengono esclusivamente all'ingegneria del software, ma sono gli obiettivi di tutti i progetti di ingegneria. Tuttavia, l'ingegneria del software è diversa dagli altri tipi di ingegneria per varie ragioni che rendono la gestione del software particolarmente impegnativa. Alcune differenze sono elencate qui di seguito.

1. *Il prodotto è intangibile* – Il manager di un progetto di costruzione navale o di ingegneria civile può vedere il prodotto che sta sviluppando. Se i tempi di avanzamento si allungano, gli effetti sul prodotto sono visibili, in quanto alcune parti risultano incomplete. Il software invece è intangibile, perché non può essere visto o toccato. Il manager di un progetto software non può vedere i progressi osservando il manufatto che si sta costruendo, ma deve fidarsi di altre persone per avere la prova che il lavoro sta progredendo.
2. *I grandi progetti software sono spesso progetti “unici”* – Di solito, un grande progetto di sviluppo del software è unico, in quanto qualsiasi ambiente dove il software è sviluppato è in qualche modo diverso da tutti gli altri. Anche i manager di lunga esperienza hanno difficoltà nel prevedere i problemi. Inoltre, i rapidi cambiamenti tecnologici dei computer e delle comunicazioni possono rendere obsoleta l'esperienza acquisita. Le lezioni imparate da precedenti progetti potrebbero non essere più valide per i nuovi progetti.
3. *I processi software sono variabili e specifici di un’organizzazione* – I processi di ingegneria per alcuni tipi di sistemi, come i ponti e gli edifici, sono ben noti. I processi di sviluppo del software, invece, variano totalmente da un'organizzazione all'altra; non si può prevedere in maniera attendibile quando un particolare processo software potrà causare problemi durante il suo sviluppo. Questo è particolarmente vero quando il progetto software è

parte di un più ampio progetto di ingegneria di sistemi o quando si sta sviluppando un software completamente nuovo.

A causa di questi problemi, non ci si può sorprendere se alcuni progetti software siano in ritardo o superino il budget preventivato. I sistemi software sono spesso nuovi, molto complessi e tecnicamente innovativi. Anche in altri progetti di ingegneria complessi e innovativi, come i nuovi sistemi di trasporto, spesso vengono superati i termini di consegna e i limiti di budget. Viste le enormi difficoltà, è davvero straordinario notare che molti progetti software siano consegnati puntualmente e rientrino nel budget.

È impossibile fornire una descrizione dei compiti standard per un manager di progetti software. I compiti variano notevolmente in funzione dell’azienda e del tipo di software che si sta sviluppando. Alcuni dei più importanti fattori che influiscono sulla gestione dei progetti software sono elencati qui di seguito.

1. *Dimensione dell’azienda* – Le piccole aziende possono operare con una gestione e comunicazioni informali, non avendo bisogno di politiche e strutture gestionali formali. Esse hanno minori overhead di gestione rispetto alle grandi organizzazioni. In queste organizzazioni, devono essere rispettate le gerarchie gestionali, report e budget formali e procedure di approvazione.
2. *Clienti del software* – Se il cliente è interno (come nel caso dello sviluppo di un prodotto software), allora le comunicazioni con il cliente possono essere informali e non c’è bisogno di adattarsi al modo di operare del cliente. Se deve essere sviluppato un software personalizzato per un cliente esterno, bisogna raggiungere un accordo sui canali di comunicazione formali. Se il cliente è un’agenzia governativa, la società di software deve operare secondo le procedure e le politiche dell’agenzia, che di solito sono burocratiche.
3. *Dimensione del software* – I piccoli sistemi software possono essere sviluppati da un piccolo team, i cui membri possono riunirsi nella stessa stanza per discutere l’avanzamento del progetto e altri problemi di gestione. I grandi sistemi software di solito richiedono più squadre di sviluppo che possono essere geograficamente distribuite e appartenere a società diverse. Il manager dei progetti deve coordinare le attività di queste squadre e predisporre gli strumenti di comunicazione tra i loro membri.
4. *Tipo di software* – Se il software da sviluppare è un prodotto per un cliente, non è necessario registrare formalmente le decisioni sulla gestione dei progetti. Se, invece, si sta sviluppando un sistema a sicurezza critica, tutte le decisioni sulla gestione dei progetti dovranno essere registrate e giustificate in quanto possono influire sulla sicurezza del sistema.
5. *Cultura aziendale* – Alcune aziende preferiscono supportare e incentivare i singoli individui, mentre altre preferiscono supportare il lavoro di gruppo. Le grandi aziende spesso sono formali e burocratiche. Alcune aziende

hanno una cultura avversa ai rischi, mentre altre sono più propense al rischio.

6. *Processo di sviluppo del software* – I processi agili tipicamente tentano di operare con una gestione “leggera”. I processi più formali richiedono il monitoraggio della gestione per garantire che la squadra di sviluppo stia seguendo il processo stabilito.

Questi fattori indicano che i project manager di aziende diverse operano in modi completamente differenti. Tuttavia, alcune attività di gestione dei progetti sono comuni a tutte le aziende.

1. *Pianificazione dei progetti* – I project manager sono responsabili della pianificazione, della stima e della tempistica dello sviluppo dei progetti; inoltre devono assegnare i compiti al personale. Hanno la supervisione del lavoro per garantire che esso sia svolto secondo gli standard, e tengono sotto controllo l'avanzamento del lavoro per verificare che esso venga sviluppato in tempo ed entro il budget previsto.
2. *Gestione dei rischi* – I project manager devono definire i rischi che potrebbero influenzare un progetto, monitorare questi rischi e svolgere le azioni appropriate quando si presenta qualche problema.
3. *Gestione del personale* – I project manager sono responsabili della gestione di una squadra di persone. Devono selezionare le persone della loro squadra e definire le modalità operative che possono migliorare l'efficienza del lavoro di gruppo.
4. *Reporting* – I project manager di solito hanno il compito di documentare il progresso di un progetto in appositi report da inviare ai clienti e ai manager della società che sviluppa il software. Devono essere in grado di produrre report di vari livelli, dai documenti tecnici dettagliati ai rapporti sintetici sulla gestione. Devono scrivere documenti concisi e coerenti che riassumono le informazioni critiche estratte dai report dettagliati dei progetti. Devono essere in grado di fornire queste informazioni durante le revisioni dell'avanzamento dei lavori.
5. *Scrittura delle proposte* – La prima fase di un progetto software potrebbe richiedere la scrittura di una proposta per aggiudicarsi un contratto di realizzazione di una parte del progetto. La proposta descrive gli obiettivi del progetto e come saranno raggiunti; di solito include le stime dei costi e dei tempi e spiega perché il contratto dovrebbe essere assegnato a una particolare società o team di sviluppo. La scrittura delle proposte è un compito critico, in quanto la sopravvivenza di molte società di software dipende dal numero di proposte accettate e di appalti aggiudicati.

La pianificazione dei progetti è un argomento molto importante, per questo lo descriverò dettagliatamente nel Capitolo 20. In questo capitolo concentreremo la nostra attenzione sulla gestione dei rischi e del personale.

19.1 Gestione dei rischi

La gestione dei rischi è uno dei compiti più importanti di un project manager. Un rischio può essere immaginato come qualcosa che preferiremmo non avvenisse mai. I rischi possono minacciare un progetto, il software in via di sviluppo o l'organizzazione. La gestione dei rischi consiste nel prevedere i rischi che potrebbero influire sulla tempistica di un progetto o sulla qualità del software che si sta sviluppando, e nel decidere le azioni da svolgere per evitare tali rischi (Hall 1998; Ould 1999).

I rischi possono essere classificati in base alla loro natura (tecnica, organizzativa ecc.), come dirò nel Paragrafo 19.1.1. Un'altra classificazione dei rischi si basa sull'oggetto dei loro effetti:

1. *rischi per il progetto*; influiscono sulle risorse e sulla tempistica del progetto. Un esempio è la perdita di un progettista esperto. Trovare un sostituto con l'esperienza e le capacità adeguate potrebbe richiedere molto tempo; di conseguenza, occorrerà più tempo per sviluppare il progetto rispetto a quanto originariamente programmato;
2. *rischi per il prodotto*; influiscono sulla qualità o sulle prestazioni del software che si sta sviluppando. Un esempio è un componente acquistato che non funziona come previsto. Questo potrebbe influire sulle prestazioni complessive del sistema, che diventa più lento del previsto;
3. *rischi per l'azienda*; influiscono sull'azienda che sta sviluppando o acquistando il software. Un esempio è un'azienda concorrente che sviluppa un nuovo prodotto. L'introduzione di un prodotto competitivo potrebbe invalidare alcune ipotesi sui volumi di vendita dei prodotti software esistenti.

Ovviamente questi tipi di rischi possono sovrapporsi. Un programmatore esperto che decide di abbandonare un progetto rappresenta un *rischio per il progetto*, perché viene ritardata la consegna del software. Inevitabilmente, occorrerà del tempo prima che il suo sostituto possa capire il lavoro che è stato fatto; quindi il sostituto non potrà essere immediatamente produttivo. Di conseguenza, la consegna del sistema sarà ritardata. La perdita di un membro della squadra può essere anche un *rischio per il prodotto*, in quanto un sostituto senza la stessa esperienza potrebbe commettere errori di programmazione. Infine, la perdita di un membro esperto della squadra potrebbe essere anche un *rischio per l'azienda*, perché la reputazione di un ingegnere esperto potrebbe essere un fattore critico per aggiudicarsi nuovi contratti d'appalto.

Per i grandi progetti, è necessario scrivere i risultati dell'analisi dei rischi in un apposito registro insieme con un'analisi delle conseguenze. Questo serve a definire le conseguenze dei rischi per il progetto, il prodotto e l'azienda. Una gestione efficiente dei rischi semplifica la risoluzione dei problemi e assicura che questi non comportino sforamenti inaccettabili nei tempi e nei costi preventivati.

Rischio	Influisce su	Descrizione
Turnover del personale	Progetto	Il personale esperto può abbandonare il progetto prima che sia finito.
Cambio della gestione	Progetto	Ci potrebbe essere un cambio nella gestione aziendale con priorità differenti.
Hardware non disponibile	Progetto	L'hardware che è essenziale per il progetto potrebbe non essere consegnato nei tempi previsti.
Modifiche dei requisiti	Progetto e prodotto	Numero di modifiche dei requisiti superiore al previsto.
Ritardi nelle specifiche	Progetto e prodotto	Le specifiche delle interfacce essenziali non sono disponibili nei tempi previsti.
Sottostima della dimensione	Progetto e prodotto	La dimensione del sistema è stata sottostimata.
Prestazioni insoddisfacenti degli strumenti software	Prodotto	Gli strumenti software che supportano il progetto non funzionano come previsto.
Nuove tecnologie	Azienda	La tecnologia su cui si basa il sistema viene sostituita da nuova tecnologia.
Concorrenza sul prodotto	Azienda	Un prodotto concorrente viene immesso sul mercato prima che il sistema sia stato completato.

Figura 19.1 Esempi di rischi comuni per il progetto, il prodotto e l'azienda.

Per i piccoli progetti, la registrazione formale dei rischi potrebbe non essere richiesta, ma il project manager dovrebbe conoscerne le conseguenze.

I rischi specifici che possono influire su un progetto dipendono dal tipo di progetto e dall'ambiente organizzativo in cui il software è sviluppato. Tuttavia, ci sono rischi comuni che sono indipendenti dal tipo di software che si sta sviluppando. Questi rischi possono presentarsi in qualsiasi progetto di sviluppo del software. Alcuni esempi di questi rischi comuni sono riportati nella Figura 19.1.

La gestione dei rischi è importante a causa delle incertezze intrinseche nello sviluppo del software. Queste incertezze scaturiscono dai requisiti definiti in modo approssimativo, dalle modifiche dei requisiti dovute alle mutate esigenze dei clienti, dalle difficoltà nella stima del tempo e delle risorse necessarie allo sviluppo del software, e dalle differenti capacità dei singoli membri della squadra di sviluppo. Bisogna saper prevedere i rischi, capire il loro impatto sul progetto, sul prodotto e sull'azienda, e prendere le decisioni appropriate per evitarli. Potrebbe essere necessario elaborare dei piani di emergenza in modo che, se si concretizza un rischio, si possa trovare immediatamente un opportuno rimedio.

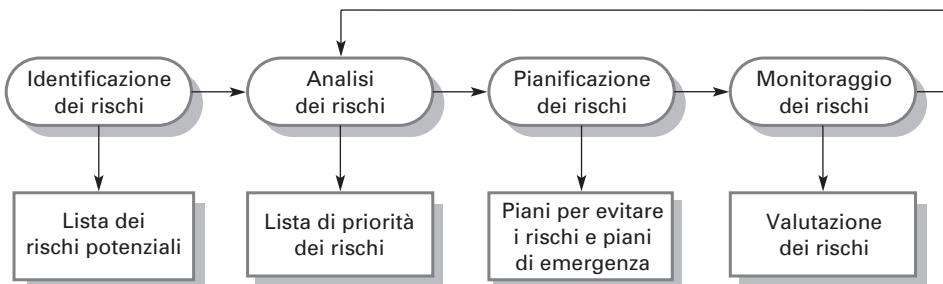


Figura 19.2 Schema della gestione dei rischi.

Uno schema generico della gestione dei rischi è illustrato nella Figura 19.2, dove sono rappresentate varie fasi:

1. *identificazione dei rischi*; vengono identificati i rischi per il progetto, il prodotto e l'azienda;
2. *analisi dei rischi*; vengono valutate le probabilità e le conseguenze di tali rischi;
3. *pianificazione dei rischi*; si definiscono i piani per evitare i rischi e i rimedi per minimizzarne gli effetti;
4. *monitoraggio dei rischi*; i rischi sono costantemente controllati e si ridefiniscono i piani per attenuarne gli effetti, non appena sono disponibili nuove informazioni.

Per i grandi progetti, si dovrebbero documentare i risultati del processo di gestione dei rischi in un apposito piano di gestione. Questo piano dovrebbe includere una descrizione dei rischi cui è sottoposto il progetto, un'analisi di questi rischi e le informazioni sulle modalità di intervento nel caso questi rischi dovessero concretizzarsi.

La gestione dei rischi è un processo iterativo che continua per tutto lo sviluppo del progetto. Una volta definito il piano iniziale di gestione dei rischi, occorre monitorare costantemente la situazione per rilevare un'eventuale insorgenza dei rischi. Se si rendono disponibili nuove informazioni, occorre analizzare di nuovo i rischi ed eventualmente stabilire le nuove priorità. Potrebbe essere necessario aggiornare anche i piani per evitare i rischi e quelli di emergenza.

La gestione dei rischi nello sviluppo agile è meno formale. Si dovrebbero svolgere le stesse attività fondamentali per analizzare i rischi, sebbene non sia necessario documentarle formalmente. Lo sviluppo agile riduce alcuni rischi, come quelli derivanti dalle modifiche dei requisiti. Tuttavia, lo sviluppo agile presenta alcuni svantaggi. A causa della sua dipendenza dal personale, il turnover del personale potrebbe avere effetti significativi sul progetto, sul prodotto e sull'azienda. A causa della mancanza di documentazione e comunicazioni formali, è molto difficile mantenere la continuità e il ritmo nello sviluppo di un progetto nel caso in cui le persone chiave abbondonino il progetto.

19.1.1 Identificazione dei rischi

L'identificazione dei rischi è la prima fase del processo di gestione dei rischi. Si tratta di identificare i rischi che potrebbero danneggiare seriamente il processo di ingegneria del software, il software che si sta sviluppando e la società di sviluppo. L'identificazione potrebbe essere un processo di squadra i cui membri si riuniscono per cercare di scoprire i potenziali rischi. In alternativa, i project manager possono identificare i rischi in base alla loro esperienza sugli errori che si sono verificati in precedenti progetti.

Come punto di partenza per l'identificazione dei rischi, si può utilizzare una lista che elenca i vari tipi di rischi, come la seguente:

1. *rischi di stima*; derivano dalle stime delle risorse che dovrebbero servire al sistema che si sta costruendo;
2. *rischi aziendali*; derivano dall'ambiente aziendale in cui si sta sviluppando il software;
3. *rischi riguardanti il personale*; sono associati alle persone che fanno parte del team di sviluppo;
4. *rischi dei requisiti*; derivano dalle modifiche dei requisiti del cliente e del processo di gestione dei requisiti;
5. *rischi tecnologici*; derivano dalle tecnologie hardware o software che sono utilizzate per sviluppare il sistema;
6. *rischi degli strumenti*; derivano dal software di supporto e dagli strumenti software che sono utilizzati per sviluppare il sistema.

La Figura 19.3 mostra alcuni esempi di possibili rischi in ciascuna di queste categorie. Una volta completato il processo di identificazione dei rischi, si dovrebbe avere una lista di potenziali rischi che potrebbero influire sul prodotto, sul processo o sull'azienda. Se il numero dei rischi è troppo elevato, potrebbe essere praticamente impossibile monitorarli tutti.

19.1.2 Analisi dei rischi

Durante il processo di analisi dei rischi, bisogna considerare i singoli rischi che sono stati identificati e valutarne la probabilità e la gravità. Non c'è un modo semplice per farlo; dovete affidarvi alle valutazioni e all'esperienza acquisita nei precedenti progetti. Non è possibile fare delle valutazioni numeriche precise della probabilità e della gravità di ciascun rischio; piuttosto, è consigliabile associare il rischio a una delle varie bande di rischio:

1. la probabilità di rischio può essere stimata molto bassa, bassa, moderata, alta o molto alta;
2. gli effetti del rischio possono essere stimati catastrofici (minacciano la sopravvivenza stessa del progetto), gravi (procurano notevoli ritardi), tollerabili (i ritardi sono entro i margini consentiti) o insignificanti.

Tipo di rischio	Possibili rischi
Stima	1. Il tempo richiesto per sviluppare il software è sottostimato. 2. Il tasso di correzione dei difetti è sottostimato. 3. La dimensione del software è sottostimata.
Azienda	4. L'organizzazione aziendale viene ristrutturata cambiando i responsabili della gestione del progetto. 5. Problemi finanziari dell'azienda impongono la riduzione del budget per il progetto.
Personale	6. Impossibilità di assumere personale con le capacità richieste. 7. Elementi chiave del personale si ammalano e non sono disponibili nei momenti critici. 8. Impossibilità di addestrare il personale come richiesto.
Requisiti	9. Le modifiche dei requisiti richiedono una significativa revisione del progetto. 10. I clienti non riescono a valutare l'impatto delle modifiche dei requisiti sul processo di sviluppo.
Tecnologia	11. Il database usato nel sistema non può elaborare il numero di transazioni al secondo che era stato previsto. 12. I difetti dei componenti software esistenti devono essere riparati prima di poter essere riutilizzati.
Strumenti	13. Il codice generato dagli strumenti software non è efficiente. 14. Gli strumenti software non possono essere integrati.

Figura 19.3 Esempi di vari tipi di rischi.

È possibile classificare i risultati di questa analisi inserendoli in una tabella ordinata secondo la gravità dei rischi. La Figura 19.4 illustra un esempio di tabella per i rischi identificati nella Figura 19.3. Ovviamente, la valutazione delle probabilità e della gravità è arbitraria. Per fare queste valutazioni, è necessario avere informazioni dettagliate sul progetto, sul processo, sul team di sviluppo e sull'azienda.

Ovviamente, la probabilità e la valutazione degli effetti di un rischio possono cambiare quando si ottengono nuove informazioni e quando si implementano i piani di gestione dei rischi. Di conseguenza, questa tabella andrebbe aggiornata dopo ogni iterazione del processo di gestione dei rischi.

Una volta che i rischi sono stati analizzati e classificati, bisognerebbe valutare quali sono quelli più significativi. Il giudizio si dovrà basare su una combinazione della probabilità di concretizzazione dei rischi e dei loro effetti. In generale i rischi catastrofici dovrebbero essere presi sempre in considerazione, così come tutti i rischi gravi che hanno probabilità di concretizzazione più che moderate.

Rischio	Probabilità	Effetti
Problemi finanziari dell'azienda impongono la riduzione del budget per il progetto (5).	Bassa	Catastrofici
Impossibilità di assumere personale con le capacità richieste (6).	Alta	Catastrofici
Elementi chiave del personale si ammalano e non sono disponibili nei momenti critici (7).	Moderata	Gravi
I difetti dei componenti software esistenti devono essere riparati prima di poter essere riutilizzati (12).	Moderata	Gravi
Le modifiche dei requisiti richiedono una significativa revisione del progetto (9).	Moderata	Gravi
L'organizzazione aziendale viene ristrutturata cambiando i responsabili della gestione del progetto (4)	Alta	Gravi
Il database usato nel sistema non può elaborare il numero di transazioni al secondo che era stato previsto (11)	Moderata	Gravi
Il tempo richiesto per sviluppare il software è sottostimato (1)	Alta	Gravi
Gli strumenti software non possono essere integrati (14)	Alta	Tollerabili
I clienti non riescono a valutare l'impatto delle modifiche dei requisiti (10).	Moderata	Tollerabili
Impossibilità di addestrare il personale come richiesto (8)	Moderata	Tollerabili
Il tasso di correzione dei difetti è sottostimato (2)	Moderata	Tollerabili
La dimensione del software è sottostimata (3)	Alta	Tollerabili
Il codice generato dagli strumenti software non è efficiente (13)	Moderata	Insignificanti

Figura 19.4 Tabella di analisi dei rischi.

Boehm (Boehm 1988) consiglia di identificare e monitorare i 10 rischi più importanti (“top 10”). Io ritengo che il numero giusto di rischi da considerare dipenda dal progetto che si sta sviluppando; potrebbe essere 5 o 15. Per i rischi identificati nella Figura 19.4, è opportuno considerare solo quelli che hanno conseguenze serie o catastrofiche (Figura 19.5).

19.1.3 Pianificazione dei rischi

Il processo di pianificazione dei rischi serve a sviluppare le strategie per gestire i rischi che minacciano il progetto. Per ogni rischio occorre considerare le azioni

Rischio	Strategia
Problemi finanziari dell'azienda	Preparare un breve documento per i dirigenti dell'azienda per dimostrare come il progetto stia apportando un contributo molto importante agli obiettivi economici e per spiegare le ragioni per le quali un taglio del budget del progetto potrebbe essere economicamente svantaggioso.
Problemi di selezione del personale	Informare il cliente sulle potenziali difficoltà e sui possibili ritardi; considerare l'acquisto di nuovi componenti.
Malattia del personale	Riorganizzare il team di sviluppo in modo che ci sia una maggiore sovrapposizione di lavoro e ogni membro del team possa capire il lavoro degli altri.
Componenti difettosi	Sostituire i componenti potenzialmente difettosi con altri di affidabilità superiore.
Modifiche dei requisiti	Acquisire le informazioni di tracciabilità per valutare l'impatto delle modifiche dei requisiti sul processo di sviluppo; massimizzare la quantità di informazioni nascoste nel progetto.
Ristrutturazione aziendale	Preparare un breve documento per i dirigenti per dimostrare come il progetto stia apportando un contributo notevole agli obiettivi economici dell'azienda.
Prestazioni del database	Esaminare la possibilità di acquistare un database di prestazioni superiori.
Tempo di sviluppo sottostimato	Considerare l'acquisto di nuovi componenti e l'utilizzo di un generatore automatico di codice.

Figura 19.5 Strategie che aiutano la gestione dei rischi.

che potrebbero essere svolte per minimizzare gli effetti dannosi sul progetto, se si verifica il problema identificato in un rischio. Occorre anche considerare quali informazioni raccogliere durante il monitoraggio del progetto, in modo che possano essere rilevati i problemi al loro insorgere, prima che il loro effetti diventino gravi.

Durante la pianificazione dei rischi, occorre porsi alcune domande del tipo “che cosa accade, se ...” per esaminare singoli rischi, combinazioni di rischi e fattori esterni che influiscono sui rischi. Ecco alcune di queste domande:

1. Che cosa accade se si ammalano più ingegneri contemporaneamente?
2. Che cosa accade se, a causa di una crisi finanziaria, il budget del progetto viene ridotto del 20%?
3. Che cosa accade se le prestazioni del software open-source non sono adeguate e l'unico esperto di tale software ha abbandonato il team di sviluppo?

4. Che cosa accade se l’azienda che fornisce e mantiene i componenti software chiude l’attività?
5. Che cosa accade se il cliente non consegna i requisiti modificati nei tempi previsti?

In base alle risposte a queste domande, è possibile definire le strategie di gestione dei rischi. La Figura 19.5 riporta le possibili strategie di gestione che sono state identificate per i rischi chiave (gravi e non tollerabili) illustrati nella Figura 19.4. Queste strategie possono essere classificate in tre categorie.

1. *Strategie per evitare i rischi* – Seguire queste strategie significa ridurre la probabilità che un rischio si concretizzi; per esempio la strategia per gestire i componenti difettosi descritta nella Figura 19.5.
2. *Strategie di minimizzazione dei rischi* – Seguire queste strategie significa ridurre l’impatto dei rischi; per esempio la strategia per la malattia del personale descritta nella Figura 19.5.
3. *Piani di emergenza* – Seguire queste strategie significa prepararsi al peggio e avere soluzioni pronte a gestire situazioni molto gravi; per esempio la strategia per i problemi finanziari aziendali descritta nella Figura 19.5.

Si può notare una chiara analogia con le strategie adottate nei sistemi critici per garantire affidabilità, protezione e sicurezza, dove occorre evitare, tollerare o risolvere un problema. Ovviamente, è meglio adottare una strategia che eviti i rischi. Se questo non fosse possibile, occorre adottare una strategia che riduca al minimo le probabilità che si abbiano effetti gravi. Infine, dovreste avere pronte le strategie per affrontare un rischio quando si concretizza. Queste strategie riducono l’impatto complessivo di un rischio sul progetto o sul prodotto.

19.1.4 Monitoraggio dei rischi

Il monitoraggio dei rischi è il processo che verifica se le ipotesi fatte sui rischi relativi al prodotto, al processo e all’azienda non sono cambiate. Occorre valutare periodicamente i rischi identificati per stabilire se la loro probabilità di concretizzarsi stia crescendo oppure no e se i loro effetti siano cambiati. Per fare questo, bisogna esaminare altri fattori, quali il numero di richieste di modifica dei requisiti, che forniscono importanti indizi sulla probabilità dei rischi e sui loro effetti. Questi fattori dipendono ovviamente dal tipo di rischio. La Figura 19.6 mostra alcuni esempi di fattori che possono essere utili per valutare questi tipi di rischi.

Il monitoraggio dei rischi dovrebbe essere effettuato regolarmente in tutte le fasi di sviluppo di un processo. In ogni revisione del progetto si dovrà analizzare separatamente ciascuno dei rischi chiave; poi, bisognerà stabilire se un rischio ha più o meno probabilità di concretizzarsi e se la sua gravità e le sue conseguenze siano cambiate.

Tipo di rischio	Potenziali indicatori
Stima	Mancato rispetto della tempistica concordata; mancata risoluzione dei difetti rilevati.
Azienda	Pettegolezzi aziendali; mancanza di interventi da parte dei dirigenti.
Personale	Morale basso delle persone; pessime relazioni tra i membri del team; alto turnover del personale.
Requisiti	Molte richieste di modifica dei requisiti; lamentele dei clienti.
Tecnologia	Consegna in ritardo di hardware o software di supporto; molti problemi tecnologici.
Strumenti	Riluttanza dei membri del team a usare gli strumenti software; lamentele per gli strumenti software; richieste di computer più veloci e con memoria potenziata.

Figura 19.6 Fattori di rischio.

19.2 Gestione del personale

Le persone che lavorano in una società di software sono l'investimento più importante. È costoso assumere e trattenere le persone più valide. I manager hanno la responsabilità di garantire che gli ingegneri che lavorano su un progetto abbiano un alto livello di produttività. Nelle società di successo, questa produttività viene raggiunta quando le persone sono apprezzate dai loro responsabili e hanno compiti che riflettono le loro capacità ed esperienze.

È importante che i responsabili dei progetti software capiscano i problemi tecnici che influenzano il lavoro di sviluppo del software. Purtroppo, un buon ingegnere informatico non sempre è un buon gestore del personale. Gli ingegneri informatici spesso hanno grandi competenze tecniche, ma mancano delle capacità che consentono loro di motivare e guidare un team di sviluppo di un progetto software. Un project manager dovrebbe conoscere i potenziali problemi connessi alla gestione del personale e dovrebbe cercare di migliorare le sue capacità di gestione del personale.

Ci sono quattro fattori critici che influiscono sulle relazioni tra un manager e il suo personale.

1. *Coerenza* – Tutte le persone che fanno parte di un team di sviluppo dovrebbero essere trattate nello stesso modo. Nessuno si aspetta di ricevere la stessa ricompensa, ma nessuno deve avere la sensazione che il suo lavoro sia sottovalutato.
2. *Rispetto* – Persone differenti hanno capacità differenti, e i manager dovrebbero rispettare queste differenze. A ogni membro di un team dovrebbe essere data l'opportunità di dare il proprio contributo. In alcuni casi, ovvia-

mente, può accadere che alcune persone non si inseriscono bene nella squadra e non possono continuare il loro lavoro; è importante, tuttavia, non trarre conclusioni affrettate su queste persone, soprattutto se si è in una fase iniziale dello sviluppo del progetto.

3. *Considerazione* – Le persone operano in modo più efficiente quando sentono che gli altri li ascoltano e prendono in considerazione le loro proposte. È importante creare un ambiente di lavoro dove tutte le opinioni, anche quelle del personale meno esperto, vengono prese in considerazione.
4. *Obiettività* – I manager dovrebbero sempre essere obiettivi su cosa sta andando bene e cosa sta andando male nel team. Dovrebbero essere obiettivi anche riguardo al loro livello di conoscenze tecniche e affidare il lavoro a persone più competenti, se necessario. Se tentano di nascondere i problemi o le loro incompetenze, alla fine potrebbero essere scoperti e perdere il rispetto dei membri del team.

La gestione pratica del personale può essere acquisita tramite l’esperienza, quindi il ruolo di questo paragrafo e del successivo sul lavoro di squadra è soltanto quello di rendere i project manager più consapevoli dei principali problemi che dovranno affrontare.

19.2.1 Motivare le persone

Uno dei compiti del project manager è motivare le persone che lavorano per lui, in modo che esse siano stimolate a dare il massimo delle loro capacità. In pratica, motivare significa organizzare il lavoro e il suo ambiente in modo da incoraggiare le persone a lavorare nel modo più efficace possibile. Se le persone non sono motivate, saranno meno interessate al lavoro che svolgono, lavoreranno lentamente, commetteranno errori con maggiore probabilità e non potranno fornire il loro contributo al raggiungimento dei più ampi obiettivi del team o dell’azienda.

Per dare questo incoraggiamento, occorre conoscere meglio che cosa s’intende per motivazione delle persone. Maslow (Maslow 1954) sostiene che le persone sono motivate quando vedono soddisfatte le loro necessità, che possono essere classificate in una serie di livelli, come illustra la Figura 19.7. Alla base di queste necessità ci sono i bisogni fondamentali, quali il cibo, il sonno e così via; poi c’è il bisogno di sentirsi sicuri nel proprio ambiente. Seguono le necessità sociali, ovvero il bisogno di sentirsi parte di un raggruppamento sociale. Al livello successivo c’è la necessità di stima, ovvero il bisogno di sentirsi rispettati dagli altri; in cima alla piramide c’è la necessità di autorealizzazione, che riguarda lo sviluppo della propria personalità. Le persone hanno bisogno di soddisfare le esigenze di livello inferiore, come la fame, prima di quelle più astratte di livello superiore.

Le persone che lavorano nelle società di sviluppo del software generalmente non sono affamate, assetate o fisicamente minacciate dal loro ambiente. È chiaro che, da un punto di vista gestionale, esse cerchino di soddisfare le esigenze sociali, la stima degli altri e la realizzazione personale nel lavoro.



Figura 19.7 Gerarchia delle necessità delle persone.

1. Per soddisfare le necessità sociali, occorre dare a una persona il tempo di incontrare i propri colleghi, lo spazio e le occasioni per conoscerli. Alcune società di software, come Google, mettono a disposizione degli spazi nei loro uffici dove le persone possono incontrarsi. Questo è relativamente semplice quando tutti i membri di un team di sviluppo lavorano nello stesso luogo; tuttavia, attualmente sempre più spesso, i membri di un team non lavorano più nello stesso edificio e neppure nella stessa città o stato; a volte lavorano per società diverse o a casa per la maggior parte del tempo.

Si possono utilizzare i social network e le teleconferenze per mettere in comunicazione le persone; tuttavia, la mia esperienza è che questi sistemi non sono molto efficaci se le persone non si conoscono già. Si potrebbero organizzare degli incontri faccia a faccia nella fase iniziale di un progetto, in modo che ogni persona possa interagire direttamente con gli altri membri del team. Grazie a questa interazione diretta, le persone diventano parte di un gruppo sociale e fanno propri gli obiettivi e le priorità del gruppo.

2. Per soddisfare le necessità di stima, i project manager devono far notare alle persone che il loro contributo è prezioso per la società. Il riconoscimento pubblico degli obiettivi raggiunti è un modo semplice ed efficace per fare questo. Ovviamente, occorre che le persone si sentano soddisfatte anche da un punto di vista economico, in base alle proprie capacità ed esperienze.
3. Per soddisfare le esigenze di autorealizzazione, occorre responsabilizzare le persone nello svolgimento del loro lavoro, assegnando compiti difficili, ma non impossibili, e offrendo loro l'opportunità di partecipare a corsi di addestramento e formazione per sviluppare le loro capacità. L'addestramento è un fattore importante nel processo di motivazione, perché alle persone piace acquisire nuove conoscenze e perfezionare le proprie capacità.

Il modello delle motivazioni di Maslow è utile fino a un certo punto; credo infatti che ci sia un problema, in quanto il modello si basa su un punto di vista esclusivamente individuale sulla motivazione delle persone. Esso non tiene conto in

Caso di studio: motivazione

Alice è un project manager che lavora in una società che produce sistemi di allarmi. La società intende entrare nel mercato in espansione della tecnologia assistenziale per aiutare le persone anziane e disabili a vivere in modo autonomo. Alice ha ricevuto l'incarico di guidare un team di sei persone per sviluppare nuovi prodotti basati sui sistemi di allarme della società.

Il progetto di sviluppo dei nuovi sistemi di tecnologia assistenziale parte bene; tra i membri del team si instaurano buone relazioni e nascono molte idee innovative. Il team decide di sviluppare un sistema di allarme che un utente può avviare e controllare tramite un telefono cellulare o un tablet. Dopo alcuni mesi dall'avvio del progetto, Alice nota che Dorothy, l'esperta di progettazione hardware, inizia ad arrivare tardi al lavoro; la qualità del suo lavoro peggiora e i suoi rapporti con gli altri membri del team diventano sempre meno frequenti.

Alice parla del problema in modo informale con gli altri membri del team per tentare di capire se la situazione personale di Dorothy sia cambiata e se ciò possa influire sul suo lavoro. Poiché i colleghi di Dorothy non sanno nulla, Alice decide di parlare direttamente con Dorothy per cercare di capire il problema.

Dopo un rifiuto iniziale di ammettere qualsiasi tipo di problema, Dorothy riconosce di aver perso interesse nel lavoro. Sperava di riuscire a sviluppare e sfruttare le sue capacità di interfacciamento hardware, ma a causa delle direttive scelte per il prodotto, ha avuto poche opportunità di sfruttare le sue conoscenze. In pratica sta collaborando con gli altri membri del team come programmatrice C nello sviluppo del software dei sistemi di allarme.

Sebbene ammetta che il lavoro sia interessante, tuttavia è preoccupata per il fatto che non sta sviluppando le sue capacità di interfacciamento hardware. È preoccupata perché crede che, dopo questo progetto, sarà difficile trovare un lavoro che richieda conoscenze di interfacciamento hardware. Poiché non vuole sconvolgere il lavoro del team rivelando che sta pensando a un altro progetto, ha deciso che sia meglio ridurre al minimo le conversazioni con loro.

Figura 19.8 Motivazione individuale.

modo adeguato del fatto che le persone sentono di far parte di un'organizzazione, di un gruppo professionale o di una o più culture. Sentirsi membro di un gruppo coeso è altamente motivante per una persona. Le persone che hanno un lavoro appagante spesso si recano al lavoro con piacere, in quanto sono motivate dai loro colleghi e dal lavoro che svolgono. Un manager, quindi, dovrebbe cercare di motivare le persone come gruppo, non solo individualmente. Tratterò questo e altri temi sul lavoro di squadra nel Paragrafo 19.3.

Nella Figura 19.8 è illustrato un problema di motivazione che i manager spesso devono affrontare. In questo esempio, un membro esperto del gruppo perde interesse nel lavoro che svolge e nel gruppo nel suo insieme. La qualità del suo lavoro si riduce fino a diventare inaccettabile. Questa situazione deve essere gestita velocemente; se non si risolve il problema, gli altri membri del gruppo inizieranno a sentirsi insoddisfatti e avranno la sensazione di condividere in maniera scorretta una parte del lavoro.

In questo esempio Alice sta tentando di scoprire se la situazione personale di Dorothy possa essere la causa del problema. Le difficoltà personali influiscono

spesso sulle motivazioni, in quanto le persone non riescono a concentrarsi sul lavoro. Un buon manager dovrebbe dare alle persone tempo e supporto per risolvere questi problemi, anche se deve essere chiaro che esse continuano ad avere delle responsabilità verso il datore di lavoro.

Il problema della motivazione di Dorothy si presenta tipicamente quando un progetto inizia a svilupparsi in una direzione imprevista. Una persone che si aspetta di svolgere un determinato compito finisce col fare qualcosa di completamente diverso. In tali circostanze, potrebbe essere opportuno che la persona interessata abbandoni la squadra e cerchi altrove nuove opportunità. Nel caso in esame, tuttavia, Alice vuole provare a convincere Dorothy che ampliare le sue esperienze è un passo positivo nella sua carriera. Offre a Dorothy una maggiore autonomia di progettazione e organizza dei corsi di ingegneria del software che consentiranno a Dorothy di avere più opportunità dopo che avrà completato il progetto corrente.

Anche il tipo di personalità psicologica influisce sulla motivazione. Bass e Duntzman (Bass e Duntzman 1963) hanno classificato i professionisti in tre categorie.

1. *Persone orientate ai compiti* – Sono persone motivate dal lavoro che svolgono. Nell'ingegneria del software, sono tecnici motivati dalla sfida intellettuale intrinseca nell'attività di sviluppo del software.
2. *Persone orientate a sé stesse* – Sono persone motivate principalmente dal successo e dal riconoscimento personale; sono interessate allo sviluppo del software come mezzo per raggiungere i propri obiettivi. Di solito, hanno obiettivi a lungo termine, come l'avanzamento di carriera, e vogliono avere successo nel lavoro per poter realizzare tali obiettivi.
3. *Persone orientate alle interazioni* – Sono persone motivate dalla presenza e dalle azioni dei colleghi di lavoro. Quanta più attenzione viene rivolta alla progettazione di interfacce utente, tanto più aumenta il coinvolgimento di questi professionisti nell'ingegneria del software.

La ricerca ha dimostrato che le persone orientate alle interazioni di solito preferiscono lavorare in gruppo, mentre quelle orientate ai compiti o a sé stesse di solito preferiscono operare individualmente. In genere, le donne sono più orientate alle interazioni rispetto agli uomini; spesso sono abili comunicatrici. Nel caso di studio descritto nella Figura 19.10 descriverò il mix di questi differenti tipi di personalità.

Le motivazioni di ciascun individuo sono formate da elementi delle tre categorie, ma un solo tipo di motivazione di solito è dominante in un qualsiasi momento. Ovviamente, gli individui possono cambiare. Per esempio, i tecnici che ritengono di non essere adeguatamente remunerati possono diventare personalità orientate a sé stesse e anteporre gli interessi personali a quelli lavorativi. Se un gruppo lavora particolarmente bene, le personalità orientate a sé stesse possono diventare personalità orientate alle interazioni.

Modelli di maturità delle capacità delle persone

Il modello di maturità delle capacità delle persone (People Capability Maturity Model o P-CMM) è uno strumento che permette di valutare come le società gestiscono lo sviluppo del loro personale. Mette in evidenza le pratiche ideali nella gestione del personale e fornisce alle aziende la base per migliorare i processi di gestione del loro personale. Si adatta bene alle grandi aziende, non alle aziende piccole e informali.

<http://software-engineering-book.com/web/people-cmm/>

19.3 Lavoro di squadra

Il software professionale viene in gran parte prodotto da team di sviluppo, che possono essere composti da due sole persone o perfino da centinaia di persone. Poiché è impossibile che coloro che fanno parte di un grande gruppo possano lavorare insieme su un singolo problema, di solito i grandi gruppi vengono suddivisi in gruppi più piccoli, ciascuno dei quali si occupa di una parte dell'intero sistema. La dimensione ideale di un gruppo di ingegneria del software varia da 4 a 6 membri, e non dovrebbe mai superare le 12 unità. Quando i gruppi sono piccoli, i problemi di comunicazione sono ridotti. Ogni membro del gruppo conosce il lavoro degli altri, e l'intero gruppo può riunirsi attorno a un tavolo per discutere il progetto e il software che si sta sviluppando.

Formare un gruppo di persone con il giusto equilibrio di competenze tecniche, esperienza e personalità è un compito gestionale critico. Ovviamente, un gruppo di sviluppo non ha successo semplicemente perché i suoi membri hanno il giusto equilibrio di capacità. Un buon gruppo ha spirito di squadra e si considera un'unità forte e compatta. Le persone che ne fanno parte sono motivate dal successo del gruppo, oltre che dal raggiungimento degli obiettivi personali.

Se un gruppo è coeso, i suoi membri ritengono che il gruppo sia più importante degli singoli individui che lo compongono. I membri di un gruppo coeso e ben diretto sono fedeli al gruppo. Ogni membro si identifica con gli obiettivi del gruppo e con gli altri membri. Ciascuno tenta di proteggere il gruppo, come un'unica entità, dalle interferenze esterne. Questo rende il gruppo forte e capace di affrontare problemi e situazioni impreviste.

Un gruppo coeso offre diversi vantaggi.

1. *Il gruppo può definire i suoi standard di qualità* – Poiché questi standard vengono definiti di comune accordo, è più probabile che essi siano rispettati, diversamente da quanto accade con gli standard imposti dall'esterno al gruppo.
2. *I membri imparano dagli altri membri e si supportano a vicenda* – Ogni membro del gruppo impara lavorando insieme agli altri membri. Le inibizioni causate dall'ignoranza sono ridotte al minimo, perché si favorisce l'apprendimento reciproco.

3. *Le conoscenze sono condivise* – La continuità del lavoro può essere mantenuta anche quando un membro del gruppo abbandona il gruppo. Altri membri possono subentrare nei compiti critici e assicurare che il progetto non sia indebitamente interrotto.
4. *Sono favoriti i miglioramenti continui* – I membri del gruppo collaborano per consegnare un prodotto di alta qualità e per risolvere i problemi, indipendentemente dagli individui che avevano originariamente creato il progetto o il programma.

Un buon project manager dovrebbe favorire sempre la coesione fra i membri del gruppo. È importante cercare di sviluppare un senso di identità del gruppo, assegnando per esempio un nome al gruppo e definendo un territorio per il gruppo. Alcuni manager creano appositamente delle attività che rafforzano il senso di appartenenza al gruppo, come giochi o competizioni sportive, anche se queste attività non sempre sono popolari tra i membri del gruppo. Alcuni eventi sociali per i membri del gruppo e le loro famiglie sono un buon modo per mantenere la coesione del gruppo.

Uno dei modi più efficaci per promuovere la coesione di un gruppo consiste nel tenere sempre in grande considerazione i suoi membri. Un project manager dovrebbe trattare i membri del gruppo come persone responsabili e degne di fiducia, lasciando loro un libero accesso alle informazioni. A volte i manager sentono di non poter rivelare determinate informazioni a tutti i membri del gruppo; questo inevitabilmente crea un clima di sfiducia. Per fare in modo che le persone si sentano apprezzate all'interno di un gruppo, basta tenerle sempre informate su ciò che sta accadendo.

Un esempio di quanto appena detto è riportato nella Figura 19.9. Alice organizza periodiche riunioni informali per spiegare ai membri del gruppo che cosa sta succedendo; fa il punto della situazione coinvolgendo le persone che stanno sviluppando il prodotto e chiedendo loro di presentare nuove idee sulla base delle proprie esperienze personali. Gli incontri “away day” sono un altro buon metodo per promuovere la coesione: le persone si rilassano insieme, mentre si aiutano a vicenda a imparare nuove tecnologie.

Che un gruppo sia efficiente oppure no dipende, in qualche misura, dalla natura del progetto e dalla società che sta svolgendo il lavoro. Se una società attraversa una fase di transizione, con costanti riorganizzazioni e instabilità finanziaria, è difficile per i membri del gruppo concentrarsi sullo sviluppo del software. Analogamente, se un progetto continua a cambiare con il rischio di essere annullato, le persone perdono interesse su di esso.

Dato un ambiente di lavoro stabile, tre sono i fattori che hanno maggiore influenza sulla squadra di sviluppo:

1. *le persone del gruppo*; occorre un buon mix di persone in un gruppo di progettazione, in quanto lo sviluppo del software richiede diverse attività, come negoziare con i clienti, programmare, testare il software e fornire la documentazione.

Caso di studio: lo spirito di squadra

Alice, project manager esperta, conosce l'importanza di creare un gruppo coeso. Poiché la sua società sta sviluppando un prodotto nuovo, ha deciso di coinvolgere tutti i membri del team nella definizione della specifica e nella progettazione del prodotto, invogliandoli a discutere le nuove tecnologie con le persone più anziane delle loro famiglie. Li ha invitati a far partecipare i loro familiari a una riunione per incontrare gli altri membri del team di sviluppo.

Alice organizza anche dei pranzi mensilmente per i componenti della squadra. Questi pranzi sono un'opportunità per tutti i membri di incontrarsi informalmente, conoscersi e parlare dei problemi di cui si occupano. Durante i pranzi, Alice riferisce ai membri le novità che riguardano la società, le politiche e le strategie aziendali e così via. Ogni membro del team riassume poi brevemente quello che sta facendo. Poi si discute un argomento generale, come le nuove idee sul prodotto emerse parlando con i familiari anziani.

Saltuariamente, Alice organizza un incontro “away day”, durante il quale il gruppo dedica due giorni all'aggiornamento tecnologico. Ogni membro prepara un aggiornamento su una tecnologia rilevante e lo presenta al gruppo. Trattandosi di un incontro fuori sede, i membri hanno anche molto tempo a disposizione per discutere e conoscersi meglio.

Figura 19.9 Coesione di un gruppo.

2. *l'organizzazione del gruppo*; un gruppo dovrebbe essere organizzato in modo che i membri possano esprimersi al massimo delle loro capacità e svolgere i loro compiti nei tempi previsti.
3. *comunicazioni tecniche e gestionali*; è essenziale una buona comunicazione tra i membri del gruppo, e tra la squadra di ingegneria del software e altri stakeholder del progetto.

Come in tutti i problemi di gestione, avere una buona squadra non significa garantire il successo di un progetto. Molti altri fattori potrebbero influire negativamente sullo sviluppo di un progetto, per esempio i cambiamenti aziendali o dell'ambiente operativo. Tuttavia, se non si presta particolare attenzione alla composizione e all'organizzazione del gruppo, e alla comunicazione tra i suoi membri, aumentano le probabilità che un progetto vada incontro a serie difficoltà.

19.3.1 Selezione dei membri del gruppo

Il compito di un manager o del leader di un team è creare un gruppo coeso e organizzare i suoi membri in modo che possano collaborare in modo efficiente. Questo compito richiede che siano selezionate persone che abbiano un giusto compromesso di competenze tecniche e personalità. A volte le persone vengono assunte all'esterno dell'azienda; più frequentemente, i gruppi di ingegneria del software vengono formati utilizzando i dipendenti dell'azienda che hanno esperienza su altri progetti. I manager raramente hanno mano libera nella selezione dei membri del gruppo; spesso devono utilizzare le persone che sono disponibili

all'interno dell'azienda, anche se non hanno le caratteristiche ideali per svolgere un determinato compito.

Molti ingegneri che sviluppano software trovano motivazioni principalmente nel lavoro che svolgono. I gruppi di sviluppo del software, quindi, sono spesso composti da persone che hanno le proprie idee su come dovrebbero essere risolti i problemi tecnici. Queste persone vogliono svolgere il lavoro nel modo migliore possibile, quindi potrebbero deliberatamente riprogettare un sistema nel modo in cui loro credono che possa essere migliorato e aggiungono nuove funzionalità che non fanno parte dei requisiti del sistema. I metodi agili incoraggiano gli ingegneri a prendere iniziative per migliorare il software. Questo, però, a volte significa che viene impiegato del tempo nel fare cose che non sono strettamente necessarie e che diversi ingegneri entrano in competizione per scrivere l'uno il codice dell'altro.

Le conoscenze e le capacità tecniche non dovrebbero essere l'unico fattore per selezionare i membri di un gruppo. Il problema degli “ingegneri in competizione” può essere ridotto se le persone del gruppo hanno motivazioni complementari. Le persone che sono motivate dal lavoro svolto di solito sono quelle tecnicamente più preparate. Le persone che sono orientate a sé stesse probabilmente sono quelle che spingono di più per portare a termine il lavoro. Le persone che sono orientate alle interazioni favoriscono il dialogo tra i membri del gruppo. Io penso che sia particolarmente importante avere persone orientate alle interazioni all'interno di un gruppo. Queste persone prediligono il dialogo, sono in grado di scoprire tensioni e controversie sul nascere, prima che questi problemi possano avere un grave impatto sull'intero gruppo.

Nel caso di studio riportato nella Figura 19.10 ho spiegato come Alice, in qualità di project manager, ha tentato di creare un gruppo con personalità complementari. Questo particolare gruppo ha un buon mix di persone orientate alle interazioni e persone orientate ai compiti; ma, come ho già descritto nella Figura 19.8, Dorothy, che ha una personalità orientata a sé stessa, ha causato dei problemi, perché non stava svolgendo il compito che si aspettava. Anche il ruolo part-time di Fred, esperto di domini, potrebbe essere un problema per il gruppo; egli è interessato principalmente alle sfide tecniche, quindi potrebbe interagire nel modo sbagliato con gli altri membri del gruppo. Il fatto che non faccia sempre parte del team significa che potrebbe non essere in completa sintonia con gli obiettivi del team.

A volte è impossibile scegliere un gruppo con personalità complementari. In questi casi, il project manager deve controllare il gruppo in modo che gli obiettivi dei singoli membri non abbiano la precedenza su quelli aziendali o del gruppo. Questo controllo è più facile da svolgere se tutti i membri del gruppo partecipano a ciascuna fase del progetto. L'iniziativa individuale è più probabile che si sviluppi quando ai membri del gruppo sono date istruzioni senza indicare la parte che il loro compito gioca nell'intero progetto.

Caso di studio: composizione del gruppo

Nel creare un gruppo per lo sviluppo di nuovi sistemi di tecnologia assistenziale, Alice conosce l'importanza di selezionare i membri che abbiano personalità complementari: durante i colloqui con le persone, ha cercato di capire quali di esse fossero orientate ai compiti, quali a sé stesse e quali alle interazioni. Alice sentiva di essere principalmente orientata a sé stessa, in quanto considerava il progetto come un'opportunità per farsi notare dai suoi dirigenti ed eventualmente ottenere una promozione. Ha cercato quindi una o due persone orientate alle interazioni e altre orientate ai compiti per completare il team. La composizione finale del gruppo è la seguente:

Alice – orientata a sé stessa
Brian – orientato ai compiti
Chun – orientato alle interazioni
Dorothy – orientata a sé stessa
Ed – orientato alle interazioni
Fiona – orientata ai compiti
Fred – orientato ai compiti
Hassan – orientato alle interazioni

Figura 19.10 Composizione del gruppo.

Per esempio, supponiamo che un ingegnere abbia ricevuto l'incarico di sviluppare un sistema software; si accorge che è possibile apportare dei miglioramenti al progetto di questo sistema. Se implementasse questi miglioramenti senza conoscere la logica che sta alla base del progetto originale, qualsiasi modifica, anche se fatta con buone intenzioni, potrebbe avere implicazioni negative sulle altre parti del sistema. Se tutti i membri del gruppo sono coinvolti nel progetto fin dalle fasi iniziali, è più facile che essi capiscano perché sono state fatte certe scelte di progettazione; successivamente, potranno riconoscere tali scelte, anziché contraddirle.

19.3.2 Organizzazione del gruppo

Il modo in cui un gruppo è organizzato influenza sulle decisioni del gruppo, sullo scambio delle informazioni e sulle interazioni tra il team di sviluppo e gli stakeholder esterni. Fra le più importanti domande che i project manager dovrebbero porsi sull'organizzazione di un gruppo di sviluppo del software ci sono le seguenti.

1. Il project manager dovrebbe essere il leader tecnico del gruppo? Il leader tecnico o architetto del sistema è responsabile delle decisioni tecniche critiche prese durante lo sviluppo del software. A volte, il project manager ha le capacità e l'esperienza per svolgere questo ruolo. Tuttavia, per i grandi progetti, è meglio separare il ruolo tecnico da quello manageriale. Il project manager dovrebbe scegliere un ingegnere esperto come project architect, che assumerà la leadership tecnica della squadra di sviluppo.

Selezione del personale appropriato

I project manager spesso hanno la responsabilità di selezionare il personale per il team di ingegneria del software. È molto importante scegliere le persone più idonee per ogni ruolo del team di sviluppo, in quanto una scelta inappropriata potrebbe mettere a serio rischio l'intero progetto.

I fattori chiave che influiscono sulla selezione del personale sono il titolo di studio, l'addestramento, le esperienze tecnologiche e applicative, la capacità di comunicazione, l'adattabilità e l'abilità di risolvere i problemi.

<http://software-engineering-book.com/web/people-selection/>

2. Chi dovrà essere coinvolto nelle decisioni tecniche critiche e come dovranno essere prese queste decisioni? Sarà il responsabile tecnico o il project manager a prendere queste decisioni? Oppure dovrà essere raggiunto un consenso ampio tra i membri del team?
3. Come saranno gestite le interazioni con gli stakeholder esterni e l'alta dirigenza aziendale? In molti casi, il project manager avrà la responsabilità di queste interazioni, assistito eventualmente dal responsabile tecnico. Un modello organizzativo alternativo consiste nel creare un apposito ruolo per le relazioni esterne e nominare qualcuno, con le capacità appropriate, a svolgere tale ruolo.
4. In che modo i gruppi possono integrare quelle persone che non si trovano fisicamente nello stesso luogo? Oggi è comune che i gruppi includano membri appartenenti ad aziende diverse e che alcuni lavorino a casa o in uffici condivisi. Questi casi devono essere presi in considerazione nei processi decisionali di un gruppo di sviluppo.
5. In che modo le informazioni possono essere condivise tra i membri di un gruppo? L'organizzazione di un gruppo influenza sulla condivisione delle informazioni, in quanto certi metodi organizzativi sono migliori di altri ai fini di tale condivisione. Tuttavia, bisognerebbe evitare di condividere troppe informazioni, perché un sovraccarico di notizie e dati potrebbe distrarre le persone dal loro lavoro.

Alcuni gruppi di programmazione di solito si organizzano in un modo informale. Il leader del gruppo è coinvolto nello sviluppo del software con altri membri del gruppo. In un gruppo informale, tutti i suoi membri discutono il lavoro da svolgere, e i compiti vengono assegnati in base alle capacità e all'esperienza. I membri più esperti potrebbero assumersi la responsabilità della progettazione architettonica, ma la responsabilità dei dettagli dell'implementazione e della progettazione è del membro del gruppo al quale è stato assegnato un particolare compito.

I team di sviluppo agile sono sempre gruppi informali. I sostenitori dei metodi agili ritengono che la struttura formale inibisca lo scambio di informazioni. Molte decisioni che sembrano decisioni gestionali (come quelle sulla tempistica) potrebbero essere delegate ai membri del gruppo. Tuttavia, c'è sempre bisogno di un project manager che abbia la responsabilità delle decisioni strategiche e delle comunicazioni con entità esterne al gruppo.

I gruppi informali possono essere molto efficaci, specialmente quando molti membri del gruppo sono persone esperte e competenti. Un gruppo siffatto è in grado di prendere decisioni all'unanimità, cosa che migliora lo spirito coesione e le prestazioni. Se, invece, un gruppo è composto perlopiù da membri poco esperti o incompetenti, l'informalità può essere un problema, in quanto, non esistendo un'autorità in grado di dirigere il lavoro, verrebbe a mancare la coordinazione tra i membri del gruppo, con un eventuale fallimento del progetto.

In un gruppo gerarchico, il leader è in cima alla scala gerarchica; ha un'autorità più formale rispetto ai membri del gruppo e, quindi, può dirigere il loro lavoro. C'è una chiara struttura organizzativa; le decisioni prese in alto nella struttura gerarchica vengono implementate dalle persone che si trovano più in basso nella gerarchia. Le comunicazioni sono soprattutto istruzioni provenienti dal personale più anziano; le persone dei livelli gerarchici più bassi di solito scambiano poche informazioni con i manager dei livelli più alti.

I gruppi gerarchici possono operare bene quando un compito può essere facilmente suddiviso in componenti software che possono essere sviluppati dalle varie parti della struttura gerarchica. Questo tipo di gruppo consente di accelerare il processo decisionale, ecco perché è adottato nelle organizzazioni militari. Tuttavia, questo modello raramente può essere adattato ai progetti complessi di ingegneria del software. Nello sviluppo del software è essenziale avere comunicazioni efficienti a tutti i livelli.

1. Una modifica del software spesso implica la modifica di numerose parti del sistema; ciò può essere fatto discutendo e negoziando a tutti i livelli gerarchici.
2. Le tecnologie del software cambiano così velocemente che il personale più giovane potrebbe conoscerle meglio del personale più anziano. Le comunicazioni top-down potrebbero impedire al project manager di venire a conoscenza delle opportunità offerte dall'utilizzo di queste nuove tecnologie.

Un problema importante che devono affrontare i project manager è la differenza tra le capacità tecniche dei membri del gruppo. I migliori programmatore potrebbero essere fino a 25 volte più produttivi dei peggiori programmatore. Ha senso utilizzare questi “super-programmatore” nel modo più efficiente e fornire loro il maggior supporto possibile.

D'altra parte, favorire i super-programmatore potrebbe essere causa di demotivazione per gli altri membri del gruppo, che potrebbero risentirsi perché poco responsabilizzati; ciò sarebbe anche fonte di preoccupazione perché potrebbe

incidere negativamente sulla loro carriera. Inoltre, se un super-programmatore abbandona la società, l'impatto sullo sviluppo del progetto potrebbe essere pesante. Quindi, adottare un modello di gruppo che si basa sulle singole individualità potrebbe comportare seri rischi.

19.3.3 Comunicazione tra i membri del gruppo

È assolutamente essenziale che i membri di un gruppo comunichino in modo efficiente ed efficace tra loro e con altri stakeholder del progetto. I membri del gruppo devono scambiarsi le informazioni sullo stato di avanzamento del lavoro, sulle decisioni relative al progetto che stanno sviluppando e sulle modifiche apportate alle precedenti decisioni. Devono risolvere i problemi che nascono con altri stakeholder e informare questi stakeholder sulle modifiche da apportare al sistema e ai piani di consegna. La buona comunicazione serve anche a rafforzare la coesione del gruppo; ciascun membro può capire meglio le motivazioni, i punti di forza e di debolezza degli altri membri.

Diversi sono i fattori chiave che influiscono sull'efficacia e sull'efficienza della comunicazione.

1. *Dimensione del gruppo* – Quando un gruppo si espande, diventano più difficili le comunicazioni tra i suoi membri. Il numero dei collegamenti di comunicazione unidirezionale è $n^*(n-1)$, dove n è la dimensione del gruppo; quindi con un gruppo di sette o otto membri, ci sono 56 possibili percorsi di comunicazione. Questo significa che alcune persone avranno poche probabilità di comunicare. Le differenze di stato tra i membri del gruppo spesso sono causa di comunicazioni unilaterali. I manager e gli ingegneri esperti tendono a prevalere sulle comunicazioni a danno del personale meno esperto, che di solito è più riluttante a iniziare una conversazione o a fare osservazioni critiche.
2. *Struttura del gruppo* – Le persone che fanno parte di gruppi strutturati in modo informale comunicano più efficacemente di quelle che fanno parte di gruppi aventi una struttura gerarchica formale. Nei gruppi gerarchici, le comunicazioni tendono a fluire su e giù nella struttura gerarchica; le persone che si trovano allo stesso livello gerarchico spesso non comunicano tra loro. Questo è un problema tipico dei grandi progetti con gruppi di sviluppo differenti. Se le persone che lavorano su diversi sottosistemi comunicano soltanto tramite i loro manager, è molto più probabile che il progetto sia soggetto a ritardi e a cattive interpretazioni.
3. *Composizione del gruppo* – Le persone con lo stesso tipo di personalità (descritte nel Paragrafo 19.2.1) potrebbero entrare in conflitto e ostacolare le comunicazioni. Di solito, le comunicazioni sono migliori nei gruppi composti da persone di sesso misto (Marshall e Heslin 1975). Le donne tendono a essere più orientate alle interazioni rispetto agli uomini e spesso svolgono un ruolo di controllo e moderazione delle interazioni tra i membri.

L'ambiente fisico di lavoro

Le comunicazioni e la produttività dei membri di un gruppo di sviluppo sono influenzate dall'ambiente del posto di lavoro. Le aree di lavoro private favoriscono la concentrazione sui dettagli tecnici, in quanto le persone subiscono un minor numero di distrazioni e interruzioni. Le aree di lavoro condivise, invece, agevolano le comunicazioni. Un ambiente di lavoro ben progettato dovrebbe essere dotato di entrambe queste aree di lavoro.

<http://software-engineering-book.com/web/workspace/>

4. *L'ambiente fisico di lavoro* – L'organizzazione del posto di lavoro influisce in modo considerevole sull'efficienza delle comunicazioni. Sebbene alcune società utilizzino uffici open-space per il loro personale, altre offrono posti che includono aree miste di lavoro pubblico e privato. In questo modo è possibile svolgere attività di gruppo e attività individuali che richiedono un alto livello di concentrazione.
5. *I canali di comunicazione* – Ci sono molte forme di comunicazione: faccia a faccia, messaggi e-mail, documenti formali, telefono e tecnologie come i social network e i wiki. Poiché i team di sviluppo stanno assumendo sempre più la forma di gruppi distribuiti, i cui membri lavorano in posti remoti, occorre sfruttare le tecnologie di interazione, come i sistemi di teleconferenza, per agevolare le comunicazioni tra i membri di questi gruppi.

I project manager di solito lavorano con tempi stretti e, di conseguenza, spesso cercano di utilizzare i canali di comunicazione che non assorbono troppo del loro tempo. Essi possono contare su riunioni e documenti formali per passare le informazioni sul progetto al personale e agli stakeholder, oppure spedire lunghe e-mail al personale. Sebbene questo possa sembrare un metodo efficace di comunicazione dal punto di vista del project manager, in effetti non lo è. Spesso, le persone hanno validi motivi per non partecipare alle riunioni, e quindi non vengono aggiornate. Molti non hanno tempo di leggere lunghi documenti o e-mail che non riguardano direttamente il loro lavoro. Quando vengono prodotte più versioni dello stesso documento, è difficile che un lettore riesca a tenere traccia delle modifiche.

Le comunicazioni sono efficienti quando sono bilaterali e le persone coinvolte possono discutere problemi, scambiarsi informazioni e stabilire una base comune di obiettivi da raggiungere e argomenti da trattare. Tutto questo può essere fatto riunendosi periodicamente, sebbene nelle riunioni spesso prevalgano le personalità più spiccate. Le discussioni informali, quando un manager incontra i membri del gruppo per bere un caffè, a volte sono le più efficienti.

I team di progettazione includono un numero sempre maggiore di membri che lavorano in posti fisicamente distanti; questo rende più difficili gli incontri. Per consentire la comunicazione e lo scambio di informazioni tra questi membri, si possono utilizzare i wiki e i blog. Grazie ai wiki, tutti i membri del gruppo pos-

sono partecipare alla creazione e all'editing dei documenti; i blog supportano le discussioni su temi e problemi proposti dai membri del gruppo, indipendentemente dal luogo in cui si trovano. Tutto questo aiuta a gestire le informazioni e a tenere traccia dei temi della discussione, che spesso diventano confusi quando sono trattati tramite le e-mail. Per risolvere quei problemi che richiedono di essere discussi, è possibile anche utilizzare la teleconferenza e la messaggistica istantanea, che possono essere facilmente configurate.

Punti chiave

- Una buona gestione della progettazione è essenziale affinché i progetti di ingegneria del software siano sviluppati entro i tempi e i costi previsti.
- La gestione del software è diversa dalle altre forme di gestione ingegneristiche. Il software è intangibile. I progetti possono essere nuovi o innovativi, quindi non esiste nessuna persona esperta che possa guidare la loro gestione. I processi software non sono così maturi come i tradizionali processi di ingegneria.
- La gestione dei rischi richiede l'identificazione e la valutazione dei principali rischi di progettazione per stabilire le loro probabilità di concretizzazione e le conseguenze sul progetto nel caso in cui un rischio si concretizzi. Occorre definire dei piani per evitare, gestire o trattare i rischi con più alta probabilità di concretizzazione.
- La gestione del personale richiede la selezione delle persone più adatte a svolgere determinati compiti di progettazione e l'organizzazione del team e del suo ambiente di lavoro, in modo da massimizzare la produttività dei membri del team.
- Le persone sono motivate dalle interazioni con altre persone, dall'apprezzamento dei colleghi e dei manager, e dalle opportunità di crescita professionale.
- I gruppi di sviluppo del software dovrebbero essere coesi e relativamente piccoli. I fattori chiave che influiscono sull'efficienza di un gruppo sono le persone che lo compongono, il modo in cui il gruppo è organizzato e le comunicazioni tra i membri del gruppo.
- Le comunicazioni all'interno di un gruppo sono influenzate da vari fattori, quali lo stato dei membri del gruppo, la dimensione del gruppo, la composizione sessuale del gruppo, le personalità e i canali di comunicazione.

Esercizi

- 19.1 Spiegate perché l'intangibilità dei sistemi software è causa di problemi speciali per la gestione della progettazione del software.
- * 19.2 Spiegate perché i migliori programmatore non sempre sono i migliori manager. Per rispondere potrebbe essere utile la lista delle attività di gestione descritte nel Paragrafo 19.1.
- 19.3 Utilizzando alcuni esempi classici di problemi di progettazione, elencate le difficoltà e gli errori di gestione che si presentano in questi progetti falliti di progettazione (vi suggerisco di leggere *The Mythical Man Month*, indicato nelle Ulteriori letture).

- * 19.4 Oltre ai rischi riportati nella Figura 19.1, identificate almeno altri sei possibili rischi che potrebbero presentarsi nei progetti software.
 - 19.5 Spiegate perché la probabilità dei rischi e le loro conseguenze possono cambiare durante lo sviluppo di un progetto.
 - * 19.6 Gli appalti a prezzo fisso, offerti dai fornitori per completare lo sviluppo di un sistema software, possono essere utilizzati per spostare i rischi di progettazione dal cliente all'appaltatore. Se qualcosa va male è l'appaltatore che deve pagare. Spiegate perché l'uso di tali contratti può aumentare la probabilità di concretizzazione dei rischi di progettazione.
 - 19.7 Spiegate perché, tenendo informati tutti i membri di un gruppo sull'avanzamento e sulle decisioni tecniche di un progetto, è possibile migliorare la coesione del gruppo.
 - * 19.8 Quali problemi potrebbero nascere nei team di programmazione estrema dove molte decisioni gestionali sono delegate ai membri dei team?
 - * 19.9 Scrivete un caso di studio nello stile che è stato utilizzato per dimostrare l'importanza delle comunicazioni in un team di progettazione. Supponete che qualche membro del team lavori in un luogo remoto e che non sia possibile riunire tutti i membri in breve tempo.
 - 19.10 Il vostro manager vi ha chiesto di consegnare il software a una certa data che può essere rispettata soltanto chiedendo al vostro team di fare degli straordinari non pagati. Tutti i membri del team hanno bambini piccoli. Pensate di accettare la richiesta del manager o vorreste convincere i membri del team a dedicare il loro tempo all'azienda, anziché alle famiglie? Quali fattori potrebbero essere determinanti per la vostra decisione?
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

The Mythical Man Month: Essays on Software Engineering (Anniversary Edition). I problemi della gestione del software non sono cambiati dagli anni '60 a oggi, e questo è uno dei migliori libri che tratta questi temi. Presenta un interessante e piacevole racconto della gestione di uno dei primi grandi progetti software, il sistema operativo IBM OS/360. L'edizione dell'anniversario (pubblicata vent'anni dopo quella originale del 1975) comprende altri articoli classici di Brooks (F. P. Brooks, 1995, Addison-Wesley).

Peopleware: Productive Projects and Teams, 2nd ed. È diventato un testo classico sull'importanza di trattare in modo opportuno le persone quando si gestiscono progetti software. È uno dei pochi libri che riconosce come il luogo in cui lavorano le persone influenzli le comunicazioni e la produttività. Fortemente consigliato (T. DeMarco e T. Lister, 1999, Dorset House).

Waltzing with Bears: Managing Risk on Software Projects. Una presentazione molto pratica e facile da leggere della gestione dei rischi (T. DeMarco e T. Lister, 2003, Dorset House).

Effective Project Management: Traditional, Agile, Extreme. 2014 (7th ed.). Un testo sulla gestione della progettazione in generale, anziché sulla gestione della progettazione del software. Si basa sul cosiddetto PMBOK (Project Management Body of Knowledge) e, diversamente da molti altri libri su questo argomento, descrive le tecniche PM per i progetti agili (R. K. Wysocki, 2014).

CAPITOLO

20

Pianificazione della progettazione

L'obiettivo di questo capitolo è presentare la pianificazione della progettazione, la tempistica dei progetti e la stima dei costi. Dopo aver letto questo capitolo:

- conoscerete gli elementi fondamentali del costo del software e i fattori che influiscono sul prezzo di un sistema software che dovrà essere sviluppato per un cliente esterno;
- saprete quali parti dovranno essere incluse in un piano di progettazione che viene creato all'interno di un processo di sviluppo guidato da piani;
- capirete le funzioni della tempistica dei progetti e dei grafici a barre per presentare il piano di sviluppo di un progetto;
- conoscerete la pianificazione agile dei progetti che si basa sul "gioco della pianificazione";
- apprenderete le tecniche di stima dei costi e come il modello COCOMO II possa essere utilizzato per stimare i costi del software.

- 20.1 Prezzo del software
- 20.2 Sviluppo guidato da piani
- 20.3 Tempistica dei progetti
- 20.4 Pianificazione agile
- 20.5 Tecniche di stima
- 20.6 Modelli di costi COCOMO

La pianificazione della progettazione è uno dei più importanti compiti di un project manager. Un manager deve saper suddividere il lavoro da svolgere in più parti per assegnarle ai membri di una squadra di sviluppo; deve saper prevedere i problemi che potrebbero nascere e preparare le soluzioni per risolvere tali problemi. Il piano di un progetto, che viene creato all'inizio della progettazione e aggiornato durante lo sviluppo del progetto, è utilizzato per spiegare come il lavoro sarà fatto e per valutare l'avanzamento del progetto.

La pianificazione della progettazione si svolge in tre fasi nel ciclo di vita di un progetto:

1. *Proposta* – Si scrive un contratto di appalto per sviluppare o fornire un sistema software. In questa fase occorre definire un piano per cercare di capire se si hanno le risorse necessarie per completare il lavoro e per valutare il prezzo da offrire al cliente.
2. *Avviamento* – Si scelgono le persone che dovranno realizzare il progetto; si definiscono le parti che dovranno essere progressivamente sviluppate per completare il progetto; si stabiliscono le modalità di allocazione delle risorse per sviluppare il progetto e così via.
3. *Revisioni* – Si rivede periodicamente il progetto, aggiornando il piano di sviluppo in base alle nuove informazioni sul software e sul suo avanzamento. Durante le revisioni si acquisiscono nuove informazioni sull'implementazione del sistema e sulle capacità del team di sviluppo. Se i requisiti del software cambiano, la suddivisione del lavoro dovrà essere modificata e la tempistica estesa. Queste informazioni consentono di effettuare delle stime più accurate del lavoro ancora da svolgere.

La pianificazione nella fase di proposta è inevitabilmente speculativa, in quanto non si dispone della definizione completa dei requisiti per il software da sviluppare. Occorre predisporre la proposta di realizzazione del progetto in base alla descrizione generale delle funzionalità richieste per il software. Di solito, la proposta include un piano credibile che illustra lo svolgimento del lavoro. Se ci si aggiudica il contratto di appalto, occorrerà rivedere il piano di realizzazione, prendendo in considerazione i cambiamenti che ci sono stati da quando è stata presentata la proposta, le nuove informazioni sul sistema, il processo di sviluppo e la squadra di sviluppo.

Per definire l'offerta da includere nel contratto, occorre valutare il prezzo da proporre al cliente per lo sviluppo del software. Come punto di partenza per calcolare questo prezzo, è necessario elaborare una stima dei costi per completare il lavoro di progettazione. La stima richiede di valutare il lavoro richiesto per completare le singole parti e, da qui, calcolare il costo totale del progetto. Occorre sempre calcolare con obiettività i costi del software, cercando di prevedere in modo accurato il costo di sviluppo del software. Una volta effettuata una stima ragionevole dei costi, è possibile calcolare il prezzo da offrire al cliente. Come

Costi di overhead

Quando si stimano i costi delle attività da svolgere in un progetto software, non bisogna moltiplicare semplicemente la retribuzione oraria delle persone per il tempo impiegato nel progetto. Occorre tenere conto anche dei costi di overhead dell’azienda (amministrazione, uffici ecc.) che devono essere coperti dai ricavi del progetto. I costi devono essere calcolati tenendo conto di questi overhead e aggiungendo i costi di ogni ingegnere che lavora al progetto.

<http://software-engineering-book.com/web/overhead-costs/>

dirò nel prossimo paragrafo, molti fattori influiscono sul prezzo di un progetto software – che non è la semplice somma dei costi e del profitto.

Occorrono tre parametri principali per calcolare i costi di sviluppo di un progetto software:

- costi delle attività svolte dai manager e dagli ingegneri;
- costi dell’hardware e del software, quali la manutenzione dell’hardware e il supporto al software;
- costi per i viaggi e l’addestramento del personale.

Per la maggior parte dei progetti, i costi più grandi sono quelli delle attività. Bisogna stimare le attività complessive (in persone-mese) che si pensa siano richieste per completare il lavoro di un progetto. Ovviamente, spesso si hanno informazioni limitate per fare tale stima; quindi, bisogna fare la migliore stima possibile e poi aggiungere qualche costo di emergenza (per attività straordinarie) nel caso in cui la stima iniziale sia ottimistica.

Per i sistemi commerciali, di solito si usa un hardware di largo consumo, che è relativamente economico. I costi del software, invece, possono essere significativi se si pagano licenze per l’uso di software middleware o di piattaforme software. Quando un progetto viene sviluppato in luoghi distanti, bisogna valutare anche i costi dei viaggi dei tecnici. Sebbene questi costi siano una piccola frazione dei costi delle attività, tuttavia il tempo trascorso per viaggiare spesso è sprecato, con conseguente aumento dei costi delle attività di progettazione. Per ridurre i viaggi e quindi avere più tempo a disposizione delle attività produttive, è possibile utilizzare la teleconferenza o altri sistemi elettronici per le riunioni dei membri di un team di sviluppo.

Una volta che il contratto di sviluppo di un sistema è stato assegnato, deve essere ridefinito lo schema iniziale del progetto in modo da creare un piano di avviamento per lo sviluppo del progetto. In questa fase, servono maggiori informazioni sui requisiti del sistema. L’obiettivo è definire un piano abbastanza dettagliato che possa agevolare le decisioni dei manager sulla selezione del personale e sul budget da destinare al progetto. Questo piano sarà la base per decidere quali risorse allocare all’interno dell’azienda o se ricorrere a risorse esterne.

Il piano dovrebbe definire anche i meccanismi di monitoraggio del progetto. Occorre controllare costantemente il progresso del progetto e confrontare l'avanzamento dei lavori e i costi effettivi con quelli previsti nel piano. Sebbene molte società abbiano delle procedure formali di monitoraggio, di solito un buon manager è in grado di avere una visione chiara della situazione discutendo informalmente con il personale del team di sviluppo. Il monitoraggio informale può prevedere alcuni potenziali problemi, mettendo in evidenza le difficoltà non appena si presentano. Per esempio, una discussione giornaliera con il personale del team potrebbe rivelare che il team ha un problema per un malfunzionamento del software nel sistema di comunicazione. Il project manager può quindi immediatamente assegnare a un esperto di comunicazioni il compito di scoprire e risolvere tale problema.

Il piano del progetto si evolve in continuazione durante il processo di sviluppo, a causa delle modifiche apportate ai requisiti, alle innovazioni tecnologiche e ai problemi di sviluppo. La pianificazione di un progetto di sviluppo serve a garantire che il piano resti un documento utile che consenta al personale di capire che cosa deve essere realizzato e quando deve essere consegnato. Per questo occorre sempre rivedere la tempistica, le stime dei costi e i rischi durante lo sviluppo del software.

Se si adotta un metodo agile, occorre comunque preparare un piano di avviamento del progetto perché, indipendentemente dall'approccio adottato, la società ha sempre bisogno di pianificare le risorse da allocare al progetto. In questo caso, tuttavia, il piano non è dettagliato; basta includere le informazioni essenziali sulla suddivisione del lavoro e sulla tempistica. Durante lo sviluppo, vengono definiti un piano informale e le stime delle attività per ogni release del software, coinvolgendo l'intero team nel processo di pianificazione. Alcuni aspetti della pianificazione agile sono stati già trattati nel Capitolo 3; altri argomenti saranno trattati nel Paragrafo 20.4.

20.1 Prezzo del software

In teoria, il prezzo di un sistema software sviluppato per un cliente è semplicemente il costo di sviluppo più il profitto per la società di software. In pratica, tuttavia, la relazione tra il costo del progetto e il prezzo praticato al cliente non è così semplice.

Per calcolare il prezzo, occorre tenere in conto fattori più ampi, di tipo organizzativo, economico, politico e aziendale (Figura 20.1). Occorre considerare i problemi organizzativi, i rischi associati al progetto e il tipo di contratto che sarà sottoscritto. Questi aspetti potrebbero richiedere una revisione del prezzo verso l'alto o verso il basso.

Fattore	Descrizione
Termini contrattuali	Un cliente potrebbe lasciare alla società di sviluppo la proprietà del codice sorgente in modo che questa possa riutilizzarlo in altri progetti. Il prezzo offerto potrebbe essere così ridotto per riflettere il valore che il codice sorgente ha per la società di sviluppo.
Incertezza nella stima dei costi	Se una società di software non è sicura delle sue stime dei costi, potrebbe aumentare il prezzo del software di un certo valore sopra il suo normale profitto.
Salute finanziaria	Le società di software che attraversano difficoltà finanziarie potrebbero abbassare il loro prezzo per aggiudicarsi un appalto. È meglio avere un profitto ridotto, piuttosto che dichiarare fallimento. Nei periodi di crisi economica, il cash flow è più importante del profitto.
Opportunità di mercato	Una società di software potrebbe fissare un prezzo basso perché vuole entrare in un nuovo segmento del mercato del software. Accettare un basso profitto su un progetto può dare all'organizzazione l'opportunità di ottenere un profitto maggiore in futuro. Inoltre, l'esperienza acquisita potrà servire alla società per sviluppare nuovi prodotti.
Volatilità dei requisiti	Se è probabile che i requisiti cambino, una società potrebbe abbassare il prezzo del software per aggiudicarsi un appalto. Dopo l'assegnazione dell'appalto, la società potrebbe aumentare il prezzo inizialmente concordato per apportare le modifiche dei requisiti.

Figura 20.1 Fattori che influiscono sul prezzo del software.

Per illustrare alcuni dei fattori che influiscono sul prezzo del software, consideriamo il seguente scenario.

Una piccola società di software, PharmaSoft, ha 10 ingegneri. Ha appena ultimato un grosso progetto, ma ha contratti che richiedono solo cinque membri nel team di sviluppo. Sta partecipando a una gara per aggiudicarsi un importante appalto con una grande società farmaceutica che richiede 30 anni-uomo in due anni. Il progetto non partirà prima di un anno ma, se la società se lo aggiudicherà, le sue finanze rifioriranno.

PharmaSoft ha l'opportunità di fare un'offerta per un progetto che richiede sei persone e che deve essere completato in dieci mesi. I costi (inclusi gli overhead di questo progetto) sono stimati in 1,2 milioni di euro. Per essere più concorrenziale, PharmaSoft offre al cliente un prezzo di 0,8 milioni di euro. Questo significa che, anche se la società ha una perdita con questo appalto, tuttavia potrà mantenere il personale specializzato per progetti futuri più redditizi.

Questo è un esempio di approccio al prezzo del software chiamato “pricing to win” (prezzo per vincere una gara di appalto). Questo approccio significa che un’azienda ha una certa *idea* del prezzo che il cliente prevede di accettare e fa un’offerta basata sulle aspettative del cliente. Sebbene il metodo possa apparire immorale e commercialmente scorretto, tuttavia offre alcuni vantaggi sia al cliente sia al fornitore del sistema.

Il costo del progetto è concordato sulla base di una proposta generale. Successivamente, il cliente e il fornitore negoziano i dettagli della specifica del sistema. La specifica resta vincolata dal costo concordato. L’acquirente e il venditore devono accordarsi sulle funzionalità del sistema. In molti progetti il fattore fisso non è rappresentato dai requisiti del progetto, ma dal costo. I requisiti possono essere modificati in modo che i costi del progetto restino entro le previsioni del budget.

Si consideri per esempio una società di software (OilSoft) che sta partecipando a una gara d’appalto per sviluppare un nuovo sistema di erogazione del carburante per una società petrolifera che pianifica le consegne di carburante alle proprie stazioni di servizio. Non c’è un documento dettagliato dei requisiti per questo sistema, quindi OilSoft ritiene che un prezzo di 900.000 euro possa essere competitivo e rientrare nel budget della società petrolifera. Dopo l’assegnazione dell’appalto, OilSoft negozia i requisiti dettagliati del sistema in modo che siano consegnate le funzionalità di base; poi stima i costi aggiuntivi per gli altri requisiti.

Questo approccio offre dei vantaggi sia allo sviluppatore del software sia al cliente. I requisiti vengono negoziati per evitare quelli che sono difficili da implementare e potenzialmente molto costosi. I requisiti flessibili semplificano il riutilizzo del software. La società petrolifera ha assegnato l’appalto a una società di cui si può fidare. Inoltre, è possibile spalmare il costo del progetto su più versioni del sistema; in questo modo, è possibile ridurre i costi di sviluppo del sistema e consentire al cliente di distribuire i costi del progetto su più anni finanziari.

20.2 Sviluppo guidato da piani

Lo sviluppo guidato da piani è un approccio all’ingegneria del software in cui il processo di sviluppo viene pianificato dettagliatamente. Un piano di progettazione viene creato per stabilire il lavoro da fare, chi dovrà farlo e la tempistica di sviluppo. I manager usano questo piano a supporto delle loro decisioni e come strumento per misurare l’avanzamento del lavoro. Lo sviluppo guidato da piani si basa sulle tecniche di gestione dei progetti di ingegneria e può essere immaginato come il modo “tradizionale” di gestire grandi progetti di sviluppo del software. Lo sviluppo agile richiede un processo di pianificazione differente (descritto nel Paragrafo 20.4) in cui le decisioni vengono ritardate.

Il problema dello sviluppo guidato da piani è che le decisioni iniziali devono essere riviste a causa delle modifiche degli ambienti nei quali il software viene

sviluppato e utilizzato. Ritardando le decisioni di pianificazione si evitano revisioni superflue. Tuttavia, chi è a favore dell'approccio guidato da piani sostiene che la pianificazione precoce permette di prendere in considerazione i problemi organizzativi (disponibilità del personale, altri progetti ecc.). Problemi e dipendenze tra le varie attività vengono scoperti prima che il progetto inizi, non dopo che è stato avviato.

Secondo il mio punto di vista, l'approccio migliore alla pianificazione della progettazione consiste in un opportuno mix di sviluppo agile e sviluppo guidato da piani; il mix dipende dal tipo di progetto e dalle capacità delle persone. A un estremo, i grandi sistemi a sicurezza critica richiedono un'approfondita analisi up-front e possono essere certificati prima di essere messi a disposizione degli utenti. Lo sviluppo di questi sistemi dovrebbe essere principalmente guidato da piani. All'altro estremo, i sistemi di piccole e medie dimensioni, da utilizzare in un ambiente competitivo in rapida evoluzione, dovrebbero essere sviluppati principalmente con metodi agili. Se più società sono impegnate nello sviluppo di un progetto, di solito si usa un approccio guidato da piani per coordinare il lavoro tra i vari siti di sviluppo.

20.2.1 Piano di progettazione

In un progetto di sviluppo guidato da piani, un piano di progettazione definisce le risorse disponibili, la suddivisione del lavoro e la tempistica per svolgere il lavoro. Il piano dovrebbe identificare l'approccio da adottare nella gestione dei rischi e descrivere i rischi del progetto e del software che si sta sviluppando. I dettagli di un piano di progettazione variano in funzione del tipo di progetto e organizzazione, ma di solito includono le seguenti parti.

1. *Introduzione* – Descrive brevemente gli obiettivi del progetto e definisce i vincoli (budget, tempistica ecc.) che influiscono sulla gestione del progetto.
2. *Organizzazione* – Descrive il modo in cui il team di sviluppo è organizzato, le persone coinvolte e i loro ruoli.
3. *Analisi del rischio* – Descrive i possibili rischi del progetto, la loro probabilità di concretizzazione e le strategie per limitarli.
4. *Requisiti delle risorse hardware e software* – Specifica l'hardware e il software di supporto richiesti per eseguire lo sviluppo. Se occorre acquistare nuove unità hardware, dovrebbero essere incluse anche le stime dei costi e dei tempi di consegna dell'hardware.
5. *Suddivisione del lavoro* – Descrive la ripartizione del progetto in varie attività e definisce gli input e gli output di ciascuna attività.
6. *Tempistica del progetto* – Mostra le dipendenze tra le varie attività, il tempo stimato per raggiungere ciascun traguardo (*milestone*), e l'allocazione delle persone alle singole attività.

Piano	Descrizione
Piano di gestione della configurazione	Describe le procedure e le strutture di gestione della configurazione.
Piano di consegne	Describe come il software e l'hardware associato (se richiesto) dovranno essere consegnati al cliente. Dovrebbe includere anche un piano per trasferire i dati dai sistemi esistenti.
Piano di manutenzione	Prevede i requisiti, i costi e le attività di manutenzione del sistema.
Piano di qualità	Describe le procedure e gli standard di qualità da seguire nella progettazione.
Piano di convalida	Describe l'approccio, le risorse e la tempistica da adottare per convalidare il sistema.

Figura 20.2 Piani di progettazione supplementari.

7. *Meccanismi di monitoraggio e reporting* – Definisce la documentazione da produrre sulla gestione, i suoi tempi di produzione, e i meccanismi di monitoraggio del progetto.

Il piano di progettazione principale dovrebbe includere sempre la valutazione dei rischi e la tempistica di realizzazione del progetto. In aggiunta, è possibile sviluppare altri piani supplementari per attività quali il testing del software e la gestione della configurazione. La Figura 20.2 illustra alcuni piani supplementari che potrebbero essere sviluppati. Questi piani di solito sono necessari nello sviluppo di sistemi grandi e complessi.

20.2.2 Processo di pianificazione

La pianificazione dei progetti è un processo iterativo che inizia quando viene creato il primo piano di progettazione durante la fase di avviamento del progetto. La Figura 20.3 mostra un diagramma UML che illustra un tipico workflow per il processo di pianificazione di un progetto. Le modifiche del piano di progettazione sono inevitabili. Man mano che si acquisiscono nuove informazioni sul sistema e sul team durante la progettazione, occorre rivedere periodicamente il piano per adattarlo ai nuovi requisiti, alla nuova tempistica e ai nuovi rischi. Anche la modifica degli obiettivi aziendali porta a cambiamenti del piano di progettazione. Se gli obiettivi aziendali cambiano, dovranno essere rivisti tutti i progetti.

All'inizio di un processo di pianificazione si dovrebbero definire i vincoli che influiscono sul progetto. Questi vincoli sono la data di consegna del progetto, il personale del team di sviluppo, il budget, gli strumenti disponibili e così via. Oltre a questi vincoli, si dovrebbero identificare anche i traguardi raggiunti e le consegne effettuate. I traguardi sono punti del piano rispetto ai quali si può valutare il progresso del progetto, per esempio, il sistema è pronto per i test. Le consegne

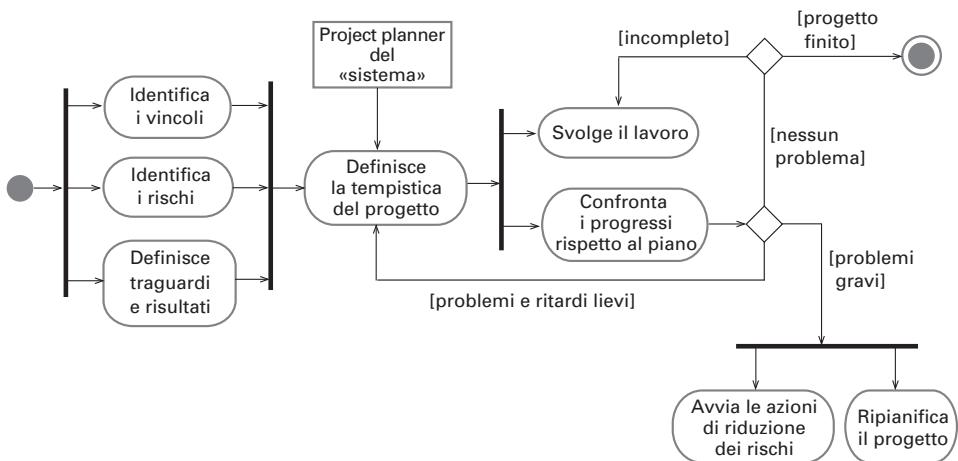


Figura 20.3 Il processo di pianificazione di un progetto.

sono prodotti che possono essere passati al cliente, per esempio, un documento sui requisiti del sistema.

Il processo poi entra in un ciclo che termina quando il progetto è completato. Si definisce un piano iniziale di realizzazione del progetto, e si avviano o si continuano le attività previste nel piano. Dopo qualche tempo (di solito due o tre settimane circa), occorre rivedere lo stato di avanzamento del progetto e mettere in evidenza eventuali discrepanze rispetto al piano programmato. Poiché le stime iniziali dei parametri di un progetto sono inevitabilmente approssimate, è normale che ci sia qualche lieve rallentamento nell'avanzamento dei lavori; per questo, occorre apportare qualche modifica al piano originale.

È bene fare delle ipotesi realistiche, anziché ottimistiche, quando si definisce il piano di un progetto. Durante lo sviluppo di un progetto possono sempre nascerne dei problemi, che possono comportare qualche ritardo nell'avanzamento del progetto. Le ipotesi e le tempistiche iniziali dovrebbero essere quindi pessimistiche, in modo da tener conto dei problemi inaspettati. Il piano dovrebbe prevedere anche alcune strategie di emergenza da adottare quando qualcosa va storto, in modo da non danneggiare irreparabilmente il piano di realizzazione del progetto.

Se, durante lo sviluppo del progetto, si presentano problemi seri che potrebbero causare significativi ritardi, occorre avviare delle opportune azioni per scongiurare il rischio del fallimento completo del progetto. Oltre a queste azioni, si dovrebbe rivedere il piano del progetto. Questo potrebbe richiedere la rinegoziazione dei vincoli del progetto con il cliente; una nuova tempistica delle attività da svolgere dovrebbe essere ridefinita e concordata con il cliente.

Se questa rinegoziazione non ha successo o se le azioni di limitazione dei rischi sono inefficaci, allora occorre organizzare una revisione tecnica formale del progetto. L'obiettivo di questa revisione è trovare un approccio alternativo che

consenta la prosecuzione del progetto. La revisione dovrebbe anche verificare che gli obiettivi del cliente siano preservati e che il progetto resti allineato con tali obiettivi.

Il risultato della revisione potrebbe essere l'annullamento di un progetto. Questo potrebbe essere dovuto a fallimenti tecnici o gestionali ma, di solito, è la conseguenza di modifiche esterne che influiscono sul progetto. Per sviluppare un grande progetto software spesso occorrono alcuni anni. In questo periodo di tempo, gli obiettivi e le priorità aziendali inevitabilmente cambiano. Questi cambiamenti potrebbero significare che il software non è più necessario o che i requisiti del progetto iniziale non sono appropriati. I manager possono quindi decidere di annullare lo sviluppo del software o di apportare sostanziali modifiche al progetto in modo da soddisfare le mutate esigenze aziendali.

20.3 Tempistica dei progetti

Nella tempistica di un progetto si stabilisce come il lavoro di un progetto dovrà essere organizzato in compiti separati, e come e quando questi compiti dovranno essere eseguiti. Occorre stimare il tempo necessario per completare ciascun compito e le attività connesse, e decidere chi dovrà svolgere i compiti che sono stati identificati. Occorre anche valutare le risorse software e hardware necessarie per completare ciascun compito. Per esempio, se si sta sviluppando un sistema integrato, occorre stimare il tempo da impiegare sull'hardware specializzato e i costi per utilizzare un simulatore di sistema. In termini di fasi di pianificazione descritte all'inizio di questo capitolo, di solito viene definita una tempistica iniziale durante la fase di avviamento del progetto. Questa tempistica viene successivamente affinata e modificata durante lo sviluppo del progetto.

Sia i metodi guidati da piani sia i metodi agili richiedono una pianificazione iniziale della progettazione, sebbene un metodo agile richieda meno dettagli. La definizione iniziale della tempistica serve a stabilire come le persone dovranno essere allocate ai vari compiti e a controllare il progresso di un progetto rispetto alle scadenze contrattuali. Nei processi di sviluppo tradizionali la tempistica viene sviluppata inizialmente e poi modificata all'avanzare del progetto. Nei processi agili deve esserci una tempistica globale che serve a identificare quando le principali fasi del progetto saranno completate. Successivamente, si applica alla tempistica un metodo iterativo per ciascuna fase del progetto.

La definizione della tempistica nei progetti guidati da piani (Figura 20.4) richiede la suddivisione di tutto il lavoro necessario per completare un progetto in compiti distinti, e la stima del tempo richiesto per completare ciascuno di questi compiti. Di solito, i singoli compiti durano da una settimana a due mesi. Una suddivisione più fine significa che dovrà essere impiegata una quantità di tempo sproporzionata per ripianificare e aggiornare il piano di un progetto. In generale, la quantità massima di tempo per ciascun compito dovrebbe variare da 6 a 8 set-

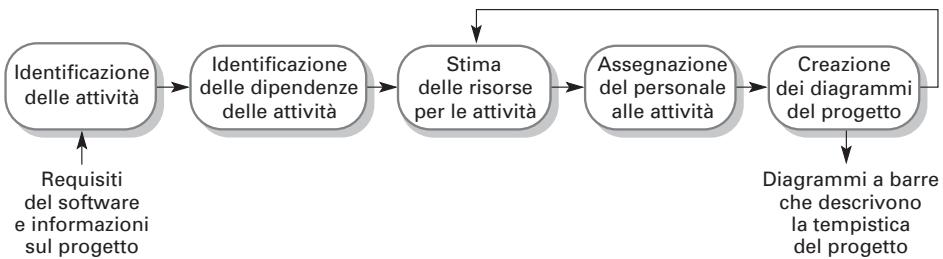


Figura 20.4 Definizione della tempistica di un progetto.

timane; se un compito dovesse richiedere più tempo, è bene suddividerlo in sotto-compiti.

Alcuni di questi compiti sono svolti in parallelo, con varie persone che lavorano sui vari componenti del sistema. Occorre coordinare questi compiti paralleli e organizzare il lavoro in modo che le persone siano utilizzate nel modo ottimale, evitando di creare inutili dipendenze tra i vari compiti. È importante evitare situazioni in cui l'intero progetto possa subire ritardi a causa di un compito critico che non è stato ancora ultimato.

Se un progetto è tecnicamente avanzato, le stime iniziali quasi sempre saranno ottimistiche, anche quando si cerca di prendere in considerazione tutte le eventualità. In questo senso, la tempistica del software non si differenzia dalla pianificazione di qualsiasi altro grande progetto avanzato. La realizzazione dei nuovi aerei o ponti, e perfino dei nuovi modelli di automobili, è spesso in ritardo a causa di problemi imprevisti. La tempistica deve quindi essere aggiornata ogni volta che si rendono disponibili nuove informazioni sull'avanzamento del progetto. Se il progetto che si sta pianificando è simile a un precedente progetto, si possono utilizzare le stime precedenti. Tuttavia, i progetti possono utilizzare metodi di realizzazione e linguaggi di implementazione differenti; quindi l'esperienza acquisita in un precedente progetto potrebbe essere inapplicabile al nuovo progetto.

Quando si definisce la tempistica di un progetto, occorre mettere in conto la possibilità che qualcosa possa andare storto. Le persone che lavorano su un progetto possono sbagliare o andarsene, l'hardware potrebbe guastarsi, e il software e l'hardware di supporto potrebbero arrivare in ritardo. Se il progetto è nuovo e tecnicamente avanzato, alcune sue parti potrebbero risultare più difficili da realizzare e richiedere più tempo di quanto originariamente previsto.

Una buona regola pratica consiste nel fare le stime come se tutto dovesse andare bene e, poi, aumentare tali stime per tenere conto dei problemi prevedibili. Le stime dovrebbero essere aumentate anche per tenere conto delle attività di emergenza necessarie per far fronte a problemi imprevisti. Queste attività di emergenza dipendono dal tipo di persone, dai parametri del processo (scadenze, standard ecc.), dalla qualità e dall'esperienza degli ingegneri che sviluppano il software. Per esempio, la stima delle attività e dei tempi richiesti per il progetto potrebbe essere aumentata del 30-50% per tenere conto delle emergenze.

Compito	Impegno (giorno-persona)	Durata (giorni)	Dipendenze
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figura 20.5 Compiti, durate e dipendenze tra le attività.

20.3.1 Presentazione della tempistica

La tempistica di un progetto può essere semplicemente documentata in una tabella o in un foglio elettronico che mostra i compiti, le attività stimate, le durate e le dipendenze tra i compiti (Figura 20.5). Tuttavia, questo stile di presentazione rende difficile vedere le relazioni e le dipendenze tra le varie attività. Per agevolare la comprensione della tempistica di un progetto, sono state sviluppate rappresentazioni grafiche alternative. Due sono i tipi di rappresentazioni grafiche più comuni.

1. I diagrammi a barre basati sul calendario mostrano, per ogni compito, le persone responsabili, la durata prevista, il tempo di inizio e di fine. I diagrammi a barre sono anche detti diagrammi di Gantt, dal suo inventore Henry Gantt.
2. Le reti delle attività mostrano le dipendenze tra le varie attività che formano un progetto. Queste reti sono descritte in una sezione web associata.

Le attività del progetto costituiscono gli elementi di base della pianificazione. Ogni attività ha:

- una durata espressa in giorni o mesi di calendario;
- un stima dell'impegno delle persone, espressa in numero di giorni-persona o mesi-persona per completare il lavoro;
- una scadenza entro la quale l'attività deve essere completata;

Diagrammi delle attività

Un diagramma delle attività è la rappresentazione della tempistica di un progetto che mostra il piano del progetto come un grafo orientato. Il diagramma illustra i compiti da svolgere in parallelo e quelli da svolgere in sequenza a causa delle loro dipendenze da attività precedenti. Se un compito dipende da molti altri compiti, allora tutti questi compiti devono essere completati prima che il compito sia avviato. Il “percorso critico” attraverso il diagramma delle attività è formato dalla sequenza più lunga dei compiti dipendenti. Questo percorso definisce la durata del progetto.

<http://software-engineering-book.com/web/planning-activities/>

- un obiettivo definito, che può essere la scrittura di un documento, lo svolgimento di una riunione di revisione, il superamento di tutti i test e così via.

Quando si pianifica un progetto, si può decidere di definire i traguardi (milestone) da raggiungere. Un traguardo è la fine logica di una fase del progetto, in corrispondenza della quale può essere rivisto l'avanzamento del lavoro. Ogni traguardo deve essere documentato da un breve rapporto (spesso una semplice e-mail) che riassume il lavoro svolto e se il lavoro è stato completato come previsto nel piano del progetto. I traguardi possono essere associati a un singolo compito o a gruppi di attività correlate. Per esempio, nella Figura 20.5, il traguardo M1 è associato alla coppia di compiti T2 e T4; non c'è un singolo traguardo alla fine di questi compiti.

Alcune attività completano le “consegne” del progetto, ovvero gli output che devono essere consegnati al cliente del software. Di solito, le consegne sono specificate esplicitamente nel contratto del progetto, e il cliente giudica l'avanzamento del progetto da queste consegne. I traguardi e le consegne non sono la stessa cosa. I traguardi sono brevi sintesi sull'avanzamento dei lavori, mentre le consegne sono risultati più sostanziosi, come il documento dei requisiti o l'implementazione iniziale del sistema.

La Figura 20.5 mostra un ipotetico gruppo di compiti, i relativi impegni, durate e interdipendenze. Da questa tabella è possibile notare che il compito T3 dipende dal compito T1. Questo significa che il compito T1 deve essere completato prima di T3, la configurazione del sistema selezionato. Non è possibile avviare la configurazione del sistema finché non sarà stato scelto e installato il sistema applicativo da modificare.

Si noti che la durata stimata di alcuni compiti è qualcosa più dell'impegno richiesto e viceversa. Se l'impegno è minore della durata, le persone assegnate a quel compito non stanno lavorando a pieno regime. Se l'impegno supera la durata, allora più membri del team si stanno occupando contemporaneamente di quel compito.

La Figura 20.6 si basa sulle informazioni della Figura 20.5 e presenta la tempistica del progetto in un diagramma a barre, che mostra le date di inizio e fine dei vari compiti in funzione del tempo. Leggendo il diagramma da sinistra a

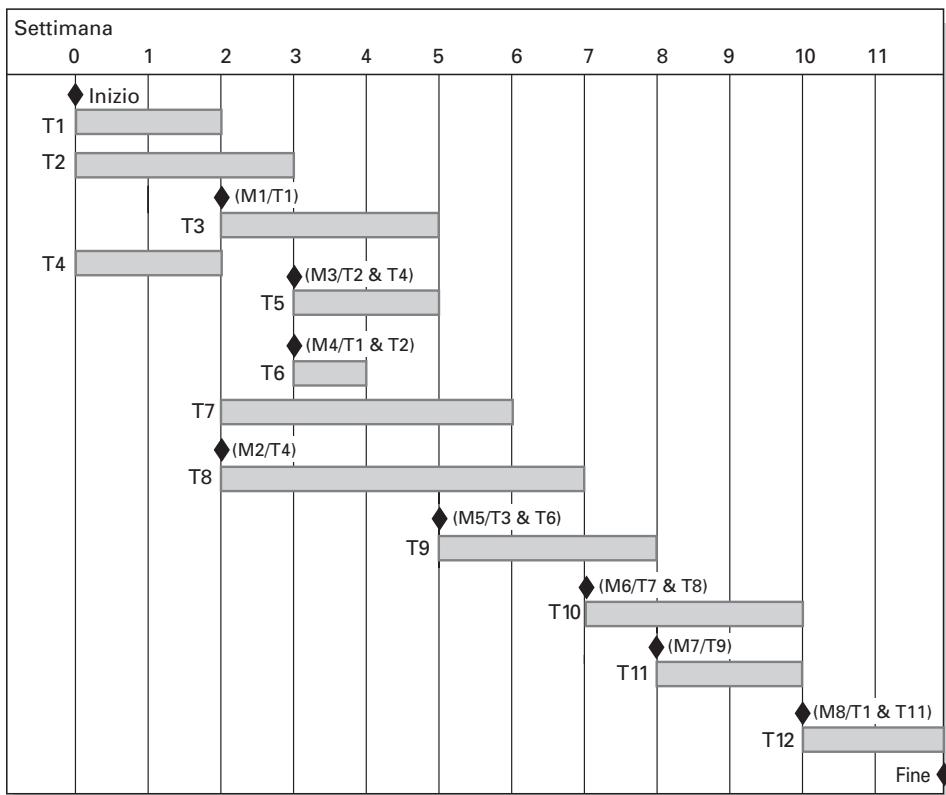


Figura 20.6 Diagramma a barre delle attività di un progetto.

destra, si può facilmente notare l'inizio e la fine dei singoli compiti. Nel diagramma sono indicati anche i traguardi (M1, M2 ecc.). Si noti che i compiti che sono indipendenti possono essere svolti in parallelo. Per esempio, i compiti T1, T2 e T4 iniziano tutti insieme all'avviamento del progetto.

Oltre a definire la tempistica delle consegne per il software, i project manager devono anche allocare le risorse ai vari compiti. Le risorse chiave, ovviamente, sono gli ingegneri che dovranno svolgere il lavoro. Essi saranno assegnati alle varie attività del progetto. L'allocazione delle risorse può essere analizzata tramite appositi strumenti di gestione della progettazione; può essere generato un diagramma a barre che indica i periodi in cui le persone lavorano sul progetto (Figura 20.7). Le persone possono occuparsi di più compiti contemporaneamente. A volte potrebbero non lavorare sul progetto in esame; potrebbero andare in ferie, lavorare su un altro progetto o partecipare a corsi di addestramento. Le barre con la diagonale indicano le attività part-time.

Le grandi società di solito impiegano un certo numero di specialisti che lavorano su un progetto quando è necessario. Nella Figura 20.7 si può notare che Mary è una specialista che si occupa di un solo compito (T5) del progetto. L'im-

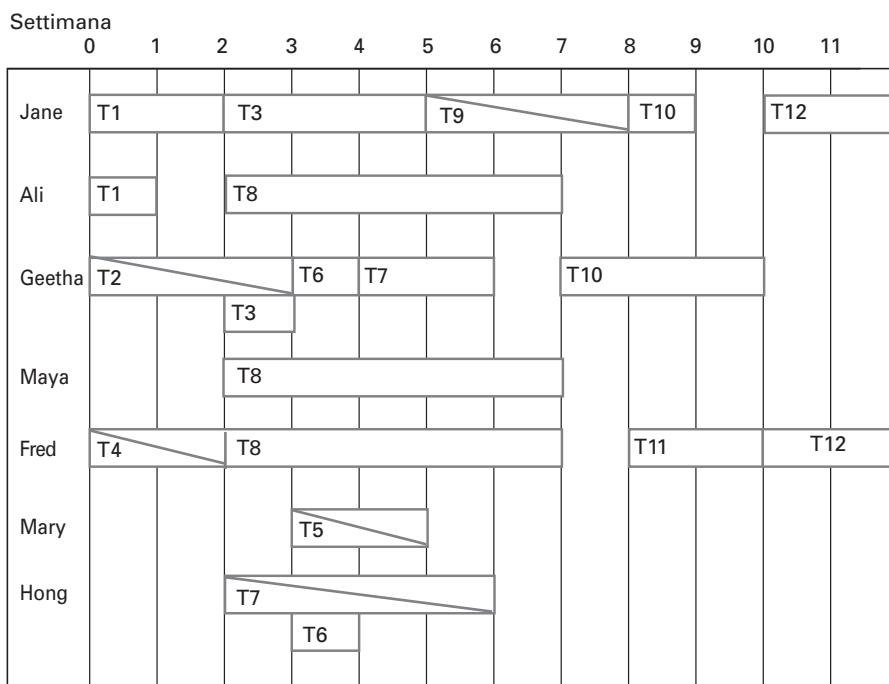


Figura 20.7 Diagramma di allocazione del personale.

piego di specialisti è inevitabile quando si sviluppano sistemi complessi, ma può causare problemi di tempistica. Se un progetto subisce un ritardo mentre uno specialista sta lavorando su di esso, questo potrebbe influire su altri progetti, dove è impegnato lo stesso specialista. Questi progetti potrebbero subire ritardi in quanto lo specialista non è disponibile.

Se un compito è in ritardo, anche i compiti successivi che dipendono da esso potrebbero subire un ritardo. Questi compiti non possono essere avviati finché il compito in ritardo non sarà completato. I ritardi possono causare problemi seri all'allocazione del personale, specialmente se le persone sono impegnate contemporaneamente su più progetti. Se un compito (T) è in ritardo, le persone assegnate a questo compito potrebbero essere assegnate a un altro lavoro (W). Per completare questo lavoro, potrebbe essere necessario un tempo maggiore del ritardo; ma, una volta avviato il lavoro W, queste persone non possono più essere riassegnate al compito originale. Questo potrebbe ritardare ulteriormente il compito T, perché le persone devono completare il lavoro W.

Di solito, si usa un apposito strumento di pianificazione della progettazione, come Basecamp o Microsoft project, per creare, aggiornare e analizzare le informazioni della tempistica dei progetti. Gli strumenti di gestione dei progetti di solito richiedono che le informazioni sul progetto vengano immesse in una tabella; queste informazioni vengono inserite in un database. I diagrammi a barre e i diagrammi delle attività vengono automaticamente generati da questo database.

20.4 Pianificazione agile

I metodi agili di sviluppo del software sono approcci iterativi nei quali il software viene sviluppato e consegnato ai clienti con incrementi successivi. Diversamente dagli approcci guidati da piani, le funzionalità di questi incrementi non vengono pianificate in anticipo, ma vengono decise durante lo sviluppo. La decisione su cosa includere in un incremento dipende dall'avanzamento dei lavori e dalle priorità del cliente. La ragione a favore di questo approccio è che le priorità e i requisiti del cliente possono cambiare, quindi ha senso avere un piano flessibile che possa accogliere queste modifiche. Il libro di Cohn (Cohn 2005) è un'eccellente introduzione alla pianificazione agile.

I metodi di sviluppo agile, come Scrum (Rubin 2013) e la programmazione estrema (Beck e Andres 2004), hanno un approccio in due fasi alla pianificazione, che corrisponde alla fase di avviamento nello sviluppo guidato da piani e nella pianificazione dello sviluppo:

1. *pianificazione delle release*; si fanno previsioni per vari mesi e si scelgono le funzionalità da includere in una release del sistema;
2. *pianificazione delle iterazioni*; si fa una previsione a breve termine e si decide la pianificazione del successivo incremento del sistema. Di solito, questo richiede da 2 a 4 settimane di lavoro del team di sviluppo.

Ho già spiegato l'approccio Scrum nel Capitolo 3, che si basa sui product backlog e sulle revisioni giornaliere del lavoro da svolgere; è orientato fondamentalmente alla pianificazione delle iterazioni. Un altro approccio alla pianificazione agile, che è stato sviluppato come parte della programmazione estrema, si basa sulle storie utente. Il cosiddetto “planning game” (gioco della pianificazione) può essere utilizzato sia nella pianificazione delle release sia nella pianificazione delle iterazioni.

La base del planning game (Figura 20.8) è una serie di storie utente (Capitolo 3) che riguardano tutte le funzionalità da includere nel sistema finale. Il team di sviluppo e il cliente del software lavorano insieme per sviluppare queste storie. I membri del team leggono e discutono le storie e le classificano in base alla quantità di tempo che essi ritengono possa essere richiesta per implementarle. Alcune storie potrebbero essere troppo grandi per essere implementate in una sola iterazione, quindi devono essere suddivise in storie più piccole.

Il problema della classificazione delle storie è che spesso è difficile stimare il tempo e l'impegno necessari per fare qualcosa. Per semplificare questo, si potrebbero utilizzare delle classificazioni relative. Il team confronta le storie a coppie e decide quale potrà richiedere più tempo e impegno, senza stabilire esattamente quanto impegno sarà necessario per implementarle. Alla fine di questo processo, la lista delle storie risulta ordinata, con le storie più impegnative poste in cima alla lista. Il team valuta il livello di impegno assegnando un punteggio appropriato alle singole storie della lista. Una storia complessa potrebbe valere 8 punti, una storia semplice 2 punti.

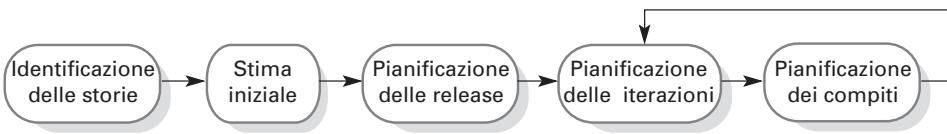


Figura 20.8 Il “planning game”.

Una volta definite le storie, le corrispondenti attività vengono utilizzate per fare una prima stima dell'impegno totale richiesto per implementare il sistema, applicando il concetto di “velocità”. La velocità è il numero di punti-impegno al giorno corrispondenti al lavoro di implementazione del team. Questo valore può essere stimato sia in base a precedenti esperienze sia sviluppando una o due storie per vedere quanto tempo è richiesto. La stima della velocità è approssimata, ma può essere affinata durante il processo di sviluppo. Una volta stimata la velocità, è possibile calcolare l'impegno totale (espresso in giorni-persona) per implementare il sistema.

La pianificazione delle release richiede la scelta e l'affinamento delle storie che potranno rispecchiare le funzioni da implementare in una release del sistema e l'ordine in cui le storie dovranno essere implementate. Il cliente deve essere coinvolto in questo processo. Si sceglie poi una data per la release, e le storie vengono esaminate per vedere se la stima degli impegni è compatibile con tale data. In caso negativo, le storie vengono aggiunte o rimosse dalla lista.

La pianificazione delle iterazioni è la prima fase nello sviluppo di un incremento completo del sistema. Durante un'iterazione vengono selezionate le storie da implementare, con il numero di storie che rispecchia il tempo per consegnare un sistema funzionante (di solito, 2 o 3 settimane) e la velocità del team. Quando arriva la data di consegna, l'iterazione dello sviluppo è completa, anche se non sono state implementate tutte le storie. Il team considera le storie che sono state implementate e aggiunge i corrispondenti punti-impegno. La velocità viene ricalcolata, e questo valore viene utilizzato per pianificare la successiva versione del sistema.

All'inizio di ogni iterazione di sviluppo, c'è una fase di pianificazione dei compiti in cui gli sviluppatori suddividono le storie in compiti di sviluppo. Un compito di sviluppo dovrebbe durare 4-16 ore. Vengono elencati tutti i compiti che devono essere completati per implementare tutte le storie di un'iterazione. I singoli sviluppatori vengono assegnati ai compiti specifici che dovranno implementare. Ogni sviluppatore conosce la sua velocità e non dovrebbe assumere più compiti di quelli che è in grado di implementare nel tempo prestabilito.

Questo approccio all'allocazione dei compiti offre due importanti vantaggi.

1. Tutti i membri del team hanno una visione dei compiti da completare in ogni iterazione. Quindi, ogni membro conosce che cosa stanno facendo gli altri membri e sa con chi parlare se identifica delle dipendenze con altri compiti.

2. I singoli sviluppatori scelgono i compiti da implementare; non è il project manager che assegna loro i compiti. Essi, quindi, sviluppano un senso di appartenenza a questi compiti, e questo potrebbe motivarli maggiormente nel lavoro.

A metà di un’iterazione, l’avanzamento dei lavori viene rivisto. In questa fase, è stata impiegata metà dei punti-impegno delle storie. Quindi, se un’iterazione richiede 24 punti per le storie e 36 compiti, dovrebbero essere stati impiegati 12 punti per le storie e completati 18 compiti. Se non è così, allora bisogna discutere con il cliente quali storie eliminare dall’incremento del sistema che si sta sviluppando.

Questo approccio alla pianificazione ha il vantaggio che l’incremento di un sistema software viene sempre consegnato alla fine di ciascuna iterazione del progetto. Se le funzionalità da includere nell’incremento non possono essere completate entro il tempo previsto, allora vengono ridotte. La tempistica delle consegne non viene mai estesa. Questo, però, può causare dei problemi, in quanto i piani del cliente potrebbero esserne influenzati. La riduzione delle funzionalità potrebbe significare un lavoro extra per il cliente, perché sarebbe costretto a utilizzare un sistema incompleto o a cambiare il modo di lavorare tra una release e l’altra.

Un problema importante nella pianificazione agile è che essa si basa sul coinvolgimento e sulla disponibilità del cliente. Questo coinvolgimento può essere difficile da gestire, in quanto i rappresentanti del cliente a volte hanno altre priorità e non sono disponibili per il planning game. In aggiunta, alcuni clienti potrebbero avere maggiore familiarità con la pianificazione tradizionale dei progetti e potrebbero trovarsi in difficoltà nei processi di pianificazione agile.

La pianificazione agile si adatta bene a team di sviluppo piccoli e stabili, perché possono facilmente riunirsi e discutere le storie da implementare. Se, invece, i team sono grandi e/o geograficamente distribuiti oppure se i membri del team cambiano frequentemente, è praticamente impossibile che tutti i membri siano coinvolti in una pianificazione partecipativa che è essenziale nella gestione della progettazione agile. Di conseguenza, i grandi progetti di solito sono pianificati utilizzando i metodi tradizionali della gestione della progettazione.

20.5 Tecniche di stima

È difficile stimare la tempistica di realizzazione di un progetto. Occorre fare delle stime iniziali sulla base di una definizione incompleta dei requisiti del cliente. Potrebbe essere necessario eseguire il software su piattaforme poco familiari o utilizzare nuove tecnologie di sviluppo. Le capacità delle persone coinvolte nel progetto potrebbero non essere note. Ci sono tante incertezze che è impossibile stimare accuratamente i costi di sviluppo di un sistema durante le fasi iniziali di un progetto. Ciononostante, le società hanno bisogno di stimare le attività e i costi

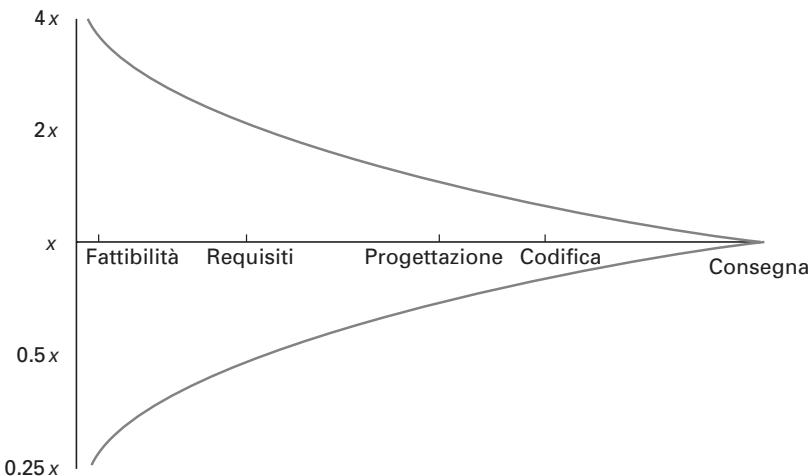


Figura 20.9 Variabilità delle stime.

per sviluppare un sistema software. Ci sono due tecniche che possono essere adottate per fare queste stime.

1. *Tecniche basate sull'esperienza* – La stima delle attività future si basa sull'esperienza acquisita dal manager nello sviluppo di precedenti progetti. Essenzialmente, il manager effettua delle valutazioni sulle attività necessarie per realizzare un progetto.
2. *Modellazione algoritmica dei costi* – Si adotta un metodo matematico per calcolare le attività del progetto in base alle stime degli attributi del prodotto, quali la dimensione, le caratteristiche del processo e l'esperienza del personale del team di sviluppo.

In entrambi i casi, occorre stimare con attenzione sia l'impegno richiesto direttamente dalle persone sia le caratteristiche dei progetti e dei prodotti. Nella fase di avviamento di un progetto, queste stime hanno un grande margine di errore. Sulla base dei dati raccolti da un gran numero di progetti, Boehm (B. Boehm et al. 1995) ha scoperto che le stime nella fase di avviamento variano significativamente. Se la stima iniziale dell'impegno delle persone è pari a x mesi, allora la variazione può essere compresa tra $0.25x$ e $4x$ dell'impegno effettivo misurato alla consegna del sistema. Durante l'avanzamento dei lavori, le stime diventano sempre più accurate, come illustra la Figura 20.9.

Le tecniche basate sull'esperienza si basano sull'esperienza acquisita dai manager nello sviluppo di precedenti progetti e sugli sforzi realmente fatti nelle attività correlate allo sviluppo del software. Tipicamente, si identificano le conseguenze effettuabili in un progetto e i vari componenti o sistemi software che devono essere sviluppati. Questi elementi, con le relative stime, vengono immessi in foglio elettronico; quindi viene calcolato l'impegno totale per sviluppare il pro-

getto. Di solito, è bene coinvolgere le persone che partecipano allo sviluppo del progetto, e chiedere a ciascuna di esse di spiegare le proprie stime. Così facendo, spesso, emergono fattori che altri non avevano considerato; il procedimento si ripete per arrivare a una stima concordata tra tutti i membri del team.

La difficoltà delle tecniche basate sull'esperienza è che un progetto software nuovo potrebbe non avere nulla in comune con i precedenti progetti. Lo sviluppo del software cambia molto rapidamente, e un progetto spesso usa tecniche non familiari, come i servizi web, la configurazione dei sistemi applicativi o HTML5. Se non avete utilizzato queste tecniche, la vostra esperienza precedente potrebbe non aiutarvi nella stima degli sforzi richiesti, rendendo più difficile formulare stime accurate dei costi e dei tempi di realizzazione di un progetto.

È impossibile dire se siano più accurati i metodi algoritmici o le tecniche basate sull'esperienza. Le stime di un progetto spesso sono auto-avveranti. La stima viene utilizzata per definire il budget di un progetto, e il prodotto viene adattato in modo da rispettare la previsione del budget. Il rispetto delle previsioni del budget potrebbe essere ottenuto a danno delle funzionalità del software che si sta sviluppando.

Per fare un confronto tra l'accuratezza delle stime di queste due tecniche, occorrerebbe fare alcuni esperimenti controllati, nei quali vengono adottati vari metodi in modo indipendente per stimare gli sforzi e i costi di un progetto. Non è consentito modificare il progetto, e devono essere confrontati soltanto gli sforzi effettuati alla fine del progetto. Il project manager non dovrebbe conoscere le stime degli sforzi, evitando così sue possibili influenze. Questo scenario, purtroppo, è impossibile da ricreare nei progetti reali, quindi non avremo mai un confronto obiettivo tra questi approcci.

20.5.1 Modellazione algoritmica dei costi

La modellazione algoritmica dei costi usa una formula matematica per prevedere i costi di progettazione basandosi sulle stime della dimensione del progetto, del tipo di software da sviluppare e di altri fattori relativi al team, al processo e ai prodotti. I modelli algoritmici dei costi possono essere sviluppati analizzando i costi e gli attributi di progetti completati, e poi trovando la formula che meglio si adatta ai costi effettivamente sostenuti.

I modelli algoritmici dei costi sono utilizzati principalmente per stimare i costi di sviluppo del software. Boehm (Boehm et al. 2000) ha descritto diverse applicazioni di questi modelli, quali la preparazione delle stime per gli investitori nelle società di software, la definizione di strategie alternative per valutare i rischi, per riutilizzare il software esistente, per riavviare il processo di sviluppo o per ricorrere a fornitori esterni (outsourcing).

Molti modelli algoritmici per la stima dello sforzo richiesto in un progetto software si basano su una semplice formula:

$$\text{Sforzo} = A \times \text{Dimensione}^B \times M$$

dove:

A è un fattore costante che dipende dalle prassi organizzative interne e dal tipo di software che si sta sviluppando;

Dimensione è una stima della dimensione del codice del software o di una funzionalità espressa in punti-funzione o punti-applicazione;

B rappresenta la complessità del software e di solito è compreso tra 1 e 1,5;

M è un fattore che tiene conto degli attributi di processo, prodotto e sviluppo, quali i requisiti di affidabilità del software e l'esperienza del team di sviluppo. Questi attributi possono aumentare o diminuire la difficoltà generale di sviluppo del sistema.

Il numero di righe del codice sorgente (SLOC) nel sistema consegnato al cliente è il parametro dimensionale che viene utilizzato in molti modelli algoritmici dei costi. Per stimare il numero di righe del codice di un sistema software, è possibile utilizzare una combinazione dei seguenti approcci.

1. Confrontare il sistema da sviluppare con analoghi sistemi preesistenti e utilizzare la dimensione del loro codice come base stimare la dimensione del nuovo sistema.
2. Stimare il numero di punti-funzione o punti-applicazione del sistema (si veda il successivo paragrafo) e convertire con una formula questo valore in righe di codice nel linguaggio di programmazione utilizzato.
3. Classificare i componenti del sistema in base a stime delle loro dimensioni relative e utilizzare un componente di riferimento noto per tradurre questa classificazione in dimensione del codice.

La maggior parte dei modelli algoritmici dei costi ha un componente esponenziale (B nella formula precedente) che aumenta con la dimensione e la complessità del sistema. Al crescere della dimensione e della complessità del software, aumentano i costi extra, perché crescono gli overhead delle comunicazioni tra team più numerosi, la gestione della configurazione diventa più complessa, l'integrazione del sistema è più difficile e così via. Quanto più è complesso il sistema, tanto più questi fattori influiscono sui costi.

L'idea di utilizzare un metodo scientifico e obiettivo per stimare i costi è interessante; purtroppo, tutti i modelli algoritmici dei costi presentano due problemi.

1. È praticamente impossibile stimare con precisione la *Dimensione* nella fase iniziale della progettazione, quando è disponibile soltanto la specifica del progetto. È più facile stimare i punti-funzione o i punti-applicazione (si veda il successivo paragrafo), anziché la dimensione del codice, sebbene anche la loro stima di solito non sia accurata.
2. Le stime dei fattori di complessità e di processo, che influiscono su B ed M , sono soggettive. Queste stime variano da una persona all'altra, in base alle personali conoscenze ed esperienze con il tipo di sistema che si sta sviluppando.

Produttività del software

La produttività del software è una stima della quantità media del lavoro di sviluppo che gli ingegneri svolgono in una settimana o in un mese. Essa può essere espressa in righe di codice/mese, punti-funzione/mese e così via.

Tuttavia, sebbene la produttività possa essere facilmente misurata quando c'è un risultato tangibile (per esempio, un amministratore elabora N reclami di viaggi al giorno), la produttività del software è più difficile da definire. Persone differenti possono implementare la stessa funzionalità in modi diversi, utilizzando un numero diverso di righe di codice. La qualità del codice è importante, ma è in qualche misura soggettiva. Quindi, non è possibile confrontare la produttività dei singoli ingegneri. Ha senso paragonare i valori di produttività soltanto per grandi team di sviluppo.

<http://software-engineering-book.com/web/productivity/>

È difficile stimare con precisione la dimensione del codice nella fase iniziale della progettazione, in quanto la dimensione del programma finale dipende dalle decisioni di progettazione che non sono state ancora prese quando viene richiesta la stima. Per esempio, se un'applicazione richiede una gestione complessa dei dati, si potrebbe usare un database commerciale oppure implementare un apposito sistema per la gestione dei dati. Nella stima iniziale dei costi, è improbabile che si sappia già che esiste un database commerciale in grado di soddisfare i requisiti di una gestione complessa dei dati. Quindi, non è possibile sapere quanto codice per la gestione dei dati sarà incluso nel sistema.

Anche il linguaggio di programmazione utilizzato nello sviluppo del software influenza sul numero di righe di codice da scrivere. Un linguaggio come Java potrebbe richiedere più righe di codice rispetto al linguaggio C. Questo codice extra, tuttavia, permette un maggiore controllo durante la fase di compilazione, quindi è probabile che i costi di convalida del software si riducano. Non è chiaro come si possa tenere conto di questo nel processo iniziale di stima dei costi. Anche la quantità di codice riutilizzato potrebbe influire sui costi; per questo, alcuni modelli stimano esplicitamente il numero di righe di codice riutilizzato. Tuttavia, se vengono riutilizzati i sistemi applicativi o i servizi esterni, allora è molto difficile calcolare il numero di righe di codice sorgente che viene rimpiazzato.

I modelli algoritmici dei costi sono metodi sistematici per stimare lo sforzo richiesto per sviluppare un sistema software. Questi modelli, purtroppo, sono complessi e difficili da utilizzare. Ci sono molti attributi e ampi margini di incertezza nella stima dei loro parametri. Questa complessità significa che l'applicazione pratica di questi modelli si è ridotta a un numero relativamente piccolo di grandi società, che operano principalmente nell'ingegneria dei sistemi di difesa e aerospaziali.

Un altro ostacolo che scoraggia l'utilizzo dei modelli algoritmici dei costi è la necessità di effettuare una calibrazione. Gli utenti dei modelli devono calibrare il loro modello e i valori degli attributi utilizzando i propri dati storici di progetta-

zione, perché questi rispecchiano la pratica e l'esperienza locali. Tuttavia, poche società hanno raccolto dati sufficienti dai precedenti progetti nella forma adatta a supportare tale calibrazione. L'utilizzo pratico dei modelli algoritmici, quindi, deve iniziare dai valori pubblicati per i parametri dei modelli. È praticamente impossibile per un modellatore sapere quanto questi parametri siano strettamente correlati alla sua società.

Se si utilizza un modello algoritmico di stima dei costi, si dovrebbe sviluppare una serie di stime (peggiore, attesa e migliore), anziché una singola stima, e applicare la formula del costo a tutte le stime. Le stime sono probabilmente più accurate quando si conosce il tipo di software che si sta sviluppando, quando il modello di costo è stato calibrato utilizzando dati locali o quando le scelte del linguaggio di programmazione e dell'hardware sono predefinite.

20.6 Modelli di costi COCOMO

Il miglior modello algoritmico di stima dei costi è COCOMO II. Questo modello empirico fu derivato dalla raccolta di dati di un ampio numero di progetti software di varie dimensioni. Questi dati furono analizzati per scoprire le formule più adatte a rappresentare le osservazioni fatte. Le formule collegavano la dimensione del sistema e i fattori del prodotto, del progetto e del team allo sforzo richiesto per sviluppare il sistema. COCOMO II è un modello disponibile gratuitamente che è supportato con strumenti open-source.

COCOMO II fu sviluppato dai precedenti modelli di stima COCOMO (Constructive Cost Modeling), che si basavano essenzialmente sullo sviluppo del codice originale (B. W. Boehm 1981; B. Boehm e Royce 1989). Il modello COCOMO II tiene conto dei moderni approcci allo sviluppo del software, come lo sviluppo rapido che usa linguaggi dinamici di programmazione, lo sviluppo con riutilizzo e la programmazione dei database. COCOMO II incorpora diversi sottomodelli che si basano su queste tecniche e che consentono di ottenere stime sempre più dettagliate.

I sottomodelli (Figura 20.10) che fanno parte del modello COCOMO II sono:

1. *Modello di composizione delle applicazioni.* Modella lo sforzo richiesto per sviluppare i sistemi creati da componenti riutilizzabili, da scripting o dalla programmazione di database. Le stime della dimensione del software si basano sui punti-applicazione, e utilizzano una semplice formula dimensione/produttività per stimare lo sforzo richiesto.
2. *Modello di progettazione iniziale.* È utilizzato nelle prime fasi della progettazione del sistema, dopo che sono stati definiti i requisiti. La stima si basa sulla formula standard che ho descritto nell'introduzione di questo capitolo. Le stime si basano sui punti-funzione, che vengono poi convertiti nel numero di righe del codice sorgente.

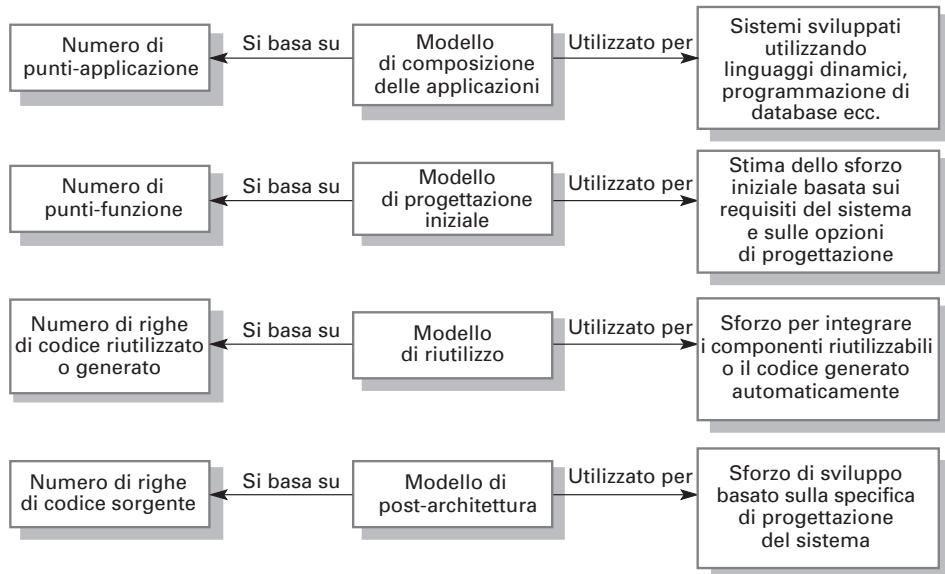


Figura 20.10 Modelli di stima COCOMO.

I punti-funzione sono un modo indipendente dal linguaggio per quantificare le funzionalità di un programma. Per calcolare il numero totale di punti-funzione di un programma, si misurano o si stimano il numero di input e output, le interazioni degli utenti, le interfacce esterne, i file o le tabelle dei database utilizzati dal sistema.

3. *Modello di riutilizzo.* Calcola lo sforzo richiesto per integrare i componenti riutilizzabili e/o il codice generato automaticamente per il programma. Di solito, viene utilizzato insieme al modello di post-architettura.
4. *Modello di post-architettura.* Una volta che l'architettura del sistema è stata progettata, si può fare una stima più accurata della dimensione del software. Anche questo modello utilizza la formula standard per la stima dei costi discussa in precedenza. Include, però, un insieme più esteso di 17 moltiplicatori che riflettono le capacità del personale e le caratteristiche del prodotto e del progetto.

Ovviamente, nei grandi sistemi, parti diverse possono essere sviluppate utilizzando tecnologie di sviluppo differenti, ed è possibile che non tutte le parti del sistema siano stimate con lo stesso livello di accuratezza. In questi casi, si può utilizzare un sottomodello appropriato per ciascuna parte del sistema e, poi, combinare i risultati per creare una stima composita.

Il modello COCOMO II è molto complesso e, per agevolarne la spiegazione, ho semplificato la sua presentazione. Potete utilizzare i modelli nel modo in cui li ho spiegati per una semplice stima dei costi. Per usare il modello COCOMO in modo appropriato, dovreste consultare il libro di Boehm e il manuale per il modello COCOMO II (B. W. Boehm et al 2000; Abts et al. 2000).

Esperienza e capacità dello sviluppatore	Molto basse	Basse	Normali	Alte	Molto alte
Maturità ed efficacia degli strumenti software	Molto basse	Basse	Normali	Alte	Molto alte
PROD (NAP/mese)	4	7	13	25	50

Figura 20.11 Produttività in punti-applicazione.

20.6.1 Modello di composizione delle applicazioni

Il modello di composizione delle applicazioni fu introdotto nel COCOMO II per stimare gli sforzi richiesti per sviluppare progetti prototipi e per realizzare progetti il cui software veniva sviluppato tramite la composizione di componenti esistenti. Si basa su una stima dei punti-applicazione (detti anche punti-oggetto) ponderati, divisa per una stima standard della produttività espressa in punti-applicazione (B. W. Boehm et al. 2000). Il numero dei punti-applicazione in un programma si ottiene da quattro stime più semplici:

- il numero di schermate o pagine web distinte che vengono visualizzate;
- il numero di report che vengono prodotti;
- il numero di moduli nel linguaggio di programmazione principale (come Java);
- il numero di righe nel linguaggio scripting o di codice di programmazione del database.

Questa stima viene poi affinata in funzione delle difficoltà di sviluppo di ciascun punto-applicazione. La produttività dipende dall'esperienza e dalle capacità dello sviluppatore e anche dall'efficacia degli strumenti software (ICASE) che vengono utilizzati a supporto dello sviluppo. La Figura 20.11 mostra i livelli di produttività (in punti-applicazione) indicati dagli sviluppatori del modello COCOMO II.

La composizione delle applicazioni di solito si basa sul riutilizzo del software esistente e sulla configurazione dei sistemi applicativi. Alcuni dei punti-applicazione nel sistema potranno quindi essere implementati tramite componenti riutilizzabili. Di conseguenza, si deve affinare la stima per tenere conto della percentuale di riutilizzo attesa. La formula finale per il calcolo dello sforzo per i prototipi di un sistema è quindi:

$$PM = (NAP \times (1 - \%riutilizzo/100)) / PROD$$

dove:

PM è lo sforzo stimato in mesi-persona;

NAP è il numero totale di punti-applicazione nel sistema consegnato;

%riutilizzo è una stima della quantità di codice riutilizzato nello sviluppo;

PROD è la produttività in punti-applicazione, come illustra la Figura 20.11.

20.6.2 Modello di progettazione iniziale

Questo modello viene utilizzato durante le fasi iniziali della progettazione, prima che sia disponibile un progetto architettonico dettagliato del sistema. Il modello di progettazione iniziale suppone che i requisiti dell’utente siano stati concordati e che le fasi iniziali del processo di progettazione del sistema siano state avviate. A questo punto, basta fare una rapida stima approssimata dei costi, sulla base di ipotesi esemplificative, supponendo per esempio che non ci sia alcuno sforzo per integrare il codice riutilizzabile.

Le stime iniziali sono particolarmente utili per analizzare quelle opzioni in cui occorre confrontare diversi modi di implementare i requisiti utente. Le stime prodotte in questa fase si basano sulla formula standard per i modelli algoritmici, ovvero:

$$\text{Sforzo} = A \times \text{Dimensione}^B \times M$$

Basandosi sul suo ricco dataset, Boehm suggerisce che il coefficiente **A** dovrebbe essere 2,94. La dimensione del sistema è espressa in KSLOC, ovvero in migliaia di righe di codice sorgente. Per calcolare KSLOC, si stima il numero di punti-funzione del software. Poi, si utilizzano delle tabelle standard, che correlano la dimensione del codice ai punti-funzione per diversi linguaggi di programmazione (QSM 2014), per ottenere una stima iniziale della dimensione del sistema in KSLOC.

L’esponente **B** riflette lo sforzo richiesto che cresce all’aumentare della dimensione del progetto. Può variare da 1,1 a 1,24, a seconda delle novità del progetto, della flessibilità di sviluppo, dei processi di risoluzione dei rischi, della coesione del team di sviluppo e del livello di maturità del processo di organizzazione aziendale (Capitolo 26 sul Web). Nel paragrafo dedicato al modello di post-architettura, spiegherò come calcolare il valore di questo esponente utilizzando tre parametri.

Questo porta a un calcolo dello sforzo come segue:

$$PM = 2,94 \times \text{Dimensione}^{(da\ 1,1\ a\ 1,24)} \times M$$

dove:

$$M = PERS \times PREX \times RCPX \times RUSE \times PDIF \times SCED \times FSIL;$$

PERS è la capacità del personale;

PREX è l’esperienza del personale;

RCPX è l’affidabilità e la complessità del prodotto;

RUSE è il riutilizzo richiesto;

PDIF è la difficoltà della piattaforma;

SCED è la tempistica;

FSIL sono le funzionalità di supporto.

Il moltiplicatore **M** si basa su sette attributi di progettazione e di processo che aumentano o riducono la stima. Questi attributi sono descritti nelle pagine web del libro. Per stimare questi attributi, si usa una scala di sei punti, dove 1 corrisponde a “molto basso” e 6 a “molto alto”; per esempio, **PERS = 6** significa che è disponibile un personale esperto per sviluppare il progetto.

20.6.3 Modello di riutilizzo

Il modello COCOMO per il riutilizzo del software è usato per stimare lo sforzo richiesto per integrare il codice riutilizzabile o generato. Come detto nel Capitolo 15, il riutilizzo del software attualmente è la norma in qualsiasi processo di sviluppo del software. Molti grandi sistemi includono una quota significativa di codice che è stato riutilizzato da precedenti progetti di sviluppo.

Il modello COCOMO II considera due tipi di codice riutilizzato. Il codice a scatola chiusa (*black-box code*) che lo sviluppatore può riutilizzare senza bisogno di capirlo o modificarlo. Esempi di questo codice sono i componenti che vengono automaticamente generati dai modelli UML o dalle librerie di applicazioni, come le librerie di grafica. Si suppone che lo sforzo di sviluppo per il codice a scatola chiusa sia nullo. La sua dimensione non viene presa in considerazione nel calcolo dello sforzo complessivo.

Il secondo tipo è il codice a scatola aperta (*white-box code*) che deve essere adattato affinché si possa integrare con il nuovo codice o con altri componenti riutilizzati. È richiesto un certo sforzo di sviluppo per il suo riutilizzo, perché questo codice deve essere capito e modificato, prima di poter operare correttamente nel sistema. Il codice a scatola aperta può essere generato automaticamente, ma richiede qualche modifica o aggiunta manuale. In alternativa, può essere formato da componenti riutilizzabili di altri sistemi che devono essere adattati per il sistema che si sta sviluppando.

Tre fattori contribuiscono allo sforzo necessario per riutilizzare i componenti del codice a scatola aperta.

1. Lo sforzo necessario per stabilire se un componente possa essere riutilizzato oppure no nel sistema che si sta sviluppando.
2. Lo sforzo necessario per capire il codice che si vuole riutilizzare.
3. Lo sforzo necessario per modificare il codice riutilizzato per adattarlo e integrarlo nel sistema che si sta sviluppando.

Lo sforzo nel modello di riutilizzo è calcolato utilizzando il modello COCOMO della progettazione iniziale e si basa sul numero totale di righe di codice del sistema. La dimensione del codice include il nuovo codice sviluppato per i componenti che non sono riutilizzati, più un fattore addizionale che tiene conto dello sforzo richiesto per riutilizzare e integrare il codice esistente. Questo fattore addizionale è detto **ESLOC**, il numero equivalente di righe del nuovo codice; ovvero, lo sforzo di riutilizzo viene espresso come lo sforzo che sarebbe richiesto per sviluppare altro codice sorgente.

La formula utilizzata per calcolare l'equivalenza del codice sorgente è:

$$\text{ESLOC} = \text{ASLOC} \times (1 - \text{AT}/100) \times \text{AAM}$$

dove:

ESLOC è il numero equivalente di righe del nuovo codice sorgente;

ASLOC è una stima del numero delle righe di codice nei componenti riutilizzati che devono essere modificate;

AT è la percentuale del codice riutilizzato nei componenti riutilizzati che deve essere modificato;

AAM (Adaptation Adjustment Multiplier) è un fattore che tiene conto dello sforzo aggiuntivo che è richiesto per riutilizzare i componenti.

In alcuni casi, le regolazioni richieste per riutilizzare il codice sono sintattiche e possono essere implementate automaticamente da un apposito strumento software. Queste regolazioni non richiedono uno sforzo significativo, quindi basta stimare quale frazione delle modifiche apportate al codice riutilizzato può essere automatizzata (**AAT**). Questo riduce il numero totale di righe di codice che devono essere adattate.

Il fattore **AAM** affina la stima per tenere conto dello sforzo aggiuntivo che è richiesto per riutilizzare il codice. La documentazione sui modelli COCOMO (Abts et al. 2000) descrive dettagliatamente come deve essere calcolato **AAM**. Semplificando, **AAM** è la somma di tre componenti.

1. Un fattore di valutazione (**AA**) che rappresenta lo sforzo per stabilire se riutilizzare oppure no i componenti. **AA** varia da 0 a 8, a seconda del tempo richiesto per analizzare e scegliere i potenziali candidati al riutilizzo.
2. Un componente di comprensione (**SU**) che rappresenta i costi necessari affinché gli ingegneri comprendano il codice da riutilizzare e familiarizzino con esso. **SU** varia da 50 per un codice complesso non strutturato, a 10 per un codice ben scritto, orientato agli oggetti.
3. Un componente di adattamento (**AAF**) che rappresenta i costi necessari per modificare il codice da riutilizzare; i costi includono le modifiche di progettazione, programmazione e integrazione.

Una volta calcolato un valore di **ESLOC**, si applica la formula standard della stima per calcolare lo sforzo totale richiesto, dove il parametro **Dimensione** = **ESLOC**. Quindi la formula per stimare lo sforzo di riutilizzo è:

$$\text{Sforzo} = A \times \text{ESLOC}^B \times M$$

dove **A**, **B** ed **M** hanno gli stessi valori utilizzati nel modello di progettazione iniziale.

Costi guida del modello COCOMO

I costi guida del modello COCOMO II sono attributi che rispecchiano alcuni fattori del prodotto, del team, del processo e dell'azienda che influiscono sullo sforzo necessario per sviluppare un sistema software. Per esempio, se è richiesto un elevato livello di affidabilità, sarà necessario uno sforzo maggiore; se occorre una rapida consegna, sarà necessario uno sforzo maggiore; se i membri del team di sviluppo cambiano, sarà necessario uno sforzo maggiore.

Il modello COCOMO II ha 17 di questi attributi, ai quali sono stati associati dei valori stimati dagli sviluppatori del modello.

<http://software-engineering-book.com/web/cost-drivers/>

20.6.4 Modello di post-architettura

Il modello di post-architettura è il più dettagliato dei modelli COCOMO II. Viene utilizzato quando è disponibile un progetto architettonico iniziale per il sistema. Il punto di partenza delle stime prodotte al livello di post-architettura è la stessa formula di base utilizzata nelle precedenti stime:

$$PM = A \times Dimensione^B \times M$$

In questa fase del progetto si è in grado di fare delle stime più accurate della dimensione del progetto, perché si sa come sarà scomposto il sistema nei vari componenti e sottosistemi. La stima della dimensione complessiva del codice nel modello di post-architettura si fa sommando tre valori:

1. una stima del numero totale di righe del nuovo codice da sviluppare;
2. una stima dei costi di riutilizzo in base al numero equivalente di righe di codice sorgente (ESLOC), calcolata utilizzando il modello di riutilizzo;
3. una stima del numero delle righe di codice che devono essere modificate a causa delle modifiche apportate ai requisiti.

L'ultimo valore della stima – il numero di righe del codice modificato – riflette il fatto che i requisiti del software cambiano sempre. Questo implica una revisione del codice e lo sviluppo di un codice extra, di cui bisogna tener conto. Ovviamente, ci sarà sempre più incertezza nel calcolo di questo valore che nella stima del nuovo codice da sviluppare.

Il termine esponenziale (**B**) nella formula del calcolo dello sforzo tiene conto del livello di complessità del progetto. Se i progetti diventano più complessi, gli effetti della dimensione crescente del sistema diventano più significativi. Il valore dell'esponente **B** si basa su cinque fattori, come illustra la Figura 20.12. Questi fattori sono classificati su una scala da 0 a 5, dove 0 significa “molto alto” e 5 “molto basso”. Per calcolare **B**, si sommano i fattori di scala, si divide per 100 e si aggiunge 1,01 al risultato.

Fattore di scala	Spiegazione
Architettura/risoluzione dei rischi	Riflette il livello di analisi dei rischi; molto basso significa poca analisi; molto alto significa un'analisi completa e dettagliata dei rischi.
Flessibilità di sviluppo	Riflette il grado di flessibilità nel processo di sviluppo. Molto basso significa che viene utilizzato un processo ben dettagliato; molto alto significa che il cliente impone solo obiettivi generali.
Precedente esperienza	Riflette il livello di esperienza precedente dell'azienda con questo tipo di progetto. Molto basso significa nessuna esperienza precedente; molto alto significa che l'azienda ha una completa familiarità con questo tipo di applicazioni.
Coesione del team	Riflette il grado di conoscenza e affiatamento tra i membri del team di sviluppo. Molto basso significa interazioni molto difficili; molto alto indica un team integrato ed efficiente, senza problemi di comunicazione.
Maturità del processo	Riflette il grado di maturità del processo nell'azienda (descritto nel Capitolo 26 sul Web). Il calcolo di questo valore dipende dal questionario di maturità CMM, ma una stima può essere fatta sottraendo il livello di maturità del processo CMM da 5.

Figura 20.12 Fattori di scala per calcolare l'esponente nel modello di post-architettura.

Per esempio, supponiamo che un'azienda stia lavorando su un progetto in un dominio nel quale ha poca esperienza. Il cliente del progetto non ha definito il processo da utilizzare e non ha concesso troppo tempo per un'approfondita analisi dei rischi. Deve essere approntato un nuovo team di sviluppo per implementare il sistema. L'azienda ha recentemente predisposto un piano di miglioramento del processo ed è stata classificata come un'organizzazione di Livello 2 secondo il modello SEI di valutazione delle capacità (Capitolo 26 sul Web). Sulla base di queste considerazioni è possibile stimare i fattori di scala utilizzati nel calcolo dell'esponente nel modo seguente:

1. *esperienza precedente*, classificata bassa (4). Si tratta di un nuovo progetto per l'azienda;
2. *flessibilità di sviluppo*, classificata molto alta (1). Non c'è coinvolgimento del cliente nel processo di sviluppo, quindi ci sono poche modifiche imposte dall'esterno;
3. *architettura/risoluzione dei rischi*, classificate molto basse (5). Non viene eseguita alcuna analisi dei rischi;
4. *coesione del team*, classificata normale (3). Si tratta di un nuovo team, quindi non si dispone di alcuna informazione sulla coesione;
5. *maturità del processo*, classificata normale (3). È presente un certo controllo del processo.

Valore dell'esponente	1,17
Dimensione del sistema (inclusi i fattori per il riutilizzo e la volatilità dei requisiti)	128 KLOC
Stima COCOMO iniziale senza costi guida	730 mesi-persona
Affidabilità	Molto alta, moltiplicatore = 1,39
Complessità	Molto alta, moltiplicatore = 1,3
Limite di memoria	Alto, moltiplicatore = 1,21
Uso di strumenti	Basso, moltiplicatore = 1,12
Tempistica	Accelerata, moltiplicatore = 1,29
Stima COCOMO affinata	2306 mesi-persona
Affidabilità	Molto bassa, moltiplicatore = 0,75
Complessità	Molto bassa, moltiplicatore = 0,75
Limite di memoria	Nessuno, moltiplicatore = 1
Uso di strumenti	Molto alto, moltiplicatore = 0,72
Tempistica	Normale, moltiplicatore = 1
Stima COCOMO affinata	295 mesi-persona

Figura 20.13 Effetto dei costi guida sulla stima dello sforzo.

La somma di questi valori è 16. Per calcolare il valore finale dell'esponente, si divide questa somma per 100 e si aggiunge 1,01 al risultato; quindi B è pari a 1,17.

La stima dello sforzo complessivo può essere affinata utilizzando un set esteso di 17 attributi, al posto dei 7 attributi del modello di progettazione iniziale. È possibile stimare questi attributi, in quanto adesso si hanno più informazioni a disposizione sul software, sui suoi requisiti non funzionali, sul team e sul processo di sviluppo.

La Figura 20.13 mostra come questi costi guida influenzano le stime dello sforzo. Si suppone che l'esponente sia pari a 1,17, come descritto nel precedente esempio. I costi guida chiave del progetto sono l'affidabilità (RELY), la complessità (CPLX), la memoria (STOR), gli strumenti (TOOL) e la tempistica (SCED). Tutti gli altri costi guida hanno un valore nominale pari a 1, quindi non influiscono sul calcolo dello sforzo.

Nella Figura 20.13 sono stati assegnati i valori massimi e minimi ai costi guida chiave per mostrare come influiscono sulla stima dello sforzo. I valori sono presi dal manuale di riferimento COCOMO II (Abts et al. 2000). Si può vedere che i valori alti dei costi guida portano a una stima dello sforzo che è più di tre volte la stima iniziale, mentre i valori bassi riducono la stima a circa un terzo dell'originale. Questo mette in evidenza la significativa differenza tra i diversi tipi di progetti e la difficoltà di trasferire l'esperienza da un dominio applicativo a un altro.

20.6.5 Durata del progetto e gestione del personale

Oltre a stimare i costi globali di un progetto e lo sforzo richiesto per sviluppare un sistema software, i project manager devono anche stimare quanto tempo richiederà lo sviluppo del software e quando il personale dovrà lavorare sul progetto. Sempre più aziende richiedono tempi di sviluppo più brevi, in modo che i loro prodotti possano essere immessi sul mercato prima dei concorrenti.

Il modello COCOMO include una formula per stimare il tempo richiesto per completare un progetto:

$$TDEV = 3 \times (PM)^{0,33 + 0,2 \times (B - 1,01)}$$

dove:

TDEV è il tempo nominale, espresso in mesi, per sviluppare il progetto, ignorando qualsiasi moltiplicatore che è correlato alla tempistica del progetto;

PM è lo sforzo calcolato dal modello COCOMO;

B è un esponente correlato alla complessità, come descritto nel Paragrafo 20.5.1.

Se **B** = 1,17 e **PM** = 60, allora

$$TDEV = 3 \times (60)^{0,36} = 13 \text{ mesi}$$

La tempistica nominale del progetto prevista dal modello COCOMO non corrisponde necessariamente alla tempistica richiesta dal cliente del software. Potrebbe essere necessario consegnare il software in anticipo o (più raramente) oltre la data indicata nella tempistica nominale. Se la tempistica deve essere compressa (ovvero il software deve essere sviluppato più rapidamente), lo sforzo richiesto per il progetto aumenta. Il moltiplicatore **SCED** tiene conto di questo fatto nel calcolo della stima dello sforzo.

Supponiamo che per un progetto sia stato stimato **TDEV** pari a 13 mesi, come indicato in precedenza, ma la tempistica effettiva richieda 10 mesi. Questo implica una compressione della tempistica di circa il 25%. Utilizzando i valori derivati dal team di Boehm per il moltiplicatore **SCED**, si nota che il moltiplicatore dello sforzo per questo livello di compressione della tempistica è 1,43. Pertanto, lo sforzo reale che sarà richiesto per soddisfare questa tempistica accelerata è di circa il 50% maggiore di quello richiesto per consegnare il software secondo la tempistica nominale.

Esiste una relazione complessa tra il numero di persone che lavorano su un progetto, lo sforzo che sarà dedicato al progetto e la tempistica di consegna del progetto. Se 4 persone possono completare un progetto in 13 mesi (pari a uno sforzo di 52 mesi-persona), allora si potrebbe pensare che, aggiungendo un'altra persona, il lavoro potrà essere completato in 11 mesi (55 mesi-persona). Il modello COCOMO, invece, indica che, in effetti, occorreranno 6 persone per completare il lavoro in 11 mesi (66 mesi-persona).

La ragione di questo è che, aggiungendo persone a un progetto, si riduce la produttività dei membri del team esistente. Quando la dimensione del progetto aumenta, i membri del team impiegano più tempo per comunicare e definire le interfacce tra le parti del sistema sviluppate da altre persone. Raddoppiando il numero delle persone, per esempio, non significa che la durata del progetto si dimezzerà.

Di conseguenza, quando aggiungete una persona extra, l'incremento effettivo dello sforzo aggiunto è minore di quello di una persona, in quanto tutte le altre persone diventano meno produttive. Se il team di sviluppo è grande, aggiungendo altre persone al progetto, in alcuni casi, la durata dello sviluppo potrebbe aumentare, anziché ridursi, a causa dell'effetto complessivo sulla produttività.

Non è possibile stimare il numero delle persone richieste per un team di sviluppo dividendo semplicemente lo sforzo totale per il tempo richiesto dal progetto. Di solito, la fase di avviamento di un progetto richiede poche persone per svolgere i compiti iniziali. Il personale del team subisce un picco durante lo sviluppo e il test del sistema; poi si riduce quando il sistema viene preparato per la consegna. Una crescita molto rapida del personale si verifica quando l'avanzamento del progetto subisce un rallentamento. Un project manager dovrebbe evitare di aggiungere troppe persone nella fase iniziale di avviamento del progetto.

Punti chiave

- Il prezzo richiesto per un sistema software non dipende semplicemente dai suoi costi di sviluppo stimati e dal profitto della società di sviluppo. Fattori aziendali potrebbero giustificare un aumento del prezzo per compensare un incremento dei rischi o una riduzione del prezzo per essere più competitivi sul mercato.
- Il prezzo del software viene spesso stabilito per poter vincere una gara di appalto; le funzionalità del sistema vengono poi regolate in funzione del prezzo stabilito.
- Lo sviluppo guidato da piani è organizzato attorno a un progetto completo che definisce le attività, la tempistica e le persone responsabili per ciascuna attività.
- La tempistica di un progetto richiede la creazione di varie rappresentazioni grafiche di parti del piano del progetto. I diagrammi a barre, che mostrano la durata e la scadenza delle varie attività dei membri del team di sviluppo, sono le rappresentazioni grafiche più comunemente utilizzate.
- Il traguardo di un progetto è il risultato prevedibile di un'attività o di una serie di attività. Per ogni traguardo deve essere preparato un rapporto formale sull'avanzamento dei lavori da presentare ai responsabili della gestione del progetto. Una consegna è il prodotto del team di sviluppo che viene presentato al cliente del progetto.
- La pianificazione agile richiede che tutto il team sia coinvolto nella pianificazione di un progetto. Il piano viene sviluppato in modo incrementale e, se sorgono dei problemi, viene regolato in modo che le funzionalità del software siano ridotte, anziché ritardare la consegna di un incremento del piano.

- Le tecniche di stima dei costi del software possono essere basate sull'esperienza, dove manager esperti sono in grado di valutare lo sforzo richiesto per lo sviluppo, o su formule algoritmiche, mediante le quali è possibile calcolare lo sforzo richiesto in base a opportuni parametri di progetto.
- Il modello COCOMO II è un modello algoritmico collaudato per stimare i costi di sviluppo del software; tiene conto degli attributi del progetto, del prodotto, dell'hardware e del personale.

Esercizi

- * 20.1 In quali circostanze una società potrebbe richiedere per il software sviluppato un prezzo molto più alto di quello derivante dalla stima del costo del software più un normale margine di profitto?
- * 20.2 Spiegate perché il processo di pianificazione dei progetti è iterativo e perché un piano deve essere continuamente rivisto durante lo sviluppo di un progetto.
- 20.3 Spiegate brevemente lo scopo di ciascuna delle sezioni di un piano di sviluppo di un progetto software.
- * 20.4 Le stime dei costi sono intrinsecamente a rischio, indipendentemente dalla tecnica di stima adottata. Indicate quattro modi in cui è possibile ridurre il rischio nella stima dei costi.
- * 20.5 La Figura 20.14 presenta un certo numero di compiti, le loro durate e le loro dipendenze. Disegnate un diagramma a barre che mostra la tempistica del progetto.
- 20.6 La Figura 20.14 mostra la durata dei compiti per le attività di progettazione del software. Supponete che sopraggiunga improvvisamente un grave problema e, anziché 10 giorni, il compito T5 ne richieda 40. Modificate il diagramma a barre per mostrare come il progetto potrebbe essere riorganizzato.
- 20.7 Il planning game si basa sul concetto di pianificazione per implementare le storie che rappresentano i requisiti del sistema. Spiegate i potenziali problemi di questo approccio quando il software ha requisiti di alte prestazioni e fidatezza.
- * 20.8 Il manager di un progetto software ha l'incarico di sviluppare un sistema software a sicurezza critica, che è progettato per controllare una macchina di radioterapia per trattare pazienti affetti da cancro. Questo sistema è integrato nella macchina e deve essere eseguito su un processore speciale con una quantità di memoria fissa (256 MB). La macchina comunica con un apposito database per ottenere i dettagli di ciascun paziente e, dopo il trattamento, registra automaticamente nel database la dose delle radiazioni emesse e altri dettagli.

Per stimare lo sforzo richiesto per sviluppare questo sistema, viene utilizzato il metodo COCOMO; il risultato della stima è di 26 mesi-persona. Tutti i moltiplicatori dei costi guida sono stati impostati a 1 per effettuare la stima.

Spiegate perché questa stima dovrebbe essere modificata per tenere conto dei fattori del progetto, del personale, del prodotto e dell'azienda. Indicate quattro fattori che potrebbero avere significativi effetti sulla stima iniziale del modello COCOMO e suggerite i possibili valori per questi fattori. Spiegate perché avete incluso ciascun fattore.

Compito	Durata (giorni)	Dipendenze
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

Figura 20.14 Esempio di tempistica di un progetto.

- 20.9 Alcuni progetti software molto grandi richiedono la scrittura di milioni di righe di codice. Spiegate perché i modelli di stima dei costi, come COCOMO, potrebbero non operare correttamente se applicati a sistemi molto grandi.
- 20.10 È moralmente corretto che una società abbassi significativamente il prezzo del software per aggiudicarsi una gara di appalto, sapendo che i requisiti sono ambigui e che potrebbero comportare un aumento del prezzo a causa delle successive modifiche richieste del cliente?

* *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Per questo capitolo sono consigliate anche le ulteriori letture del Capitolo 19.

“Ten Unmyths of Project Estimation.” Un articolo pragmatico che descrive le difficoltà pratiche nella stima dei costi di progettazione e critica alcune ipotesi fondamentali in quest’area (P. Armour, Comm. ACM, 45(11), November 2002). <http://dx.doi.org/10.1145/581571.581582>

Agile Estimating and Planning. Questo libro descrive in modo esauriente la pianificazione basata sulle storie come viene applicata nella programmazione estrema; è anche un testo fondamentale per imparare a utilizzare i metodi agili nella pianificazione dei progetti. Il libro spiega molto bene i principali problemi della pianificazione (M. Cohn, 2005, Prentice-Hall).

“Achievements and Challenges in COCOMO-based Software Resource Estimation.” Questo articolo narra la storia dei modelli COCOMO e delle influenze che hanno subito; descrive le varianti che sono state sviluppate per questi modelli. Presenta anche i possibili sviluppi dei modelli COCOMO (B. W. Boehm e R. Valeridi, *IEEE Software*, 25 (5), September/October 2008). <http://dx.doi.org/10.1109/MS.2008.133>

All About Agile; Agile Planning. Questo sito web sui metodi agili include un'eccellente serie di articoli sulla pianificazione agile, scritti da vari autori (2007-2012). <http://www.allaboutagile.com/category/agile-planning/>

Project Management Knowhow: Project Planning. Questo sito web include alcuni utili articoli sulla gestione della progettazione in generale. Sono destinati alle persone che non hanno particolare esperienza in quest'area (P. Stoemmer, 2009-2014). http://www.project-management-knowhow.com/project_planning.html

CAPITOLO

21

Gestione della qualità

L'obiettivo di questo capitolo è presentare la gestione della qualità e le misure del software. Dopo aver letto questo capitolo:

- conoscerete il processo di gestione della qualità e capirete perché è importante la pianificazione della qualità;
- comprenderete l'importanza degli standard nel processo di gestione della qualità e saprete come utilizzare gli standard per garantire la qualità;
- capirete come le revisioni e le ispezioni sono utilizzate come un meccanismo per garantire la qualità del software;
- apprenderete che la gestione della qualità nei metodi agili si basa sulla cultura della qualità nel team di sviluppo;
- capirete come le misure possono essere utili per valutare alcuni attributi della qualità del software e apprenderete il concetto di analitica del software e le limitazioni di queste misure.

- 21.1 Qualità del software
- 21.2 Standard del software
- 21.3 Revisioni e ispezioni
- 21.4 Gestione della qualità e sviluppo agile
- 21.5 Misure del software

La gestione della qualità del software si occupa di garantire che i sistemi software sviluppati siano conformi agli scopi prestabiliti, ovvero che i sistemi soddisfino le esigenze dei loro utenti, siano eseguiti in modo efficiente e affidabile e siano consegnati in tempo ed entro i limiti dei costi previsti nel budget. L'uso di tecniche di gestione della qualità insieme a nuove tecnologie software e metodi di testing ha prodotto significativi miglioramenti nel livello di qualità del software negli ultimi 20 anni.

La gestione della qualità (QM o quality management) formalizzata è particolarmente importante nei team che sviluppano grandi sistemi software di lunga durata, che richiedono parecchi anni di sviluppo. Questi sistemi sono sviluppati per clienti esterni, utilizzando di solito un processo guidato da piani. Per questi sistemi, la gestione della qualità è un problema di natura sia organizzativa sia progettuale.

1. A livello organizzativo, la gestione della qualità si occupa di definire processi e standard che possano portare allo sviluppo di un software di alta qualità. Il team QM dovrebbe avere la responsabilità di definire i processi da utilizzare nello sviluppo del software e gli standard da applicare al software e la relativa documentazione, inclusi i requisiti, il progetto e il codice del sistema.
2. A livello progettuale, la gestione della qualità richiede l'applicazione di specifici processi di qualità, verificando che questi processi pianificati siano seguiti e garantendo che gli output del progetto soddisfino gli standard definiti per il progetto. I responsabili della qualità possono anche definire un piano di qualità per il progetto. Questo piano dovrebbe stabilire gli obiettivi della qualità per il progetto e definire quali processi e standard utilizzare.

Le tecniche di gestione della qualità del software traggono origine dalle tecniche e dai metodi che sono stati sviluppati nelle industrie manifatturiere, dove i termini “garanzia della qualità” e “controllo della qualità” sono largamente utilizzati. La garanzia della qualità è la definizione di processi e standard che dovrebbero portare a prodotti di alta qualità e l'introduzione di processi di qualità nel processo manifatturiero. Il controllo della qualità è l'applicazione di questi processi di qualità per eliminare quei prodotti che non hanno raggiunto il livello qualitativo richiesto. La garanzia della qualità e il controllo della qualità fanno parte entrambi della gestione della qualità.

Nell'industria del software, alcune società vedono la garanzia della qualità come la definizione di procedure, processi e standard per garantire che venga raggiunta la qualità desiderata per il software. In altre società, la garanzia della qualità include anche la gestione della configurazione, la verifica e la convalida delle attività che si applicano dopo che un prodotto è stato consegnato dal team di sviluppo.

La gestione della qualità fornisce un controllo indipendente sui processi di sviluppo del software. Il team QM controlla le consegne del progetto per garanti-

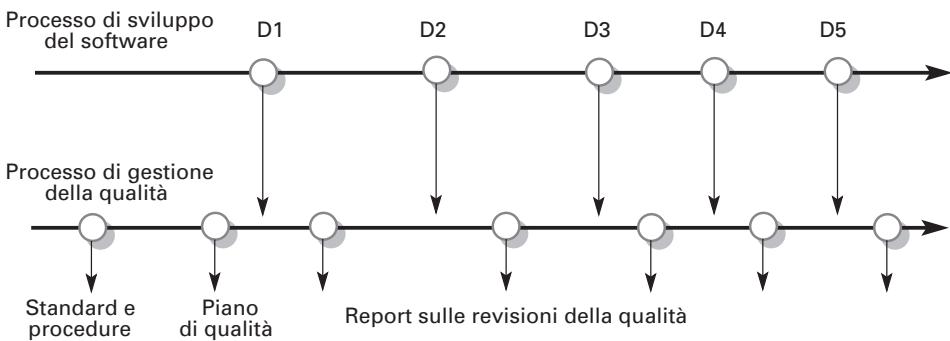


Figura 21.1 Gestione della qualità e sviluppo del software.

re che esse siano coerenti con gli standard e gli obiettivi aziendali (Figura 21.1). Il team QM controlla anche la documentazione dei processi, dove sono registrati i compiti che sono stati completati da ciascun membro del team di sviluppo. Il team QM usa questa documentazione per verificare che non siano stati dimenticati importanti compiti o che un gruppo non abbia fatto ipotesi errate sul lavoro svolto da altri gruppi.

Il team QM di una grande società di solito è responsabile della gestione dei processi di testing delle release. Come detto nel Capitolo 8, questo significa che essi gestiscono i test del software prima che questo sia consegnato ai clienti. In aggiunta, essi hanno la responsabilità di controllare se i test del sistema forniscano una copertura adeguata dei requisiti e che siano mantenute apposite registrazioni dei test effettuati.

Il team QM dovrebbe essere indipendente e non far parte del team di sviluppo, in modo da avere una visione oggettiva della qualità del software. Il team QM può esprimersi sulla qualità del software, senza essere influenzato dai problemi di sviluppo del software. Teoricamente, il team QM dovrebbe avere responsabilità di alto livello sulla gestione della qualità, e dovrebbe informare i manager di livello superiore a quello del project manager.

Poiché i project manager devono rispettare il budget e la tempistica dei progetti, potrebbero essere tentati di compromettere la qualità del prodotto pur di rispettare la tempistica. Un team QM indipendente assicura che gli obiettivi aziendali della qualità non siano influenzati da considerazioni di breve termine sul budget e sulla tempistica. Nelle società più piccole, invece, tutto questo è praticamente impossibile. La gestione della qualità e lo sviluppo del software sono inevitabilmente intrecciati, con persone che hanno contemporaneamente responsabilità di sviluppo del software e di controllo della qualità.

La pianificazione formale della qualità è parte integrante dei processi di sviluppo guidato da piani. È il processo che sviluppa un piano di qualità per un progetto. Il piano di qualità dovrebbe definire le qualità desiderate per il software e descrivere come queste qualità possano essere valutate; definisce che cosa signi-

fica veramente software di “alta qualità” per un particolare sistema. In questo modo, gli ingegneri hanno un’idea comune sui più importanti attributi della qualità del software.

Humphrey (Humphrey 1989), nel suo classico libro sulla gestione del software, ha indicato una struttura guida per un piano di qualità. Questa struttura include le seguenti parti.

1. *informazioni sul prodotto*: una descrizione del prodotto, del mercato a cui è rivolto e le aspettative di qualità;
2. *piani del prodotto*: le date critiche di rilascio e le responsabilità per il prodotto, insieme con i piani per la distribuzione e l’assistenza del prodotto;
3. *descrizioni del processo*: i processi di sviluppo e di assistenza che dovrebbero essere utilizzati per lo sviluppo e la gestione del prodotto;
4. *obiettivi di qualità*: gli obiettivi e i piani di qualità del prodotto, inclusa l’identificazione degli attributi critici della qualità del prodotto;
5. *rischi e gestione dei rischi*: i rischi chiave che possono influire sulla qualità del prodotto e le azioni per evitarli.

I piani di qualità, che sono sviluppati come parte del processo di pianificazione generale di un progetto, differiscono nei dettagli, a seconda della dimensione e del tipo di sistema che si sta sviluppando. Tuttavia, quando si scrive un piano di qualità, occorre essere molto concisi. Se il documento è troppo lungo, le persone non lo leggono, facendo fallire l’obiettivo del piano di qualità.

La gestione della qualità tradizionale è un processo formale che si basa su un’ampia documentazione sui test e sulla convalida dei sistemi e su come i processi devono essere eseguiti. A questo proposito, essa è diametralmente opposta allo sviluppo agile, dove l’obiettivo è impiegare il minor tempo possibile per scrivere documenti e formalizzare le modalità di sviluppo del software. Nel Paragrafo 21.4 descriverò la gestione della qualità e lo sviluppo agile.

21.1 Qualità del software

L’industria manifatturiera ha stabilito i fondamenti della gestione della qualità con l’obiettivo di migliorare la qualità dei prodotti. Come parte di questo impegno, l’industria ha elaborato il concetto di qualità, che si basa sulla conformità dei prodotti a una determinata specifica. L’ipotesi sottostante è che per ogni prodotto è possibile definire sia una specifica dettagliata sia le procedure per controllare la conformità di un prodotto alla sua specifica. Ovviamente, i prodotti non potranno soddisfare esattamente una specifica, quindi è ammessa qualche tolleranza. Se un prodotto è “quasi conforme” alla specifica, esso deve essere classificato come accettabile.

La qualità del software non è direttamente paragonabile con la qualità dell’industria manifatturiera. Il concetto di tolleranza è applicabile a prodotti simili, non

al software. Inoltre, spesso è impossibile arrivare a un giudizio oggettivo sul fatto che un sistema software soddisfi oppure no la sua specifica.

1. È difficile scrivere specifiche del software chiare e complete. Gli sviluppatori del software e i clienti potrebbero interpretare i requisiti in vari modi, e quindi potrebbe essere impossibile raggiungere un accordo sulla conformità di un sistema software alla sua specifica.
2. Le specifiche, di solito, integrano i requisiti definiti da varie classi di stakeholder. Questi requisiti sono inevitabilmente un compromesso e potrebbero non includere i requisiti di tutti i gruppi di stakeholder. Di conseguenza, gli stakeholder esclusi potrebbero percepire il sistema come un sistema di bassa qualità, anche se esso implementa i requisiti concordati.
3. È impossibile misurare direttamente alcune caratteristiche della qualità (per esempio, la manutenibilità); quindi, non è possibile specificare queste caratteristiche in modo chiaro. Le difficoltà nel misurare alcune caratteristiche della qualità sono descritte nel Paragrafo 21.4.

A causa di questi problemi il giudizio sulla qualità del software è un processo soggettivo. Il team di gestione della qualità usa i suoi criteri per decidere se è stato raggiunto un livello di qualità accettabile. Essi decidono se il software soddisfa oppure no i requisiti richiesti. La decisione richiede che si risponda ad alcune domande sulle caratteristiche del sistema, come quelle qui elencate.

1. Il software è stato opportunamente provato? Sono stati implementati tutti i requisiti?
2. Il software è sufficientemente affidabile da poter essere utilizzato?
3. Le prestazioni del software sono accettabili per il normale utilizzo?
4. Il software è utilizzabile?
5. Il software è ben strutturato e comprensibile?
6. Sono stati seguiti gli standard di programmazione e documentazione durante il processo di sviluppo?

Un presupposto fondamentale nella gestione della qualità è che un sistema software dovrà essere provato per verificare che soddisfi i suoi requisiti. Il giudizio sul fatto che il software svolga le funzionalità richieste si deve basare sui risultati di questi test. Quindi, il team QM dovrebbe rivedere i test che sono stati sviluppati ed esaminare le loro registrazioni per verificare che siano stati svolti correttamente. In alcune società, il team di gestione della qualità svolge il test finale del sistema; in altre, un apposito team documenta l'esito dei test e lo presenta al responsabile della qualità del sistema.

La soggettività della qualità di un sistema software dipende principalmente dalle sue caratteristiche non funzionali. Ciò rispecchia l'esperienza pratica degli utenti – se una funzionalità del software non è quella attesa, gli utenti spesso aggirano tale difetto e trovano altri modi di fare ciò che vorrebbero fare. Se, inve-

Sicurezza	Comprensibilità	Portabilità	Protezione	Testabilità
Utilizzabilità	Affidabilità	Adattabilità	Riutilizzabilità	Resilienza
Modularità	Efficienza	Robustezza	Complessità	Facilità di apprendimento

Figura 21.2 Attributi di qualità del software.

ce, il software è inaffidabile o troppo lento, è praticamente impossibile per loro raggiungere i loro obiettivi.

La gestione della qualità, dunque, non consiste semplicemente nel verificare che le funzionalità del software siano state implementate, ma nel controllare anche che il software abbia le necessarie caratteristiche non funzionali, riportate nella Figura 21.2. Queste caratteristiche riflettono la fidatezza, l'utilizzabilità, l'efficienza e la manutenibilità del software.

Non è possibile ottimizzare qualsiasi sistema per tutti questi attributi di qualità. Per esempio, un miglioramento della protezione potrebbe portare a una riduzione delle prestazioni. Il piano di qualità dovrebbe definire gli attributi di qualità più importanti per il software che si sta sviluppando. Se l'efficienza è il fattore più importante, allora per raggiungerla si possono sacrificare gli altri fattori. Se il piano di qualità mette in evidenza l'importanza dell'efficienza, gli ingegneri che sviluppano il software possono cooperare per raggiungere tale obiettivo. Il piano dovrebbe anche includere una definizione del processo di valutazione della qualità. Questo processo dovrebbe essere un modo concordato per stabilire se qualche attributo di qualità, per esempio la portabilità o la robustezza, è presente nel prodotto.

La gestione tradizionale della qualità del software si basa sull'ipotesi che la qualità del software sia direttamente correlata alla qualità del processo di sviluppo del software. Questa ipotesi deriva dai sistemi manifatturieri, dove la qualità dei prodotti è intimamente correlata al processo produttivo. Un processo manifatturiero si basa sulla configurazione, l'impostazione e il funzionamento delle macchine coinvolte nel processo. Una volta che le macchine funzionano correttamente, la qualità del prodotto viene automaticamente raggiunta. È possibile misurare la qualità del prodotto e modificare il processo finché non sarà raggiunto il livello qualitativo richiesto. La Figura 21.3 illustra questo approccio per ottenere la qualità di un prodotto.

C'è un chiaro collegamento tra il processo e la qualità del prodotto nell'industria manifatturiera, in quanto il processo è relativamente facile da standardizzare e monitorare. Una volta calibrati i sistemi di produzione, essi possono continuare a produrre prodotti di alta qualità. Il software, invece, è progettato diversamente dai prodotti manifatturieri, e la relazione tra qualità del processo e qualità del prodotto è più complessa. La progettazione del software è un processo creativo, quindi l'influenza delle capacità e delle esperienze delle persone è significativa.

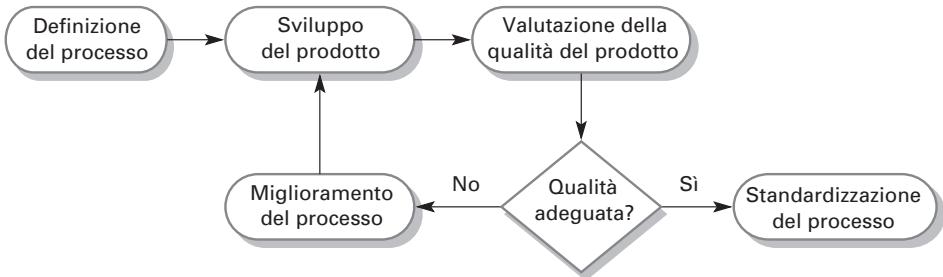


Figura 21.3 La qualità basata sul processo.

Fattori esterni, come la novità di un'applicazione o le pressioni commerciali per anticipare una release, influiscono anch'essi sulla qualità del prodotto, indipendentemente dal processo utilizzato.

Indubbiamente, il processo di sviluppo utilizzato ha una significativa influenza sulla qualità del software, ed è più probabile che i buoni processi portino a un software di buona qualità. La gestione e il miglioramento della qualità dei processi può ridurre il numero di difetti nel software che si sta sviluppando. Tuttavia, è difficile valutare gli attributi di qualità, come l'affidabilità e la manutenibilità, senza utilizzare il software per un lungo periodo di tempo. Di conseguenza, è difficile stabilire come le caratteristiche del processo possano influire su questi attributi. Inoltre, a causa del ruolo svolto dalla creatività nella progettazione di un processo software, la standardizzazione del processo in alcuni casi potrebbe soffocare la creatività, con conseguente impoverimento, anziché arricchimento, della qualità del software.

Definire i processi è importante, ma i responsabili della qualità dovrebbero anche cercare di sviluppare una “cultura della qualità”: chiunque abbia una responsabilità nello sviluppo del software dovrà impegnarsi a raggiungere un alto livello di qualità per il prodotto che sta sviluppando. I team dovrebbero essere incoraggiati ad assumersi le responsabilità della qualità del loro lavoro. I responsabili della qualità dovrebbero sviluppare nuovi approcci al miglioramento della qualità. Sebbene standard e procedure siano la base della gestione della qualità, i manager della buona qualità sanno che ci sono aspetti intangibili nella qualità del software (per esempio, l'eleganza, la chiarezza) che non possono essere integrati negli standard. Questi manager dovrebbero supportare le persone che sono interessate agli aspetti intangibili della qualità e incoraggiare il comportamento professionale in tutti i membri dei team.

21.2 Standard del software

Gli standard del software svolgono un ruolo importante nella gestione della qualità del software. Come detto in precedenza, una parte importante della garanzia della qualità è la definizione o selezione degli standard da applicare al processo

Documentazione standard

I documenti di un progetto sono un modo tangibile di descrivere le varie rappresentazioni di un sistema software (requisiti, UML, codice ecc.) e il suo processo produttivo. Gli standard sulla documentazione definiscono l'organizzazione dei vari tipi di documenti e dei loro formati. Questi standard sono importanti perché semplificano le verifiche relative all'omissione di materiale indispensabile nella documentazione e assicurano che tutti i documenti del progetto abbiano un aspetto e una struttura comuni. Gli standard possono riguardare i contenuti, le modalità di scrittura e lo scambio dei documenti.

<http://software-engineering-book.com/web/documentation-standards/>

di sviluppo del software o a un prodotto software. Come parte di questo processo, possono essere scelti anche gli strumenti e i metodi che supportano l'uso di questi standard. Una volta che gli standard sono stati scelti, devono essere definiti i processi specifici dei progetti per monitorare l'uso degli standard e controllare che essi siano applicati.

Gli standard del software sono importanti per tre motivi.

1. Gli standard racchiudono la conoscenza di ciò che è più prezioso e appropriato per una società. Questa conoscenza spesso è acquisita soltanto dopo una lunga serie di prove ed errori; includerla in uno standard aiuta la società a sfruttare questa esperienza, evitando gli errori commessi in precedenza.
2. Gli standard forniscono un ambiente idoneo per definire il significato di qualità in un particolare contesto. Come detto in precedenza, la qualità del software è soggettiva e, tramite gli standard, è possibile stabilire una base comune per stabilire se è stato raggiunto il livello qualitativo desiderato. Ovviamente, ciò dipende dalla scelta degli standard che rispecchiano le aspettative dell'utente in termini di fidatezza, utilizzabilità e prestazioni del software.
3. Gli standard favoriscono la continuità quando il lavoro svolto da una persona viene continuato da un'altra persona. Gli standard assicurano che tutti gli ingegneri di una società adottino le stesse tecniche. Di conseguenza, si riduce lo sforzo di apprendimento richiesto quando si avvia un nuovo lavoro.

Due tipi correlati di standard per l'ingegneria del software possono essere definiti e sviluppati nella gestione della qualità del software.

1. *Standard di prodotto* – Si applicano al prodotto software che si sta sviluppando. Includono gli standard per i documenti, come la struttura del documento dei requisiti, gli standard per la documentazione, come le intestazioni dei commenti nella definizione delle classi di oggetti, e gli standard per la codifica, che definiscono come utilizzare un linguaggio di programmazione.

Standard di prodotto	Standard di processo
Modulo di revisione del progetto	Procedura di revisione del progetto
Struttura del documento dei requisiti	Invio di nuovo codice per la costruzione del sistema
Formato delle intestazioni dei metodi	Processo di release delle versioni
Stile di programmazione Java	Processo di approvazione del piano del progetto
Formato del piano del progetto	Processo di controllo delle modifiche
Modulo di richiesta delle modifiche	Processo di registrazione dei test

Figura 21.4 Standard di prodotto e di processo.

2. *Standard di processo* – Definiscono i processi da seguire durante lo sviluppo del software. Specificano le buone pratiche da seguire nel processo di sviluppo. Possono includere le definizioni dei processi di specifica, progettazione e convalida del software, la descrizione degli strumenti di supporto e dei documenti che dovrebbero essere scritti durante l'esecuzione di questi processi.

La Figura 21.4 elenca alcuni esempi di standard di prodotto e di processo.

Gli standard devono fornire valore, nella forma di una maggiore qualità del prodotto. Non ha senso definire standard, che sono costosi in termini di tempo e di impegno, che portino miglioramenti solo marginali della qualità. Gli standard di prodotto devono essere progettati in modo da poter essere applicati e controllati in maniera efficiente. Gli standard di processo dovrebbero includere la definizione dei processi che consentono di verificare se gli standard di prodotto sono stati applicati.

Gli standard dell'ingegneria del software che sono utilizzati all'interno di una società, di solito, vengono ricavati da standard nazionali e internazionali più generali. Sono stati sviluppati standard nazionali e internazionali che includono la terminologia dell'ingegneria del software, linguaggi di programmazione come Java e C++, notazioni come i simboli dei diagrammi, procedure per derivare e scrivere i requisiti del software, procedure di garanzia della qualità e processi di verifica e convalida del software (IEEE 2003). Sono stati sviluppati standard più specifici per la sicurezza e la protezione dei sistemi critici.

Gli ingegneri del software a volte considerano gli standard come ostacoli burocratici e strumenti irrilevanti per l'attività tecnica di sviluppo del software. Questa sensazione è ancora più forte quando gli standard richiedono tediose documentazioni con la registrazione delle attività svolte. Di solito, gli ingegneri sono concordi nell'accettare degli standard generali; spesso però trovano delle buone ragioni per giudicare gli standard poco appropriati al loro particolare progetto. I manager della qualità, che definiscono gli standard, dovrebbero prendere delle iniziative appropriate per far capire agli ingegneri l'importanza degli standard.

1. *Coinvolgere gli ingegneri del software nella scelta degli standard di prodotto* – Se gli sviluppatori capiscono perché gli standard sono stati scelti, è più probabile che essi si impegnino a rispettarli. Teoricamente, il documento degli standard non dovrebbe indicare semplicemente gli standard da seguire, ma dovrebbe spiegare anche i motivi delle decisioni di standardizzazione.
2. *Rivedere e modificare gli standard regolarmente per adeguarli ai cambiamenti tecnologici* – Gli standard sono costosi da sviluppare, per questo rischiano di restare “rinchiusi” nel manuale aziendale degli standard. Il management spesso è riluttante a modificarli, a causa delle discussioni e dei costi richiesti. Il manuale degli standard è essenziale, ma dovrebbe evolversi per riflettere i cambiamenti ambientali e le innovazioni tecnologiche.
3. *Verificare che siano disponibili gli strumenti software per supportare lo sviluppo basato sugli standard*. Gli sviluppatori spesso trovano gli standard fastidiosi, quando la loro applicazione richiede un lavoro manuale noioso, che potrebbe essere svolto da uno strumento software. Se è disponibile uno strumento di supporto, gli standard possono essere applicati con un piccolo sforzo aggiuntivo. Per esempio, gli standard del layout di un programma possono essere definiti e implementati da un sistema di editing dei programmi orientato alla sintassi.

Tipi di software differenti richiedono processi di sviluppo differenti, quindi gli standard devono essere facilmente adattabili. Non ha senso prescrivere un particolare modo di lavorare, se è inappropriato a un progetto o a un team di sviluppo. Ogni project manager dovrebbe avere l'autorità di modificare gli standard di processo in base alle singole situazioni. Quando vengono apportate le modifiche, è importante verificare che esse non portino a un abbassamento del livello qualitativo del prodotto.

I responsabili del progetto e i responsabili della qualità possono evitare il problema della inadeguatezza degli standard pianificando accuratamente la qualità all'avviamento del progetto. Devono decidere quali standard aziendali utilizzare senza modifiche, quali standard modificare e quali ignorare. Possono creare nuovi standard in risposta a particolari richieste da parte del cliente o del team di progettazione. Per esempio, potrebbero essere necessari nuovi standard per le specifiche formali, se questi standard non sono stati utilizzati nei progetti precedenti.

21.2.1 Standard ISO 9001

Con il nome ISO 9000 si identificano tutti gli standard internazionali utilizzati nello sviluppo di sistemi per la gestione della qualità in tutti i tipi di industrie. Questi standard possono essere applicati a varie aziende, da quelle manifatturiere e quelle dei servizi. Lo standard ISO 9001, il più generale di questi standard, si applica alle aziende che progettano, sviluppano e mantengono prodotti, incluso il software. Lo standard ISO 9001 fu sviluppato originariamente nel 1987. Qui

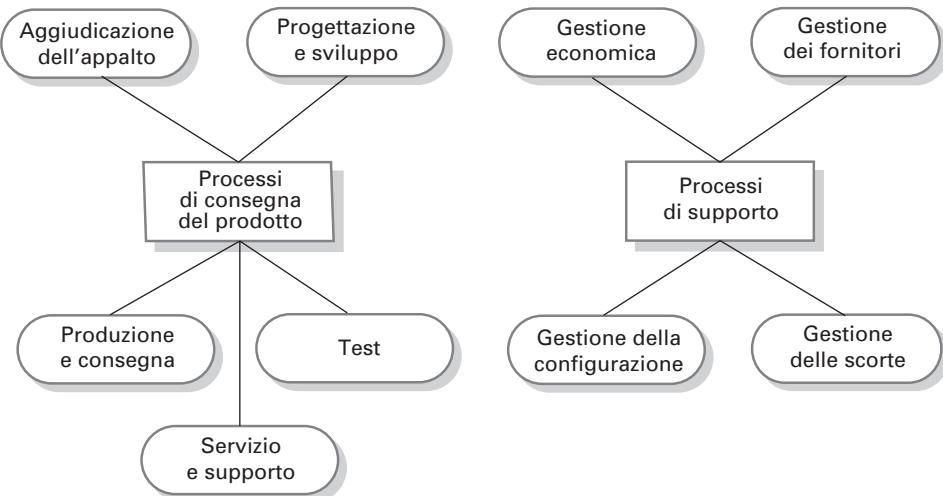


Figura 21.5 Processi fondamentali dell'ISO 9001.

descriverò la versione del 2008 di questo standard, ma lo standard potrebbe essere stato modificato nel 2015, anno in cui è programmata una nuova versione.

Lo standard ISO 9001 non è uno standard per lo sviluppo del software, ma bensì è un contesto per sviluppare gli standard per il software. Definisce i principi generali della qualità, descrive i processi di qualità ed espone le norme e le procedure organizzative che dovrebbero essere definite. Tutto questo dovrebbe essere documentato in un manuale di qualità di ciascuna azienda.

Una revisione importante dello standard ISO 9001 nel 2001 definì nove processi fondamentali (Figura 21.5). Se un'azienda vuole adottare lo standard ISO 9001, deve documentare come i suoi processi siano correlati a questi processi fondamentali; inoltre, deve definire e mantenere le registrazioni che dimostrano che i processi aziendali definiti sono stati seguiti. Il manuale di qualità dell'azienda dovrebbe descrivere i processi principali e i dati dei processi che devono essere raccolti e mantenuti.

Lo standard ISO 9001 non definisce né prescrive specifici processi di qualità che devono essere adottati da un'azienda. Per essere conformi a questo standard, un'azienda deve definire i tipi di processi illustrati nella Figura 21.5 e avere le procedure che dimostrano che i suoi processi di qualità sono seguiti. Questo significa flessibilità tra i vari settori industriali e tra aziende di differenti dimensioni.

Gli standard di qualità possono essere definiti appropriati per il tipo di software che si sta sviluppando. Le piccole aziende possono avere processi semplici, senza molti documenti, e rimanere conformi all'ISO 9001. Tuttavia, questa flessibilità significa che non è possibile fare ipotesi sulle somiglianze o differenze tra i processi di aziende diverse che sono conformi all'ISO 9001. Alcune aziende potrebbero avere processi di qualità molto rigidi con registrazioni dettagliate, mentre altre potrebbero essere meno formali, con livelli minimi di documentazione.

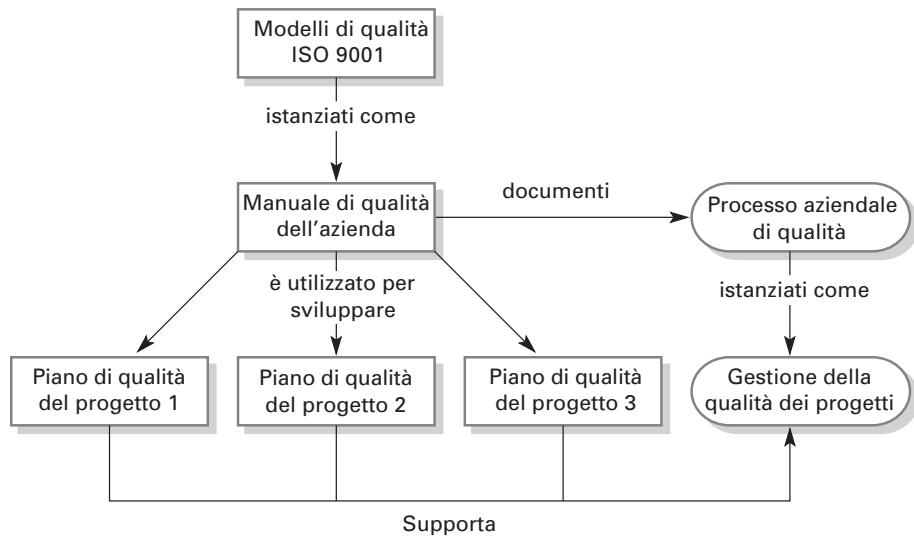


Figura 21.6 ISO 9001 e gestione della qualità.

Le relazioni tra ISO 9001, il manuale di qualità di un’azienda e i piani di qualità dei singoli progetti sono illustrate nella Figura 21.6. Questo diagramma è stato ricavato da un modello ideato da Ince (Ince 1994), che spiega come lo standard generale ISO 9001 può essere utilizzato come base per i processi di gestione della qualità del software. Bamford e Deibler (Bamford e Deibler 2003) spiegano come il successivo standard ISO 9001: 2000 può essere applicato alle società di software.

Alcuni clienti di sistemi software richiedono che i loro fornitori abbiano la certificazione ISO 9001. I clienti hanno così la garanzia che la società che sviluppa il software adotta un sistema di gestione della qualità approvato. Enti indipendenti di certificazione esaminano i processi di gestione della qualità e la documentazione dei processi e decidono se questi processi ricoprono tutte le aree specificate nell’ISO 9001. In caso affermativo, certificano che i processi di qualità della società, così come sono definiti nel manuale di qualità, sono conformi allo standard ISO 9001.

Alcuni credono erroneamente che la certificazione ISO 9001 significhi che la qualità del software prodotto dalle società certificate sia sempre migliore di quella delle società non certificate. L’obiettivo principale dello standard ISO 9001 è garantire che una società abbia definito le procedure di gestione della qualità e che queste procedure siano applicate. Non c’è alcuna garanzia che le società certificate ISO 9001 utilizzino le migliori pratiche di sviluppo del software o che i loro processi generino un software di alta qualità.

La certificazione ISO 9001, secondo il mio punto di vista, non è adeguata, in quanto definisce la qualità come conformità agli standard. Non tiene conto della qualità come viene avvertita dagli utenti del software. Per esempio, una società

potrebbe definire lo standard per eseguire i test specificando che tutti i metodi negli oggetti devono essere chiamati almeno una volta. Questo standard, purtroppo, potrebbe essere soddisfatto da un test del software incompleto che non include i test con altri parametri dei metodi. Se le procedure dei test sono eseguite e vengono mantenute le registrazioni dei test, la società può essere certificata ISO 9001.

21.3 Revisioni e ispezioni

Le revisioni e le ispezioni sono attività della garanzia della qualità che controllano la qualità delle consegne dei prodotti. Questo richiede il controllo del software, la sua documentazione e la registrazione dei processi per scoprire errori e omissioni, come pure le violazioni degli standard. Come detto nel Capitolo 8, le revisioni e le ispezioni sono utilizzate durante i test dei programmi come parte del processo generale di verifica e convalida del software.

Durante una revisione, molte persone esaminano il software e la documentazione associata, cercando potenziali problemi e discordanze con gli standard. Il team di revisione esprime dei giudizi sul livello di qualità del software o sulla documentazione del progetto. I project manager possono servirsi di questi giudizi nelle loro decisioni di pianificazione e di allocazione delle risorse al processo di sviluppo.

Le revisioni della qualità si basano sui documenti che sono stati prodotti durante il processo di sviluppo del software. Oltre alle specifiche del software, anche i progetti, il codice, i modelli, i piani di test, le procedure di gestione della configurazione, gli standard dei processi e i manuali degli utenti dovrebbero essere rivisti. La revisione dovrebbe controllare la coerenza e la completezza dei documenti e del codice e, se sono stati definiti gli standard di qualità, verificare che questi standard siano stati applicati.

Le revisioni non servono solo a verificare la conformità agli standard; servono anche a scoprire problemi e omissioni nella documentazione del software e dei progetti. Le conclusioni delle revisioni dovrebbero essere registrate formalmente come parte del processo di gestione della qualità. Se sono stati identificati dei problemi, i commenti dei revisori dovrebbero essere passati all'autore del software o a chiunque abbia la responsabilità di correggere gli errori o le omissioni.

Lo scopo delle revisioni e delle ispezioni è migliorare la qualità del software, non giudicare l'operato delle persone del team di sviluppo. La revisione è un processo pubblico di identificazione degli errori, mentre il processo di test dei componenti è più riservato. Inevitabilmente, gli errori commessi dai singoli vengono rivelati a tutti i membri del team di programmazione. Per garantire che tutti gli sviluppatori collaborino costruttivamente con il processo di revisione, i project manager dovrebbero essere sensibili alle preoccupazioni dei singoli membri del team; dovrebbero alimentare una cultura lavorativa che offre un supporto alla collaborazione, senza incolpare nessuno quando si scopre un errore.

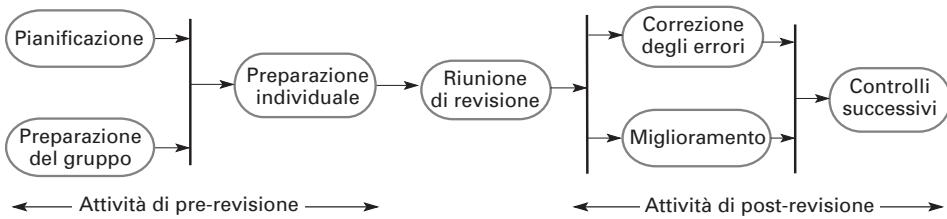


Figura 21.7 Il processo di revisione del software.

Le revisioni della qualità non sono revisioni dell'avanzamento della gestione, sebbene le informazioni sulla qualità del software possano essere utilizzate nel processo decisionale della gestione della qualità. Le revisioni dell'avanzamento confrontano il progresso effettivo di un progetto software con il progresso pianificato. Il loro primo obiettivo è stabilire se il progetto sarà in grado oppure no di consegnare un software utilizzabile in tempo ed entro il budget. Le revisioni dell'avanzamento tengono conto di fattori esterni, e i cambiamenti dell'ambiente operativo potrebbero significare che il software che si sta sviluppando non è più richiesto o deve essere radicalmente modificato. I progetti che hanno prodotto un software di alta qualità potrebbero essere annullati, in seguito a nuovi orientamenti dell'azienda o a modifiche del suo ambiente operativo.

21.3.1 Il processo di revisione

Sebbene ci siano molte varianti nei dettagli delle revisioni, i processi di revisione (Figura 21.7) possono essere strutturati in tre fasi.

1. *Attività di pre-revisione* – Sono le attività preparatorie che sono essenziali per la riuscita della revisione. Tipicamente, queste attività riguardano la pianificazione e la preparazione della revisione. La pianificazione della revisione consiste nella scelta del team di revisione, nella definizione dei tempi e dei luoghi della revisione e nella distribuzione dei documenti da rivedere. Durante la preparazione della revisione, i membri del team potrebbero incontrarsi per fare una panoramica sul software da rivedere. I membri del team di revisione leggono e capiscono il software, la documentazione e gli standard; lavorano in modo autonomo per trovare errori, omissioni e discordanze dagli standard. I revisori possono fornire commenti scritti sul software se non possono partecipare alle riunioni di revisione.
2. *Riunione di revisione* – Durante la riunione di revisione, un autore del documento o del programma oggetto della revisione dovrebbe “attraversare” il documento con il team di revisione. La revisione stessa dovrebbe essere relativamente breve – due ore al massimo. Un membro del team dovrebbe presiedere la revisione, e un altro dovrebbe registrare formalmente tutte le decisioni di revisione e le azioni da svolgere. Durante la revisione, il mode-

Ruoli nel processo di ispezione

Quando l'IBM adottò l'ispezione dei programmi (Fagan 1986), furono definiti alcuni ruoli formali per i membri del team di ispezione, tra cui il moderatore, il lettore del codice e lo scrittore. Altri utenti delle ispezioni hanno modificato questi ruoli, ma è generalmente riconosciuto che un'ispezione dovrebbe coinvolgere l'autore del codice, un ispettore e uno scrittore, e dovrebbe essere presieduta da un moderatore.

<http://software-engineering-book.com/web/qm-roles>

ratore ha la responsabilità di garantire che tutti i commenti presentati siano presi in considerazione. Il moderatore dovrebbe firmare le registrazioni dei commenti e delle azioni concordate durante la riunione.

3. *Attività di post-revisione* – I problemi che sono emersi durante la riunione di revisione devono essere risolti. Le azioni concordate nella riunione possono riguardare la correzione di bug nel codice, la modifica del software in modo che esso sia conforme agli standard qualitativi o la riscrittura dei documenti. A volte, i problemi scoperti durante la revisione della qualità sono tali che è necessaria anche una revisione della gestione per decidere se servono nuove risorse per correggere tali problemi. Dopo che sono state apportate le modifiche, il moderatore dovrebbe verificare che i commenti di revisione siano stati presi tutti in considerazione. In alcuni casi, potrebbe essere necessaria un'ulteriore revisione per verificare che le modifiche fatte riguardino tutti i precedenti commenti di revisione.

I team di revisione di solito sono formati da un nucleo di tre o quattro persone che sono selezionate come revisori principali. Un membro del team dovrebbe essere un progettista esperto che si assume la responsabilità di prendere decisioni tecniche significative. I revisori principali possono invitare altri membri del team di progettazione, come i progettisti di sottosistemi, in modo che possano apportare il loro contributo al processo di revisione; essi non dovrebbero essere coinvolti nella revisione dell'intero documento, ma dovrebbero concentrarsi soltanto su quelle parti che interessano il loro lavoro. In alternativa, il team di revisione potrebbe far circolare il documento e chiedere dei commenti scritti a un numero significativo di membri del team di progettazione. Non è necessario coinvolgere il project manager nella revisione, a meno che i problemi che emergono non richiedano la modifica del piano di progettazione.

I processi indicati per la revisione presuppongono che i membri del team di revisione abbiano un incontro faccia a faccia per discutere il software o i documenti che stanno rivedendo. I team di progettazione oggi sono spesso distribuiti, a volte su regioni o continenti diversi, quindi è impossibile incontrare faccia a faccia i membri di questi team. È possibile supportare le revisioni a distanza, utilizzando documenti in condivisione, dove ogni membro del team di revisione può annotare i propri commenti. Gli incontri faccia a faccia potrebbero essere impossibili a

causa degli impegni di lavoro delle persone o per il fatto che le persone lavorano in luoghi con fusi orari differenti. Il moderatore ha la responsabilità di coordinare i commenti e di discutere le modifiche singolarmente con i membri del team di revisione.

21.3.2 Ispezioni dei programmi

Le ispezioni dei programmi sono revisioni nelle quali i membri del team collaborano per trovare i bug nel codice che stanno sviluppando. Come detto nel Capitolo 8, le ispezioni possono essere parte dei processi di verifica e convalida. Sono complementari ai test, in quanto non richiedono che il programma sia eseguito. Possono essere verificate versioni incomplete del sistema, e possono essere controllate rappresentazioni del sistema, come i modelli UML. I test dei programmi possono essere oggetto di revisione. Le revisioni dei test spesso permettono di scoprire problemi con i test e, quindi, migliorano la loro efficienza nel rilevare i bug del codice.

Le ispezioni dei programmi coinvolgono le diverse esperienze dei membri del team che effettuano accurate revisioni, riga per riga, del codice sorgente dei programmi. Essi ricercano difetti e problemi nel codice e li descrivono durante le riunioni di ispezione. I difetti possono essere errori logici, anomalie nel codice che potrebbero indicare condizioni erronee o caratteristiche che sono state omesse dal codice. Il team di revisione esamina dettagliatamente i modelli del progetto o il codice del programma e mette in evidenza le anomalie e i problemi da risolvere.

Durante un’ispezione, spesso viene utilizzata una lista di controllo con i tipici errori di programmazione per facilitare la ricerca dei bug nel codice. Questa lista si basa su esempi ricavati da libri o sulla conoscenza dei difetti che sono comuni in un particolare dominio di applicazioni. È possibile utilizzare varie liste di controllo per diversi linguaggi di programmazione, in quanto ciascun linguaggio presenta i suoi errori caratteristici. Humphrey (Humphrey 1989), durante una presentazione delle ispezioni, ha fornito un certo numero di liste di controllo.

I controlli che potrebbero essere fatti durante il processo di ispezione sono illustrati nella Figura 21.8. Le società dovrebbero sviluppare le proprie liste di controllo in base agli standard e alle pratiche locali. Queste liste dovrebbero essere aggiornate periodicamente, quando vengono scoperti nuovi difetti. Gli elementi di una lista di controllo variano in funzione del linguaggio di programmazione, in quanto durante la compilazione sono possibili diversi livelli di controlli a seconda del linguaggio utilizzato. Per esempio, un compilatore Java controlla che le funzioni abbiano il numero corretto di parametri, mentre un compilatore C non effettua questo controllo.

Le società che usano le ispezioni hanno scoperto che esse sono molto efficienti nell’individuare i bug nel codice. In una vecchia ricerca, Fagan (Fagan 1986) ha riportato che oltre il 60% degli errori in un programma sono stati identificati utilizzando ispezioni informali del codice. McConnell (McConnell 2004) ha con-

Classe di errore	Controllo di ispezione
Errori nei dati	<ul style="list-style-type: none"> ■ Tutte le variabili del programma sono inizializzate prima che i loro valori siano utilizzati? ■ Tutte le costanti hanno avuto un nome? ■ Il limite superiore di ogni array deve essere uguale alla dimensione dell'array o alla Dimensione – 1? ■ Se si utilizzano stringhe di caratteri, viene assegnato esplicitamente un delimitatore? ■ Ci sono rischi di overflow dei buffer?
Errori di controllo	<ul style="list-style-type: none"> ■ Per ogni istruzione condizionale, la condizione è corretta? ■ È sicuro che ogni ciclo sarà ultimato? ■ Le istruzioni composte sono messe correttamente fra parentesi? ■ Nelle istruzioni case, sono stati previsti tutti i casi possibili? ■ È stato inserito un break dopo ogni istruzione case, nel caso in cui il break sia richiesto?
Errori di input/output	<ul style="list-style-type: none"> ■ Sono state utilizzate tutte le variabili di input? ■ È stato assegnato un valore a tutte le variabili di output, prima che il programma ne richieda il valore? ■ Input imprevisti possono causare errori del programma?
Errori di interfaccia	<ul style="list-style-type: none"> ■ Le chiamate delle funzioni e dei metodi hanno il numero corretto di parametri? ■ I tipi di parametri formali e reali corrispondono? ■ I parametri sono nel giusto ordine? ■ I componenti che accedono alla memoria condivisa hanno lo stesso modello di struttura della memoria condivisa?
Errori nella gestione della memoria	<ul style="list-style-type: none"> ■ Se una struttura collegata viene modificata, i collegamenti vengono correttamente riassegnati? ■ Se si utilizza la memoria dinamica, lo spazio viene allocato correttamente? ■ Lo spazio viene esplicitamente deallocated se non è più richiesto?
Errori di gestione delle eccezioni	<ul style="list-style-type: none"> ■ Sono state considerate tutte le possibili condizioni d'errore?

Figura 21.8 Una lista di controllo per l'ispezione dei programmi.

frontato i test delle unità, dove il tasso di identificazione dei difetti è del 25% circa, con le ispezioni, dove questo tasso era del 60%. Questi confronti sono stati effettuati prima della vasta diffusione dei test automatizzati. Noi non sappiamo come le ispezioni possano essere confrontate con questo approccio.

Nonostante la loro ben nota caratteristica di efficienza, molte società di sviluppo del software sono riluttanti a utilizzare le ispezioni o le revisioni. Gli ingegneri del software esperti di test dei programmi, a volte, non riescono ad accettare il fatto che le ispezioni possano essere più efficaci dei test nell'identificare i difetti

del codice. I manager potrebbero essere sospettosi, perché le ispezioni richiedono costi aggiuntivi durante la progettazione e lo sviluppo del software; non vogliono correre il rischio di non avere le corrispondenti riduzioni dei costi dei test dei programmi.

21.4 Gestione della qualità e sviluppo agile

I metodi agili dell’ingegneria del software si focalizzano sullo sviluppo del codice; minimizzano la documentazione e i processi che non sono direttamente correlati con lo sviluppo del codice ed enfatizzano l’importanza delle comunicazioni informali tra i membri del team di sviluppo, anziché le comunicazioni basate sui documenti di progettazione. Qualità, nello sviluppo agile, significa qualità del codice; pratiche quali la rifattorizzazione e lo sviluppo guidato da test sono utilizzate per garantire che venga prodotto un codice di alta qualità.

La gestione della qualità nello sviluppo agile è informale, anziché basarsi sulla documentazione. Si affida a una cultura della qualità, dove tutti i membri del team di sviluppo si sentono responsabili della qualità del software e prendono decisioni che garantiscono il mantenimento della qualità. La comunità dello sviluppo agile si oppone essenzialmente a ciò che essa considera come sovraccosti burocratici dei metodi basati sugli standard e dei processi di qualità, come quelli inclusi nell’ISO 9001. Le aziende che adottano i metodi dello sviluppo agile raramente si occupano di certificazioni ISO 9001.

Nello sviluppo agile, la gestione della qualità si basa su buone pratiche condivise, anziché su documenti formali. Alcuni esempi di buone pratiche sono elencati qui di seguito.

1. *Verifica prima della consegna.* I programmatore hanno la responsabilità di organizzare le revisioni del loro codice con altri membri del team di sviluppo prima che il codice sia inserito nel sistema software.
2. *Non spezzare mai la compilazione.* Non è ammesso che ciascun membro del team di sviluppo accetti un codice che provochi il fallimento dell’intero sistema software. Di conseguenza, ogni membro deve verificare le modifiche del suo codice rispetto all’intero sistema ed essere certo che tutto il codice operi come previsto. Se la compilazione viene interrotta, la persona responsabile deve dare la massima priorità alla risoluzione del problema.
3. *Correggere gli errori quando vengono scoperti.* Il codice del sistema appartiene al team, non ai singoli membri. Quindi, se un programmatore scopre un errore o un’anomalia nel codice sviluppato da un altro membro del team, deve risolvere direttamente questi problemi, anziché segnalarli allo sviluppatore originario.

I processi agili raramente usano processi formali di revisione o ispezione. Nell’approccio Scrum, i membri del team di sviluppo si incontrano dopo ogni iterazione per discutere i problemi di qualità. Il team può decidere di modificare

il modo di operare per evitare i problemi di qualità che sono emersi. Potrebbe essere concordata una decisione per mettere in evidenza la rifattorizzazione e il miglioramento della qualità durante uno sprint, anziché aggiungere nuove funzionalità al sistema.

Le revisioni del codice potrebbero essere affidate alla responsabilità dei singoli membri (verifica prima della consegna) o alla programmazione in coppia. Come detto nel Capitolo 3, la programmazione in coppia è un metodo nel quale due persone sono responsabili dello sviluppo del codice e operano insieme per realizzarlo. Il codice sviluppato da un programmatore viene quindi costantemente esaminato e rivisto da un altro membro del team di sviluppo. Due persone controllano ogni singola riga del codice prima che questo sia accettato.

La programmazione in coppia porta a una profonda conoscenza di un programma, in quanto entrambi i programmatori devono capire dettagliatamente il programma per proseguire nel suo sviluppo. Questa conoscenza approfondita a volte è difficile da conseguire in altri processi di ispezione; per questo la programmazione in coppia è in grado di scoprire bug che le ispezioni formali non sarebbero in grado di scoprire. Tuttavia, è da notare che le due persone coinvolte nello sviluppo del codice potrebbero non essere così obiettive come un team esterno di ispezione, in quanto devono giudicare il loro stesso lavoro. La programmazione in coppia presenta due potenziali problemi.

1. *Stessi errori.* Entrambi i membri di una coppia di programmazione potrebbero commettere gli stessi errori nell'interpretare i requisiti del sistema. Le discussioni tra i due membri potrebbero confermare tali errori.
2. *Stesso obiettivo.* Le coppie potrebbero essere riluttanti a cercare gli errori, perché non vogliono rallentare l'avanzamento del loro lavoro.
3. *Relazioni lavorative.* L'abilità di una coppia nello scoprire i difetti del codice potrebbe essere compromessa dalle strette relazioni lavorative che spesso rendono riluttanti a criticare l'operato dell'altro membro della coppia.

L'approccio informale alla gestione della qualità adottato nei metodi agili è particolarmente efficace nello sviluppo di prodotti software, dove la società che sviluppa il software controlla anche la sua specifica. Non è necessario consegnare rapporti sulla qualità a un cliente esterno, né è richiesta l'integrazione con altri team di gestione della qualità. Tuttavia, se si sta sviluppando un grande sistema per un cliente esterno, gli approcci agili alla gestione della qualità con una documentazione minima potrebbero essere impraticabili.

1. Se il cliente è una grande azienda, potrebbe disporre dei suoi processi di gestione della qualità e potrebbe aspettarsi che la società di sviluppo del software fornisca rapporti sull'avanzamento del lavoro che siano compatibili con tali processi. In questi casi, il team di sviluppo dovrà fornire un piano formale sulla qualità e una documentazione sulla qualità che sia conforme alle esigenze del cliente.

2. Quando più team di sviluppo sono distribuiti geograficamente, e magari appartengono a società differenti, le comunicazioni informali potrebbero essere impraticabili. Società differenti potrebbero avere approcci diversi alla gestione della qualità e, quindi, potrebbe essere necessario produrre qualche documento formale.
3. Per i sistemi di lunga durata, i membri di un team di sviluppo potrebbero cambiare nel tempo. Se non c'è alcuna documentazione, per i nuovi membri del team potrebbe essere impossibile capire perché sono state fatte alcune scelte di sviluppo.

In base a queste considerazioni, l'approccio informale alla gestione della qualità nei metodi agili deve essere rivisto in modo da introdurre qualche documento e processo formali. In generale, questo approccio viene integrato con il processo di sviluppo iterativo. Anziché sviluppare il software, uno degli sprint o una delle iterazioni dovrebbe occuparsi della produzione di una documentazione essenziale sul software.

21.5 Misure del software

Le misure del software riguardano la quantificazione di qualche attributo di un sistema software, come la complessità o l'affidabilità. Confrontando i valori misurati tra loro e con gli standard che si applicano in una società, è possibile trarre delle conclusioni sulla qualità del software o valutare l'efficienza dei processi software, degli strumenti e dei metodi adottati. In un mondo ideale, la gestione della qualità potrebbe affidarsi alle misure degli attributi che influiscono sulla qualità del software; dopodiché si potrebbero stabilire le modifiche dei processi e degli strumenti che consentono di migliorare la qualità del software.

Per esempio, supponiamo di lavorare per una società che prevede di introdurre un nuovo strumento di test del software. Prima di introdurre lo strumento, registriamo il numero di difetti del software scoperti in un determinato periodo di tempo. Questa è la base di riferimento per stabilire l'efficienza dello strumento. Dopo aver utilizzato lo strumento per un certo periodo di tempo, ripetiamo questo procedimento. Se scopriamo più difetti nello stesso periodo di tempo, dopo l'introduzione dello strumento, allora possiamo concludere che lo strumento fornisce un utile supporto al processo di convalida del software.

L'obiettivo a lungo termine delle misure del software è utilizzare i risultati delle misure per giudicare la qualità del software. Teoricamente, un sistema potrebbe essere valutato utilizzando un range di metriche per misurare i suoi attributi. Dalle misure fatte si potrebbe dedurre un valore della qualità del sistema software. Se il software ha raggiunto la soglia richiesta della qualità, potrebbe essere approvato senza revisioni. Gli strumenti di misura potrebbero mettere in evidenza aree del software che potrebbero essere migliorate. Purtroppo, siamo

ancora molto lontani da questa situazione ideale, e il giudizio automatizzato della qualità è improbabile che diventi una realtà nel prossimo futuro.

Una metrica del software è una caratteristica del sistema software, della documentazione del sistema o del processo di sviluppo che può essere effettivamente misurata. Esempi di metriche sono: la dimensione di un prodotto espressa in righe di codice; l'indice Fog, che misura della leggibilità del testo narrativo; il numero di guasti rilevati in un prodotto software consegnato; il numero di giorni-persona richiesti per sviluppare un componente del sistema.

Le metriche del software possono essere metriche di controllo o di previsione. Le prime sono a supporto della gestione dei processi, mentre le seconde aiutano a prevedere le caratteristiche del software. Le metriche di controllo di solito sono associate ai processi software. Esempi di metriche di controllo o di processo sono lo sforzo medio e il tempo richiesto per riparare i difetti rilevati. Ci sono tre tipi di metriche di processo che possono essere utilizzate.

1. *Il tempo impiegato da un particolare processo per essere completato.* Può essere il tempo totale dedicato al processo, il tempo dedicato al processo da alcuni ingegneri e così via.
2. *Le risorse richieste da un particolare processo.* Potrebbero essere lo sforzo complessivo in giorni-persona, i costi dei viaggi o le risorse del computer.
3. *Il numero di occorrenze di un particolare evento.* Esempi di eventi che potrebbero essere monitorati includono il numero di difetti scoperti durante l'ispezione del codice, il numero delle modifiche richieste per i requisiti, il numero di bug identificati in un sistema consegnato al cliente, e il numero medio delle righe di codice modificate in seguito a una modifica dei requisiti.

Le metriche di previsione (dette anche metriche di prodotto) sono associate al software stesso. Esempi di queste metriche sono la complessità ciclomatica di un modulo, la lunghezza media degli identificatori in un programma e il numero di operazioni e attributi associati alle classi di oggetti in un progetto. Le metriche di controllo e di previsione influiscono entrambe sulle decisioni di gestione, come illustra la Figura 21.9. I manager usano le metriche di processo per decidere se le modifiche dei processi devono essere apportate, e le metriche di previsione per decidere se le modifiche del software sono necessarie e se il software è pronto per essere consegnato.

In questo capitolo tratterò dettagliatamente le metriche di previsione, i cui valori sono automaticamente stabiliti analizzando il codice o i documenti. Il Capitolo 26 spiega le metriche di controllo e il loro impiego per migliorare i processi.

Le misure di un sistema software possono essere utilizzate in due modi.

1. *Assegnare un valore agli attributi di qualità di un sistema.* Misurando le caratteristiche dei componenti di un sistema e poi aggregando queste misure, è possibile valutare gli attributi di qualità del sistema, come la munite- nività.

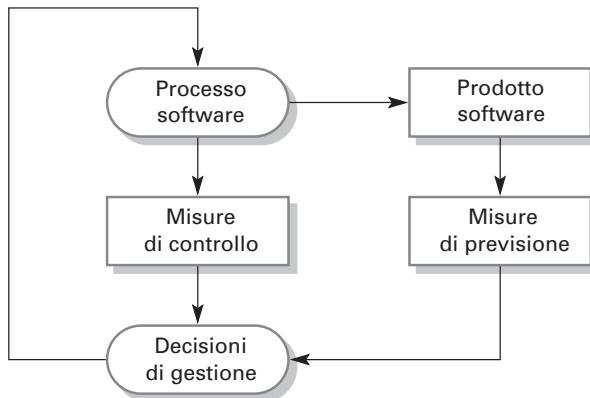


Figura 21.9 Misure di controllo e di previsione.

2. *Identificare i componenti del sistema la cui qualità è inferiore allo standard.* Le misure possono identificare quei componenti le cui caratteristiche non sono conformi alla norma. Per esempio, è possibile misurare i componenti per scoprire quelli con complessità più alta. Questi componenti hanno maggiori probabilità di contenere bug, in quanto la complessità aumenta il rischio di errori da parte di chi sviluppa questi componenti.

È difficile fare delle misure dirette di molti degli attributi di qualità del software riportati nella Figura 21.2. Gli attributi di qualità come la manutenibilità, la comprensibilità e l'utilizzabilità sono attributi esterni che sono correlati al modo in cui gli sviluppatori e gli utenti sperimentano il software. Questi attributi sono influenzati da fattori soggettivi, come l'esperienza e le conoscenze delle persone, e quindi non possono essere misurati in modo obiettivo. Per valutare questi attributi, occorre misurare qualche attributo interno del software (come la dimensione e la complessità) e supporre che questi attributi abbiano una correlazione con le caratteristiche di qualità che interessano.

La Figura 21.10 mostra alcuni attributi esterni di qualità e gli attributi interni che, intuitivamente, potrebbero essere correlati con essi. Il diagramma suggerisce che ci possono essere relazioni tra attributi esterni e interni, ma non dice come questi attributi sono correlati. Kitchenham (Kitchenham 1990) suggerisce che, affinché la misura di un attributo interno sia un predittore utile di una caratteristica esterna del software, devono essere soddisfatte tre condizioni:

1. l'attributo interno deve essere misurato accuratamente; la misura, purtroppo, non è mai semplice e potrebbe richiedere appositi strumenti di misura;
2. deve esistere una relazione tra l'attributo che può essere misurato e l'attributo esterno di qualità che interessa. In altre parole, il valore dell'attributo di qualità deve essere correlato, in qualche modo, al valore dell'attributo che può essere misurato;

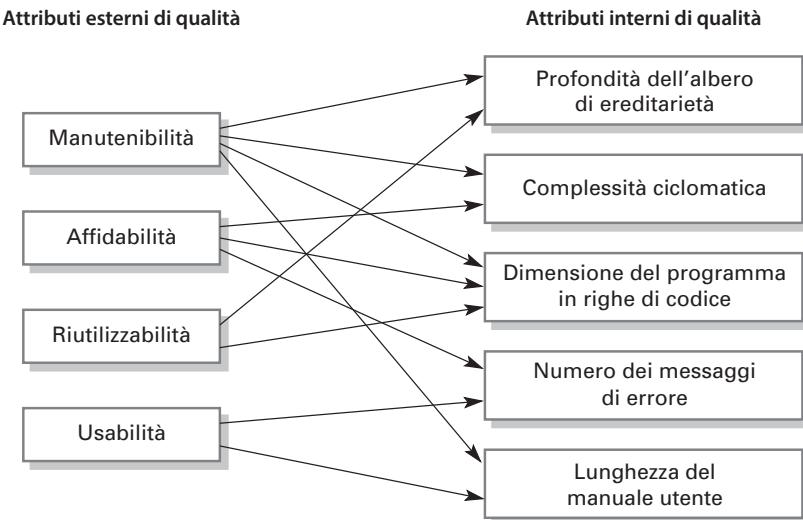


Figura 21.10 Relazioni tra attributi interni ed esterni del software.

3. La relazione tra l'attributo interno e quello esterno deve essere convalidata ed espressa in termini di formula o modello. La formulazione di un modello richiede l'identificazione della forma funzionale del modello (lineare, esponenziale ecc.) tramite l'analisi dei dati raccolti, la definizione dei parametri che devono essere inclusi nel modello e la calibrazione di questi parametri utilizzando i dati esistenti.

Recenti studi sull'analitica del software (Zhang et al. 2013) hanno utilizzato tecniche di data mining e apprendimento automatico per analizzare i repository di prodotti software e dati di processo. L'idea che sta alla base dell'analitica del software (Menzies e Zimmermann 2013) è che, in effetti, non si ha bisogno di un modello che riflette le relazioni tra la qualità del software e i dati raccolti. Piuttosto, se ci sono dati a sufficienza, le correlazioni possono essere scoperte ed è possibile fare delle previsioni sugli attributi del software. L'analitica del software è trattata nel Paragrafo 21.5.4.

Abbiamo pochissime informazioni sulle misure sistematiche del software effettuate nelle industrie. Molte aziende raccolgono informazioni sul loro software, come il numero di richieste di modifica dei requisiti o il numero di difetti scoperti durante i test. Tuttavia, non è chiaro se queste misure vengono utilizzate in modo sistematico per confrontare i prodotti e i processi software o per valutare l'impatto delle modifiche sui processi e sugli strumenti software. Tutto questo è difficile per varie ragioni.

1. È impossibile quantificare il ritorno degli investimenti relativi all'introduzione di metriche aziendali o di un programma di analitica del software. Abbiamo visto significativi miglioramenti nella qualità del software negli

ultimi anni senza l'utilizzo di metriche, quindi è difficile giustificare i costi iniziali per introdurre misure e valutazioni sistematiche del software.

2. Non ci sono standard per le metriche del software né processi standardizzati per effettuare le misure e le analisi. Molte aziende sono riluttanti a introdurre programmi per misure del software finché non saranno disponibili gli standard e gli strumenti di supporto.
3. Le misure potrebbero richiedere lo sviluppo e la manutenzione di strumenti software speciali. È difficile giustificare i costi per sviluppare questi strumenti quando non si conoscono i vantaggi che ne derivano.
4. In molte aziende, i processi software non sono standardizzati, non sono ben controllati e sono definiti in modo incompleto. Pertanto, c'è troppa variabilità all'interno di una stessa azienda sulle misure da adottare in maniera significativa.
5. Molte delle ricerche sulle misure e sulle metriche del software si sono focalizzate sulle metriche basate sul codice e sui processi di sviluppo guidati da piani. Tuttavia, una parte sempre più consistente di software attualmente viene sviluppata riutilizzando e configurando sistemi applicativi esistenti o utilizzando metodi agili. Non sappiamo come le precedenti ricerche sulle metriche possano essere applicate a queste tecniche di sviluppo del software.
6. L'introduzione di misure del software aumenta gli overhead dei processi. Questo contraddice lo scopo dei metodi agili, che raccomandano l'eliminazione delle attività di quei processi che non sono direttamente correlati con lo sviluppo dei programmi. Le aziende che hanno adottato i metodi agili difficilmente investiranno su un programma di metriche.

Le misure e le metriche del software sono la base dell'ingegneria empirica del software. In quest'area di ricerca, sono stati utilizzati esperimenti su sistemi software e dati raccolti da progetti reali per formare e convalidare ipotesi su metodi e tecniche di ingegneria del software. I ricercatori che operano in quest'area ritengono che possiamo essere sicuri del valore dei metodi e delle tecniche di ingegneria del software soltanto se siamo in grado di fornire prove concrete che essi apportino i benefici promessi dai loro inventori.

La ricerca sull'ingegneria empirica del software, tuttavia, non ha avuto un significativo impatto sulle pratiche di ingegneria del software. È difficile correlare una generica ricerca con un particolare progetto che è diverso dallo studio oggetto della ricerca. È probabile che molti fattori locali siano più importanti dei risultati empirici generali della ricerca. Per questo motivo, i ricercatori che si occupano di analitica del software sostengono che gli analisti non dovrebbero cercare di trarre conclusioni generali, ma dovrebbero analizzare i dati per specifici sistemi.

21.5.1 Metriche di prodotto

Le metriche di prodotto sono metriche di previsione che vengono utilizzate per quantificare gli attributi interni di un sistema software. Esempi di metriche di prodotto sono la dimensione del sistema, misurata in righe di codice, e il numero di metodi associati a ciascuna classe di oggetti. Purtroppo, come ho spiegato all'inizio di questo paragrafo, le caratteristiche del software che possono essere facilmente misurate, quali la dimensione e la complessità ciclomatica, non hanno una relazione chiara e coerente con gli attributi di qualità, quali la comprensibilità e la manutenibilità. Le relazioni variano in funzione dei processi di sviluppo, delle tecnologie utilizzate e del tipo di sistema che si sta sviluppando.

Le metriche di prodotto si suddividono in due classi:

1. *metriche dinamiche*; sono raccolte dalle misure effettuate su un progetto in esecuzione. Queste metriche possono essere raccolte durante i test del sistema o dopo che il sistema è consegnato agli utenti. Un esempio potrebbe essere il numero di report sui bug o il tempo impiegato per completare un calcolo;
2. *metriche statiche*; sono raccolte dalle misure eseguite sulle rappresentazioni del sistema, come il progetto, il programma o la documentazione. Esempi di metriche statiche sono illustrate nella Figura 21.11.

Questi tipi di metriche sono correlati a diversi attributi di qualità. Le metriche dinamiche aiutano a valutare l'efficienza e l'affidabilità di un sistema. Le metriche statiche aiutano a valutare la complessità, la comprensibilità e la manutenibilità di un sistema o dei suoi componenti.

Di solito esiste una chiara relazione tra metriche dinamiche e caratteristiche della qualità del software. È abbastanza semplice misurare il tempo richiesto per eseguire una data funzione e per valutare il tempo richiesto per avviare un sistema. Queste funzioni sono correlate direttamente all'efficienza del sistema. Analogamente, il numero e il tipo di malfunzionamenti di un sistema possono essere registrati e correlati direttamente all'affidabilità del software. Nel Capitolo 12 ho spiegato come misurare l'affidabilità di un sistema software.

Le metriche statiche, elencate nella Figura 21.11, hanno una relazione indiretta con gli attributi di qualità. Sono state proposte numerose metriche e sono stati effettuati molti esperimenti per derivare e convalidare le relazioni tra queste metriche e gli attributi, come la complessità e la manutenibilità di un sistema. Nessuno di questi esperimenti ha portato a conclusioni definitive, tuttavia la dimensione del programma e la complessità del controllo sembrano i predittori più affidabili della comprensibilità, della complessità e della manutenibilità di un sistema.

Metrica statica	Descrizione
Fan-in/Fan-out	Fan-in è una misura del numero di funzioni o di metodi che chiamano una funzione o un metodo X. Fan-out è il numero di funzioni che sono chiamate dalla funzione X. Un alto valore di fan-in implica che X è strettamente collegata al resto del progetto e che eventuali modifiche di X produrranno ampi effetti a catena. Un alto valore di fan-out indica che la complessità generale di X può essere alta a causa della complessità della logica di controllo necessaria per coordinare i componenti chiamati.
Lunghezza del codice	Misura la dimensione di un programma. In generale, quanto più è grande la dimensione del codice di un componente, tanto più è probabile che il componente sia complesso e a rischio di errore. È stato dimostrato che la lunghezza del codice è una delle metriche più affidabili per prevedere quanto i componenti siano a rischio di errore.
Complessità ciclomatica	Misura la complessità del controllo di un programma. La complessità del controllo è correlata alla comprensibilità del programma.
Lunghezza degli identificatori	È una misura della lunghezza media degli identificatori (nomi delle variabili, classi, metodi ecc.) in un programma. Quanto più sono lunghi gli identificatori, tanto più è probabile che abbiano significato e, quindi, che il programma sia più comprensibile.
Profondità dei cicli annidati	È una misura della profondità di annidamento delle istruzioni if in un programma. Istruzioni if profondamente annidate sono difficili da capire e a rischio di errore.
Indice Fog	È una misura della lunghezza media delle parole e delle frasi nei documenti. Quanto più è alto il valore dell'indice Fog, tanto più è difficile capire il documento.

Figura 21.11 Metriche di prodotto statiche.

Le metriche nella Figura 21.11 possono essere applicate a qualsiasi programma, ma sono state proposte anche alcune metriche più specifiche orientate agli oggetti. La Figura 21.12 riassume la serie di sei metriche orientate agli oggetti, che sono state proposte da Chidamber e Kemerer (Chidamber e Kemerer 1994), dette anche metriche o suite CK. Sebbene queste metriche siano state proposte originalmente agli inizi degli anni '90, esse sono ancora le metriche OO (orientate agli oggetti) più utilizzate. Alcuni strumenti di progettazione UML raccolgono automaticamente i valori per queste metriche, quando vengono creati i diagrammi UML.

El-Amam, in un eccellente studio delle metriche orientate agli oggetti (El-Amam 2001), ha analizzato le metriche CK e altre metriche OO. Ha concluso che non ci sono ancora prove sufficienti per capire come queste e altre metriche ori-

Metrica CK	Descrizione
Metodi pesati per classe (WMC)	È il numero di metodi in una classe, pesati dalla complessità di ciascun metodo. Per esempio, un metodo semplice può avere complessità 1, mentre un metodo grande e complesso può avere un valore molto più alto. Quanto più alto è il valore di questa metrica, tanto più complessa è la classe. Gli oggetti complessi sono più difficili da comprendere; potrebbero non essere logicamente collegati e quindi non potrebbero essere riutilizzati in modo efficiente come superclassi in un albero di ereditarietà.
Profondità dell'albero di ereditarietà (DIT)	Rappresenta il numero di livelli discreti nell'albero di ereditarietà, dove le sottoclassi ereditano gli attributi e le operazioni (metodi) dalle superclassi. Quanto più profondo è l'albero di ereditarietà, tanto più complesso è il progetto. Occorre studiare molte classi per poter capire quelle che sono definite al livello delle foglie dell'albero.
Numero di figli (NOC)	Misura il numero delle sottoclassi dirette di una classe. È un indice dell'ampiezza della gerarchia di una classe, mentre DIT misura la sua profondità. Un alto valore di NOC potrebbe indicare un maggiore riutilizzo. Potrebbe significare che occorre uno sforzo maggiore per convalidare le classi di base, a causa del numero di sottoclassi che dipendono da esse.
Accoppiamento tra le classi di oggetti (CBO)	Le classi sono accoppiate quando i metodi in un classe usano metodi o variabili di istanza definiti in un'altra classe. CBO misura l'entità di questo accoppiamento. Un alto valore di CBO significa che le classi sono fortemente interdipendenti; pertanto, è più probabile che la modifica di una classe influisca sulle altre classi del programma.
Risposta per una classe (RFC)	Misura il numero di metodi che potenzialmente potrebbero essere eseguiti in risposta a un messaggio ricevuto da un oggetto di una classe. RFC è correlato alla complessità. Quanto più è alto il valore di RFC, tanto più complessa è una classe e, quindi, è più probabile che essa includa errori.
Mancanza di coesione nei metodi (LCOM)	Questa metrica è calcolata considerando coppie di metodi in una classe. LCOM è la differenza tra il numero di coppie di metodi senza attributi condivisi e il numero di coppie di metodi con attributi condivisi. Il valore di questa metrica è stato ampiamente dibattuto, ed esiste in numerose varianti. Non è chiaro se LCOM fornisce nuove e utili informazioni oltre a quelle fornite dalle altre metriche.

Figura 21.12 Metriche CK orientate agli oggetti.

tate agli oggetti si correlino alle qualità esterne del software. Questa situazione non è cambiata dalla sua analisi del 2001. Non sappiamo ancora come utilizzare le misure dei programmi orientati agli oggetti per trarre delle conclusioni affidabili sulla loro qualità.

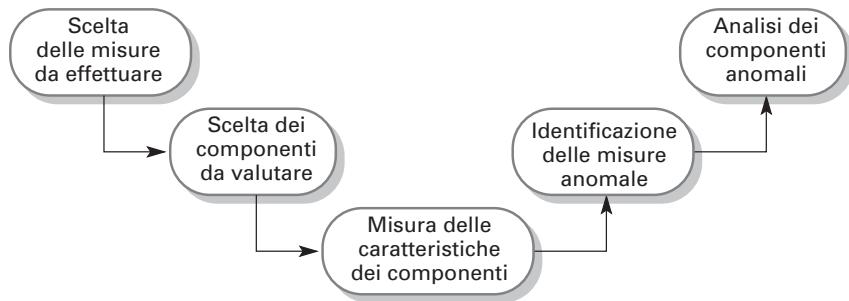


Figura 21.13 Il processo di misurazione di un prodotto.

21.5.2 Analisi dei componenti software

Nella Figura 21.13 è illustrato un processo di misurazione che può essere parte di un processo di controllo della qualità. Ciascun componente del sistema viene analizzato separatamente utilizzando una serie di metriche. I valori di queste metriche possono essere confrontati tra vari componenti ed, eventualmente, con i dati storici delle misure effettuate in precedenti progetti. Le misure anomale, che si scostano significativamente dalla norma, di solito indicano problemi di qualità per questi componenti.

Il processo di misurazione è composto da cinque fasi chiave.

1. *Scelta delle misure da effettuare* – Si dovrebbero formulare le domande alle quali le misure intendono rispondere; poi si dovrebbero definire le misure da effettuare per rispondere a queste domande. Non occorre effettuare misure che non sono direttamente pertinenti a queste domande.
2. *Scelta dei componenti da valutare* – Potrebbe non essere necessario valutare le metriche per tutti i componenti del sistema software. In alcuni casi, si può fare una selezione rappresentativa dei componenti per le metriche da valutare, in modo da poter formulare un giudizio complessivo sulla qualità del sistema. In altri casi, si potrebbero valutare i componenti principali che sono in uso quasi costantemente. La qualità di questi componenti è più importante di quella dei componenti che vengono eseguiti raramente.
3. *Misura delle caratteristiche dei componenti* – Vengono misurati i componenti selezionati e vengono calcolate le metriche associate. Questa fase normalmente richiede l'elaborazione della rappresentazione dei componenti (progetto, codice ecc.) attraverso uno strumento di raccolta automatica dei dati. Questo strumento può essere scritto specificatamente per questo compito oppure può essere incorporato negli strumenti di progettazione già in uso nella società.
4. *Identificazione delle misure anomale* – Una volta effettuate le misure dei componenti, si dovrebbero confrontare i valori ottenuti tra di loro e con quelli precedentemente registrati in un database. Si dovrebbero cercare i

valori insolitamente alti o bassi per ogni metrica, in quanto questi valori anomali indicano la presenza di potenziali problemi nel componente che li ha prodotti.

5. *Analisi dei componenti anomali* – I componenti che presentano valori anomali per alcune metriche dovrebbero essere esaminati per decidere se questi valori indicano che la qualità dei componenti è compromessa. Un valore anomalo per la metrica della complessità, per esempio, non indica necessariamente un componente di scarsa qualità; potrebbero esserci altre ragioni che spiegano questo valore anomalo.

Se possibile, tutti i dati raccolti dovrebbero essere mantenuti come una risorsa aziendale; le registrazioni storiche di tutti i progetti dovrebbero essere conservate in un database, anche quando i dati non vengono utilizzati durante un particolare progetto. Una volta che è stato memorizzato un numero sufficientemente grande di misure, è possibile fare dei confronti della qualità del software per vari progetti e convalidare le relazioni tra gli attributi interni dei componenti e le caratteristiche della qualità.

21.5.3 Ambiguità delle misure

Una volta raccolti i dati quantitativi sul software e sui processi software, occorre analizzare questi dati per capire il loro significato. È facile commettere errori di interpretazione e fare delle deduzioni sbagliate. Tuttavia, non basta un semplice esame dei dati; occorre anche considerare il contesto in cui i dati sono stati raccolti.

Per spiegare come i dati raccolti possano essere interpretati in modi differenti, consideriamo il seguente scenario, che riguarda il numero di richieste di modifiche fatte dagli utenti di un sistema software.

Un manager decide di misurare il numero di richieste di modifica fatte dai clienti basandosi sull'ipotesi che ci sia una relazione tra queste richieste e l'utilizzabilità e l'adattabilità del prodotto. Egli suppone che quanto più è alto il numero di richieste di modifica, tanto meno il software soddisfa le necessità del cliente.

Gestire queste richieste e modificare il software sono attività costose. La società decide quindi di modificare i suoi processi con lo scopo di migliorare la soddisfazione del cliente e allo stesso tempo di ridurre i costi delle modifiche. L'obiettivo è che le modifiche dei processi si traducano in prodotti migliori e si riduca anche il numero di richieste di modifica da parte dei clienti. I processi vengono modificati aumentando il coinvolgimento dei clienti nella progettazione del software. Viene introdotto il beta testing di tutti i prodotti e le modifiche richieste dai clienti vengono incorporate nel prodotto consegnato.

Una volta modificati i processi, le richieste di modifica da parte dei clienti continuano a essere contate. Vengono consegnate le nuove versioni dei prodotti, che sono stati sviluppati con i processi modificati. In alcuni casi, il numero di richieste di modifica si riduce; in altri casi, aumenta. Il manager è confuso, non riuscendo a valutare gli effetti delle modifiche dei processi sulla qualità dei prodotti.

Per capire il motivo di questo tipo di ambiguità, si deve capire perché gli utenti richiedono le modifiche. Le principali ragioni di queste richieste sono elencate qui di seguito.

1. Il software consegnato non è molto valido e non fa ciò che i clienti vorrebbero che facesse. Quindi, i clienti richiedono delle modifiche in modo da ottenere le funzionalità di cui hanno effettivamente bisogno.
2. Il software è molto valido e viene usato frequentemente e da molte persone. Le richieste di modifica scaturiscono dal fatto che alcuni utenti ritengono che il software possa svolgere nuove funzioni.

Aumentando il coinvolgimento dei clienti nel processo di sviluppo del software, si potrebbe ridurre il numero di richieste di modifica per quei prodotti che non soddisfano i clienti. Le modifiche dei processi sono state efficaci e hanno reso il software più utilizzabile e adattabile. Se, invece, le modifiche dei processi non sono state efficaci, i clienti potrebbero decidere di cercare un sistema alternativo. Il numero di richieste di modifica diminuisce perché il prodotto ha ceduto una fetta di mercato a un prodotto rivale e, quindi, si è ridotto il numero di utenti di questo prodotto.

D'altra parte, le modifiche dei processi potrebbero aver soddisfatto un maggior numero di nuovi clienti che, quindi, desiderano prendere parte al processo di sviluppo del software; questo potrebbe aumentare le richieste di modifica. I cambiamenti del processo di gestione delle richieste di modifica potrebbero contribuire a questo incremento. Se la società è molto reattiva alle richieste di modifica, i clienti potrebbero effettuare nuove richieste, perché sanno che le loro richieste saranno prese in seria considerazione; credono che i loro suggerimenti molto probabilmente saranno incorporati nelle successive versioni del software. In alternativa, il numero di richieste di modifica potrebbe aumentare perché i siti di beta testing non erano propriamente conformi al tipico utilizzo del software.

Per analizzare i dati delle richieste di modifica non occorre semplicemente conoscerne il numero. Bisogna sapere chi ha fatto le richieste e perché, e come utilizza il software. Occorre anche avere delle informazioni su alcuni fattori esterni, quali i cambiamenti della procedura di richiesta delle modifiche o delle condizioni di mercato che potrebbero avere un effetto sulle richieste. Disponendo di tali informazioni, è più facile scoprire se le modifiche dei processi sono state efficaci nel migliorare il livello qualitativo del prodotto.

Questo dimostra le difficoltà nell'interpretare gli effetti delle modifiche. L'approccio "scientifico" a questo problema consiste nel ridurre il numero di fattori

che potrebbero influire sulle misure fatte. Tuttavia, i processi e i prodotti che si stanno misurando non sono isolati dal loro ambiente. L'ambiente aziendale è in continua evoluzione, ed è impossibile evitare di modificare le pratiche operative, semplicemente perché queste potrebbero invalidare il confronto dei dati. I dati quantitativi sulle attività umane non si possono sempre prendere come valori di confronto. Le ragioni per cui un valore misurato cambia sono spesso ambigue. Si dovrebbero analizzare dettagliatamente queste ragioni, prima di trarre qualsiasi conclusione dai dati misurati.

21.5.4 Analitica del software

Negli ultimi anni, si è diffuso il concetto di “big data analysis” come strumento per scoprire idee e relazioni attraverso l’analisi automatica di grandi volumi di dati raccolti in modo automatizzato. È possibile scoprire relazioni tra elementi di dati che non potrebbero essere scoperte tramite analisi e modelli manuali dei dati. L’analitica del software è l’applicazione di tali tecniche ai dati relativamente al software e ai processi software.

Due fattori hanno reso possibile l’analitica del software.

1. La raccolta automatizzata dei dati dell’utente di un sistema software da parte della società che ha prodotto il sistema. Se il software ha un malfunzionamento, le informazioni su questo guasto e sullo stato del sistema possono essere inviate tramite Internet dal computer dell’utente ai server dello sviluppatore del prodotto. Di conseguenza, grandi volumi di dati sui singoli prodotti, come Internet Explorer o Photoshop, diventano disponibili per essere analizzati.
2. L’uso di software open-source disponibile nelle piattaforme, come Sourceforge e GitHub, e di repository open-source di dati di ingegneria del software (Menzies e Zimmermann 2013). Il codice sorgente del software open-source è disponibile per l’analisi automatica e, a volte, può essere collegato ai dati di un repository open-source.

Menzies e Zimmermann (Menzies e Zimmermann 2013) definiscono l’analitica del software in questo modo.

L’analitica del software è l’analisi dei dati per manager e ingegneri del software che ha lo scopo di consentire ai singoli membri e ai team di sviluppo del software di trarre conclusioni e condividere idee dai loro dati per prendere decisioni migliori.

Menzies e Zimmermann enfatizzano il fatto che l’analitica del software non consiste nell’estrappolare generiche teorie sul software, ma nell’identificare specifici argomenti che possono interessare i manager e gli sviluppatori del software. L’obiettivo dell’analitica è fornire informazioni su questi argomenti in tempo reale, in modo che possano essere svolte le azioni appropriate in risposta alle informazioni fornite dall’analisi. In uno studio sui manager della Microsoft, Buse e Zim-

mermann (Buse e Zimmermann 2012) hanno identificato le informazioni necessarie su particolari temi, quali la definizione dei test, le ispezioni e la riferitorizzazione, quando si rilascia un prodotto software, e le tecniche per capire le esigenze dei clienti del software.

Diversi strumenti di analisi e data mining possono essere utilizzati nell’analitica del software (Witten, Frank e Hall 2011). In generale, è impossibile sapere quali sono gli strumenti migliori da utilizzare in una particolare situazione. Occorre provare diversi strumenti per scoprire quali sono i più efficaci. Buse e Zimmermann suggeriscono alcune linee guida per la scelta di questi strumenti.

- Gli strumenti dovrebbero essere facili da utilizzare, in quanto è improbabile che i manager abbiano dimestichezza con l’analisi del software.
- Gli strumenti dovrebbero essere veloci e dovrebbero fornire output concisi, anziché grandi volumi di informazioni.
- Gli strumenti dovrebbero effettuare molte misure utilizzando il maggior numero di parametri. È impossibile sapere in anticipo quali relazioni potranno emergere dall’analisi.
- Gli strumenti dovrebbero essere interattivi e consentire ai manager e agli sviluppatori di esaminare i risultati dell’analisi. Dovrebbero essere in grado di riconoscere che manager e sviluppatori hanno interessi differenti. Non dovrebbero fare previsioni, ma dovrebbero fornire un supporto al processo decisionale sulla base dell’analisi dei dati correnti e passati.

Zhang e i suoi colleghi (Zhang et al. 2013) hanno descritto un’eccellente applicazione pratica dell’analitica del software per il debugging delle prestazioni. Il software dell’utente venne dotato di appositi strumenti per raccogliere dati sui tempi di risposta e sui corrispondenti stati del sistema. Quando il tempo di risposta era maggiore del previsto, i dati venivano inviati per essere analizzati. L’analisi automatica metteva in evidenza i colli di bottiglia delle prestazioni del software. Il team di sviluppo poteva così migliorare gli algoritmi per eliminare i colli di bottiglia e, quindi, perfezionare le prestazioni del software nella successiva release.

Attualmente, l’analitica del software è ancora immatura, ed è troppo presto per dire quali effetti potrà avere. Non soltanto ci sono problemi generali per elaborare grandi quantità di dati (Harford 2013), ma c’è sempre il problema che le nostre conoscenze dipendono dai dati raccolti dalle grandi società. Questi dati provengono principalmente dai prodotti software, e non è chiaro se le tecniche e gli strumenti che sono appropriati per i prodotti possano essere utilizzati anche con il software dei clienti. È improbabile che le società più piccole investano in sistemi di raccolta dei dati che sono necessari per l’analisi automatica; quindi, queste società non potranno applicare l’analitica del software.

Punti chiave

- La gestione della qualità del software serve a garantire che il software abbia un basso numero di difetti e che raggiunga gli standard richiesti di manutenibilità, affidabilità, portabilità e così via. Prevede anche la selezione degli standard per i processi e i prodotti e la definizione dei processi che controllano che questi standard siano applicati.
- Gli standard sono importanti per garantire la qualità del software, in quanto identificano le “pratiche migliori” da seguire. Quando si sviluppa il software, gli standard forniscono una solida base sulla quale costruire un software di buona qualità.
- Le revisioni delle consegne del software coinvolgono un team di persone che controllano che siano stati seguiti gli standard di qualità. Le revisioni sono la tecnica più utilizzata per valutare la qualità del software.
- Durante un’ispezione o revisione di un programma, un piccolo team controlla sistematicamente il codice. I membri del team esaminano il codice dettagliatamente e cercano possibili errori e omissioni. I problemi rilevati vengono poi discussi durante le riunioni di revisione.
- La gestione agile della qualità di solito non si affida a un apposito team per garantire la qualità del software; piuttosto, si cerca di sviluppare la cultura della qualità, secondo la quale ciascun membro del team di sviluppo deve operare per migliorare la qualità del software.
- Le misure del software possono essere utilizzate per raccogliere dati quantitativi sul software e sul processo software. È possibile utilizzare i valori delle metriche del software che sono stati raccolti per scoprire eventuali relazioni tra le qualità del prodotto e del processo.
- Le metriche di qualità del prodotto sono particolarmente utili per identificare componenti anomali che potrebbero avere problemi di qualità. Questi componenti dovrebbero essere analizzati più approfonditamente.
- L’analitica del software è l’analisi automatizzata di grandi volumi di dati relativi a prodotti e processi software, con l’obiettivo di scoprire le relazioni che potrebbero fornire utili idee ai manager e agli sviluppatori dei sistemi software.

Esercizi

- 21.1 Spiegate perché un processo software di alta qualità dovrebbe generare prodotti software di alta qualità. Discutete i possibili problemi di questo sistema di gestione della qualità.
- * 21.2 Spiegate come si potrebbero usare gli standard per catturare le conoscenze aziendali sui metodi efficaci di sviluppo del software. Suggerite quattro tipi di conoscenze che potrebbero essere inclusi negli standard aziendali.
- 21.3 Discutete la valutazione della qualità del software secondo gli attributi di qualità mostrati nella Figura 21.2. Analizzate ciascun attributo e spiegate come potrebbe essere valutato.

- 21.4 Descrivete brevemente gli standard che si potrebbero adottare per:
- usare i costrutti di controllo in C, C++ o Java;
 - i rapporti che potrebbero essere presentati per un progetto a termine in una università;
 - apportare e approvare le modifiche di un programma (Capitolo 26);
 - acquistare e installare un nuovo computer.
- * 21.5 Supponete di lavorare per una società che sviluppa software per database per piccole aziende. La società è interessata a quantificare il processo di sviluppo del software. Scrivete un rapporto suggerendo le metriche appropriate e il metodo per raccoglierle.
- * 21.6 Spiegate perché le ispezioni sono una tecnica efficace per scoprire gli errori in un programma. Quali tipi di errori è difficile scoprire con questa tecnica?
- * 21.7 Quali problemi potrebbero nascere se si usano le ispezioni formali in un’azienda dove parte del software è sviluppata tramite metodi agili?
- * 21.8 Spiegate perché è difficile convalidare le relazioni tra gli attributi interni di un prodotto, come la complessità ciclomatica, e gli attributi esterni, come la manutenibilità.
- 21.9 Supponete di lavorare per una società di software e che il vostro manager abbia letto un articolo sull’analitica del software. Vi chiede di effettuare delle ricerche su questo argomento. Consultate la documentazione sull’analitica del software e scrivete un rapporto che riassuma le attività e i problemi da considerare se si decide di adottare l’analitica del software.
- 21.10 Un vostro collega, che è un ottimo programmatore, produce un software con pochissimi errori, ma ignora sistematicamente gli standard di qualità aziendali. Come dovrebbero reagire i manager della società a questo comportamento?
- * *Gli esercizi contrassegnati con l’asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Software Quality Assurance: From Theory to Implementation. Un eccellente e ancora valido libro sui principi e la pratica della garanzia della qualità del software. Include una descrizione degli standard, come l’ISO 9001 (D. Galin, Addison-Wesley, 2004).

“Misleading Metrics and Unsound Analyses.” Un articolo eccellente scritto da eminenti ricercatori che tratta le difficoltà di capire il significato effettivo delle metriche (B. Kitchenham, R. Jeffrey e C. Connaughton, *IEEE Software*, 24 (2), March-April 2007). <http://dx.doi.org/10.1109/MS.2007.49>

“A Practical Guide to Implementing an Agile QA Process on Scrum Projects.” Una serie di slide che fornisce una panoramica su come integrare la gestione della qualità del software con lo sviluppo agile utilizzando il metodo Scrum (S. Rayhan 2008). https://www.scrumalliance.org/system/resource_files/0000/0459/agileqa.pdf

“Software Analytics: So What?” Un buon articolo introduttivo che spiega il significato dell’analitica del software e perché questa tecnica sta assumendo maggiore importanza. Tratta un tema speciale sull’analitica del software; include anche molti altri articoli su questo argomento che aiutano a capire meglio l’analitica del software (T. Menzies e T. Zimmermann, *IEEE Software*, 30 (4), July-August 2013). <http://dx.doi.org/10.1109/MS.2013.86>

CAPITOLO

22

Gestione della configurazione

L'obiettivo di questo capitolo è presentare i processi e gli strumenti per la gestione della configurazione del software. Dopo aver letto questo capitolo:

- conoscerete le funzionalità essenziali che dovrebbero essere fornite da un sistema di controllo delle versioni e vedrete come questo viene realizzato nei sistemi centralizzati e distribuiti;
- conoscerete le sfide della costruzione di un sistema software e i benefici dell'integrazione continua;
- capirete perché è importante la gestione delle modifiche del software e conoscerete le attività essenziali nel processo di gestione delle modifiche;
- apprenderete i concetti fondamentali della gestione delle release del software e come questa si differenzia dalla gestione delle versioni.

- 22.1 Gestione delle versioni
- 22.2 Costruzione dei sistemi
- 22.3 Gestione delle modifiche
- 22.4 Gestione delle release

I sistemi software cambiano costantemente durante lo sviluppo e l'utilizzo. I bug vengono scoperti e devono essere corretti. I requisiti del sistema cambiano, e occorre implementare queste modifiche in una nuova versione del sistema. Quando vengono rilasciate nuove versioni delle piattaforme hardware e software, occorre adattare i sistemi esistenti a queste novità. Quando i concorrenti introducono nuove caratteristiche nei loro sistemi, occorre adattare il proprio sistema a queste caratteristiche. Ogni volta che vengono apportate delle modifiche al software, viene creata una nuova versione del sistema. Molti sistemi, quindi, possono essere considerati come un insieme di versioni, ciascuna delle quali può essere mantenuta e gestita.

La gestione della configurazione (CM, *Configuration Management*) si occupa di politiche, processi e strumenti per gestire un sistema software in evoluzione (Aiello e Sachs 2011). Occorre gestire i sistemi in evoluzione perché è facile perdere le tracce delle modifiche e dei componenti che sono stati incorporati in ciascuna versione del sistema. Le versioni implementano proposte di modifiche, correzioni di errori e adattamenti per hardware e sistemi operativi differenti. Possono esserci più versioni in via di sviluppo e in uso nello stesso momento. Se non si dispone di procedure efficaci per la gestione della configurazione, si corre il rischio di sprecare preziose energie modificando o consegnando la versione sbagliata di un sistema, o perfino di dimenticare dove è stato memorizzato il codice sorgente del software di una particolare versione del sistema o di un componente.

La gestione della configurazione è utile per i singoli progetti, in quanto è facile che una persona dimentichi quali modifiche sono state fatte. È essenziale per i team di progettazione dove molti sviluppatori lavorano contemporaneamente sullo stesso sistema software. A volte questi sviluppatori lavorano tutti nello stesso luogo fisico; tuttavia, oggi sempre più team sono geograficamente distribuiti, con i membri che si trovano in località sparse su tutta la terra. Il sistema di gestione della configurazione fornisce ai membri di un team l'accesso al sistema che si sta sviluppando e gestisce le modifiche che vengono apportate al codice.

La gestione della configurazione di un prodotto software è formata da quattro attività strettamente correlate (Figura 22.1).

1. *Controllo delle versioni*. Consiste nel tenere traccia delle varie versioni dei componenti di un sistema e nel garantire che le modifiche apportate ai componenti dai diversi sviluppatori non interferiscano l'una con l'altra.
2. *Costruzione del sistema*. È il processo che assembla i componenti, i dati e le librerie di un programma, poi compila e collega tutto questo per creare un sistema eseguibile.
3. *Gestione delle modifiche*. È il processo che tiene traccia delle richieste di modifica del software da parte dei clienti e degli sviluppatori, valuta i costi e l'impatto di queste modifiche, e decide se e quando implementare le modifiche.



Figura 22.1 Attività della gestione della configurazione.

4. *Gestione delle release.* Prepara il software per le release esterne e tiene traccia delle versioni del sistema che sono state rilasciate per essere utilizzate dai clienti.

A causa degli enormi volumi di informazioni da gestire e delle relazioni tra gli elementi di configurazione, è essenziale disporre di appositi strumenti di supporto alla gestione della configurazione. Questi strumenti sono utilizzati per memorizzare le versioni dei componenti del sistema, per costruire i sistemi da questi componenti, per tenere traccia delle release delle versioni del sistema e delle proposte di modifica. Gli strumenti CM possono essere semplici strumenti che supportano un solo compito della gestione della configurazione, come l'identificazione dei bug, o perfino ambienti integrati che supportano tutte le attività di gestione della configurazione.

Lo sviluppo agile, dove i componenti e i sistemi vengono modificati più volte al giorno, sarebbe impossibile senza l'utilizzo di strumenti CM. Le versioni definitive dei componenti vengono conservate in un repository di progettazione. Gli sviluppatori queste versioni copiano nella loro area di lavoro; modificano il codice e poi usano gli strumenti di costruzione del sistema per creare un nuovo sistema sul loro computer, dove possono provarlo. Se sono soddisfatti delle modifiche apportate, restituiscono i componenti modificati al repository di progettazione. In questo modo, i componenti modificati sono disponibili a tutti gli altri membri del team.

Lo sviluppo di un prodotto software o di un sistema software personalizzato si svolge in tre fasi distinte.

1. Una *fase di sviluppo* dove il team di sviluppo è responsabile della gestione della configurazione del software e dove vengono aggiunte nuove funzionalità al software. Il team di sviluppo decide le modifiche da apportare al sistema.

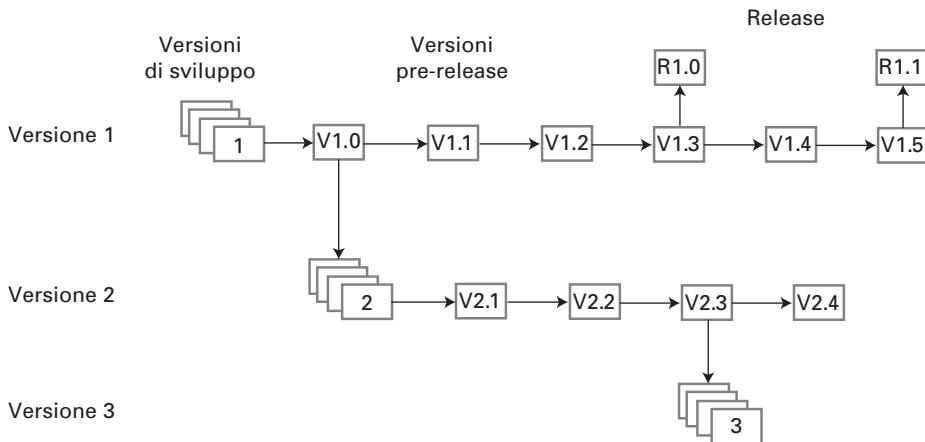


Figura 22.2 Sviluppo di più versioni dello stesso sistema.

2. Una *fase di test del sistema* dove viene completata una versione interna del sistema per essere sottoposta ai test. I test possono essere eseguiti dal team di gestione della qualità o da uno o più membri del team di sviluppo. In questa fase, nessuna funzionalità viene aggiunta al sistema. Le modifiche apportate in questa fase sono correzioni di bug, miglioramenti delle prestazioni o riparazioni dei punti di vulnerabilità del codice. In questa fase, i clienti potrebbero essere coinvolti come beta tester.
3. Una *fase di release* dove il software viene rilasciato ai clienti per essere utilizzato. Dopo che la release è stata distribuita, i clienti potrebbero segnalare eventuali bug e richiedere alcune modifiche del sistema. Nuove versioni della release del sistema potrebbero essere sviluppate per correggere i bug e i punti di vulnerabilità e per includere nuove funzionalità suggerite dai clienti.

Per i grandi sistemi software, non c’è mai una sola versione del sistema “che funziona”; ci sono sempre diverse versioni del sistema durante le varie fasi di sviluppo. Parecchi team potrebbero essere coinvolti nello sviluppo delle varie versioni del sistema. La Figura 22.2 mostra i casi in cui vengono sviluppate tre versioni di un sistema.

1. La versione 1.5 del sistema è stata sviluppata per riparare i bug e migliorare le prestazioni della prima release del sistema. È la base della seconda release del sistema (R1.1).
2. La versione 2.4 viene testata con l’idea che sarà la release 2.0 del sistema. Nessuna funzionalità nuova viene aggiunta in questa fase.
3. La versione 3 è un sistema di sviluppo dove vengono aggiunte nuove funzionalità per soddisfare le richieste di modifica dei clienti e del team di sviluppo. Alla fine, questo sistema sarà distribuito come release 3.0.

Queste differenti versioni hanno molti componenti in comune; ci sono anche componenti o versioni di componenti che sono unici per una particolare versione del sistema. Il sistema CM tiene traccia dei componenti che fanno parte di ciascuna versione e li include nella costruzione del sistema.

Nei grandi progetti software, la gestione della configurazione a volte fa parte della gestione della qualità (trattata Capitolo 21). In questi casi, il manager della qualità è responsabile sia della gestione della qualità sia della gestione della configurazione. Quando è pronta una pre-release del sistema, il team di sviluppo la consegna al team di gestione della qualità, che verifica se la qualità del sistema è accettabile. In caso affermativo, la pre-release diventa un sistema controllato; questo vuol dire che tutte le modifiche apportate al sistema devono essere concordate e registrate prima di essere implementate.

Molti termini specializzati vengono utilizzati nella gestione della configurazione (Figura 22.3). Purtroppo questi termini non sono standardizzati. I sistemi software militari sono stati i primi sistemi nei quali è stata utilizzata la gestione della configurazione del software; quindi la terminologia per questi sistemi riflette i processi e i termini utilizzati nella gestione della configurazione dell'hardware. Gli sviluppatori dei sistemi commerciali non conoscevano la terminologia e le procedure militari e, così, spesso inventavano i loro termini. Anche per i metodi agili è stata ideata una nuova terminologia per poter distinguere l'approccio agile dai tradizionali metodi CM.

La definizione e l'uso degli standard per la gestione della configurazione sono essenziali per la certificazione di qualità secondo l'ISO 9000 e secondo il modello di maturità delle capacità SEI (Bamford e Deibler 2003; Chrissis, Konrad e Shrum 2011). All'interno di una società, gli standard CM potrebbero basarsi su standard generici, come IEEE 828-2012, uno standard IEEE per la gestione della configurazione. Questi standard si focalizzano sui processi CM e sui documenti prodotti durante il processo CM (IEEE 2012). Utilizzando standard esterni come punto di partenza, le società possono poi sviluppare standard più dettagliati e adatti alle loro specifiche esigenze. I metodi agili raramente usano questi standard, in quanto la documentazione da produrre è considerata un overhead.

22.1 Gestione delle versioni

La gestione delle versioni è il processo che tiene traccia delle differenti versioni dei componenti software e dei sistemi nei quali questi componenti sono utilizzati. Questo processo deve garantire che le modifiche apportate dai vari sviluppatori a queste versioni non interferiscano tra loro. In altre parole, la gestione delle versioni è il processo che gestisce le codeline e le baseline.

La Figura 22.4 mostra le differenze tra codeline e baseline. Una codeline è una sequenza di versioni del codice sorgente, dove le ultime versioni sono derivate dalle precedenti. Le codeline di solito si applicano ai componenti dei sistemi,

Termino	Spiegazione
Baseline	Una raccolta di versioni di componenti che formano un sistema. Le baseline sono controllate, nel senso che le versioni utilizzate in una baseline non possono essere modificate. È sempre possibile ricostruire una baseline dai componenti di cui è costituita.
Branching	Creazione di una nuova codeline da una versione di una codeline esistente. La nuova codeline e quella esistente possono poi essere sviluppate in modo indipendente.
Codeline	Una serie di versioni di un componente software e di altri elementi di configurazione da cui dipende il componente.
Controllo della configurazione	Il processo che garantisce che le versioni dei sistemi e dei componenti siano registrate e mantenute, in modo che le modifiche possano essere gestite e tutte le versioni dei componenti possano essere identificate e registrate per tutta la durata del sistema.
Elemento di configurazione del software (SCI)	Tutto ciò che è associato a un progetto software (codice, dati di prova, documenti ecc.) che è stato posto sotto il controllo della configurazione. Gli elementi di configurazione hanno sempre un identificatore unico.
Mainline	Una sequenza di baseline che rappresenta le diverse versioni di un sistema.
Merging	La creazione di una nuova versione di un componente software, ottenuta unendo versioni distinte in codeline differenti. Queste codeline potrebbero essere state create da un precedente branching di una delle codeline coinvolte.
Release	La versione di un sistema che è stata rilasciata ai clienti (o ad altri utenti di una società) per essere utilizzata.
Repository	Un database condiviso di versioni di componenti software e meta-information sulle modifiche apportate a questi componenti.
Costruzione del sistema (system building)	La creazione di una versione eseguibile del sistema, ottenuta compilando e collegando le versioni appropriate dei componenti e delle librerie che formano il sistema.
Versione	Un'istanza di un elemento di configurazione che differisce, in qualche modo, dalle altre istanze dell'elemento. Ogni versione dovrebbe avere sempre un identificatore unico.
Area di lavoro (workspace)	Area di lavoro privata dove uno sviluppatore può modificare il software senza influire sul lavoro degli altri sviluppatori, che potrebbero utilizzare o modificare contemporaneamente lo stesso software.

Figura 22.3 Terminologia della gestione della configurazione.

quindi ci sono differenti versioni per ogni componente. Una baseline è la definizione di un determinato sistema. La baseline specifica le versioni dei componenti che sono inclusi nel sistema e identifica le librerie utilizzate, i file di configura-

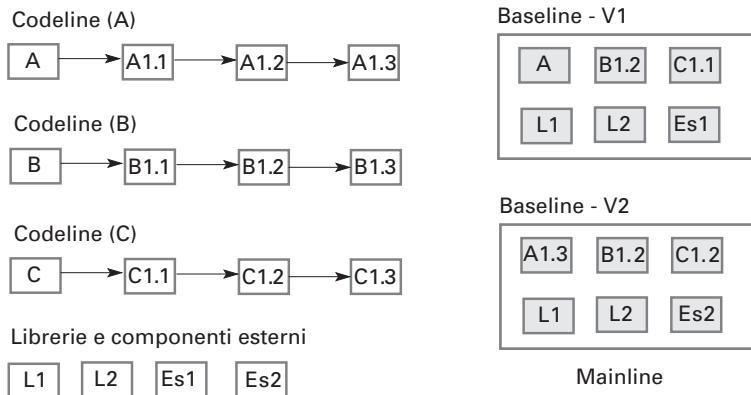


Figura 22.4 Codeline e baseline.

zione e altre informazioni sul sistema. Nella Figura 22.4 è possibile notare che differenti baseline usano differenti versioni di componenti da ciascuna codeline. Nel diagramma ho ombreggiato i box che rappresentano i componenti nella definizione della baseline per indicare che questi sono realmente i riferimenti ai componenti in una codeline.

Le baseline possono essere specificate utilizzando un linguaggio di configurazione in cui sono definiti i componenti da includere in una specifica versione del sistema. È possibile specificare esplicitamente una versione del componente (per esempio, X.1.2) o semplicemente l'identificatore del componente (X). Se si include semplicemente l'identificatore del componente nella descrizione della configurazione, dovrà essere utilizzata la versione più recente del componente.

Le baseline sono importanti perché potrebbe essere necessario ricostruire una singola versione del sistema. Per esempio, una linea di prodotti potrebbe essere istanziata in modo che ci sia una specifica versione del sistema per ogni cliente. Potrebbe essere necessario ricostruire la versione consegnata a un cliente, se vengono scoperti dei bug che devono essere corretti.

I sistemi di controllo delle versioni (CV) identificano, registrano e controllano l'accesso alle differenti versioni dei componenti. Ci sono due tipi di moderni sistemi di controllo delle versioni.

1. *Sistemi centralizzati*. Un solo repository principale conserva tutte le versioni dei componenti software che devono essere sviluppati. Subversion (Pilato, Collins-Sussman e Fitzpatrick 2008) è un esempio molto diffuso di sistema CV centralizzato.
2. *Sistemi distribuiti*. Esistono contemporaneamente più versioni del repository dei componenti. Git (Loeliger e McCullough 2012) è un esempio molto diffuso di sistema CV distribuito.

I sistemi CV centralizzati e distribuiti offrono funzionalità simili, ma implementano queste funzionalità in modo differente. Le caratteristiche chiave di questi sistemi sono elencati qui di seguito.

1. *Identificazione delle versioni e delle release.* Le versioni di un componente sono assegnate a identificatori unici quando sono incluse nel sistema. Questi identificatori consentono di gestire differenti versioni dello stesso componente, senza cambiare il nome del componente. Le versioni possono essere anche attributi assegnati, con il set di attributi utilizzati per identificare in modo univoco ciascuna versione.
2. *Registrazione della storia delle modifiche.* Il sistema CV tiene traccia delle modifiche che sono state fatte per creare una nuova versione di un componente da una precedente versione. In alcuni sistemi, queste modifiche possono essere utilizzate per selezionare una particolare versione del sistema. Questo richiede che i componenti siano etichettati con parole chiave che descrivono le modifiche fatte. Queste etichette possono essere poi utilizzate per selezionare i componenti da includere in una baseline.
3. *Sviluppo indipendente.* Diversi sviluppatori potrebbero lavorare sullo stesso componente contemporaneamente. Il sistema di controllo delle versioni tiene traccia dei componenti che sono stati designati alle modifiche e assicura che le modifiche apportate a un componente da sviluppatori differenti non interferiscano tra loro.
4. *Supporto ai progetti.* Un sistema di controllo delle versioni potrebbe fornire un supporto allo sviluppo di più progetti, che condividono i componenti. Di solito è possibile verificare e accettare tutti i file associati a un progetto, anziché dover lavorare con un solo file o una sola directory alla volta.
5. *Gestione delle registrazioni.* Anziché mantenere copie separate di tutte le versioni di un componente, il sistema di controllo delle versioni può utilizzare dei meccanismi efficienti per garantire che le copie identiche degli stessi file non siano conservate. Se ci sono solo piccole differenze tra i file, il sistema CV potrebbe registrare solo queste differenze, anziché mantenere più copie di file. La versione di uno specifico componente potrebbe essere automaticamente ricostruita applicando le differenze a una versione master del componente.

Gran parte delle attività di sviluppo del software è svolta dal team, quindi è probabile che più membri del team lavorino sullo stesso componente contemporaneamente. Per esempio, supponiamo che Alice stia apportando qualche modifica al sistema, che riguarda la modifica dei componenti A, B e C. Nello stesso tempo, Bob sta lavorando sulle modifiche da apportare ai componenti X, Y e C. Alice e Bob, quindi, stanno modificando C. È importante evitare situazioni in cui le modifiche interferiscono tra loro, ovvero che le modifiche di Bob annullino quelle di Alice o viceversa.

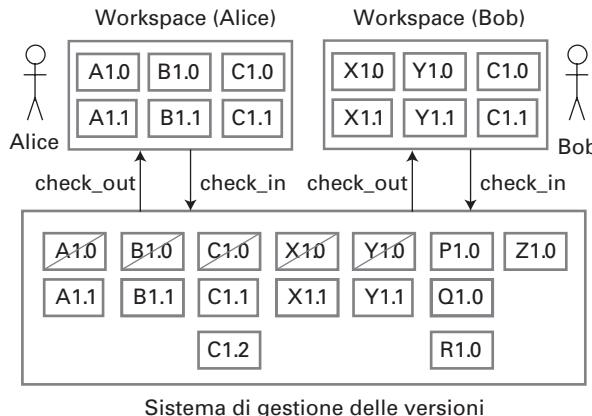


Figura 22.5 Check-in e check-out da un repository centralizzato di versioni.

Per supportare lo sviluppo indipendente senza interferenze, tutti i sistemi di controllo delle versioni usano i concetti di repository dei progetti e di area di lavoro privata (workspace). Il repository dei progetti conserva la versione “master” di tutti i componenti, che viene utilizzata per creare le baseline per la costruzione del sistema. Per modificare un componente, gli sviluppatori copiano (check-out) il componente da questo repository nel loro workspace e lavorano su questa copia. Quando hanno completato le modifiche, il componente modificato viene restituito (check-in) al repository. Si noti, tuttavia, che i sistemi CV centralizzati e distribuiti supportano lo sviluppo indipendente dei componenti condivisi in modi differenti.

Nei sistemi centralizzati, gli sviluppatori copiano i componenti o le directory di componenti dal repository dei progetti nella loro area di lavoro privata e lavorano su queste copie. Quando hanno ultimato le modifiche, restituiscono i componenti al repository. In questo modo, si crea una nuova versione del componente che può essere condivisa, come illustra la Figura 22.5.

In questa rappresentazione, Alice ha copiato le versioni A1.0, B1.0 e C1.0; ha lavorato su queste versioni e ha creato le nuove versioni A1.1, B1.1 e C1.1, che restituisce nel repository. Bob copia le versioni X1.0, Y1.0 e C1.0. Crea nuove versioni di questi componenti e le restituisce nel repository. Tuttavia, Alice ha già creato una nuova versione di C, mentre Bob sta lavorando su di essa. Quando Bob restituisce la copia modificata nel repository, viene creata un’altra versione C1.2, in modo che le modifiche di Alice non siano annullate.

Se due o più persone stanno lavorando contemporaneamente su un componente, ciascuna di esse deve copiare il componente dal repository. Se un componente è stato copiato, il sistema di controllo avverte gli utenti che il componente che vogliono copiare è stato già copiato da un altro utente. Il sistema garantisce anche che, quando i componenti modificati saranno restituiti nel repository, le differenti versioni saranno associate a identificatori differenti e registrate separatamente.

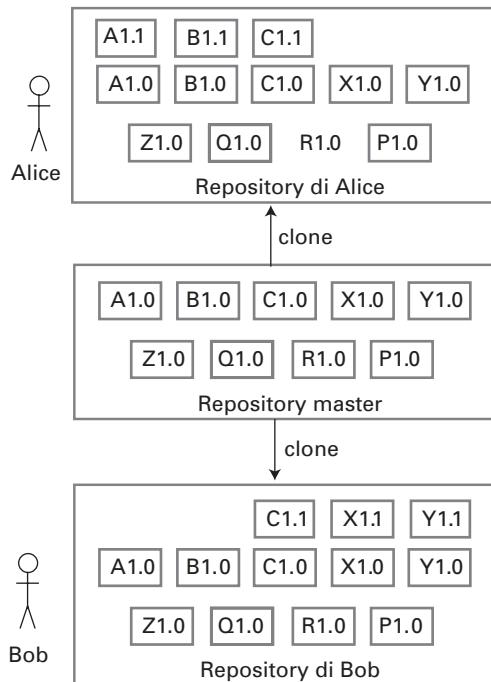


Figura 22.6 Clonazione del repository master.

In un sistema CV distribuito, come Git, l’approccio adottato è differente. Viene creato un repository “master” su un server che conserva il codice prodotto dal team di sviluppo. Anziché copiare semplicemente i file di cui ha bisogno, uno sviluppatore crea un clone del repository che viene scaricato e installato nel suo computer.

Gli sviluppatori lavorano sui file e conservano le nuove versioni nei repository privati dei loro computer. Quando hanno ultimato le modifiche, “confermano” queste modifiche e aggiornano il loro repository privato. Successivamente, possono “trasmettere” queste modifiche al repository master o comunicare al manager del repository che sono disponibili delle versioni modificate. Il manager può “accettare” questi file nel repository master (Figura 22.6). In questo esempio, Bob e Alice hanno clonato il repository master e hanno aggiornato i file, ma non hanno ancora trasmesso le versioni modificate al repository master.

Questo modello di sviluppo offre alcuni vantaggi.

1. Fornisce un meccanismo di backup per il repository. Se il repository si danneggia, il lavoro può continuare e il repository può essere ripristinato dalle copie locali.
2. Consente di lavorare offline, nel senso che gli sviluppatori possono completare le loro modifiche anche se non hanno la connessione di rete.

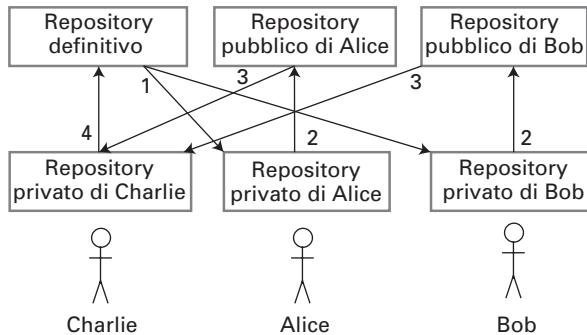


Figura 22.7 Sviluppo open-source.

3. Il supporto ai progetti è il modo di default di operare. Gli sviluppatori possono compilare e testare l'intero sistema sulle loro macchine locali e controllare la correttezza delle modifiche che hanno apportato.

Il controllo distribuito delle versioni è essenziale per lo sviluppo open-source, dove più persone possono lavorare contemporaneamente sullo stesso sistema senza un coordinamento centrale. Non c'è modo per il “manager” del sistema open-source di sapere quando le modifiche saranno apportate. In questo caso, oltre a un repository privato sul loro computer, gli sviluppatori dispongono anche di un repository pubblico nel quale riversano le nuove versioni dei componenti che hanno modificato. È responsabilità del “manager” del sistema open-source decidere quando accettare queste modifiche nel repository definitivo. Questa organizzazione è illustrata nella Figura 22.7.

In questo esempio, Charlie è il manager responsabile dell'integrazione per il sistema open-source. Alice e Bob lavorano autonomamente per sviluppare il sistema e clonano il repository definitivo (1). Oltre ai loro repository privati, Alice e Bob hanno anche un repository privato su un server cui può accedere Charlie. Dopo aver apportato le modifiche e dopo averle verificate, Alice e Bob trasmettono le versioni modificate dai repository privati ai loro repository pubblici e informano Charlie che questi repository sono disponibili (2). Charlie accetta queste nuove versioni e le copia nel suo repository privato per provarle (3). Dopo aver verificato che le modifiche possono essere accettate, Charlie aggiorna il repository definitivo (4).

Una conseguenza dello sviluppo indipendente dello stesso componente è che le codeline possono ramificarsi (branching). Anziché una sequenza lineare di versioni che riflettono le modifiche di un componente nel tempo, ci possono essere più sequenze indipendenti, come illustra la Figura 22.8. Questo è normale durante lo sviluppo di un sistema, dove più sviluppatori operano in modo autonomo su differenti versioni del codice sorgente e lo modificano in vari modi. In generale, quando si lavora su un sistema, si consiglia di creare una nuova ramificazione, in modo che le modifiche non interrompano accidentalmente il funzionamento del sistema.

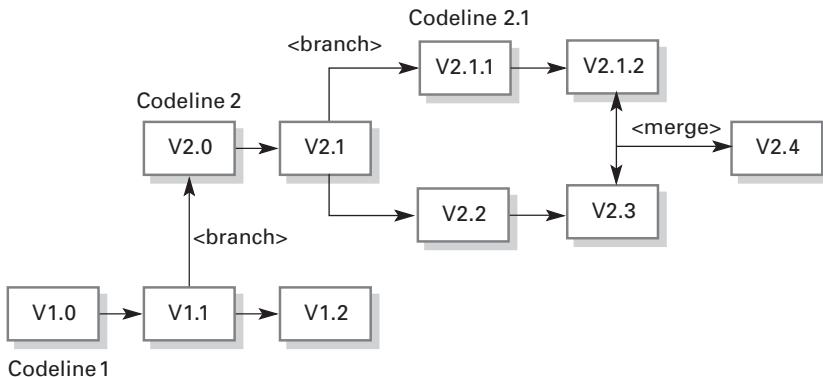


Figura 22.8 Branching e merging.

A un certo punto, potrebbe essere necessario fondere i rami delle codeline per creare una nuova versione di un componente che include tutte le modifiche che sono state apportate. Questo è illustrato nella Figura 22.8, dove le versioni 2.1.2 e 2.3 vengono fuse per creare la versione 2.4. Se le modifiche fatte interessano parti completamente differenti del codice, le versioni del componente possono essere fuse automaticamente dal sistema di controllo delle versioni. Questa è la modalità operativa normale quando vengono aggiunte nuove caratteristiche. Le modifiche del codice vengono fuse nella copia master del sistema. Tuttavia, le modifiche apportate da differenti sviluppatori a volte si sovrappongono, causando incompatibilità e interferenze tra le varie versioni. In questi casi, uno sviluppatore deve controllare dove si verificano le interferenze e apportare le opportune modifiche per risolvere le incompatibilità tra le differenti versioni.

Quando furono sviluppati i primi sistemi di controllo delle versioni, la gestione dello spazio della memoria era una delle funzioni più importanti. Lo spazio su disco era costoso, ed era importante ridurre al minimo lo spazio utilizzato dalle varie copie dei componenti. Anziché mantenere una copia completa per ogni versione, il sistema registra una lista delle differenze (delta) tra le varie versioni. Applicando questo concetto alla versione master (di solito, la versione più recente), è possibile ricostruire una versione target. Questo è illustrato nella Figura 22.9.

Quando viene creata una nuova versione, il sistema registra semplicemente un delta, una lista di differenze, tra la nuova versione e la vecchia versione che è stata utilizzata per creare quella nuova. Nella Figura 22.9, i box ombreggiati rappresentano le versioni precedenti di un componente che vengono automaticamente ricostruite dalla versione più recente. I delta di solito sono registrati come liste di linee modificate e, applicandole automaticamente, è possibile ricostruire la versione di un componente da un'altra versione. Poiché è probabile che la versione più recente di un componente sia quella più utilizzata, molti sistemi memorizzano questa versione per intero. I delta quindi definiscono come ricostruire le versioni precedenti di un sistema.

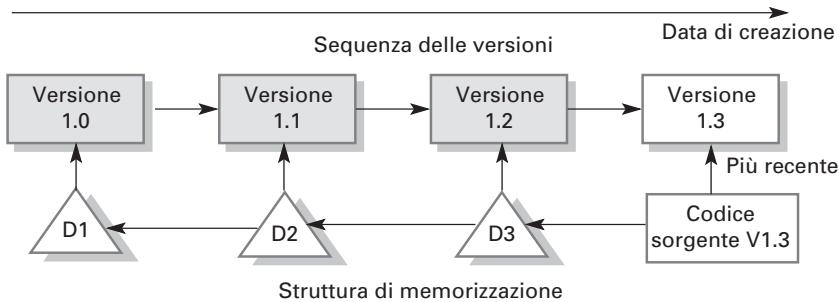


Figura 22.9 Gestione della memoria tramite i delta.

Uno dei problemi dell’approccio basato sui delta per la gestione dello spazio di memoria è che l’applicazione di tutti i delta potrebbe essere un processo lungo. Poiché oggi lo spazio su disco è relativamente economico, Git usa un altro approccio più veloce. Git non usa i delta, ma applica un algoritmo di compressione che memorizza i file e le loro meta-information associate; non memorizza copie di file. Per recuperare un file, basta decomprimere, senza bisogno di applicare una sequenza di operazioni. Git sfrutta anche il concetto di packfile, dove più file di piccole dimensioni vengono combinati in un unico file indicizzato. In questo modo, si riducono gli overhead associati a un gran numero di piccoli file. I delta sono utilizzati all’interno dei packfile per ridurre ulteriormente la loro dimensione.

22.2 Costruzione dei sistemi

La costruzione di un sistema è il processo che crea un sistema eseguibile completo, tramite la compilazione e il collegamento dei componenti del sistema, delle librerie esterne, dei file di configurazione e di altre informazioni. Gli strumenti di costruzione di un sistema e gli strumenti di controllo delle versioni devono essere integrati quando il processo di costruzione prende le versioni dei componenti dal repository gestito dal sistema di controllo delle versioni.

La costruzione dei sistemi richiede l’assemblaggio di una grande quantità di informazioni sul software e sul suo ambiente operativo. Di conseguenza, ha senso utilizzare uno strumento che automatizza il processo di costruzione (Figura 22.10). Si noti che non basta avere semplicemente i file del codice sorgente che sono coinvolti nel processo di costruzione; potrebbe essere necessario collegare questi file con librerie esterne, file di dati (per esempio, il file con i messaggi di errore) e file di configurazione che definiscono l’installazione del target. Potrebbe essere necessario specificare le versioni del compilatore e di altri strumenti software che sono richiesti dal processo di costruzione. Teoricamente, bisognerebbe essere in grado di costruire un sistema completo con un singolo comando o un clic del mouse.

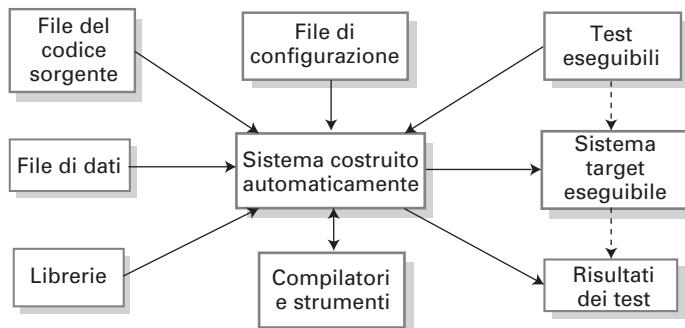


Figura 22.10 Costruzione di un sistema.

Gli strumenti per l'integrazione e la costruzione di un sistema includono alcune o tutte queste caratteristiche.

1. *Generazione dello script per la costruzione del sistema.* Il sistema di costruzione dovrebbe analizzare il progetto che si sta costruendo, identificare i componenti e generare automaticamente uno script per la costruzione del sistema (file di configurazione). Il sistema dovrebbe supportare anche la creazione e l'editing manuali degli script.
2. *Integrazione del sistema di controllo delle versioni.* Il sistema di costruzione dovrebbe accettare le versioni richieste dei componenti dal sistema di controllo delle versioni.
3. *Ricompilazione minima.* Il sistema di costruzione dovrebbe stabilire quale codice sorgente deve essere ricompilato e, se occorre, impostare la compilazione.
4. *Creazione di un sistema eseguibile.* Il sistema di costruzione dovrebbe collegare i file degli oggetti tra loro e con gli altri file richiesti, come i file di configurazione e delle librerie, per creare un sistema eseguibile.
5. *Automazione dei test.* Alcuni sistemi di costruzione possono eseguire automaticamente i test utilizzando appositi strumenti di automazione, come JUnit. Questi strumenti controllano che il processo di costruzione non sia “spezzato” dalle modifiche.
6. *Reporting.* Il sistema di costruzione dovrebbe fornire un report sul successo o sul fallimento del processo di costruzione e sui test che sono stati effettuati.
7. *Generazione della documentazione.* Il sistema di costruzione dovrebbe essere in grado di generare alcune note sul processo di costruzione e alcune pagine di guida per il sistema.

Lo script per la costruzione del sistema è la definizione del sistema da costruire; include informazioni sui componenti e sulle loro interdipendenze, sulle versioni degli strumenti da utilizzare per compilare e collegare il sistema. Il linguaggio di configurazione utilizzato per definire lo script include i costrutti che descrivono i

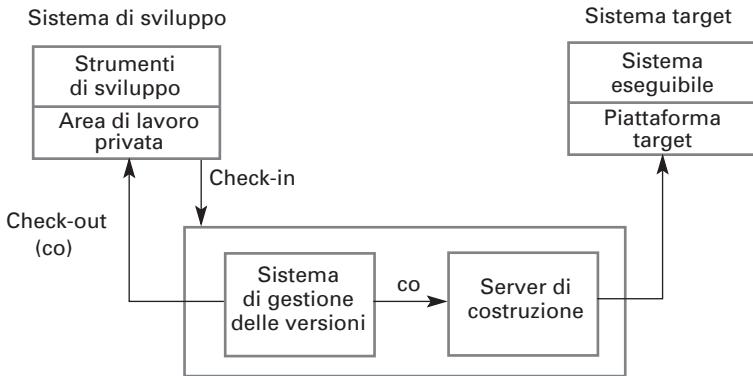


Figura 22.11 Piattaforme di sviluppo, costruzione e target.

componenti del sistema che devono essere inclusi nel processo di costruzione e le loro interdipendenze.

La costruzione di un sistema è un processo complesso, potenzialmente soggetto a errore, in quanto può includere tre differenti piattaforme di sistema (Figura 22.11).

1. *Sviluppo del sistema.* Include gli strumenti di sviluppo, come i compilatori e gli editor del codice sorgente. Gli sviluppatori copiano (chek-out) il codice dal sistema di controllo delle versioni nelle loro aree di lavoro private prima di modificare il sistema. Potrebbero costruire una versione del sistema per testarla nel loro ambiente di sviluppo prima di trasmettere (chek-in) le modifiche al sistema di controllo delle versioni. Questo richiede l'impiego di strumenti locali che usano le versioni copiate dei componenti nell'area di lavoro privata.
2. *Server di costruzione.* È utilizzato per realizzare versioni eseguibili definitive del sistema. Questo server conserva le versioni definitive di un sistema. Tutti gli sviluppatori trasmettono il codice al sistema di controllo delle versioni che viene poi utilizzato dal server di costruzione del sistema.
3. *Ambiente target.* È la piattaforma nella quale viene eseguito il sistema; può essere dello stesso tipo di computer che è stato utilizzato per sviluppare e costruire il sistema. Tuttavia, per i sistemi real-time e integrati, l'ambiente target spesso è più piccolo e più semplice dell'ambiente di sviluppo (per esempio, un telefono cellulare). Per i grandi sistemi, l'ambiente target può includere database e altri sistemi applicativi che non possono essere installati nelle macchine di sviluppo. In questi casi, non è possibile costruire e testare il sistema sul computer di sviluppo o sul server di costruzione.

I sostenitori dei metodi agili ritengono che i processi molto frequenti di costruzione di un sistema dovrebbero essere eseguiti all'esterno, con strumenti di test automatizzati per identificare eventuali problemi nel software. Le costruzioni

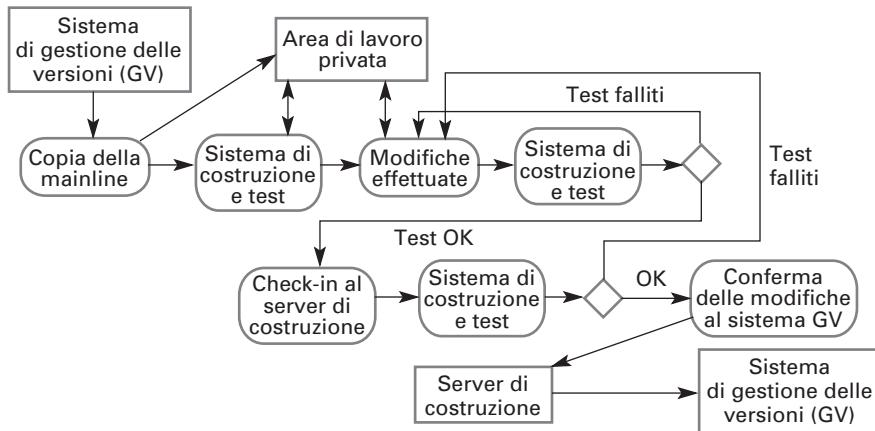


Figura 22.12 Integrazione continua.

frequenti sono parte di un processo di integrazione continua, come mostra la Figura 22.12. Considerando il concetto dell'approccio agile di effettuare molte piccole modifiche, l'integrazione continua richiede la ricostruzione frequente della mainline, dopo che sono state apportate piccole modifiche al codice sorgente. I passi dell'integrazione continua sono qui elencati.

1. Estrarre la mainline dal sistema di gestione delle versioni (GV) nell'area di lavoro privata dello sviluppatore.
 2. Costruire il sistema ed eseguire i test automatici per garantire che il sistema costruito superi tutti i test. In caso di fallimento dei test, il sistema è spezzato, e deve essere informato lo sviluppatore che ha trasferito l'ultima baseline nel sistema. Questo sviluppatore ha la responsabilità di risolvere il problema.
 3. Apportare le modifiche ai componenti del sistema.
 4. Costruire il sistema in un'area di lavoro privata e ripetere i test del sistema. Se i test non vengono superati, continuare l'editing del codice.
 5. Una volta superati i test, trasferire il sistema nel server di costruzione, ma senza confermarlo come una nuova baseline nel sistema CV.
 6. Costruire il sistema sul server ed eseguire i test. In alternativa, se state utilizzando Git, potete trasferire le ultime modifiche dal server nella vostra area di lavoro privata. Questo è necessario nel caso in cui altri sviluppatori hanno modificato i componenti da quando voi avete copiato il sistema. In questo caso, copiate i componenti che non hanno superato i test e modificatevi in modo che superino i test nella vostra area di lavoro privata.
 7. Se il sistema supera i test sul server di costruzione, confermare le modifiche fatte come una nuova baseline nella mainline del sistema.

A supporto dell'integrazione continua si usano appositi strumenti come Jenkins (Smart 2011). Questi strumenti possono essere configurati per costruire un sistema non appena lo sviluppatore ha completato un aggiornamento del repository.

Il vantaggio dell'integrazione continua è che permette di scoprire e risolvere il più presto possibile i problemi causati dalle interazioni tra i vari sviluppatori. Il sistema più recente nella mainline è il sistema definitivo. Sebbene l'integrazione continua sia una buona idea, tuttavia non è sempre possibile implementarla per costruire i sistemi per vari motivi.

1. Se il sistema è molto grande, il tempo per costruire e testare il sistema potrebbe essere molto lungo, specialmente se è richiesta l'integrazione con altri sistemi applicativi. Potrebbe essere impraticabile costruire il sistema diverse volte al giorno.
2. Se la piattaforma di sviluppo è diversa dalla piattaforma target, potrebbe essere impossibile eseguire i test del sistema nell'area di lavoro privata di uno sviluppatore. Potrebbero esserci differenze nell'hardware, nel sistema operativo o nel software installato, con conseguente aumento del tempo richiesto per testare il sistema.

Per i grandi sistemi o per i sistemi dove la piattaforma di esecuzione è diversa da quella di sviluppo, l'integrazione continua di solito è impossibile. In questi casi, la costruzione frequente del sistema è supportata tramite un sistema di costruzione giornaliero:

1. L'organizzazione responsabile dello sviluppo impone un'ora di consegna (per esempio, 14:00) per i componenti del sistema. Se gli sviluppatori hanno nuove versioni dei componenti che essi stanno scrivendo, devono consegnarle nell'ora prestabilita. I componenti possono essere incompleti, ma devono includere qualche funzionalità di base che può essere testata.
2. Viene costruita una nuova versione del sistema da questi componenti, compilandoli e collegandoli per formare un sistema completo.
3. Il sistema viene quindi consegnato al team di test, che effettua una serie di test predefiniti.
4. Gli errori che vengono scoperti durante i test vengono documentati, e gli sviluppatori del sistema vengono informati. Gli sviluppatori eliminano questi errori in una successiva versione del componente.

Il vantaggio di utilizzare frequenti costruzioni del software è che aumentano le possibilità di scoprire i problemi derivanti dalle interazioni dei componenti nelle prime fasi del processo di sviluppo. La costruzione frequente agevola il test approfondito dei componenti. Psicologicamente, gli sviluppatori vengono sottoposti alla pressione di non “spezzare la costruzione”, ovvero cercano di evitare di consegnare versioni di componenti che potrebbero causare il fallimento dell'intero sistema. Per questo, sono riluttanti a consegnare nuove versioni di componenti che non sono state opportunamente testate. Ne consegue che occorre meno tempo

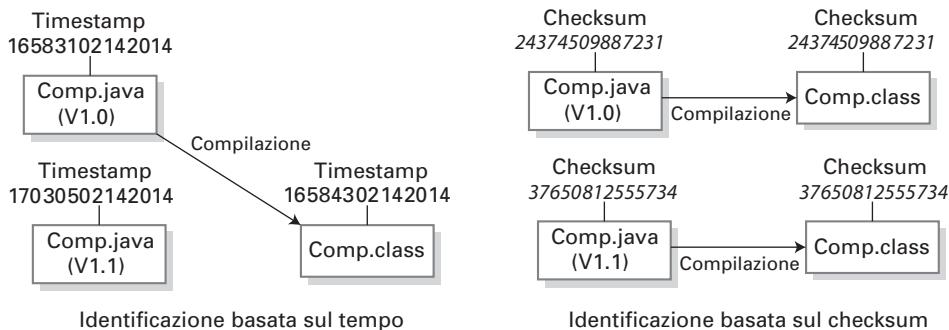


Figura 22.13 Collegare il codice sorgente e il codice oggetto.

durante i test del sistema per identificare e correggere errori che possono essere scoperti dallo sviluppatore.

Quando la compilazione è un processo che richiede calcoli intensivi, possono essere progettati appositi strumenti di supporto alla costruzione del sistema per ridurre al minimo il processo di compilazione. Questo può essere fatto verificando se è disponibile una versione compilata di un componente. Se questa versione esiste, non occorre ricompilare il componente. Quindi, ci deve essere un modo per collegare in modo inequivocabile il codice sorgente di un componente con il suo equivalente codice oggetto.

Questo collegamento è realizzato associando una firma unica a ciascun file dove è memorizzato il componente del codice sorgente. Il corrispondente codice oggetto, che è stato compilato dal codice sorgente, ha una firma correlata. La firma identifica ciascuna versione del codice sorgente, e cambia ogni volta che il codice sorgente viene modificato. Confrontando le firme nei file del codice sorgente e del codice oggetto, è possibile decidere se il componente del codice sorgente è stato utilizzato per generare il componente del codice oggetto.

È possibile utilizzare due tipi di firme, come illustra la Figura 22.13.

1. *Timestamp delle modifiche.* La firma nel file del codice sorgente è formata dall'ora e dalla data di modifica del file. Se il file del codice sorgente di un componente è stato modificato dopo il corrispondente file del codice oggetto, allora il sistema presuppone che la ricompilazione è necessaria per creare un nuovo file del codice oggetto.

Per esempio, supponiamo che i componenti Comp.java e Comp.class abbiano, rispettivamente, le firme di modifica 17:03:05:02:14:2014 e 16:58:43:02:14:2014. Questo significa che il codice Java è stato modificato 3 minuti e 5 secondi dopo le ore 17 del 14 febbraio 2014, e che la versione compilata è stata modificata 58 minuti e 43 secondi dopo le ore 16 del 14 febbraio 2014. In questo caso, il sistema ricompila automaticamente Comp.java, in quanto la versione compilata ha una data di modifica antecedente alla versione più recente del componente.

2. *Checksum del codice sorgente.* La firma nel file del codice sorgente è un checksum calcolato dalla data nel file. Una funzione di checksum calcola un numero unico utilizzando il testo sorgente come input. Se il codice sorgente viene modificato (anche di un solo carattere), si genera un checksum differente. Pertanto, si ha la certezza che i file di codice sorgente con checksum differenti sono effettivamente diversi. Il checksum è assegnato al codice sorgente appena prima della compilazione e identifica in modo univoco il file sorgente. Il processo di costruzione del sistema etichetta il file del codice oggetto generato con una firma di checksum. Se non c'è un file del codice oggetto con la stessa firma del file del codice sorgente da includere in un sistema, allora è necessario ricompilare il codice sorgente.

Poiché i file del codice oggetto di solito non sono associati a versioni, il primo approccio significa che soltanto il file del codice oggetto compilato più di recente viene conservato nel sistema. Questo file viene collegato al file del codice sorgente tramite il nome; ovvero ha lo stesso nome del file del codice sorgente, ma un suffisso diverso. Per esempio, il file sorgente Comp.java può generare il file oggetto Comp.class. Poiché il file sorgente e il file oggetto sono collegati dal nome, di solito non è possibile costruire contemporaneamente nella stessa directory versioni differenti di un componente di codice sorgente. Il compilatore genererebbe file oggetto con lo stesso nome, quindi sarà disponibile soltanto la versione compilata più di recente.

Il metodo del checksum ha il vantaggio che permette di conservare contemporaneamente più versioni del codice oggetto di un componente. La firma, non il nome del file, è il collegamento tra il codice sorgente e il codice oggetto. I file del codice sorgente e del codice oggetto hanno la stessa firma. Quindi, se viene ricompilato un componente, il codice oggetto non viene sovrascritto, come avviene nel caso in cui si usa il timestamp. Piuttosto, viene generato un nuovo file oggetto che viene etichettato con la firma del codice sorgente. La compilazione parallela è possibile, quindi possono essere compilate contemporaneamente differenti versioni di un componente.

22.3 Gestione delle modifiche

La modifica è un fatto naturale in un grande sistema software. Le esigenze e le richieste di un'azienda cambiano nel corso della vita di un sistema software, i bug devono essere eliminati e i sistemi devono essere adattati ai cambiamenti del loro ambiente operativo. Per assicurarsi che le modifiche siano applicate al sistema in modo controllato, occorre un insieme di processi di gestione delle modifiche supportati da appositi strumenti software. La gestione delle modifiche serve a garantire che l'evoluzione del sistema sia controllata e che le modifiche più urgenti ed economicamente vantaggiose abbiano priorità.

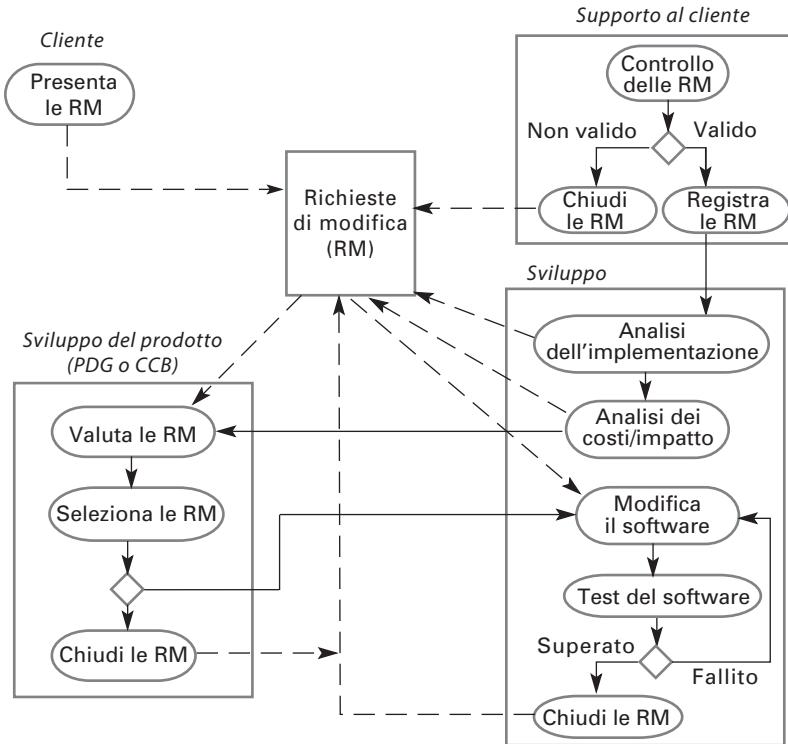


Figura 22.14 Processo di gestione delle modifiche.

La gestione delle modifiche è il processo che analizza i costi e i benefici delle modifiche proposte, approvando quelle modifiche che sono economicamente vantaggiose e tenendo traccia dei componenti del sistema che devono essere modificati. La Figura 22.14 presenta un modello di processo di gestione delle modifiche che illustra le principali attività di gestione delle modifiche. Questo processo dovrebbe diventare effettivo quando il software viene consegnato per essere rilasciato ai clienti o per essere sviluppato all'interno di una società.

Esistono molte varianti di questo processo, a seconda che il software sia un sistema personalizzato, una linea di prodotti o un prodotto off-the-shelf. Anche la dimensione della società influenza – le piccole società usano processi meno formali di quelle più grandi che operano con clienti pubblici o privati. Tuttavia, tutti i processi di gestione delle modifiche dovrebbero includere qualche strumento per controllare, valutare e approvare le modifiche.

Gli strumenti che supportano la gestione delle modifiche possono essere sistemi relativamente semplici per monitorare problemi o bug, o applicazioni più complesse che vengono integrate con un pacchetto di gestione della configurazione nei sistemi di larga scala, come Rational Clearcase. I sistemi di monitoraggio dei problemi consentono di descrivere un bug e di suggerire una modifica del

sistema, e tengono traccia del modo in cui il team di sviluppo ha pensato di risolvere questi problemi. Questi sistemi non impongono un processo agli utenti e, quindi, possono essere utilizzati con varie impostazioni. I sistemi più complessi sono costruiti attorno a un modello di processo per la gestione delle modifiche; automatizzano l'intero processo di gestione delle richieste di modifica, dalla proposta iniziale del cliente all'approvazione finale e alla presentazione delle modifiche al team di sviluppo.

Il processo di gestione delle modifiche viene avviato quando uno stakeholder del sistema completa e presenta una richiesta di modifica da apportare al sistema. Questa richiesta potrebbe essere il report su un bug, con la descrizione dei sintomi del bug, o una nuova funzionalità da aggiungere al sistema. Alcune società gestiscono i report sui bug e le richieste di nuove funzionalità in modo diverso, ma, in entrambi i casi, si tratta di richieste di modifica (RM). Le richieste di modifica possono essere presentate utilizzando un apposito modulo (MRM). Gli stakeholder possono essere i proprietari del sistema, gli utenti, i beta tester, gli sviluppatori o il dipartimento di marketing di una società.

I moduli elettronici delle richieste di modifica registrano le informazioni che sono condivise tra tutti i gruppi coinvolti nella gestione delle modifiche. Quando viene elaborata una richiesta di modifica, le informazioni vengono aggiunte nell'MRM per registrare le decisioni prese in ciascuna fase del processo. In qualsiasi momento, l'MRM rappresenta una istantanea dello stato della richiesta di modifica. Oltre a registrare la modifica richiesta, l'MRM memorizza anche i suggerimenti sulla modifica, i costi stimati della modifica, e le date di richiesta, approvazione, implementazione e convalida della modifica. L'MRM può anche includere una sezione dove uno sviluppatore spiega come implementare la modifica. Il livello di formalizzazione dell'MRM varia a seconda della dimensione e del tipo di società che sta sviluppando il sistema.

La Figura 22.15 mostra un esempio di un tipo di MRM che potrebbe essere utilizzato in un grande progetto di ingegneria di un sistema software. Per progetti più piccoli, è consigliabile registrare formalmente le richieste di modifica; l'MRM dovrebbe descrivere accuratamente la modifica richiesta, con meno enfasi sui problemi di implementazione. Gli sviluppatori del sistema decidono come implementare la modifica e stimano il tempo necessario per completare l'implementazione.

Dopo che una richiesta di modifica è stata presentata, occorre controllare che sia valida. Il controllore può essere un cliente o un membro del team di supporto oppure, per le richieste interne, un membro del team di sviluppo. La richiesta di modifica potrebbe essere rifiutata in questa fase. Se la richiesta è un report su un bug, il bug potrebbe già essere stato segnalato e corretto. A volte, ciò che le persone credono che sia un problema in effetti è soltanto una cattiva interpretazione di ciò che il sistema dovrebbe fare. In alcuni casi, le persone richiedono funzionalità che sono state già implementate, ma di cui non sanno nulla. Se una di queste funzionalità esiste, il problema è risolto e il modulo viene aggiornato con

Modulo di richiesta di modifica (MRM)	
Progetto: SICSA/AppProcessing	Numero: 23/02
Richiedente: I. Sommerville	Data: 20/07/12
Modifica richiesta: lo stato di ogni richiesta (rifiutata, accettata ecc.) dovrebbe essere visibile nella lista delle modifiche richieste.	
Analista delle modifiche: R. Looek	Data di analisi: 25/07/12
Componenti influenzati: ApplicantListDisplay, StatusUpdater	
Componenti associati: StudentDatabase	
Valutazione della modifica: relativamente semplice da implementare, perché basta cambiare il colore di visualizzazione in base allo stato della richiesta. Deve essere aggiunta una tabella che mette in relazione gli stati di una richiesta di modifica con i colori. Non sono richieste modifiche ai componenti associati.	
Priorità: media	
Implementazione della modifica:	
Tempo stimato: 2 ore	
Data di invio alla CCB: 28/07/12	Data di decisione della CCB: 30/07/12
Decisione della CCB: modifica accettata, da implementare nella Release 1.2.	
Implementatore della modifica:	Data della modifica:
Data di invio al controllo qualità:	Decisione del controllo qualità:
Data invio alla gestione della configurazione:	
Commenti:	

Figura 22.15 Un modulo di richiesta di modifica parzialmente completato.

la spiegazione della chiusura della richiesta di modifica. Se, invece, la richiesta di modifica è valida, viene registrata come richiesta da sottoporre ad analisi successiva.

Se una richiesta di modifica è valida, la fase successiva del processo è valutare i costi della modifica. Questo compito di solito spetta al team di sviluppo o di manutenzione, in quanto i loro membri sono in grado stabilire che cosa richiede l'implementazione della modifica. Deve essere valutato l'impatto della modifica sul resto del sistema. Per fare questo, occorre identificare tutti i componenti che sono influenzati dalla modifica. Se la modifica implica che altre parti del sistema debbano essere modificate, questo ovviamente aumenta i costi di implementazione della modifica. Successivamente, vengono valutate le modifiche richieste ai moduli del sistema. Infine, viene stimato il costo della modifica, tenendo conto dei costi per modificare i componenti correlati.

Dopo questa analisi, un gruppo separato decide se è economicamente conveniente per la società apportare le modifiche al software. Per i sistemi militari e governativi, questo gruppo spesso è chiamato CCB (Change Control Board, commissione di controllo delle modifiche). Per le aziende industriali, il gruppo

Clienti e modifiche

I sostenitori dei metodi agili enfatizzano l'importanza di coinvolgere i clienti nel processo che definisce le priorità delle modifiche. Il rappresentante di un cliente aiuta il team a scegliere quali modifiche implementare nella successiva iterazione del processo di sviluppo. Questo approccio può essere efficace nei sistemi che vengono sviluppati per un singolo cliente, ma può essere problematico nello sviluppo di un prodotto dove nessun cliente reale collabora con il team. In questi casi, il team deve prendere le sue decisioni sulla priorità delle modifiche.

<http://software-engineering-book.com/web/agile-changes/>

potrebbe essere chiamato PDG (Product Development Group, gruppo di sviluppo dei prodotti) e ha la responsabilità di decidere se un sistema software deve evolversi oppure no. Questo gruppo dovrebbe esaminare e approvare tutte le richieste di modifica, a meno che le modifiche non riguardino semplicemente la correzione di piccoli errori di visualizzazione delle schermate, delle pagine web o dei documenti. Queste piccole richieste dovrebbero essere passate al team di sviluppo per una immediata implementazione.

Il PDG e la CCB valutano l'impatto delle richieste di modifica da un punto di vista strategico e organizzativo, anziché tecnico; decidono se le richieste di modifica sono economicamente giustificate e assegnano le priorità di implementazione alle modifiche che sono state accettate; queste modifiche vengono passate al gruppo di sviluppo. Le modifiche rifiutate vengono chiuse, senza ulteriori interventi. I fattori che influiscono sulla decisione di implementare una richiesta di modifica sono elencati qui di seguito.

1. *Le conseguenze di non apportare una modifica.* Quando si valuta una richiesta di modifica, occorre considerare che cosa potrà accadere se la modifica non viene implementata. Se la modifica è associata a un guasto documentato del sistema, deve essere valutata la gravità del guasto. Se il guasto causa il blocco del sistema, allora è molto grave, e se non si risolve il problema, si corre il rischio di compromettere l'utilizzo del sistema. D'altra parte, se il guasto ha effetti secondari, come la visualizzazione errata dei colori sullo schermo, allora non è importante risolvere il problema rapidamente; questo tipo di modifica dovrebbe avere una bassa priorità.
2. *I benefici della modifica.* La modifica gioverà a molti utenti del sistema o soltanto a chi ha richiesto la modifica?
3. *Il numero di utenti coinvolti nella modifica.* Se la modifica interessa solo pochi utenti, allora può avere una bassa priorità. In effetti, è sconsigliabile apportare una modifica se questa significa che la maggioranza degli utenti del sistema dovrà adattarsi ad essa.
4. *I costi della modifica.* Se una modifica influenza su molti componenti del sistema (aumentando le possibilità di introdurre nuovi bug) e/o richiede troppo tempo per essere implementata, allora può essere rifiutata.

5. *Il ciclo delle release del prodotto.* Se una nuova versione del software è stata appena rilasciata ai clienti, potrebbe avere senso ritardare l'implementazione della modifica fino alla successiva release (si veda il Paragrafo 22.4).

La gestione delle modifiche per i prodotti software (per esempio, un prodotto CAD), anziché per i sistemi personalizzati appositamente sviluppati per un determinato cliente, avviene in vari modi. Nei prodotti software, il cliente non è direttamente coinvolto nelle decisioni sull'evoluzione del software, quindi l'influenza di una modifica sulle attività dei clienti non è un problema. Le richieste di modifica per questi prodotti provengono dal team di supporto ai clienti, dal team di marketing della società e dagli stessi sviluppatori.

Il team di supporto ai clienti potrebbe richiedere delle modifiche perché i clienti hanno scoperto alcuni bug dopo che il software è stato rilasciato. I clienti potrebbero utilizzare una pagina web o una e-mail per descrivere il bug. Il team per la gestione dei bug controlla che le descrizioni dei bug siano valide e le traduce in richieste formali di modifica. Il personale del marketing potrebbe incontrare i clienti e indagare sui prodotti della concorrenza; potrebbe suggerire alcune modifiche per migliorare le vendite di una nuova versione del prodotto ai nuovi e ai vecchi clienti. Gli sviluppatori stessi del prodotto potrebbero avere qualche buona idea su nuove funzionalità da aggiungere al sistema.

Il processo di richiesta delle modifiche illustrato nella Figura 22.14 inizia dopo che un sistema è stato rilasciato ai clienti. Durante lo sviluppo, quando nuove versioni del sistema vengono create tramite costruzioni giornaliere (o anche più frequenti) del sistema, non è necessario un processo formale di gestione delle modifiche. I problemi e le richieste di modifica vengono registrati in un apposito sistema di monitoraggio dei problemi e discussi in riunioni giornaliere. Le modifiche che riguardano soltanto singoli componenti vengono passate direttamente allo sviluppatore del sistema, che le può accettare o rifiutare. Tuttavia, un'autorità indipendente, come l'architetto del sistema, dovrebbe valutare le modifiche e stabilire le priorità delle modifiche che interessano i moduli del sistema prodotti da differenti team di sviluppo.

In alcuni metodi agili, i clienti sono direttamente coinvolti nelle decisioni di implementazione delle modifiche. Quando propongono una modifica dei requisiti del sistema, i clienti collaborano con il team per valutare l'impatto di tale modifica e poi decidono se dare maggiore priorità alla modifica rispetto alle funzionalità programmate per il successivo incremento del sistema. Tuttavia, le modifiche che riguardano il miglioramento del software sono lasciate alla discrezione dei programmatore che lavorano sul sistema. La rifattorizzazione – processo nel quale il software viene continuamente migliorato – non viene considerato come overheard, ma come una parte necessaria del processo di sviluppo.

Quando il team di sviluppo modifica i componenti del software, dovrebbero essere registrate tutte le modifiche apportate a ciascun componente; questo processo costituisce la storia di derivazione dei componenti. Un buon modo per conservare la storia di derivazione dei componenti consiste nell'inserire un com-

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Oggetto: currentRole
// Autore: R. Looek
// Data di creazione: 13/11/2012
//
// © St Andrews University 2012
//
// Storia delle modifiche
// Versione Modificatore Data Modifica Motivo
// 1.0 J. Jones 11/11/2009 Nuova intestazione Presentata a GM
// 1.1 R. Looek 13/11/2009 Nuovo campo Modifica n. R07/02
```

Figura 22.16 Esempio di storia di derivazione di un componente.

mento standardizzato all'inizio del codice sorgente di un componente (Figura 22.16). Questo commento può essere elaborato da script che scansionano le storie di derivazione di tutti i componenti e generano i report di modifica dei componenti. Per i documenti, le registrazioni delle modifiche incorporate in ciascuna versione di solito sono mantenute in una pagina separata all'inizio di un documento. Tutto questo è descritto nel capitolo web sulla documentazione (Capitolo 30).

22.4 Gestione delle release

La release di un sistema è una versione del sistema software che è distribuita ai clienti. Per il software di larga diffusione, di solito è possibile identificare due tipi di release: le release principali, che includono un numero significativo di nuove funzionalità, e le release secondarie, dove sono stati corretti i bug e sono stati risolti i problemi segnalati dai clienti. Per esempio, questo libro è stato scritto su un computer Mac dove il sistema operativo è OS 10.9.2; questo indica una release secondaria 2 della release principale 9 del sistema operativo 10. Le release principali sono economicamente importanti per il venditore di software, in quanto di solito i clienti devono pagare per averle. Le release secondarie, invece, di solito vengono distribuite gratuitamente.

La release di un prodotto software non è semplicemente il codice eseguibile del sistema, in quanto può includere anche:

- i file di configurazione che definiscono come la release dovrebbe essere configurata per particolari installazioni;
- i file di dati, come i file con i messaggi di errore in diverse lingue, che sono necessari per il corretto funzionamento del sistema;

- un programma di installazione che viene utilizzato per agevolare l'installazione del sistema sull'hardware di destinazione;
- documentazione elettronica e cartacea che descrive il sistema;
- imballaggio e pubblicità appositamente progettati per la release.

Preparare e distribuire le release di un sistema è un processo costoso, in particolare per i prodotti software di larga diffusione. Oltre al lavoro tecnico necessario per predisporre la distribuzione di una release, deve essere preparato anche il materiale pubblicitario. Le strategie di marketing sono progettate per convincere i clienti a comprare la nuova release del sistema. Particolare attenzione richiede la tempistica di uscita delle release. Se le release sono troppo frequenti o richiedono aggiornamenti dell'hardware, i clienti potrebbero non passare alla nuova release, specialmente se non è gratuita. Se le release di sistema sono poco frequenti, si potrebbero perdere fette di mercato, perché i clienti potrebbero scegliere sistemi alternativi più nuovi.

La Figura 22.17 elenca i vari fattori tecnici e aziendali da prendere in considerazione per decidere quando creare una nuova release di un sistema.

La creazione della release di un sistema è il processo nel quale vengono raccolti i file e i documenti che interessano tutti i componenti della release. Questo processo è composto da varie fasi.

1. Il codice eseguibile dei programmi e tutti i file di dati associati devono essere identificati nel sistema di controllo delle versioni ed etichettati con gli identificatori delle release.
2. Potrebbe essere necessario preparare le descrizioni delle configurazioni per unità hardware e sistemi operativi differenti.
3. Potrebbe essere necessario scrivere istruzioni aggiornate per i clienti che hanno bisogno di configurare i loro sistemi.
4. Potrebbe essere necessario predisporre gli script per il programma di installazione.
5. Devono essere create le pagine web che descrivono la release, con i link alla documentazione del sistema.
6. Infine, quando tutte le informazioni sono disponibili, deve essere preparata un'immagine eseguibile del software, che dovrà essere distribuita ai clienti o ai negozi di vendita.

Per il software personalizzato o per le linee di prodotti software, la complessità del processo di gestione delle release dipende dal numero di clienti. Potrebbe essere necessario produrre delle release speciali del sistema per ciascun cliente. I singoli clienti potrebbero utilizzare contemporaneamente release differenti su unità hardware differenti. Quando il software è parte di un sistema complesso di sistemi, potrebbe essere necessario creare più varianti dei singoli sistemi. Per esempio, nei veicoli antincendio, ogni tipo di veicolo può avere la sua versione del sistema software che è adatta alla sua particolare attrezzatura.

Fattore	Descrizione
Concorrenza	Per il software di larga diffusione, una nuova release del sistema può essere necessaria perché un prodotto concorrente ha introdotto nuove funzionalità e si potrebbero perdere fette di mercato se queste funzionalità non vengono fornite ai clienti esistenti.
Esigenze di marketing	Il dipartimento di marketing di un'azienda potrebbe essersi impegnato a distribuire una nuova release in una particolare data. Per ragioni di marketing, potrebbe essere necessario includere nuove funzionalità nel sistema, in modo che gli utenti possano convincersi a passare alla nuova release.
Modifiche della piattaforma	Potrebbe essere necessario creare una nuova release di un'applicazione software quando viene rilasciata una nuova versione della piattaforma del sistema operativo.
Qualità tecnica del sistema	Se vengono riportati gravi errori che influiscono sul modo in cui molti clienti usano il sistema, potrebbe essere necessario predisporre una nuova release per eliminare tali errori. Gli errori più lievi possono essere riparati rilasciando apposite patch (spesso distribuite su Internet) da applicare alla release corrente del sistema.

Figura 22.17 Fattori che influiscono sulla pianificazione delle release di un sistema.

Una società di software potrebbe gestire decine o perfino centinaia di differenti release del software che produce. I suoi sistemi e processi di gestione della configurazione dovrebbero essere progettati per fornire informazioni su quali clienti utilizzano determinate versioni del sistema e sulle relazioni tra release e versioni del sistema. In questo modo, nel caso di problemi con un sistema già consegnato a un cliente, è possibile risalire a tutte le versioni dei componenti utilizzate in quel particolare sistema.

Ne consegue che, quando viene prodotta la release di un sistema, questa deve essere documentata per garantire che possa essere ricostruita con esattezza in futuro. Questo è particolarmente importante per i sistemi integrati personalizzati e di lunga durata, come i sistemi militari e quelli che controllano macchine complesse. Questi sistemi possono avere una vita molto lunga, fino a 30 anni in alcuni casi. I clienti potrebbero usare una singola release di questi sistemi per molti anni e potrebbero richiedere modifiche specifiche in modo che la release possa durare a lungo anche dopo che è stata sostituita.

Per documentare una release, occorre registrare le versioni specifiche dei componenti di codice sorgente che sono stati utilizzati per creare il codice eseguibile. Occorre conservare le copie dei file di codice sorgente, i corrispondenti file eseguibili, i file di dati e quelli di configurazione. Potrebbe essere necessario conservare le copie di vecchi sistemi operativi e altro software di supporto, perché potrebbero essere ancora in uso. Fortunatamente, questo non significa che anche il vecchio hardware debba essere conservato. I vecchi sistemi operativi possono essere eseguiti su una macchina virtuale.

Occorre memorizzare anche le versioni del sistema operativo, le librerie, i compilatori e altri strumenti che sono stati utilizzati per costruire il software. Questi strumenti potrebbero essere necessari per ricostruire esattamente lo stesso sistema in una data futura. Analogamente, potrebbe essere necessario memorizzare copie del software della piattaforma e degli strumenti utilizzati per creare il sistema nel sistema di controllo delle versioni, insieme con il codice sorgente del sistema target.

Quando si pianifica l'installazione di nuove release di un sistema, non è possibile prevedere che tutti i clienti le installeranno. Alcuni utenti del sistema potrebbero essere soddisfatti del sistema esistente e potrebbero ritenere ingiustificati i costi di installazione di una nuova release. Le nuove release del sistema, quindi, non possono fare affidamento sull'installazione di precedenti release. Per spiegare questo problema, consideriamo il seguente scenario:

1. viene distribuita e messa in uso la release 1 di un sistema;
2. la release 2 richiede l'installazione di nuovi file di dati, ma alcuni clienti non hanno bisogno delle funzionalità della release 2, quindi mantengono la release 1;
3. la release 3 non porta nuovi dati, ma richiede i dati dei file installati nella release 2.

Il distributore del software non può supporre che i file richiesti per la release 3 siano già stati installati dappertutto; alcuni clienti potrebbero passare direttamente dalla release 1 alla release 3, saltando la 2; altri potrebbero aver modificato i file di dati associati alla release 2 per adeguarli alle situazioni locali. Pertanto i file di dati devono essere distribuiti e installati con la release 3 del sistema.

Un vantaggio offerto dal software come servizio (SaaS) è che esso evita tutti questi problemi; semplifica sia la gestione delle release sia l'installazione del sistema per i clienti. Lo sviluppatore del software è responsabile della sostituzione della release esistente di un sistema con una nuova release, che viene messa a disposizione di tutti i clienti contemporaneamente. Tuttavia, questo approccio richiede che tutti i server che forniscono i servizi siano aggiornati contemporaneamente. Per supportare gli aggiornamenti dei server, sono stati sviluppati appositi strumenti di gestione della distribuzione, come Puppet (Loope 2011), che “forzano” il nuovo software sui server.

Punti chiave

- La gestione della configurazione è il processo che gestisce l'evoluzione di un sistema software. Quando si mantiene un sistema, il team di gestione della configurazione si occupa di garantire che le modifiche siano incorporate nel sistema in modo controllato e che siano conservate le registrazioni dettagliate delle modifiche che sono state implementate.

- I principali processi di gestione della configurazione riguardano il controllo delle versioni, la costruzione del sistema, la gestione delle modifiche e la gestione delle release. Sono disponibili strumenti software appositamente studiati per svolgere tutti questi compiti.
- Il controllo delle versioni consiste nel tenere traccia delle differenti versioni dei componenti software che vengono create quando si modificano i componenti.
- La costruzione di un sistema è il processo che assembla i componenti del sistema in un programma che può essere eseguito su un sistema di computer di destinazione.
- Il software dovrebbe essere ricostruito frequentemente e testato subito dopo che viene costruita una nuova versione. Questo agevola il processo di identificazione di bug e problemi che sono stati introdotti nell'ultima costruzione del sistema.
- La gestione delle modifiche consiste nel valutare le proposte di modifica fatte dai clienti del sistema software e da altri stakeholder e nel decidere se sia economicamente vantaggioso implementare le modifiche in una nuova release del sistema.
- Le release di un sistema includono il codice eseguibile, i file dei dati, i file di configurazione e la documentazione. La gestione delle release consiste nel decidere le date delle release, nel preparare le informazioni necessarie per la distribuzione delle release e nel documentare ciascuna release.

Esercizi

- 22.1 Indicate cinque possibili problemi che potrebbero presentarsi se una società non sviluppa strategie e processi efficienti per la gestione della configurazione.
- 22.2 Spiegate perché è essenziale che ciascuna versione di un componente sia identificata in modo univoco. Commentate i problemi che potrebbero nascere se si usa uno schema di identificazione delle versioni che si basa semplicemente sui numeri delle versioni.
- * 22.3 Supponete che due sviluppatori stiano modificando contemporaneamente tre differenti componenti software. Quali problemi potrebbero nascere se tentano di combinare insieme le modifiche che hanno fatto?
- 22.4 Il software adesso viene spesso sviluppato da team di sviluppo distribuiti, i cui membri operano in località differenti e con fusi orari differenti. Indicate quali funzionalità dovrebbe avere un sistema di controllo delle versioni per supportare lo sviluppo del software distribuito.
- * 22.5 Descrivete le difficoltà che potrebbero presentarsi quando si costruisce un sistema dai suoi componenti. Quali particolari problemi potrebbero nascere quando un sistema è costruito su un computer host per qualche macchina target?
- * 22.6 Con riferimento alla costruzione di un sistema, spiegate perché a volte potrebbe essere necessario mantenere obsoleti i computer nei quali sono stati sviluppati grandi sistemi software.
- 22.7 Un problema tipico della costruzione dei sistemi si presenta quando i nomi fisici dei file vengono incorporati nel codice di un sistema e la struttura dei file implicita in questi nomi è diversa da quella della macchina target. Scrivete una serie di linee guida che aiuti un programmatore a evitare questo e altri problemi che potrebbero presentarsi durante la costruzione di un sistema.

- 22.8 Quali sono i vantaggi di utilizzare un modulo di richiesta di modifica come documento centrale nel processo di gestione delle modifiche?
- * 22.9 Descrivete sei funzionalità essenziali che dovrebbero essere incluse in uno strumento di supporto ai processi di gestione delle modifiche.
- * 22.10 Descrivete cinque fattori che gli ingegneri dovrebbero tenere in considerazione durante il processo di costruzione di una release di un grande sistema software.
- * *Gli esercizi contrassegnati con l'asterisco hanno la soluzione nella Piattaforma abbinata al testo.*

Ulteriori letture

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Un libro facile da leggere e relativamente breve che offre buoni consigli pratici sulla gestione della configurazione, specialmente per i metodi di sviluppo agile (S. P. Berczuk e B. Appleton, Addison-Wesley, 2003).

“Agile Configuration Management for Large Organizations.” Questo articolo descrive le tecniche di gestione della configurazione che possono essere utilizzate nei processi di sviluppo agile, con particolare enfasi su come queste tecniche possono essere adattate a grandi progetti e società (P. Schuh 2007). <http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html>

Configuration Management Best Practices. Un libro ben scritto che offre una panoramica ampia sulla gestione della configurazione, inclusa la gestione della configurazione dell'hardware. È particolarmente indicato per la progettazione di grandi sistemi software; non tratta i problemi dello sviluppo agile (Bob Aiello e Leslie Sachs, Addison-Wesley, 2011).

“A Behind the Scenes Look at Facebook Release Engineering.” Un interessante articolo che tratta i problemi del rilascio di nuove release di grandi sistemi nel cloud; un argomento che non ho trattato in questo capitolo. La sfida qui è garantire che tutti i server siano aggiornati contemporaneamente in modo che gli utenti non “vedano” le differenti versioni del sistema (P. Ryan, arstechnica.com, 2012). <http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/>

“Git SVn Comparison.” Questo wiki confronta i sistemi di controllo delle versioni Git e Subversion (2013, <https://git.wiki.kernel.org/index.php/GitSvnComparison>).

Glossario

Ada

Linguaggio di programmazione sviluppato per il Dipartimento della Difesa degli Stati Uniti negli anni '80 come linguaggio standard per lo sviluppo di software militare. Si basa sulle ricerche sui linguaggi di programmazione degli anni '70 e include costrutti come i tipi di dati astratti e il supporto alla simultaneità. È ancora utilizzato nello sviluppo di complessi sistemi militari e aerospaziali.

Affidabilità

La capacità di un sistema di fornire servizi come stabilito dalle sue specifiche. L'affidabilità può essere specificata quantitativamente come la probabilità di fallimenti del sistema dopo la richiesta di un servizio (POFOD) o come numero di guasti osservati in un periodo di tempo (ROCOF).

Analisi statica

Analisi, basata su strumenti, del codice sorgente di un programma con l'obiettivo di scoprire errori e anomalie. Anomalie, come l'uso di variabili non inizializzate, possono essere indicatori di errori di programmazione.

Analitica del software

Analisi automatizzata di dati statici e dinamici sui sistemi software per scoprire le relazioni tra questi dati. Le relazioni possono fornire indicazioni sui possibili modi di migliorare la qualità del software.

API (Application Program Interface)

Un'interfaccia, in genere specificata come un insieme di operazioni, che permette di accedere alle funzionalità di un programma applicativo. Ciò significa che queste funzionalità possono essere chiamate direttamente da altri programmi, non soltanto tramite l'interfaccia utente.

Architettura client-server

Modello architettonico per sistemi distribuiti dove le funzionalità di un sistema sono presentate sotto forma di servizi forniti da un server. I computer client accedono al server per usufruire dei servizi del sistema. Alcune varianti di questo approccio, come le architetture client-server a più livelli, usano più server.

Architettura del software

Un modello della struttura e dell'organizzazione di un sistema software.

Architettura di riferimento

Una generica architettura idealizzata che include tutte le caratteristiche che i sistemi dovrebbero avere. È un modo per presentare ai progettisti la struttura generale di una classe di sistemi, anziché la base per creare l'architettura specifica di un sistema.

Architettura tollerante ai guasti

Architettura di un sistema software progettata per consentire il recupero dell'esecuzione dopo un errore del software. È realizzata utilizzando componenti software ridondanti e differenti.

B, metodo

Un metodo formale di sviluppo del software che si basa sull'implementazione di un sistema tramite una sistematica trasformazione di una specifica formale del sistema.

BPMN

Business Process Modeling Notation. Una notazione per definire i flussi di lavoro che descrivono i processi aziendali e la composizione dei servizi.

Brownfield software development

Sviluppo del software per un ambiente dove esistono già vari sistemi nei quali occorre integrare il sistema che si vuole sviluppare.

C

Linguaggio di programmazione originariamente sviluppato per implementare il sistema Unix. C è un linguaggio di implementazione relativamente di basso livello, che permette l'accesso all'hardware del sistema e che può essere compilato in codice efficiente. È ancora molto utilizzato per la programmazione di sistemi a basso livello e per lo sviluppo di sistemi integrati.

C#

Linguaggio di programmazione orientato agli oggetti, sviluppato dalla Microsoft; ha molto in comune con C++, ma include caratteristiche che permettono maggiori controlli durante la compilazione.

C++

Linguaggio di programmazione orientato agli oggetti, che è un superinsieme di C.

Calcolo cloud

Consiste nel fornire servizi di calcolo e/o applicazioni su Internet utilizzando una "nuvola" (cloud) di server gestiti da un fornitore esterno. La "nuvola" è implementata utilizzando un gran numero di computer potenti e tecnologie di virtualizzazione per rendere efficiente l'uso di questi sistemi.

Caso d'uso

Specifico di un tipo di interazione con un sistema.

Caso di fidatezza

Documento strutturato che viene usato per registrare le dichiarazioni fatte da uno sviluppatore sulla fidatezza di un sistema. Esempi specifici sono i casi di sicurezza e i casi di protezione.

Caso di sicurezza

Una prova o una dimostrazione strutturata che indica che un sistema è sicuro e/o protetto. Molti sistemi critici devono essere associati a casi di sicurezza che sono definiti e approvati da autorità di controllo esterne, prima che sia certificata l'idoneità all'uso del sistema.

CBSE (Component-Based Software Engineering)

Sviluppo di software tramite la composizione di componenti indipendenti e distribuibili, che sono coerenti con un modello di componenti.

Ciclo di vita dello sviluppo del software

Espressione spesso utilizzata per indicare il processo software; originariamente fu coniata per indicare il modello a cascata del processo software.

Classe di oggetti

Una classe di oggetti definisce gli attributi e le operazioni degli oggetti. Gli oggetti sono creati a run-time istanziando la definizione di classe. Il nome della classe di oggetti può essere usato come nome di tipo in alcuni linguaggi orientati agli oggetti.

CMM (Capability Maturity Model)

Modello di stima della maturità delle capacità, elaborato dal Software Engineering Institute; è utilizzato per stabilire il livello di maturità dello sviluppo del software all'interno di una società. Recentemente è stato sostituito dal modello CMMI, ma è ancora ampiamente utilizzato.

CMMI (Capability Maturity Model Integration)

Approccio integrato alla modellazione della maturità delle capacità di sviluppo del software; si basa sull'adozione di buone pratiche dell'ingegneria del software e della gestione integrata della qualità. Supporta la modellazione continua e discreta della maturità e integra i modelli di maturità dei processi di sistema e di ingegneria del software. È stato sviluppato dal modello CMM.

COCOMO

Famiglia di modelli algoritmici di stima dei costi. COCOMO fu proposto agli inizi degli anni '80; successivamente è stato modificato e aggiornato per adattarsi alle nuove tecnologie e pratiche di ingegneria del software. COCOMO II è la sua più recente versione; è un modello algoritmico di stima dei costi, disponibile gratuitamente, che è supportato dagli strumenti software open-source.

Codice etico e di pratica professionale

Insieme delle linee guida che definisce il comportamento etico e professionale atteso dagli ingegneri del software. È stato definito dalle più importanti associazioni professionali statunitensi (ACM e IEEE) e classifica il comportamento etico in otto categorie: pubblico, cliente e datore di lavoro, prodotto, giudizio, gestione, colleghi e privato.

Componente

Unità software indipendente, distribuibile, che è completamente definita e accessibile tramite una serie di interfacce.

Computer-Aided Software Engineering (CASE)

Termine coniato negli anni '80 per descrivere il processo di sviluppo del software mediante un supporto di strumenti automatici. Virtualmente tutto lo sviluppo del software oggi dipende da un supporto automatico, quindi il termine CASE non è più molto utilizzato.

Controllo delle versioni

Il processo che gestisce le modifiche da apportare a un sistema software e ai suoi componenti, in modo che sia possibile sapere quali modifiche devono essere implementate in ciascuna versione del sistema/componente e come recuperare/ricostruire precedenti versioni del sistema/componente.

Convalida

Il processo che verifica se un sistema soddisfa le esigenze e le aspettative del cliente.

CORBA (Common Object Request Broker Architecture)

Insieme di standard proposto dall'OMG (Object Management Group) che definisce i modelli e le comunicazioni per i componenti distribuiti. Influente nello sviluppo di sistemi distribuiti, ma non più largamente utilizzato.

Costruzione del sistema

Il processo che compila i componenti o le unità che formano un sistema, collegandoli ad altri componenti per creare un programma eseguibile. La costruzione del sistema di solito è automatizzata in modo da minimizzare la ricompilazione. Questa automazione può essere integrata nel sistema di elaborazione dei linguaggi (come in Java) o può includere strumenti software che supportano la costruzione del sistema.

CVS

Strumento software open-source largamente utilizzato nella gestione delle versioni.

Database multi-tenant

Un database dove sono registrate le informazioni provenienti da più aziende. Utilizzato nell'implementazione del software come servizio.

Denial-of-service attack

Un attacco a un sistema software basato sul Web che tenta di sovraccaricare il sistema in modo che questo non possa fornire agli utenti i servizi normali.

Diagramma a barre (diagramma di Gantt)

Un diagramma utilizzato dai project manager per illustrare i compiti di un progetto, la tempestività associata a questi compiti e le persone che dovranno svolgere tali compiti. Il diagramma mostra le date di inizio e fine dei compiti e il personale che svolge ciascun compito in funzione del tempo.

Diagramma dei casi d'uso

Tipo di diagramma UML utilizzato per identificare i casi d'uso e rappresentare graficamente gli utenti coinvolti. Deve essere associato a informazioni addizionali per completare la descrizione dei casi d'uso.

Diagramma delle classi

Tipo di diagramma UML che mostra le classi degli oggetti in un sistema e le loro relazioni.

Diagramma di attività

Diagramma usato dai project manager per mostrare le interdipendenze tra le attività che devono essere completate. Il diagramma mostra le attività, il tempo previsto per il loro completamento e le loro dipendenze. Il percorso critico è il percorso più lungo (in termini di tempo richiesto per completare le attività) attraverso il diagramma, e definisce il tempo minimo richiesto per completare il progetto. Detto anche diagramma PERT.

Diagramma di sequenza

Un diagramma che mostra la sequenza delle interazioni richieste per completare qualche operazione. Nel linguaggio UML, i diagrammi di sequenza possono essere associati ai casi d'uso.

Diagramma di stato

Un tipo di diagramma UML che mostra gli stati e gli eventi che innescano una transizione da uno stato all'altro.

Digital learning environment

Un insieme integrato di strumenti software, applicazioni e informazioni a supporto dell'apprendimento.

Dinamica evolutiva dei programmi

Lo studio dei modi in cui cambia un sistema software in evoluzione. Si ritiene che la dinamica dell'evoluzione dei programmi sia governata dalle leggi di Lehman.

Disponibilità

La prontezza di un sistema nel fornire servizi quando sono richiesti. La disponibilità di solito è espressa sotto forma di numero decimale, per esempio, una disponibilità di 0,999 indica che il sistema può fornire servizi per 999 unità di tempo su 1000.

Dominio

Attività aziendali o problemi specifici in cui si utilizzano sistemi software. Esempi di domini includono il controllo in tempo reale, l'elaborazione dei dati e la commutazione nelle telecomunicazioni.

DSDM (Dynamic System Development Method)

Metodo di sviluppo dinamico dei sistemi. È considerato uno dei primi metodi di sviluppo agile.

EJB (Enterprise Java Beans)

Un modello di componenti basati su Java.

Elemento di configurazione

Unità leggibile dalla macchina, come un documento o un file di codice sorgente, che è soggetta a cambiamenti e le cui modifiche devono essere controllate da un sistema di gestione della configurazione.

Etnografia

Tecnica di osservazione che può essere usata nella ricerca e nell'analisi dei requisiti. L'etnografo si immerge nell'ambiente degli utenti e osserva le loro abitudini quotidiane di lavoro. I requisiti per il supporto software possono essere derivati da queste osservazioni.

Famiglia di applicazioni

Insieme di applicazioni software che hanno un'architettura comune e funzionalità generiche. Le applicazioni possono essere adattate alle necessità di specifici clienti modificando i componenti e i parametri dei programmi delle applicazioni.

Fault avoidance

Sviluppare il software in modo da non introdurre errori nel codice.

Fault detection

Utilizzare processi e controlli a run-time per individuare ed eliminare gli errori in un programma, prima che questi provochino un malfunzionamento del sistema.

Fault tolerance

La capacità di un sistema di continuare la sua esecuzione anche dopo che si è verificato un guasto nel sistema.

Fidatezza

La fidatezza di un sistema è una proprietà aggregata che tiene conto di vari attributi, quali la sicurezza, l'affidabilità, la disponibilità, la protezione e la resilienza. La fidatezza riflette il livello di fiducia degli utenti verso un sistema.

Framework applicativo

Un insieme di classi concrete e astratte riutilizzabili che implementano funzionalità comuni a molte applicazioni di un particolare dominio (per esempio, le interfacce utente). Le classi di un framework applicativo sono specializzate e istanziate per creare un'applicazione.

Generatore di programmi

Un programma che genera un altro programma da una specifica astratta di alto livello. Il generatore incorpora le conoscenze che vengono riutilizzate in ogni attività di generazione.

Gestione dei requisiti

Il processo che gestisce le modifiche dei requisiti in modo da garantire che le modifiche apportate siano opportunamente analizzate e tracciate durante lo sviluppo del sistema.

Gestione dei rischi

Il processo che identifica i rischi, ne valuta la gravità, pianifica le azioni per contrastarli nel caso in cui essi si concretizzano, e tiene costantemente sotto controllo il software e i processi per individuare l'insorgere dei rischi.

Gestione della configurazione

Il processo che gestisce le modifiche di un prodotto software in evoluzione. La gestione della configurazione riguarda la gestione delle versioni, la costruzione del sistema, la gestione delle modifiche e delle release.

Gestione della qualità (QM o quality management)

L'insieme dei processi che definiscono come raggiungere un determinato livello di qualità nel software e come la società che sviluppa il software può stabilire che il software ha raggiunto il livello di qualità richiesto.

Gestione delle modifiche

Un processo per registrare, controllare, analizzare, stimare e implementare le modifiche proposte per un sistema software.

Git

Gestione distribuita delle versioni e strumento di costruzione di un sistema dove gli sviluppatori utilizzano copie complete del repository dei progetti per consentire il lavoro in parallelo.

GitHub

Un server che contiene un gran numero di repository Git. I repository possono essere privati o pubblici. I repository per molti progetti open-source sono mantenuti su GitHub.

Information hiding

Utilizzo dei costrutti di un linguaggio di programmazione per nascondere la rappresentazione delle strutture dei dati e per controllare l'accesso esterno a queste strutture.

Ingegneria dei sistemi

Il processo che specifica un sistema, ne integra i componenti e verifica che i requisiti del sistema sono soddisfatti. L'ingegneria dei sistemi si occupa dell'intero sistema socio-tecnico (software, hardware e processi operativi), non solo del software.

Interfaccia

Una specifica delle operazioni e degli attributi associati a un componente software. L'interfaccia è utilizzata come strumento per accedere alle funzionalità del componente.

ISO 9000/9001

Un gruppo di standard per i processi di gestione della qualità che è definito dall'International Standard Organisation (ISO). ISO 9001 è lo standard ISO che è applicabile particolarmente allo sviluppo del software. Questi standard possono essere utilizzati per certificare i processi di gestione della qualità di un'azienda.

Ispezione dei programmi

Processo in cui un gruppo di ispettori esamina il codice di programma, riga per riga, con l'obiettivo di individuare gli errori. Questo processo spesso è guidato seguendo una lista di tipici errori di programmazione.

J2EE

Java 2 Platform Enterprise Edition. Un sistema middleware complesso che supporta lo sviluppo di applicazioni web basate sui componenti. Include un modello di componenti per componenti Java, API, servizi e altro.

Java

Linguaggio di programmazione orientato agli oggetti molto diffuso; fu progettato da Sun (adesso Oracle) con lo scopo di realizzare un linguaggio indipendente dalla piattaforma.

Leggi di Lehman

Una serie di ipotesi sui fattori che influiscono sull'evoluzione di sistemi software complessi.

Linea di prodotti software

Si veda Famiglia di applicazioni.

.NET

Un framework molto esteso utilizzato per sviluppare applicazioni per sistemi Microsoft Windows. Include un modello di componenti che definisce gli standard per i componenti nei sistemi Windows e il middleware associato per supportare l'esecuzione dei componenti.

Manifesto per lo sviluppo agile

Insieme di principi che racchiudono i concetti che sono alla base dei metodi dello sviluppo agile del software.

Manutenzione

Il processo che apporta le modifiche a un sistema dopo che questo è stato messo in esercizio.

MDA (Model-Driven Architecture)

Architettura guidata da modelli. È un approccio allo sviluppo del software basato sulla costruzione di un insieme di modelli che possono essere elaborati automaticamente o semiautomaticamente per generare un sistema eseguibile.

Mean Time To Failure (MTTF)

Il tempo medio osservato tra due guasti del sistema. Utilizzato nella specifica dell'affidabilità dei sistemi.

Mentcare

Un sistema per la gestione delle cartelle cliniche di pazienti psichiatrici. È utilizzato per registrare le informazioni sulle visite mediche e sulle cure prescritte a persone con problemi psichiatrici. È utilizzato come caso di studio in questo libro.

Metodi agili

Metodi di sviluppo del software che si basano sulla consegna rapida del software. Il software viene sviluppato e consegnato per incrementi, riducendo la documentazione e le formalità burocratiche del processo. Il cuore dello sviluppo è il codice stesso, anziché la documentazione.

Metodi formali

Metodi di sviluppo dove il software viene modellato utilizzando costrutti matematici formali, come i predicati e gli insiemi. La trasformazione formale converte i modelli in codice. Sono utilizzati principalmente nella specifica e nello sviluppo di sistemi critici.

Metodi strutturati

Metodi di progettazione del software che definiscono i modelli del sistema che si vuole sviluppare, le regole e le linee guida che si applicano a questi modelli e un processo da seguire nello sviluppo dei progetti.

Metrica del software

Un attributo di un sistema o processo software che può essere espresso numericamente e misurato. Le metriche di processo sono attributi del processo, come il tempo richiesto per completare un compito; le metriche di prodotto sono attributi del software stesso, come la dimensione o la complessità.

Metrica di controllo

Una metrica software che consente ai manager di prendere decisioni sulla pianificazione in base alle informazioni sul processo o sul prodotto software che si sta sviluppando. Molte metriche di controllo sono metriche di processo.

Middleware

Il software di infrastruttura in un sistema distribuito. Aiuta a gestire le interazioni tra le entità distribuite nel sistema e i database del sistema. Esempi di middleware sono i mediatori di richieste di oggetti e i sistemi di gestione delle transazioni.

Miglioramento di un processo

Modificare un processo di sviluppo del software con l'obiettivo di renderlo più efficiente o di migliorare la qualità dei suoi output. Per esempio, se lo scopo è ridurre il numero di difetti di un software che è stato consegnato, si potrebbe migliorare un processo aggiungendo nuove attività di convalida.

Modellazione algoritmica dei costi

Approccio alla stima dei costi del software in cui si utilizza una formula per stimare il costo del progetto. I parametri della formula sono attributi del progetto e del software stesso.

Modello a cascata

Un modello di processo software formato da fasi di sviluppo discrete: specifica, progettazione, implementazione, test e manutenzione. In teoria, una fase deve essere completata prima di passare a quella successiva; in pratica, c'è un'iterazione significativa tra le varie fasi.

Modello a spirale

Un modello di processo di sviluppo in cui il processo è rappresentato da una spirale, le cui spire incorporano i diversi stadi del processo. Spostandosi da una spira all'altra si ripetono tutti gli stadi del processo.

Modello del formaggio svizzero

Un modello di difesa dei sistemi software contro gli errori degli operatori o gli attacchi cibernetici.

Modello di componenti

Insieme di standard per l'implementazione, la documentazione e la distribuzione dei componenti. Gli standard riguardano le interfacce che possono essere fornite da un componente, i nomi da associare ai componenti, l'interazione tra i componenti e la loro composizione. I modelli dei componenti costituiscono la base per il middleware che supporta l'esecuzione dei componenti.

Modello di crescita dell'affidabilità

Il modello che esprime come cambia (migliora) l'affidabilità di un sistema quando il sistema viene testato e vengono rimossi i difetti del software.

Modello di dominio

Definizione delle astrazioni di un dominio, come le politiche, le procedure, gli oggetti, le relazioni e gli eventi. Serve come base di conoscenza su determinati problemi.

Modello di maturità del processo

Modello del grado in cui un processo include buone pratiche di sviluppo e tecniche di misurazione che sono orientate al miglioramento del processo.

Modello di oggetti

Modello di un sistema software che è strutturato e organizzato come un insieme di classi di oggetti e di relazioni tra queste classi. Possono esistere diverse prospettive sul modello, come la prospettiva di stato e quella di sequenza.

Modello di processo

Rappresentazione astratta di un processo. Un modello di processo può essere sviluppato da varie prospettive e può mostrare le attività svolte in un processo, gli artefatti utilizzati nel processo, i vincoli che si applicano al processo e i ruoli svolti dalle persone nel processo.

OCL (Object Constraint Language)

Un linguaggio, parte dell'UML, utilizzato per definire i predicati che si applicano alle classi di oggetti e alle interazioni in un modello UML. L'uso dell'OCL per specificare i componenti è una parte fondamentale dello sviluppo guidato da modelli.

OMG (Object Management Group)

Un gruppo di società che si è costituito per sviluppare standard per lo sviluppo orientato agli oggetti. Esempi di standard promossi dall'OMG sono CORBA, UML e MDA.

Open source

Un approccio allo sviluppo del software dove il codice sorgente per un sistema è a disposizione di tutti e gli utenti esterni sono incoraggiati a partecipare allo sviluppo del sistema.

P-CMM (People Capability Maturity Model)

Modello di maturità dei processi che riflette il livello di efficienza di una società nella gestione delle capacità, dell'addestramento e dell'esperienza del suo personale.

Piano di qualità

Un piano che definisce i processi e le procedure di qualità che devono essere utilizzati. Richiede la scelta di standard per prodotti e processi e la definizione dei principali attributi di qualità del sistema.

Planning game

Un approccio alla pianificazione dei progetti basato sulla stima del tempo richiesto per implementare le storie utente. È utilizzato in alcuni metodi di sviluppo agile.

Pompa di insulina

Unità medicale controllata dal software che può rilasciare dosi controllate di insulina ai pazienti diabetici. Utilizzata in un caso di studio in questo libro.

Probability of failure on demand (POFOD)

Una metrica di affidabilità che si basa sulla probabilità che un sistema software fallisca in seguito alla richiesta di un suo servizio.

Processo guidato da piani

Un processo software dove tutte le attività sono pianificate prima di sviluppare il software.

Processo software

Le attività e i processi che sono coinvolti nello sviluppo e nell’evoluzione di un sistema software.

Profilo operativo

Un insieme di input artificiali del sistema che riflette lo schema degli input che saranno elaborati in un sistema operativo. È utilizzato nei test di affidabilità.

Progettazione dell’interfaccia utente

Il processo che definisce il modo in cui gli utenti possono accedere alle funzionalità del sistema e come devono essere visualizzate le informazioni prodotte dal sistema.

Programmazione a coppie

Un modo di programmare in cui i programmatori lavorano in coppie, anziché individualmente, per sviluppare il codice. È una parte fondamentale della programmazione estrema.

Programmazione estrema (XP)

Metodo agile largamente utilizzato nello sviluppo del software che include pratiche come i requisiti basati su scenari, sviluppo con test iniziali e programmazione a coppie.

Proprietà emergente

Una proprietà che diventa apparente soltanto dopo che tutti i componenti di un sistema sono stati integrati per creare il sistema.

Protezione

La capacità di un sistema di proteggersi da intrusioni accidentali o deliberate. La protezione include la riservatezza, l’integrità e la disponibilità.

Python

Un linguaggio di programmazione con tipi dinamici, che è particolarmente adatto allo sviluppo di sistemi basati sul Web.

Rate of occurrence of failure (ROCOF)

Una metrica di affidabilità che si basa sul numero di guasti osservati in un sistema in un determinato periodo di tempo.

Rational Unified Process (RUP)

Un generico processo software che presenta lo sviluppo come un’attività iterativa di quattro fasi: avvio, elaborazione, costruzione e transizione. L’avvio stabilisce un business case per il sistema; l’elaborazione definisce l’architettura, la costruzione implementa il software, e la transizione installa il sistema nell’ambiente operativo del cliente.

Reingegnerizzazione

Cambiare un sistema software per agevolarne la lettura e la modifica. La reingegnerizzazione spesso richiede la ristrutturazione e l’organizzazione del software e dei dati, la semplificazione dei programmi e la produzione di nuovi documenti.

Release

Una versione di un sistema software che viene messa a disposizione dei clienti del sistema.

Requisito di fidatezza

Requisito di sistema che è incluso per aiutare a raggiungere il livello di fidatezza richiesto per il sistema. I requisiti non funzionali di fidatezza specificano il valore degli attributi di affidabilità; i requisiti funzionali di fidatezza permettono di evitare, individuare, tollerare o risolvere errori e guasti di un sistema.

Requisito funzionale

Definizione di una funzione o caratteristica che deve essere implementata in un sistema.

Requisito non funzionale

Definizione di un vincolo o di un comportamento atteso che si applica a un sistema. Il vincolo può riferirsi alle proprietà principali del software che si sta sviluppando o al processo di sviluppo.

Resilienza

Una valutazione su come un sistema può mantenere la continuità dei servizi critici quando si verificano eventi distruttivi, come il guasto di un'unità hardware o un attacco cibernetico.

REST (Representational State Transfer)

Metodo di sviluppo basato sulla semplice interazione client-server che usa il protocollo HTTP per le comunicazioni. Il concetto di base del metodo REST è che una risorsa è identificabile tramite un URI. Tutte le interazioni con le risorse si basano su quattro azioni HTTP: POST, GET, PUT e DELETE. È largamente utilizzato per implementare i servizi web con bassi overhead (servizi RESTful).

Rifattorizzazione

Cambiare un programma per migliorarne la struttura e la leggibilità, senza modificare le sue funzionalità.

Rischio

Una minaccia al raggiungimento di qualche obiettivo. Un rischio di processo minaccia il rispetto della tempistica o dei costi di un processo. Un rischio di prodotto può determinare il mancato soddisfacimento di alcuni requisiti del sistema. Un rischio della sicurezza è una misura della probabilità che un rischio possa produrre un guasto del sistema.

Ruby

Un linguaggio di programmazione con tipi dinamici che è particolarmente adatto alla programmazione di applicazioni web.

SaaS (Software as a Service)

Si veda Software come servizio.

SAP

Azienda tedesca che ha sviluppato un sistema ERP molto noto e diffuso. Si usa anche per far riferimento allo stesso sistema ERP.

Scenario

Descrizione di un tipico modo in cui un sistema è utilizzato o un utente svolge qualche attività.

Schema di progettazione

Soluzione collaudata di un problema comune che racchiude l'esperienza e la buona pratica di progettazione in una forma che può essere riutilizzata. È una rappresentazione astratta che può essere istanziata in vari modi.

Scrum

Un metodo di sviluppo agile, che si basa sugli sprint – sviluppo breve, cicli. Il metodo Scrum può essere utilizzato come base per la gestione agile dei progetti insieme con i metodi agili come XP.

SEI (Software Engineering Institute)

Un centro di ricerca di ingegneria del software fondato con l'obiettivo di migliorare gli standard dell'ingegneria del software nelle società americane.

Server

Un programma che fornisce un servizio ad altri programmi (client).

Servizio web

Componente software indipendente il cui accesso avviene tramite Internet utilizzando i protocolli standard. È completamente autonomo, senza dipendenze esterne. Sono stati sviluppati standard basati su XML, come SOAP (Standard Object Access Protocol) per lo scambio di

informazioni tra i servizi web, e WSDL (Web Service Definition Language) per la definizione delle interfacce dei servizi web. Per l'implementazione dei servizi web può essere utilizzato anche il metodo REST.

Sicurezza

La capacità di un sistema di operare senza danneggiare persone, dati o software del sistema.

Sistema

Un sistema è una collezione di componenti intercorrelati, di vario tipo, che operano insieme per fornire una serie di servizi al suo proprietario e ai suoi utenti.

Sistema applicativo configurabile

Un prodotto applicativo, sviluppato da un produttore di sistemi software, le cui funzionalità possono essere configurate per essere utilizzate da società differenti e in ambienti diversi.

Sistema applicativo integrato

Un sistema applicativo che è creato integrando due o più sistemi applicativi configurabili o ereditati.

Sistema basato sugli eventi

Un sistema dove il controllo delle operazioni è determinato dagli eventi che si generano nell'ambiente del sistema. La maggior parte dei sistemi reali sono sistemi basati sugli eventi.

Sistema COTS

Il termine COTS (Commercial Off-The-Shelf) è utilizzato principalmente per identificare sistemi software militari. Si veda Sistema applicativo configurabile.

Sistema critico

Un sistema informatico dove un guasto può provocare significative perdite economiche, umane e ambientali.

Sistema di elaborazione dei dati

Un sistema il cui scopo è elaborare grandi quantità di dati strutturati. Questi sistemi in genere elaborano dati in sequenza e seguono un modello di processi input-elaborazione-output. Esempi di sistemi di elaborazione dei dati sono i sistemi di tariffazione e fatturazione e i sistemi di pagamento.

Sistema di elaborazione dei linguaggi

Un sistema che traduce un linguaggio in un altro. Per esempio, un compilatore è un sistema di elaborazione dei linguaggi che traduce il codice sorgente di un programma in un codice oggetto.

Sistema di elaborazione delle transazioni

Un sistema che garantisce che le transazioni siano elaborate in modo tale che non interferiscono tra loro e che il malfunzionamento di una transazione non influisca sulle altre transazioni o sui dati del sistema.

Sistema di sistemi

Un sistema creato integrando due o più sistemi esistenti.

Sistema distribuito

Un sistema software i cui sottosistemi o componenti software vengono eseguiti su processori differenti.

Sistema ereditato

Un sistema socio-tecnico che è utile o essenziale per un'azienda, ma che è stato sviluppato usando tecnologie o metodi obsoleti. Poiché i sistemi ereditati spesso eseguono funzioni aziendali critiche, devono essere mantenuti.

Sistema ERP (Enterprise Resource Planning)

Un sistema software su larga scala che include un certo numero di funzionalità che supportano le tipiche operazioni svolte nelle aziende e che offrono gli strumenti per condividere le informazioni. Per esempio, un sistema ERP può includere un supporto per la gestione, la

fabbricazione e la distribuzione di una catena di rifornimenti. Un sistema ERP viene configurato in base alle specifiche esigenze dell'azienda che lo utilizza.

Sistema iLearn

Ambiente di apprendimento digitale a supporto dell'apprendimento scolastico. Utilizzato come caso di studio in questo libro.

Sistema in tempo reale

Un sistema che riconosce ed elabora gli eventi esterni in “tempo reale”. La correttezza di un sistema in tempo reale non dipende semplicemente da ciò che fa, ma anche dalla rapidità con cui lo fa. I sistemi in tempo reale di solito sono organizzati come un insieme di processi contemporanei.

Sistema integrato

Un sistema software che è integrato in un’unità hardware; per esempio, il software installato in un telefono cellulare. I sistemi software di solito sono sistemi in tempo reale e quindi devono rispondere in modo tempestivo agli eventi che si verificano nel loro ambiente.

Sistema peer-to-peer

Un sistema distribuito dove non c’è distinzione tra client e server. I computer nel sistema possono operare sia come cliente sia come server. Le applicazioni peer-to-peer includono la condivisione dei file, la messaggistica istantanea e i sistemi di supporto alla cooperazione.

Sistema per una stazione meteorologica

Un sistema che riceve i dati sulle condizioni atmosferiche da postazioni remote. È utilizzato come caso di studio in questo libro.

Sistema socio-tecnico

Un sistema, inclusi i componenti hardware e software, che ha processi operativi definiti seguiti da operatori umani e che funziona all’interno di un’organizzazione. È dunque influenzato da politiche, procedure e strutture aziendali.

Sistemi di controllo delle versioni

Strumenti software che sono stati sviluppati per supportare i processi di controllo delle versioni. Possono utilizzare repository centralizzati o distribuiti.

Software come servizio

Applicazioni software il cui accesso avviene in modo remoto tramite un browser web, anziché tramite l’installazione sui computer locali. L’impiego di questo tipo di software è in continua crescita perché permette di offrire i servizi direttamente agli utenti finali.

SQL (Structured Query Language)

Un linguaggio standard per la programmazione di database relazionali.

Stile architettonurale

Descrizione astratta di un’architettura software che è stata provata in un certo numero di sistemi software. La descrizione include le informazioni sulle applicazioni appropriate dello stile architettonurale e sull’organizzazione dei componenti dell’architettura.

Storia utente

Una descrizione nel linguaggio naturale di una situazione che spiega come un sistema potrebbe essere utilizzato e le interazioni con altri sistemi.

Strumento CASE

Uno strumento software, come un editor di progetti o un debugger di programmi, utilizzato per supportare un’attività nel processo di sviluppo del software.

Subversion

Uno strumento open-source largamente utilizzato per la costruzione dei sistemi e il controllo delle versioni; disponibile per varie piattaforme.

Sviluppo guidato da modelli

Un approccio allo sviluppo del software basato su modelli di sistemi che sono espressi in UML, anziché nel codice del linguaggio di programmazione. Questo consente all'architettura guidata dai modelli di considerare le attività diverse dallo sviluppo, quali l'ingegneria e il test dei requisiti.

Sviluppo guidato dai test

Approccio allo sviluppo del software dove i test eseguibili sono scritti prima del codice del programma. I test vengono eseguiti automaticamente dopo ogni modifica del programma.

Sviluppo host-target

Modalità di sviluppo secondo la quale il software viene sviluppato su un computer diverso da quello in cui sarà eseguito. È l'approccio normale allo sviluppo di sistemi software integrati e mobili.

Sviluppo incrementale

Un approccio allo sviluppo dove il software viene consegnato e installato in successivi incrementi.

Sviluppo iterativo

Un approccio allo sviluppo del software nel quale i processi di specifica, progettazione, programmazione e test sono intrecciati.

Sviluppo orientato agli oggetti

Un approccio allo sviluppo del software nel quale le astrazioni fondamentali nel sistema sono oggetti indipendenti. Lo stesso tipo di astrazione è utilizzato durante la specifica, la progettazione e lo sviluppo del software.

Test a scatola bianca (white-box test)

Un approccio ai test dei programmi dove i test si basano sulla conoscenza della struttura di un programma e dei suoi componenti. L'accesso al codice sorgente è essenziale per eseguire i test a scatola bianca.

Test a scatola nera (black-box test)

Metodi di test dove chi effettua i test non ha accesso al codice sorgente di un sistema o dei suoi componenti. I test sono derivati dalla specifica del sistema.

Test delle unità

I test di singole unità di programma effettuati da uno sviluppatore del software o dal team di sviluppo.

Test dello scenario

Un approccio al test del software dove i casi di test sono derivati da uno scenario d'uso del sistema.

Test di accettazione

Test effettuati per decidere se il sistema software è adeguato a soddisfare le esigenze dei clienti e se quindi può essere accettato da un fornitore.

Test di un sistema

L'insieme delle prove da effettuare su un sistema prima di consegnarlo ai clienti.

Tipo di dato astratto

Un tipo di dato la cui rappresentazione è definita dalle sue operazioni, anziché dalla sua rappresentazione. La rappresentazione è privata e l'accesso ad essa è consentito soltanto alle operazioni definite.

Transazione

Unità di interazione con un sistema di computer. Le transazioni sono indipendenti e indivisibili (non possono essere suddivise in parti più piccole) e sono unità fondamentali di recupero, consistenza e simultaneità.

UML (Unified Modeling Language)

Un linguaggio grafico utilizzato nello sviluppo orientato agli oggetti che include vari tipi di modelli che forniscono prospettive differenti di un sistema software. UML è diventato de facto lo standard per la modellazione orientata agli oggetti.

Verifica

Il processo che verifica se un sistema soddisfa la sua specifica.

Verifica dei modelli

Metodo di verifica statica dove un modello di stati di un sistema viene integralmente analizzato con lo scopo di individuare stati irraggiungibili.

Vista architetturale

Descrizione di un'architettura software da un particolare punto di vista.

Wicked problem

Un problema che non può essere completamente specificato o capito a causa della complessità delle interazioni tra gli elementi che generano il problema.

Workbench CASE

Insieme integrato di strumenti CASE che lavorano insieme per supportare un'attività di processo, come la progettazione del software o la gestione della configurazione. Oggi è anche detto ambiente di programmazione.

Workflow

Definizione dettagliata di un processo aziendale che ha lo scopo di svolgere un determinato compito. Il workflow di solito viene rappresentato graficamente per indicare le singole attività di un processo e le informazioni che vengono prodotte e utilizzate in ciascuna attività.

WSDL (Web Service Definition Language)

Una notazione basata su XML per definire l'interfaccia dei servizi web.

XML (Extended Markup Language)

Linguaggio di markup testuale che supporta lo scambio di dati strutturati. Ogni campo di dati è delimitato da tag che forniscono informazioni sui campi. Attualmente, XML è largamente utilizzato ed è diventato la base dei protocolli per i servizi web.

XP

Si veda Programmazione estrema.

Z

Un linguaggio di specifiche formali, basato sui modelli, sviluppato dall'Università di Oxford, in Inghilterra.

Bibliografia

Capitolo 1

- Bott, F. 2005. *Professional Issues in Information Technology*. Swindon, UK: British Computer Society.
- Duquenoy, P. 2007. *Ethical, Legal and Professional Issues in Computing*. London: Thomson Learning.
- Freeman, A. 2011. *The Definitive Guide to HTML5*. New York: Apress.
- Gotterbarn, D., K. Miller e S. Rogerson. 1999. "Software Engineering Code of Ethics Is Approved." *Comm. ACM* 42 (10): 102-107. doi:10.1109/MC.1999.796142.
- Holdener, A. T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.
- Jacobson, I., P-W. Ng, P. E. McMahon, I. Spence e S. Lidman. 2013. *The Essence of Software Engineering*. Boston: Addison-Wesley.
- Johnson, D. G. 2001. *Computer Ethics*. Englewood Cliffs, NJ: Prentice-Hall.
- Laudon, K. 1995. "Ethical Concepts and Information Technology." *Comm. ACM* 38 (12): 33-39. doi:10.1145/219663.219677.
- Naur, P. e Randell, B. 1969. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Brussels. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf>
- Tavani, H. T. 2013. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing*, 4th ed. New York: John Wiley & Sons.
- Vogel, L. 2012. *Eclipse 4 Application Development: The Complete Guide to Eclipse 4 RCP Development*. Sebastopol, CA: O'Reilly & Associates.

Capitolo 2

- Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.
2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Boehm, B. W. (1988). "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, 21 (5), 61-72. doi:10.1145/12944.12948
- Boehm, B. W. e R. Turner. 2004. "Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods." In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.

- Chrassis, M. B., M. Konrad e S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Boston: Addison-Wesley.
- Humphrey, W. S. 1988. "Characterizing the Software Process: A Maturity Framework." *IEEE Software* 5 (2): 73-79. doi:10.1109/2.59.
- Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.
- Krutch, P. 2003. *The Rational Unified Process – An Introduction*, 3rd ed. Reading, MA: Addison-Wesley.
- Royce, W. W. 1970. "Managing the Development of Large Software Systems: Concepts and Techniques." In *IEEE WESTCON*, 1-9. Los Angeles, CA.
- Wheeler, W. e J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

Capitolo 3

- Ambler, S. W. 2010. "Scaling Agile: A Executive Guide." http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling_agile_an_executive_guide10/
- Arisholm, E., H. Gallis, T. Dyba e D. I. K. Sjoberg. 2007. "Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise." *IEEE Trans. on Software Eng.* 33 (2): 65-86. doi:10.1109/TSE.2007.17.
- Beck, K. 1998. "Chrysler Goes to 'Extremes.'" *Distributed Computing* (10): 24-28.
1999. "Embracing Change with Extreme Programming." *IEEE Computer* 32 (10): 70-78. doi:10.1109/2.796139.
- Bellouiti, S. 2009. "How Scrum Helped Our A-Team." <http://www.scrumalliance.org/community/articles/2009/2009-june/how-scrum-helped-our-team>
- Bird, J. 2011. "You Can't Be Agile in Maintenance." <http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html>
- Deemer, P. 2011. "The Distributed Scrum Primer." <http://www.goodagile.com/distributedscrumprimer/>.
- Fowler, M., K. Beck, J. Brant, W. Opdyke e D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Hopkins, R. e K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Jeffries, R. e G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24-30. doi:10.1109/MS.2007.75.
- Kilner, S. 2012. "Can Agile Methods Work for Software Maintenance." <http://www.vlegaci.com/can agile-methods-work-for-software-maintenance-part-1/>
- Larman, C. e V. R. Basili. 2003. "Iterative and Incremental Development: A Brief History." *IEEE Computer* 36 (6): 47-56. doi:10.1109/MC.2003.1204375.
- Leffingwell, D. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.
- Leffingwell, D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Boston: Addison-Wesley.
- Mulder, M. e M. van Vliet. 2008. "Case Study: Distributed Scrum Project for Dutch Railways." InfoQ. <http://www.infoq.com/articles/dutch-railway-scrum>

- Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.
- Schatz, B. e I. Abdelshafi. 2005. "Primavera Gets Agile: A Successful Transition to Agile Development." *IEEE Software* 22 (3): 36-42. doi:10.1109/MS.2005.74.
- Schwaber, K. e M. Beedle. 2001. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.
- Stapleton, J. 2003. *DSDM: Business Focused Development*, 2nd ed. Harlow, UK: Pearson Education.
- Tahchiev, P., F. Leme, V. Massol e Gregory. 2010. *JUnit in Action*, 2/e. Greenwich, CT: Manning Publications.
- Weinberg, G. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand.
- Williams, L., R. R. Kessler, W. Cunningham e R. Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17 (4): 19-25. doi:10.1109/52.854064.

Capítulo 4

- Crabtree, A. 2003. *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.
- Davis, A. M. 1993. *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice-Hall.
- IBM. 2013. "Rational Doors Next Generation: Requirements Engineering for Complex Systems." <https://jazz.net/products/rational-doors-next-generation/>
- IEEE. 1998. "IEEE Recommended Practice for Software Requirements Specifications." In *IEEE Software Engineering Standards Collection*. Los Alamitos, CA: IEEE Computer Society Press.
- Jacobsen, I., M. Christerson, P. Jonsson e G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.
- Martin, D. e I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59-89. doi:10.1145/972648.972651.
- Rittel, H. e M. Webber. 1973. "Dilemmas in a General Theory of Planning." *Policy Sciences* 4: 155-169. doi:10.1007/BF01405730.
- Robertson, S. e J. Robertson. 2013. *Mastering the Requirements Process*, 3rd ed. Boston: Addison-Wesley.
- Sommerville, I., T. Rodden, P. Sawyer, R. Bentley e M. Twidale. 1993. "Integrating Ethnography into the Requirements Engineering Process." In RE'93, 165-173. San Diego, CA: IEEE Computer Society Press. doi:10.1109/ISRE.1993.324821.
- Stevens, P. e R. Pooley. 2006. *Using UML: Software Engineering with Objects and Components*, 2nd ed. Harlow, UK: Addison-Wesley.
- Suchman, L. 1983. "Office Procedures as Practical Action: Models of Work and System Design." *ACM Transactions on Office Information Systems* 1 (3): 320-328. doi:10.1145/357442.357445.
- Viller, S. e I. Sommerville. 2000. "Ethnographically Informed Analysis for Software Engineers." *Int. J. of Human-Computer Studies* 53 (1): 169-196. doi:10.1006/ijhc.2000.0370.

Capítulo 5

- Amblar, S. W. 2004. *The Object Primer: Agile Model-Driven Development with UML 2.0*, 3rd ed. Cambridge, UK: Cambridge University Press.

- Ambler, S. W. e R. Jeffries. 2002. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.
- Booch, G., J. Rumbaugh e I. Jacobson. 2005. *The Unified Modeling Language User Guide*, 2nd ed. Boston: Addison-Wesley.
- Brambilla, M., J. Cabot e M. Wimmer. 2012. *Model-Driven Software Engineering in Practice*. San Rafael, CA: Morgan Claypool.
- Den Haan, J. 2011. "Why There Is No Future for Model Driven Development." <http://www.theenterprisearchitect.eu/archive/2011/01/25/why-there-is-no-future-for-model-driven-development/>
- Erickson, J. e K Siau. 2007. "Theoretical and Practical Complexity of Modeling Methods." *Comm. ACM* 50 (8): 46-51. doi:10.1145/1278201.1278205.
- Harel, D. 1987. "Statecharts: A Visual Formalism for Complex Systems." *Sci. Comput. Programming* 8 (3): 231-274. doi:10.1016/0167-6423(87)90035-9.
- Hull, R. e R King. 1987. "Semantic Database Modeling: Survey, Applications and Research Issues." *ACM Computing Surveys* 19 (3): 201-260. doi:10.1145/45072.45073.
- Hutchinson, J., M. Rouncefield e J. Whittle. 2012. "Model-Driven Engineering Practices in Industry." In *34th Int. Conf. on Software Engineering*, 633-642. doi:10.1145/1985793.1985882.
- Jacobsen, I., M. Christerson, P. Jonsson e G. Overgaard. 1993. *Object-Oriented Software Engineering*. Wokingham, UK: Addison-Wesley.
- Koegel, M. 2012. "EMF Tutorial: What Every Eclipse Developer Should Know about EMF." <http://eclipsesource.com/blogs/tutorials/emf-tutorial/>
- Mellor, S. J. e M. J. Balcer. 2002. *Executable UML*. Boston: Addison-Wesley.
- Mellor, S. J., K. Scott e D. Weise. 2004. *MDA Distilled: Principles of Model-Driven Architecture*. Boston: Addison-Wesley.
- OMG. 2012. "Model-Driven Architecture: Success Stories." http://www.omg.org/mda/products_success.htm
- Rumbaugh, J., I. Jacobson e G Booch. 2004. *The Unified Modelling Language Reference Manual*, 2nd ed. Boston: Addison-Wesley.
- Stahl, T. e M. Voelter. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.
- Zhang, Y. e S. Patel. 2011. "Agile Model-Driven Development in Practice." *IEEE Software* 28 (2): 84-91. doi:10.1109/MS.2010.85.

Capitolo 6

- Bass, L., P. Clements e R. Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Boston: Addison-Wesley.
- Berczuk, S. P. e B. Appleton. 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley.
- Booch, G. 2014. "Handbook of Software Architecture." <http://handbookofsoftwarearchitecture.com/>
- Bosch, J. 2000. *Design and Use of Software Architectures*. Harlow, UK: Addison-Wesley.

- Buschmann, F., K. Henney e D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., R. Meunier, H. Rohnert e P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Chen, L., M. Ali Babar e B. Nuseibeh. 2013. "Characterizing Architecturally Significant Requirements." *IEEE Software* 30 (2): 38-45. doi:10.1109/MS.2012.174.
- Coplien, J. O. e N. B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice-Hall.
- Gamma, E., R. Helm, R. Johnson e J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Garlan, D. e M. Shaw. 1993. "An Introduction to Software Architecture." In *Advances in Software Engineering and Knowledge Engineering*, pubblicato da V. Ambriola e G. Tortora, 2:1-39. London: World Scientific Publishing Co.
- Hofmeister, C., R. Nord e D. Soni. 2000. *Applied Software Architecture*. Boston: Addison-Wesley.
- Kircher, M. e P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.
- Krutch, P. 1995. "The 4+1 View Model of Software Architecture." *IEEE Software* 12 (6): 42-50. doi:10.1109/52.469759.
- Lange, C. F. J., M. R. V. Chaudron e J. Muskens. 2006. "UML Software Architecture and Design Description." *IEEE Software* 23 (2): 40-46. doi:10.1109/MS.2006.50.
- Lewis, P. M., A. J. Bernstein e M. Kifer. 2003. *Databases and Transaction Processing: An Application-Oriented Approach*. Boston: Addison-Wesley.
- Martin, D. e I. Sommerville. 2004. "Patterns of Cooperative Interaction: Linking Ethnomethodology and Design." *ACM Transactions on Computer-Human Interaction* 11 (1) (March 1): 59-89. doi:10.1145/972648.972651.
- Nii, H. P. 1986. "Blackboard Systems, Parts 1 and 2." *AI Magazine* 7 (2 and 3): 38-53 e 62-69. <http://www.aaai.org/ojs/index.php/aimagazine/article/view/537/473>
- Schmidt, D., M. Stal, H. Rohnert e F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.
- Shaw, M. e D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall.
- Usability Group. 1998. "Usability Patterns". University of Brighton. <http://www.it.bton.ac.uk/Research/patterns/home.html>

Capitolo 7

- Abbott, R. 1983. "Program Design by Informal English Descriptions." *Comm. ACM* 26 (11): 882-894. doi:10.1145/182.358441.
- Alexander, C. 1979. *A Timeless Way of Building*. Oxford, UK: Oxford University Press.
- Baytersdorfer, M. 2007. "Managing a Project with Open Source Components." *ACM Interactions* 14 (6): 33-34. doi:10.1145/1300655.1300677.

- Beck, K. e W. Cunningham. 1989. "A Laboratory for Teaching Object-Oriented Thinking." In Proc. OOPSLA'89 (Conference on Object-Oriented Programming, Systems, Languages and Applications), 1-6. ACM Press. doi:10.1145/74878.74879.
- Buschmann, F., K. Henney e D. C. Schmidt. 2007a. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- 2007b. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., R. Meunier, H. Rohnert e P. Sommerlad. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Chapman, C. 2010. "A Short Guide to Open-Source and Similar Licences." *Smashing Magazine*. <http://www.smashingmagazine.com/2010/03/24/a-short-guide-to-open-source-and-similar-licenses/>.
- Gamma, E., R. Helm, R. Johnson e J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA.: Addison-Wesley.
- Kircher, M. e P. Jain. 2004. *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.
- Loeliger, J. e M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly & Associates.
- Pilato, C., B. Collins-Sussman e B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly & Associates.
- Raymond, E. S. 2001. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol. CA: O'Reilly & Associates.
- Schmidt, D., M. Stal, H. Rohnert e F. Buschmann. 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.
- St. Laurent, A. 2004. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly & Associates.
- Vogel, L. 2013. *Eclipse IDE: A Tutorial*. Hamburg, Germany: Vogella Gmbh.
- Wirfs-Brock, R., B. Wilkerson e L. Weiner. 1990. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice-Hall.

Capitolo 8

- Andrea, J. 2007. "Envisioning the Next Generation of Functional Testing Tools." *IEEE Software* 24 (3): 58-65. doi:10.1109/MS.2007.73.
- Beck, K. 2002. *Test Driven Development: By Example*. Boston: Addison-Wesley.
- Bezier, B. 1990. *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold.
- Boehm, B. W. 1979. "Software Engineering; R & D Trends and Defense Needs." In *Research Directions in Software Technology*, pubblicato da P. Wegner, 1-9. Cambridge, MA: MIT Press.
- Cusamano, M. e R. W. Selby. 1998. *Microsoft Secrets*. New York: Simon & Schuster.
- Dijkstra, E. W. 1972. "The Humble Programmer." *Comm. ACM* 15 (10): 859-866. doi:10.1145/355604.361591.
- Fagan, M. E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems J.* 15 (3): 182-211.

- Jeffries, R. e G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24-30. doi:10.1109/MS.2007.75.
- Kaner, C. 2003. "An Introduction to Scenario Testing." *Software Testing and Quality Engineering* (October 2003).
- Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In RE' 93, 126-133. San Diego CA: IEEE. doi:0.1109/ISRE.1993.324825.
- Martin, R. C. 2007. "Professionalism and Test-Driven Development." *IEEE Software* 24 (3): 32-36. doi:10.1109/MS.2007.85.
- Powell, S. J., C. J. Trammell, R. C. Linger e J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.
- Tahchiev, P., F. Leme, V. Massol e G. Gregory. 2010. *JUnit in Action*, 2nd ed. Greenwich, CT: Manning Publications.
- Whittaker, J. A. 2009. *Exploratory Software Testing*. Boston: Addison-Wesley.

Capítulo 9

- Banker, R. D., S. M. Datar, C. F. Kemerer e D. Zweig. 1993. "Software Complexity and Maintenance Costs." *Comm. ACM* 36 (11): 81-94. doi:10.1145/163359.163375.
- Coleman, D., D. Ash, B. Lowther e P. Oman. 1994. "Using Metrics to Evaluate Software System Maintainability." *IEEE Computer* 27 (8): 44-49. doi:10.1109/2.303623.
- Davidson, M. G. e J. Krogstie. 2010. "A Longitudinal Study of Development and Maintenance." *Information and Software Technology* 52 (7): 707-719. doi:10.1016/j.infsof.2010.03.003.
- Erlikh, L. 2000. "Leveraging Legacy System Dollars for E-Business." *IT Professional* 2 (3 (May/June 2000)): 17-23. doi:10.1109/6294.846201.
- Fowler, M., K. Beck, J. Brant, W. Opdyke e D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Hopkins, R. e K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Jones, T. C. 2006. "The Economics of Software Maintenance in the 21st Century." www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf.
- Kerievsky, J. 2004. *Refactoring to Patterns*. Boston: Addison-Wesley.
- Kozlov, D., J. Koskinen, M. Sakkinen e J. Markkula. 2008. "Assessing Maintainability Change over Multiple Software Releases." *J. of Software Maintenance and Evolution* 20 (1): 31-58. doi:10.1002/smrv.361.
- Lientz, B. P. e E. B. Swanson. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.
- Mitchell, R. M. 2012. "COBOL on the Mainframe: Does It Have a Future?" *Computerworld US*. <http://features.techworld.com/applications/3344704/cobol-on-the-mainframe-does-it-have-a-future/>
- O'Hanlon, C. 2006. "A Conversation with Werner Vogels." *ACM Queue* 4 (4): 14-22. doi:10.1145/1142055.1142065.
- Rajlich, V. T. e K. H. Bennett. 2000. "A Staged Model for the Software Life Cycle." *IEEE Computer* 33 (7): 66-71. doi:10.1109/2.869374.

Ulrich, W. M. 1990. "The Evolutionary Growth of Software Reengineering and the Decade Ahead." *American Programmer* 3 (10): 14-20.

Warren, I. (ed.). 1998. *The Renaissance of Legacy Systems*. London: Springer.

Capitolo e10

Abrial, J. R. 2009. "Faultless Systems: Yes We Can." *IEEE Computer* 42 (9): 30-36. doi:10.1109/MC.2009.283.

2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.

Avizienis, A., J. C. Laprie, B. Randell e C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. on Dependable and Secure Computing* 1 (1): 11-33. doi:10.1109/TDSC.2004.2.

Badeau, F. e A. Amelot. 2005. "Using B as a High Level Programming Language in an Industrial Project: Roissy VAL." In *Proc. ZB 2005: Formal Specification and Development in Z* and B. Guildford, UK: Springer. doi:10.1007/11415787_20.

Ball, T., E. Bouimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusk, S. K. Rajamani e A. Ustuner. 2006. "Thorough Static Analysis of Device Drivers." In *Proc. EuroSys 2006*. Leuven, Belgium. doi:10.1145/1218063.1217943.

Bochot, T., P. Virelizier, H. Waeselynck e V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18-27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.

Calinescu, R. C. e M. Z. Kwiatkowska. 2009. "Using Quantitative Analysis to Implement Autonomic IT Systems." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 100-10. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE.2009.5070512.

Douglass, B. 2013. "Agile Analysis Practices for Safety-Critical Software Development." <http://www.ibm.com/developerworks/rational/library/agile-analysis-practices-safety-critical-development/>.

Hall, A. e R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18-25. doi:10.1109/52.976937.

Jhala, R. e R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4), Article 21. doi:1145/1592434.1592438.

Miller, S. P., E. A. Anderson, L. G. Wagner, M. W. Whalen e M. P. E. Heimdahl. 2005. "Formal Verification of Flight Critical Software." In *Proc. AIAA Guidance, Navigation and Control Conference*. San Francisco. doi:10.2514/6.2005-6431.

Parnas, D. 2010. "Really Rethinking Formal Methods." *IEEE Computer* 43 (1): 28-34. doi:10.1109/MC.2010.22.

Parnas, D., J. van Schouwen e P. K. Shu. 1990. "Evaluation of Safety-Critical Software." *Comm. ACM* 33 (6): 636-651. doi:10.1145/78973.78974.

Swartz, A. J. 1996. "Airport 95: Automated Baggage System?" *ACM Software Engineering Notes* 21 (2): 79-83. doi:10.1145/227531.227544.

Trimble, J. 2012. "Agile Development Methods for Space Operations." In *SpaceOps 2012*. Stockholm. doi:10.2514/6.2012-1264554.

Woodcock, J., P. G. Larsen, J. Bicarregui e J. Fitzgerald. 2009. "Formal Methods: Practice and Experience." *Computing Surveys* 41 (4): 1-36. doi:10.1145/1592434.1592436.

Capitolo e11

- Avizienis, A. A. 1995. "A Methodology of N-Version Programming." In *Software Fault Tolerance*, a cura di M. R. Lyu, 23-46. Chichester, UK: John Wiley & Sons.
- Brilliant, S. S., J. C. Knight e N. G. Leveson. 1990. "Analysis of Faults in an N-Version Software Experiment." *IEEE Trans. On Software Engineering* 16 (2): 238-247. doi:10.1109/32.44387.
- Hatton, L. 1997. "N-Version Design Versus One Good Version." *IEEE Software* 14 (6): 71-76. doi:10.1109/52.636672.
- Leveson, N. G. 1995. *Safeware: System Safety and Computers*. Reading, MA: Addison-Wesley.
- Musa, J. D. 1998. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.
- Prowell, S. J., C. J. Trammell, R. C. Linger e J. H. Poore. 1999. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley.
- Pullum, L. 2001. *Software Fault Tolerance Techniques and Implementation*. Norwood, MA: Artech House.
- Randell, B. 2000. "Facing Up To Faults." *Computer J.* 45 (2): 95-106. doi:10.1093/comjnl/43.2.95.
- Torres-Pomales, W. 2000. "Software Fault Tolerance: A Tutorial." NASA. http://ntrs.nasa.gov/archive/nasa/casi/20000120144_2000175863.pdf
- Voas, J. e G. McGraw. 1997. *Software Fault Injection: Innoculating Programs Against Errors*. New York: John Wiley & Sons.

Capitolo e12

- Abrial, J. R. 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Ball, T., V. Levin e S. K. Rajamani. 2011. "A Decade of Software Model Checking with SLAM." *Communications of the ACM* 54 (7) (July 1): 68. doi:10.1145/1965724.1965743.
- Behm, P., P. Benoit, A. Faivre e J-M. Meynadier. 1999. "Meteor: A Successful Application of B in a Large Project." In *Formal Methods' 99*, 369-387. Berlin: Springer-Verlag. doi:10.1007/3-540-48119-2_22.
- Bishop, P. e R. E. Bloomfield. 1998. "A Methodology for Safety Case Development." In *Proc. Safety-Critical Systems Symposium*. Birmingham, UK: Springer. <http://www.adelard.com/papers/ss98web.pdf>
- Bochot, T., P. Virelizier, H. Waeselynck e V. Wiels. 2009. "Model Checking Flight Control Systems: The Airbus Experience." In *Proc. 31st International Conf. on Software Engineering, Companion Volume*, 18-27. Leipzig: IEEE Computer Society Press. doi:10.1109/ICSE-COMPANION.2009.5070960.
- Dehbonei, B. e F. Mejia. 1995. "Formal Development of Safety-Critical Software Systems in Railway Signalling." In *Applications of Formal Methods*, a cura di M. Hinckey e J. P. Bowen, 227-252. London: Prentice-Hall.
- Graydon, P. J., J. C. Knight e E. A. Strunk. 2007. "Assurance Based Development of Critical Systems." In *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, 347-357. Edinburgh, Scotland. doi:10.1109/DSN.2007.17.
- Holzmann, G. J. 2014. "Mars Code." *Comm ACM* 57 (2): 64-73. doi:10.1145/2560217.2560218.
- Jhala, R. e R. Majumdar. 2009. "Software Model Checking." *Computing Surveys* 41 (4). doi:10.1145/1592434.1592438.

- Kwiatkowska, M., G. Norman e D. Parker. 2011. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In *Proc. 23rd Int. Conf. on Computer Aided Verification*, 585-591. Snowbird, UT: Springer-Verlag. doi:10.1007/978-3-642-22110-1_47.
- Leveson, N. G., S. S. Cha e T. J. Shimeall. 1991. "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software* 8 (4): 48-59. doi:10.1109/52.300036.
- Lopes, R., D. Vicente e N. Silva. 2009. "Static Analysis Tools, a Practical Approach for Safety-Critical Software Verification." In *Proceedings of DASIA 2009 Data Systems in Aerospace*. Noordwijk, Netherlands: European Space Agency.
- Lutz, R. R. 1993. "Analysing Software Requirements Errors in Safety-Critical Embedded Systems." In *RE' 93*, 126-133. San Diego, CA: IEEE. doi:0.1109/ISRE.1993.324825.
- Moy, Y., E. Ledinot, H. Delseny, V. Wiels e B. Monate. 2013. "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience." *IEEE Software* 30 (3) (May 1): 50-57. doi:10.1109/MS.2013.43.
- Perrow, C. 1984. *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.
- Regan, P. e S. Hamilton. 2004. "NASA's Mission Reliable." *IEEE Computer* 37 (1): 59-68. doi:10.1109/MC.2004.1260727.
- Schneider, S. 1999. *Concurrent and Real-Time Systems: The CSP Approach*. Chichester, UK: John Wiley & Sons.
- Souyris, J., V. Weils, D. Delmas e H. Delseny. 2009. "Formal Verification of Avionics Software Products." In *Formal Methods' 09: Proceedings of the 2nd World Congress on Formal Methods*, 532-546. Springer-Verlag. doi:10.1007/978-3-642-05089-3_34.
- Storey, N. 1996. *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.
- Veras, P. C., E. Villani, A. M. Ambrosio, N. Silva, M. Vieira e H. Madeira. 2010. "Errors in Space Software Requirements: A Field Study and Application Scenarios." In *21st Int. Symp. on Software Reliability Engineering*. San Jose, CA. doi:10.1109/ISSRE.2010.37.
- Zheng, J., L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl e M. A. Vouk. 2006. "On the Value of Static Analysis for Fault Detection in Software." *IEEE Trans. on Software Eng.* 32 (4): 240-253. doi:10.1109/TSE.2006.38.

Capitolo e13

- Anderson, R. 2008. *Security Engineering*, 2nd ed. Chichester, UK: John Wiley & Sons.
- Cranor, L. and S. Garfinkel. 2005. *Designing Secure Systems That People Can Use*. Sebastopol, CA: O'Reilly Media Inc.
- Firesmith, D. G. 2003. "Engineering Security Requirements." *Journal of Object Technology* 2 (1): 53-68. http://www.jot.fm/issues/issue_2003_01/column6
- Hall, A. e R. Chapman. 2002. "Correctness by Construction: Developing a Commercially Secure System." *IEEE Software* 19 (1): 18-25. doi:10.1109/52.976937.
- Hewlett-Packard. 2012. "Securing Your Enterprise Software: Hp Fortify Code Analyzer." <http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-2455ENW&cc=us&lc=en>
- Jenney, P. 2013. "Static Analysis Strategies: Success with Code Scanning." <http://msdn.microsoft.com/en-us/security/gg615593.aspx>
- Lane, A. 2010. "Agile Development and Security." <https://securosis.com/blog/agile-development-and-security>

- Pfleeger, C. P. e S. L. Pfleeger. 2007. *Security in Computing*, 4th ed. Boston: Addison-Wesley.
- Safecode. 2012. "Practical Security Stories and Security Tasks for Agile Development Environments." http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf
- Schneier, B. 1999. "Attack Trees." *Dr Dobbs Journal* 24 (12): 1-9. <https://www.schneier.com/paperattacktrees-ddj-ft.html>
2000. *Secrets and Lies: Digital Security in a Networked World*. New York: John Wiley & Sons.
- Schoenfield, B. 2013. "Agile and Security: Enemies for Life?" <http://brookschoenfield.com/?p=151>
- Sindre, G. e A. L. Opdahl. 2005. "Eliciting Security Requirements through Misuse Cases." *Requirements Engineering* 10 (1): 34-44. doi:10.1007/s00766-004-0194-4.
- Spafford, E. 1989. "The Internet Worm: Crisis and Aftermath." *Comm ACM* 32 (6): 678-687. doi:10.1145/63526.63527.
- Stallings, W. e L. Brown. 2012. *Computer Security: Principles and Practice*. (2nd ed.) Boston: Addison-Wesley.
- Viega, J. e G. McGraw. 2001. *Building Secure Software*. Boston: Addison-Wesley.
- Wheeler, D. A. 2004. *Secure Programming for Linux and Unix*. Self-published. <http://www.dwheeler.com/secure-programs/>

Capitolo e14

- Ellison, R. J., R. C. Linger, T. Longstaff e N. R. Mead. 1999. "Survivable Network System Analysis: A Case Study." *IEEE Software* 16 (4): 70-77. doi:10.1109/52.776952.
- Ellison, R. J., R. C. Linger, H. Lipson, N. R. Mead e A. Moore. 2002. "Foundations of Survivable Systems Engineering." *Crosstalk: The Journal of Defense Software Engineering* 12: 10-15. http://resources.sei.cmu.edu/asset_files/WhitePaper/2002_019_001_77700.pdf
- Hollnagel, E. 2006. "Resilience—the Challenge of the Unstable." In *Resilience Engineering: Concepts and Precepts*, a cura di E. Hollnagel, D. D. Woods e N.G. Leveson, 9-18.
2010. "RAG – The Resilience Analysis Grid." In *Resilience Engineering in Practice*, a cura di E. Hollnagel, J. Paries, D. Woods e J. Wreathall, 275-295. Farnham, UK: Ashgate Publishing Group.
- InfoSecurity. 2013. "Global Cybercrime, Espionage Costs \$100-\$500 Billion Per Year." <http://www.infosecurity-magazine.com/view/33569/global-cybercrime-espionage-costs-100500-billion-per-year>
- Laprie, J-C. 2008. "From Dependability to Resilience." In *38th Int. Conf. on Dependable Systems and Networks*. Anchorage, Alaska. http://2008.dsn.org/fastabs/dsn08fastabs_laprie.pdf
- Reason, J. 2000. "Human Error: Models and Management." *British Medical J.* 320: 768-770. doi:10.1136/bmjj.320.7237.768.

Capitolo 15

- Baumer, D., G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle e H. Zullighoven. 1997. "Framework Development for Large Systems." *Comm. ACM* 40 (10): 52-59. doi:10.1145/262793.262804.
- Boehm, B. e C. Abts. 1999. "COTS Integration: Plug and Pray?" *Computer* 32 (1): 135-138. doi:10.1109/2.738311.
- Fayad, M.E. e D.C. Schmidt. 1997. "Object-Oriented Application Frameworks." *Comm. ACM* 40 (10): 32-38. doi:10.1145/262793.262798.
- Gamma, E., R. Helm, R. Johnson e J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

- Garlan, D., R. Allen e J. Ockerbloom. 1995. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software* 12 (6): 17-26. doi:10.1109/52.469757.
2009. "Architectural Mismatch: Why Reuse Is Still so Hard." *IEEE Software* 26 (4): 66-69. doi:10.1109/MS.2009.86.
- Holdener, A.T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.
- Jacobsen, I., M. Griss e P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.
- Monk, E. e B. Wagner. 2013. *Concepts in Enterprise Resource Planning*, 4th ed. Independence, KY: CENGAGE Learning.
- Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Schmidt, D. C., A. Gokhale e B. Natarajan. 2004. "Leveraging Application Frameworks." *ACM Queue* 2 (5 (July/August)): 66-75. doi:10.1145/1016998.1017005.
- Scott, J. E. 1999. "The FoxMeyer Drug's Bankruptcy: Was It a Failure of ERP." In *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*. Milwaukee, WI. <http://www.uta.edu/faculty/weltman/OPMA5364TW/FoxMeyer.pdf>
- Torchiano, M. e M. Morisio. 2004. "Overlooked Aspects of COTS-Based Development." *IEEE Software* 21 (2): 88-93. doi:10.1109/MS.2004.1270770.

Capitolo 16

- Councill, W. T. e G. T. Heineman. 2001. "Definition of a Software Component and Its Elements." In *Component-Based Software Engineering*, a cura di G. T. Heineman e W. T. Councill, 5-20. Boston: Addison-Wesley.
- Jacobsen, I., M. Griss e P. Jonsson. 1997. *Software Reuse*. Reading, MA: Addison-Wesley.
- Kotonya, G. 2003. "The CBSE Process: Issues and Future Visions." In *2nd CBSEnet Workshop*. Budapest, Hungary. <http://miro.sztaki.hu/projects/cbsenet/budapest/presentations/Gerald-CBSEProcess.ppt>
- Lau, K-K. e Z. Wang. 2007. "Software Component Models." *IEEE Trans. on Software Eng.* 33 (10): 709-724. doi:10.1109/TSE.2007.70726.
- Meyer, B. 1992. "Applying Design by Contract." *IEEE Computer* 25 (10): 40-51. doi:10.1109/2.161279.
2003. "The Grand Challenge of Trusted Components." In *Proc. 25th Int. Conf. on Software Engineering*. Portland, OR: IEEE Press. doi:10.1109/ICSE.2003.1201252.
- Mili, H., A. Mili, S. Yacoub e E. Addy. 2002. *Reuse-Based Software Engineering*. New York: John Wiley & Sons.
- Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.
- Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Harlow, UK: Addison-Wesley.
- Warmer, J. e A. Kleppe. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Boston: Addison-Wesley.
- Weinreich, R. e J. Sametinger. 2001. "Component Models and Component Services: Concepts and Principles." In *Component-Based Software Engineering*, a cura di G. T. Heineman e W. T. Councill, 33-48. Boston: Addison-Wesley.
- Wheeler, W. e J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.

Capitolo 17

- Bernstein, P. A. 1996. "Middleware: A Model for Distributed System Services." *Comm. ACM* 39 (2): 86-97. doi:10.1145/230798.230809.
- Coulouris, G., J. Dollimore, T. Kindberg e G. Blair. 2011. *Distributed Systems: Concepts and Design*, 5th ed. Harlow, UK: Addison-Wesley.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, CA.: O'Reilly & Associates.
- McDougall, P. 2000. "The Power of Peer-to-Peer." *Information Week* (August 28, 2000). <http://www.informationweek.com/801/peer.htm>
- Oram, A. 2001. "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology." Sebastopol, CA: O'Reilly & Associates.
- Orfali, R., D. Harkey e J. Edwards. 1997. *Instant CORBA*. Chichester, UK: John Wiley & Sons.
- Pope, A. 1997. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.
- Sanderson, D. 2012. *Programming with Google App Engine*. Sebastopol, CA: O'Reilly Media Inc.
- Sarris, S. 2013. *HTML5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Tanenbaum, A. S. e M. Van Steen. 2007. *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall.
- Wallach, D. S. 2003. "A Survey of Peer-to-Peer Security Issues." In *Software Security: Theories and Systems*, a cura di M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda e A. Yonezawa, 42-57. Heidelberg: Springer-Verlag. doi:10.1007/3-540-36532-X_4.

Capitolo 18

- Erl, T. 2004. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice-Hall.
2005. *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice-Hall.
- Fielding, R. 2000. "Representational State Transfer." *Architectural Styles and the Design of Network-Based Software Architecture*. https://www.ics.uci.edu/~fielding/pubs/.../fielding_dissertation.pdf
- Lovelock, C, S Vandermerwe e B Lewis. 1996. *Services Marketing*. Englewood Cliffs, NJ.: Prentice-Hall.
- Newcomer, E. e G. Lomow. 2005. *Understanding SOA with Web Services*. Boston: Addison-Wesley.
- OMG. 2011. "Documents Associated with Business Process Model and Notation (BPMN) Version 2.0." <http://www.omg.org/spec/BPMN/2.0/>
- Pautasso, C., O. Zimmermann e F. Leymann. 2008. "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision." In *Proc. WWW 2008*, 805-14. Beijing, China. doi:10.1145/1367497.1367606.
- Richardson, L. e S. Ruby. 2007. *RESTful Web Services*. Sebastopol, CA: O'Reilly Media Inc.
- Rosenberg, F., F. Curbera, M. Duftler e R. Khalaf. 2008. "Composing RESTful Services and Collaborative Workflows: A Lightweight Approach." *IEEE Internet Computing* 12 (5): 24-31. doi:10.1109/MIC.2008.98.
- W3C. 2012. "OWL 2 Web Ontology Language." <http://www.w3.org/TR/owl2-overview/>
2013. "Web of Services." <http://www.w3.org/standards/webofservices/>

White, S. A. e D. Miers. 2008. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL. USA: Future Strategies Inc.

Capitolo 19

Bass, B. M. e G. Dunteman. 1963. "Behaviour in Groups as a Function of Self, Interaction and Task Orientation." *J. Abnorm. Soc. Psychology*. 66 (4): 19-28. doi:10.1037/h0042764.

Boehm, B. W. 1988. "A Spiral Model of Software Development and Enhancement." *IEEE Computer* 21 (5): 61-72. doi:10.1109/2.59.

Hall, E. 1998. *Managing Risk: Methods for Software Systems Development*. Reading, MA: Addison-Wesley.

Marshall, J. E. e R. Heslin. 1975. "Boys and Girls Together: Sexual Composition and the Effect of Density on Group Size and Cohesiveness." *J. of Personality and Social Psychology* 35 (5): 952-961. doi:10.1037/h0076838.

Maslow, A. A. 1954. *Motivation and Personality*. New York: Harper & Row.

Ould, M. 1999. *Managing Software Quality and Business Risk*. Chichester, UK: John Wiley & Sons.

Capitolo 20

Abts, C., B. Clark, S. Devnani-Chulani e B. W. Boehm. 2000. "COCOMO II Model Definition Manual." Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

Beck, K. e C. Andres. 2004. *Extreme Programming Explained*. 2nd ed. Boston: Addison-Wesley.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy e R. Selby. 1995. "Cost Models for Future Software Life Cycle Processes: COCOMO 2." *Annals of Software Engineering*: 1-31. doi:10.1007/BF02249046.

Boehm, B. e W. Royce. 1989. "Ada COCOMO and the Ada Process Model." In *Proc. 5th COCOMO Users' Group Meeting*. Pittsburgh: Software Engineering Institute. <http://www.dtic.mil/cgi/tr/fulltext/u2/a243476.pdf>

Boehm, B. W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W., C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. Reifer e B. Steele. 2000. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall.

Cohn, M. 2005. *Agile Estimating and Planning*. Englewood-Cliffs, NJ: Prentice Hall.

QSM. 2014. "Function Point Languages Table." <http://www.qsm.com/resources/function-pointlanguages-table>

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.

Capitolo 21

Bamford, R. e W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.

Buse, R. P. L. e T. Zimmermann. 2012. "Information Needs for Software Development Analytics." In *Int. Conf. on Software Engineering*, 987-996. doi:10.1109/ICSE.2012.6227122.

Chidamber, S. e C. Kemerer. 1994. "A Metrics Suite for Object-Oriented Design." *IEEE Trans. on Software Eng.* 20 (6): 476-493. doi:10.1109/32.295895.

- El-Amam, K. 2001. "Object-Oriented Metrics: A Review of Theory and Practice." National Research Council of Canada. <http://seg.iit.nrc.ca/English/abstracts/NRC44190.html>.
- Fagan, M. E. 1986. "Advances in Software Inspections." *IEEE Trans. on Software Eng.* SE-12 (7): 744-751. doi:10.1109/TSE.1986.6312976.
- Harford, T. 2013. "Big Data: Are We Making a Big Mistake?" *Financial Times*, March 28. <http://timharford.com/2014/04/big-data-are-we-making-a-big-mistake/>
- Humphrey, W. 1989. *Managing the Software Process*. Reading, MA: Addison-Wesley.
- IEEE. 2003. *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, CA: IEEE Computer Society Press.
- Ince, D. 1994. *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill.
- Kitchenham, B. 1990. "Software Development Cost Models." In *Software Reliability Handbook*, a cura di P. Rook, 487-517. Amsterdam: Elsevier.
- McConnell, S. 2004. *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Seattle, WA: Microsoft Press.
- Menzies, T. e T. Zimmermann. 2013. "Software Analytics: So What?" *IEEE Software* 30 (4): 31-37. doi:10.1109/MS.2013.86.
- Witten, I. H., E. Frank e M. A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Burlington, MA: Morgan Kaufmann.
- Zhang, D, S. Han, Y. Dang, J-G. Lou, H. Zhang e T. Xie. 2013. "Software Analytics in Practice." *IEEE Software* 30 (5): 30-37. doi:10.1109/MS.2013.94.

Capítulo 22

- Aiello, B. e L. Sachs. 2011. *Configuration Management Best Practices*. Boston: Addison-Wesley.
- Bamford, R. e W. J. Deibler. 2003. "ISO 9001:2000 for Software and Systems Providers: An Engineering Approach." Boca Raton, FL: CRC Press.
- Chrissis, M. B., M. Konrad e S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Boston: Addison-Wesley.
- IEEE. 2012. "IEEE Standard for Configuration Management in Systems and Software Engineering" (IEEE Std 828-2012). doi:10.1109/IEEESTD.2012.6170935.
- Loeliger, J. e M. McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. Sebastopol, CA: O'Reilly and Associates.
- Loope, J. 2011. *Managing Infrastructure with Puppet*. Sebastopol, CA: O'Reilly and Associates.
- Pilato, C., B. Collins-Sussman, and B. Fitzpatrick. 2008. *Version Control with Subversion*. Sebastopol, CA: O'Reilly and Associates.
- Smart, J. F. 2011. *Jenkins: The Definitive Guide*. Sebastopol, CA: O'Reilly and Associates.

Indice analitico

A

accettabilità, 8
ACM, Association for Computing Machinery, 16
aggregazione, 149
agile
 pianificazione, 478
 sviluppo, 516
AJAX, 15
algoritmici, modelli, 482, 488
alpha test, 253
analitica del software, 529
applicazioni
 autonome, 11
 interattive, 12
 riutilizzo, 306
architettura
 a strati, 173, 176
 client-server, 178, 380, 382
 delle applicazioni, 182
 di riferimento, 190
 guidata da modelli, 156
 in grande, 165
 in piccolo, 165
 orientata ai servizi, 404
peer-to-peer, 388
pipe-and-filter, 180
progettazione, 46
repository, 177
ASM, metodo, 87

Association for Computing Machinery (ACM), 16
astrazione, 134, 157, 165, 214
attacchi di hacker
 falsificazione, 370
 intercettazione, 370
 interruzione, 370
 modifica, 370

B

baseline, 538
batch, 12
BCS, British Computer Society, 16
beta test, 49, 253
big data, 12
Boehm, processo a spirale, 38
branching, 538
British Computer Society (BCS), 16
brownfield, sistemi, 86

C

C#, 313
C++, 313
CAD (Computer Aided Design), 177
callback, metodi, 314
cambiamenti
 anticipazione dei, 50
 tolleranza ai, 50

casi di studio, 19
apprendimento digitale, 26
clinica psichiatrica, 22
pompa di insulina, 20
stazioni meteorologiche, 24
casi d'uso, 119, 120
modellazione dei, 140
CBSE, 336
con riutilizzo, 350
per il riutilizzo, 347
processi, 346
CCB, Change Control Board, 554
Change Control Board (CCB), 554
CIM, Computation Independent Model,
157
classi di oggetti, 202
client-server
architettura, 178, 380, 382
calcolo, 375
cluster, sistemi ereditati, 273
COBOL, 269
COCOMO, 485
codeline, 538
competenza, 16
componenti software
acquisizione, 347
analisi dei, 526
certificazione, 347
composizione, 354
distribuiti, 383
interfacce, 343
modelli, 338, 342
riutilizzo dei, 306
scomposizione, 168
test, 47
Computation Independent Model (CIM),
157
comunicazione, link di, 200
configurazione, 533
gestione, 534
terminologia, 538
controllo
errori di, 515
inversione del, 314
CORBA, 369
cultura aziendale, 437

D

database
estensione, 394
prestazioni, 445
progettazione, 46
repository, 538
data mining, 386
dati
analisi, 12
raccolta, 12
debugging, 47, 234
semplificato, 247
denial-of-service, 179, 370
descrizioni astratte, 33, 111
design pattern, 209
diritti di proprietà intellettuale, 16
disponibilità, requisito, 169

E

Eclipse, 219
editing, 183, 191, 219
efficienza, 8
elaborazione batch, 12
ereditati, sistemi, 267
ERP (Enterprise Resource Planning), 8,
325
etnografia, 109
eventi, 152
eXtreme Programming (XP), 66, 246

F

fat-client, modello, 380
fattibilità, studi di, 97
fidatezza, 8
framework, 312

G

generalizzazione, 148, 149
gerarchia
della generalizzazione, 148, 149
delle eredità, 204
gestibilità, 370

- Git, 217, 539
 Github, 217
 GNU build system, 217
 guasti
 gestione, 368
 tolleranza, 366
 GUI, Graphical User Interface, 313
- H**
- hacker, 370
 HTML5, 15
- I**
- IDE, Integrated Development Environment, 42, 177, 219
 architettura repository, 178
 identificazione
 dei componenti, 351
 dei requisiti, 127, 350
 dei rischi, 441, 442
 dei servizi, 413
 delle classi di oggetti, 202
 delle modifiche, 263
 IEEE, Institute of Electrical and Electronic Engineers, 16, 124
 standard, 122
 iLearn, 27, 111, 112
 architettura, 176
 implementazione, problemi di, 213
 ingegneria dei requisiti, 94, 104
 interviste, 108
 processi, 104
 ingegneria dei servizi, 413
 ingegneria del software
 basato sui componenti, 336
 codice etico, 17
 etica, 15
 Internet, 14
 orientata al riutilizzo, 41
 pratica professionale, 17
 input
 convalida, 324
 di un'attività, 469
 errori, 515
- messaggi, 407
 Institute of Electrical and Electronic Engineers (IEEE), 16, 124
 standard, 122
 intangibile, prodotto, 436
 Integrated Development Environment (IDE), 219
 Interactive Development Environment (IDE), 42
 interazioni, 198
 interfaccia
 astratta, 314, 407, 417
 errori, 515
 grafica utente (GUI), 313
 specifica, 208
 interviste, 108
 inversione del controllo, 314
 ISO 9001, 508
 ispezioni, 511
- J**
- Java, 313
 JUnit, 219
- L**
- legacy system, 267
 linguaggi, elaborazione dei, 188
 linguaggi di programmazione, 269
 obsoleti, 280
 orientati agli oggetti, 313
 linguaggio naturale
 problemi, 117
 specifica, 115
 strutturato, 114
 link di comunicazione, 200
- M**
- malfunzionamenti, 8, 20, 102, 103, 177, 179, 230, 276
 mantenibilità, requisito, 169
 manutenzione, 36
 previsione, 282
 marketing, strategie di, 558

- master-slave, architettura, 378
MDA, Model-Driven Architecture, 155, 157
MDE, Model-Driven Engineering, 155
Mentcare, sistema, 22, 95, 138
 casi d'uso, 141
 organizzazione, 176
 requisiti, 249
 specifica, 136
 storia utente, 251
 story card, 69
 test, 250
metodi callback, 314
metriche di prodotto, 523
middleware, 45, 315, 336, 345, 374
milestone, 469, 475
misure del software, 518
Model-Driven Architecture (MDA), 155
Model-Driven Engineering (MDE), 155
modellazione, 12
 algoritmica dei costi, 482
 basata sugli stati, 154
 dei casi d'uso, 140
 dei database, 147
 dei sistemi, 134
 del controllo, 168
 delle interazioni, 139
 guidata dagli eventi, 153
 orientata agli oggetti, 120
 UML, 33
 up-front, 160
modelli
 astratti, 45, 157, 204
 comportamentali, 150
 contestuali, 136
 di componenti, 342
 di interazione, 139, 198, 372
 di macchine a stati, 205
 dinamici, 205
 di progettazione iniziale, 488
 di riutilizzo, 489
 di sequenza, 205
 di sottosistema, 205
 fat-client, 380
 guidati dagli eventi, 152
 guidati dai dati, 151
post-architettura, 491
strutturali, 145, 205
thin-client, 380
Model-View-Controller (MVC), 173, 313
modifiche
 analisi, 129
 costi, 129
 gestione, 128, 534, 551
 implementazione, 129
 tracciabilità, 128
moduli, 46, 319
 architettonici, 380
 di richiesta di modifica, 554
multi-site, sviluppo, 217
multi-team, sviluppo, 217
multi-tenancy, 393
MVC (Model-View-Controller), 173, 313
mySQL, 221
- O**
- Object Management Group (OMG), 155
open-source
 licenze, 222
 sviluppo, 220
operatività, 36
 dettagli, 46
 vincoli, 43
output
 di un'attività, 469
 errori, 515
 messaggi, 407
overhead, 55
 costi, 465
- P**
- p2p
 architettura, 390
 sistemi, 388
pagine web dinamiche, 314
paradigmi, 33
path testing, 239
pattern, 209
peer-to-peer, architettura, 378, 388
personale, gestione del, 447, 494

- pianificazione
 agile, 478
 delle iterazioni, 478
 delle release, 478
 piattaforma, 320
 software, 45
 PIM, Platform Independent Model, 157
 pipe-and-filter, 181
 pipeline, 181
 plan-driven, 33
 planning game, 478
 Platform Independent Model (PIM), 157
 Platform Specific Model (PSM), 157
 pompa di insulina, 19
 attività del software, 152
 requisiti, 116
 software di controllo, 118
 struttura, 21
 post-architettura, modello, 491
 prestazioni, test, 252
 processi software, 32
 attività, 42
 evolutivi, 263
 guidati da piani di prova, 49
 maturità, 55
 miglioramento, 55
 modelli, 33
 pipeline di, 181
 standard, 507
 prodotti software, 316
 generici, 7
 intangibili, 436
 metriche, 523
 personalizzati, 7
 standard, 506
 progettazione
 architettonica, 167, 200
 gestione agile, 75
 modelli, 204
 orientata agli oggetti, 197
 pianificazione, 464
 schemi, 209
 progetti
 pianificazione, 438
 tempistica, 472
 programmazione
 a coppie, 74
 estrema, 67
 programmi, ispezioni, 514
 proposizioni astratte, 44
 proprietà intellettuale, 16
 protezione, 8, 169, 368
 dei servizi web, 406
 modelli, 157
 qualità del software, 504
 requisiti, 101
 sistemi critici, 37
 WAF, 314
 prototipazione, 125
 obiettivi, 52
 prototipo, 51
 PSM, Platform Specific Model, 157
 punto di vista, 107
 Python, 313
- Q**
- QoS, Quality of Service, 371
 qualità, gestione, 500, 516
 Quality of Service (QoS), 371
- R**
- Rational Unified Process (RUP), 34, 35
 refactoring, 51, 70
 reingegnerizzazione
 dei dati, 285
 del software, 284
 release
 gestione, 217, 535, 557
 test, 248
 reporting, 470
 repository, architettura, 177
 Requirements Engineering (RE), 94
 requisiti
 analisi, 44
 classificazione, 106
 convalida, 44, 124
 deduzione, 44, 105, 108
 dell’utente, 94

- del prodotto, 100
del sistema, 94
di dominio, 99
documentazione, 106 , 121, 123
duraturi, 127
esterni, 101
evoluzione, 126
funzionali, 97, 98
gestione, 127
ingegneria, 43, 94
modifica, 126
negoziazione e priorità, 106
non funzionali, 97, 99
organizzativi, 100
perfezionamento, 41
punti di vista, 107
scoperta, 106
specifiche, 41, 44, 114
 strutturate, 116
tracciabilità, 129
volatili, 127
- RESTful, 409
reverse engineering, 285
revisioni, 511
rifattorizzazione, 71, 286
rischi
 analisi, 442
 gestione, 439
 identificazione, 442
 monitoraggio, 446
 pianificazione, 444
riservatezza, 16
riutilizzo, 307
 di sistemi applicativi, 323
 su vasta scala, 165
- Ruby, 313
- RUP, Rational Unified Process, 34
- S**
- SAP, sistemi, 8
scenari, 66, 111
 test, 250
schemi architetturali, 172
scomposizione, 165
- Scrum, metodo di sviluppo, 75, 89
 distribuito, 80
 terminologia, 76
SDLC, Software Development Life Cycle, 33
SEMAT, iniziativa, 11
server
 architettura client-server, 167, 178
 del database, 188
 di costruzione, 547
 interazione client-server, 376
 web, 188
- Service-Oriented Architeture (SOA), 404
- servizi
 componenti, 406
 composizione, 423
 di piattaforma, 344
 di supporto, 344
 forniti, 340
 implementazione, 421
 ingegneria, 413
 interfaccia, 417
 RESTful, 409
 richiesti, 341
- sicurezza, 169
- simulazione, 12
- sistema software
 astratto, 4
 cambiamenti, 50
 cloud, 12
 componenti, 41, 46
 configurabile, 325
 consegna incrementale, 51, 53
 costruzione, 545
 distribuito, 366
 documentazione, 247
 ereditato, 267
 grande, 86
 intangibile, 4
 integrazione, 36
 legacy, 267
 manutenzione, 36
 modellazione, 134
 modelli contestuali, 136
 modelli di interazione, 139

- modelli strutturali, 145
progettazione, 36
prototipazione, 51
refactoring, 51
requisiti, 36, 94
scalabilità, 368
test, 36, 48, 231, 243
- sistemi
applicativi integrati, 328
centralizzati, 539
di controllo integrati, 12
di elaborazione batch, 12
di intrattenimento, 12
di sistemi, 12
distribuiti, 539
ereditati, 267, 272
cluster, 273
per la modellazione e la simulazione, 12
per la raccolta e l'analisi dei dati, 12
- SOAP, Standard Object Access Protocol, 405
- SOA, Service-Oriented Architecture, 403, 404
- software, 6
accettabilità, 8
analitica del, 529
come servizio, 391
convalida, 10, 32, 47
di mezzo, 45
di supporto, 267
efficienza, 8
evoluzione, 10, 32, 49, 260
fidatezza, 8
implementazione, 44
ingegneria, 5, 9
mantenibilità, 8
manutenzione, 277
mercato, 230
misure, 518
prezzo, 466
produttività, 484
professionale, 5
progettazione, 44
protezione, 8
qualità, 500, 502
- reingegnerizzazione, 284
revisione, 512
riutilizzo, 214, 306
scopo, 230
specifiche, 9, 32, 43
standard, 505, 506
sviluppo, 10, 32
incrementale, 38, 40
test, 228
tipi, 10
- Software Development Life Cycle (SDLC), 33
- Software Requirements Specification (SRS), 121
- sopravvivenza dei progetti, 442
- sottocomponenti, 168
- sottosistemi, 46
- specifiche
matematiche, 114
strutturate, 116
- SQL, Structured Query Language, 382
- stakeholder, 43, 165
- standard ISO 9001, 508
- Standard Object Access Protocol (SOAP), 405
- stazione meteorologica, 20, 24
casi d'uso, 200
software integrato, 198
test, 234
- stili architetturali, 172, 378
client-server a due livelli, 378
client-server a più livelli, 378
componenti distribuiti, 378
master-slave, 378
peer-to-peer, 378
stimoli, 150, 200
storie utente, 66, 69, 111
story card, 68, 69
- strategia
di gestione dei rischi, 444, 469
di marketing, 558
di protezione, 368
di sviluppo incrementale, 86
- Structured Query Language (SQL), 382
- Subversion, 217, 539

superpeer, 390
superstato, 154
sviluppo del software
 approccio agile, 55
 configurazione, 40
 consegna incrementale, 53
 con test iniziali, 71
 guidato da piani, 468
 guidato da test, 246
 host-target, 214, 218
 integrazione, 40
 metodi agili, 64, 79, 83
 metodi guidati da piani, 83
 modello a cascata, 35
 multi-site, 217
 multi-team, 217
 open-source, 220
 refactoring, 70
 rifattorizzazione, 71
 strumenti, 42
 test, 233

T

target, ambiente, 547
task, 44, 66, 68
team di sviluppo
 organizzazione, 456
 personale, 447
 selezione dei membri, 454
tempistica dei progetti, 472
test
 basato sui requisiti, 249
 degli scenari, 250
 degli utenti, 253
 dei componenti, 240
 dei percorsi, 239
 delle prestazioni, 252
 delle release, 248
 delle unità, 234
 del sistema, 243
 di accettazione, 253, 254
 di regressione, 247
 di sviluppo, 233
test case, 48, 125, 236
thin-client, modello, 380

tolleranza
 ai cambiamenti, 50
 ai guasti, 366
 nelle specifiche, 502
transazioni, 12, 184

U

UDDI, Universal Description, Discovery and Integration, 406
UML (Unified Modeling Language), 21, 119, 135
 diagrammi, 136
 di classe, 145
 di rilascio, 220
 di sequenza, 142
 eseguibile, 160
Unified Modeling Language (UML), 21, 119, 135
Uniform Resource Identifier (URI), 406
Universal Description, Discovery and Integration (UDDI), 406
URI, Uniform Resource Identifier, 406

V

versioni
 controllo delle, 534
 gestione delle, 537
view, 177
Virtual Learning Environment (VLE), 27
virus, 16
vista, 177
 architetturale, 169
VLE, Virtual Learning Environment, 27
vulnerabilità
 del codice, 536
 della sicurezza, 265
 delle protezioni, 269
 del software, 278
V&V, Verification and Validation, 47

W

WAF, Web Application Framework, 313
Web Application Framework (WAF), 313

Web Service Description Language (WSDL), 407
wicked problem, 126
workflow, 423, 426
workspace, 538
WS-Addressing, 406
WS-BPEL, 405
WSDL, Web Service Description Language, 405, 407

WS-Reliable Messaging, 406
WS-Security, 406
WS-Transactions, 406

X

XML, 405
XP, eXtreme Programming, 66, 246
XSD, 405

