

Binato • Fuggetta • Sfardini

Ingegneria del software

Creatività e metodo

Il software è una delle componenti più critiche della nostra società. Tutti i più importanti prodotti e servizi sono attualmente basati sul software, in una logica che lo vede protagonista non solo delle applicazioni gestionali tradizionali, ma anche quale elemento per "governare" prodotti innovativi come, ad esempio, elettrodomestici, sistemi biomedicali e di controllo di processo, e in grado addirittura di rivoluzionare lo stesso mondo dell'entertainment.

Partendo da questo presupposto, il libro si presenta come uno strumento agile e versatile, formativo (e non solo informativo) per un insieme diversificato di utenti: non si limita alla semplice esposizione di tecniche o di concetti astratti ma affronta anche il ruolo del software e dell'ingegnere del software nella società contemporanea. Il testo si snoda intorno ai temi centrali dello studio del problema e della progettazione della soluzione, tenendo sempre ben distinti l'aspetto metodologico dall'uso dei linguaggi, in particolare UML. Senza dimenticare che, per gestire progetti sempre più articolati e complessi occorrono, accanto all'indispensabile supporto di una solida cultura ingegneristica, intuito e creatività.

Annalisa Binato ha conseguito il Master in Information Technology con una tesi su sistemi peer-to-peer di nuova generazione. È ricercatrice presso il CEFRIEL dove si occupa di progetti legati all'ingegneria del software e alla sicurezza dei sistemi informatici.

Alfonso Fuggetta è professore ordinario presso il Politecnico di Milano e Amministratore Delegato del CEFRIEL, centro di eccellenza per il trasferimento tecnologico nel settore dell'ICT. È inoltre membro di gruppi di lavoro dell'Unione Europea sulle tecnologie del software e Faculty Associate dell'Institute for Software Research della University of California, Irvine.

Laura Sfardini è manager presso il CEFRIEL dove si occupa di progetti di e-government e di ingegneria del software. È cultore della materia presso il Politecnico di Milano per il corso di Ingegneria del Software.

All'indirizzo <http://webbook.cefriel.it> si trova un WebBook dedicato a questo testo, dove studenti e lettori potranno trovare lucidi, materiali e link sulle tematiche d'interesse. Il WebBook è stato sviluppato originariamente nell'ambito del corso tenuto al Politecnico di Milano dal prof. Alfonso Fuggetta, ed è in continua estensione e arricchimento.

€ 25,00



Visitate il nostro sito web
<http://hpe.pearsoned.it>

Ingegneria del software

Creatività e metodo

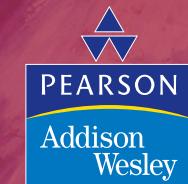
Binato
Fuggetta
Sfardini

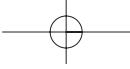


**Annalisa Binato
Alfonso Fuggetta
Laura Sfardini**

Ingegneria del software

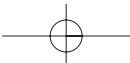
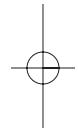
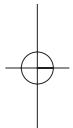
Creatività e metodo

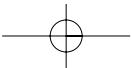
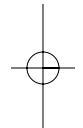
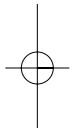




Ingegneria del software

Creatività e metodo



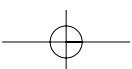




Annalisa Binato
Alfonso Fuggetta
Laura Sfardini

Ingegneria del software

Creatività e metodo



Copyright © 2006 Pearson Education Italia S.r.l.
Via Fara, 28 - 20124 Milano
Tel. 026739761 Fax 02673976503
E-mail: hpeitalia@pearson.com
Web: <http://hpe.pearsoned.it>

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Education Italia o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

I diritti di riproduzione e di memorizzazione elettronica totale e parziale con qualsiasi mezzo, compresi i microfilm e le copie fotostatiche, sono riservati per tutti i paesi.
LA FOTOCOPIATURA DEI LIBRI È UN REATO L'editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume. Le richieste di riproduzione vanno inoltrate ad AIDRO (Associazione Italiana per i Diritti di Riproduzione delle Opere dell'Ingegno), Via delle Erbe, 2 - 20121 Milano - Tel. e Fax 02.80.95.06.

Immagine di copertina di Umberto Boccioni, *Dinamismo di un ciclista*, 1913.
Copyright di copertina: Archivio Iconografico, S.A./CORBIS.

Copy-editing: Federica Sonzogno
Composizione: Totem di Andrea Astolfi
Grafica di copertina: Sabrina Miraglia
Stampa: Legoprint – Lavis (TN)

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

ISBN13: 978-8-8719-2274-4
ISBN10: 88-7192-274-3

Printed in Italy

1^a edizione: febbraio 2006

Sommario

Prefazione	IX
Capitolo 1 Lo sviluppo del software: creatività e ingegneria	1
1.1 Linguaggi, descrizioni e qualità delle descrizioni	3
1.2 Lo spazio del problema	5
1.3 Lo spazio della soluzione	7
1.4 Riferimenti bibliografici	10
Capitolo 2 UML: Unified Modeling Language	11
2.1 Use case diagram	11
2.2 Class diagram	16
2.3 Package diagram	26
2.4 Activity diagram	28
2.5 State machine diagram	35
2.6 Sequence diagram	37
2.7 Component diagram	42
2.8 Composite structure diagram	44
2.9 Deployment diagram	49
2.10 Riferimenti bibliografici	51
Capitolo 3 Qualità del software	53
3.1 Il modello di qualità interna ed esterna	55
3.1.1 Funzionalità (functionality)	55
3.1.2 Usabilità (usability)	57
3.1.3 Affidabilità (reliability)	58
3.1.4 Efficienza (efficiency)	59
3.1.5 Manutenibilità (maintainability)	59
3.1.6 Portabilità (portability)	60
3.2 Il modello di qualità in uso	61
3.3 Le metriche del software	62
3.4 Principi di progettazione come strumenti di qualità	63
3.4.1 Modularità	64
3.4.2 Information hiding	64
3.4.3 Coesione	64
3.4.4 Disaccoppiamento	65
3.5 Riferimenti bibliografici	65

VI Sommario

Capitolo 4 Descrivere il problema	67
4.1 Il dominio applicativo	68
4.2 I requisiti utente	73
4.2.1 La comunicazione con gli stakeholder	83
4.3 Specifica dell'interfaccia	84
4.4 Alcune considerazioni relative a UML	88
4.5 Riferimenti bibliografici	90
Capitolo 5 I problem frame	91
5.1 Required behaviour	92
5.2 Commanded behaviour	93
5.3 Information display	96
5.4 Simple workpiece	100
5.5 Transformation	102
5.6 Combinazione di frame	106
5.7 Riferimenti bibliografici	108
Capitolo 6 Progettare la soluzione	109
6.1 Viste architetturali	110
6.1.1 Logical-functional view	111
6.1.2 Module view	115
6.1.3 Deployment view	120
6.1.4 Execution view	121
6.2 Cos'è un componente?	127
6.3 Architetture hardware e software	127
6.4 Riferimenti bibliografici	128
Capitolo 7 Stili architetturali e design pattern	129
7.1 Client-server	130
7.2 Peer-to-peer	134
7.3 Publish-subscribe	137
7.4 Codice mobile	138
7.5 Combinazioni di stili	142
7.6 Design pattern	143
7.7 Riferimenti bibliografici	146
Capitolo 8 Cicli di vita e gestione dei progetti	147
8.1 Il processo software e i cicli di vita	148
8.2 Pianificazione di progetto e controllo di avanzamento	159
8.2.1 La pianificazione di progetto	160
8.2.2 Il controllo di avanzamento	168
8.2.3 Il ciclo di pianificazione e controllo	171
8.3 Configuration management	172
8.4 Riferimenti bibliografici	177

Sommario VII

Capitolo 9 Qualità del prodotto e del processo	179
9.1 Verifica e validazione	180
9.1.1 Testing	181
9.1.2 Tipologie di test	187
9.1.3 Tecniche di analisi manuali	188
9.2 Miglioramento del processo	189
9.2.1 Miglioramento per stadi: il CMMI	191
9.2.2 Standard ISO 9000	193
9.2.3 Miglioramento continuo: QIP (Quality Improvement Paradigm)	195
9.2.4 Le metriche e il metodo GQM	196
9.3 Riferimenti bibliografici	200
Capitolo 10 Il middleware e le tecnologie per lo sviluppo software	201
10.1 RMI	202
10.2 JNDI e JMS	205
10.3 J2EE	209
10.3.1 Componenti web tier	212
10.3.2 Componenti business tier	215
10.3.3 Java DataBase Connectivity	217
10.4 SOAP e web service	219
10.5 Uno sguardo allargato	221
10.5.1 Metodi	222
10.5.2 Tecnologie di sviluppo	223
10.5.3 Tecnologie a supporto del processo	224
10.5.4 Tecnologie infrastrutturali	225
10.6 Riferimenti bibliografici	226
Capitolo 11 Dal problema alla soluzione	227
11.1 Alcune linee guida	227
11.1.1 Completezza, formalità, rigore	227
11.1.2 La conoscenza di dominio	230
11.2 Le scelte di progetto	231
11.2.1 Il ciclo di vita e la pianificazione di progetto	231
11.2.2 La stima dei costi	232
11.2.3 Quali e quanti linguaggi di descrizione?	233
11.2.4 Quali strumenti e tecnologie?	234
11.2.5 Gli aspetti organizzativi	235
11.3 Le scelte architettonicali	235
11.3.1 Come si passa dal problema alla soluzione?	236
11.3.2 Come si sceglie un'architettura?	238
11.3.3 Il ruolo del middleware	240
11.3.4 Bottom-up o top-down	241

VIII Sommario

11.4	Dal progetto al codice	242
11.4.1	Il ruolo di UML	242
11.4.2	Come si scrive il codice	243
11.5	Affidabilità e robustezza del codice	247
11.5.1	Design by contract	247
11.5.2	Asserzioni	250
11.5.3	JML	251
11.6	Riferimenti bibliografici	254
	Appendice A	255
A.1	Il caso di studio	256
A.2	Descrizione del problema	257
A.2.1	Context diagram	257
A.2.2	Problem diagram	258
A.2.3	La descrizione del problema in UML	259
A.2.4	La descrizione dei requisiti	261
A.2.5	La specifica dell'interfaccia utente	267
A.3	Progetto della soluzione	268
A.3.1	Functional view	269
A.3.2	Module view	271
A.3.3	Deployment view	274
A.3.4	Execution view	275
A.4	Il codice	277
A.5	I casi di test	287
A.5.1	Casi di test per la ricerca diretta	287
A.5.2	Casi di test per la ricerca guidata	289
A.5.3	Casi di test per la ricerca per insiemi predefiniti	291
A.5.4	Caso di test per l'affidabilità del sistema	292
	Bibliografia	293
	Postfazione	297
	Indice analitico	301

Prefazione

Un altro testo di ingegneria del software?

Perché scrivere un nuovo libro di ingegneria del software? È una bella domanda che ci siamo ripetuti per diverso tempo. Esistono molti libri, scritti da autori prestigiosi, che hanno alle spalle diverse edizioni e migliaia di copie vendute. Perché investire tempo e risorse per andare in qualche modo a competere con questi “leader del mercato”?

I motivi sono diversi.

In primo luogo, l’ingegneria del software è una materia che si è evoluta molto nel corso degli anni. Il corpus di conoscenze si è accresciuto in modo significativo: fino a qualche anno fa si poteva immaginare di scrivere un libro di ingegneria del software che includesse l’insieme delle tematiche che costituiscono la disciplina. Oggi non è più così. Chi volesse scrivere un testo di tipo enciclopedico sull’ingegneria del software si troverebbe di fronte un compito immane, che produrrebbe come risultato certamente più di un volume. Paradossalmente, molti testi, anche tra quelli più noti, conservano questo approccio enciclopedico: si occupano di tutto. Inevitabilmente, ciò porta ad uno stile di presentazione di tipo *informativo* piuttosto che *formativo*. Si parla di tanti argomenti in modo “leggero”, in quanto non c’è lo spazio per affrontare nel dettaglio i problemi. Centinaia di pagine che coprono una molteplicità di argomenti non trattabili nell’ambito di un singolo corso. Capita così che lo studente si trova a dover acquistare testi (costosi) che usa solo in parte e senza avere il giusto livello di approfondimento. All’altro estremo, esistono molti libri che descrivono specifiche tecnologie, come UML o il middleware, senza però spiegare come usarle, secondo quali linee guida e principi. In pratica, nell’erogare i corsi di ingegneria del software per gli studenti dei corsi universitari e per i professionisti che si avvicinano per la prima volta in modo organico al tema, ci siamo accorti che non avevamo del materiale didattico adatto. In molti casi, siamo stati costretti come docenti a ideare slide che coniugassero la presentazione dei concetti fondamentali con l’approfondimento necessario a formare le persone.

Un secondo motivo risiede nel riassetto del corso degli studi universitari. Tale riassetto impone nei fatti un nuovo approccio alla didattica. Servono corsi agili che possano essere fruiti in momenti diversi del percorso formativo di uno studente. Può essere, per esempio, che uno studente decida di seguire il primo corso di ingegneria del software nell’ambito della laurea di primo livello. Altri, al contrario, possono decidere di fruirne durante la laurea specialistica, magari perché hanno conseguito una laurea di primo livello su tematiche di tipo non strettamente informatico (e non hanno quindi mai seguito corsi di ingegneria del software). Altri ancora possono decidere di affrontare l’argo-

X Prefazione

mento nei corsi di master o durante il dottorato. Questo perché il percorso formativo è molto più flessibile che in passato.

Ma c'è anche altro. *Il software è divenuto una delle componenti più critiche della nostra società.* Tutti i più importanti prodotti e servizi sono oggi basati sul software. Non si tratta più semplicemente di produrre applicazioni informatiche per sistemi gestionali tradizionali (settore peraltro molto importante!). Oggi il software governa il funzionamento di Internet e di tutti i moderni sistemi di telecomunicazione (si pensi ai cellulari e agli smartphone). Il software controlla il funzionamento di auto, moto, aerei e navi. Elettrodomestici, sistemi biomedicali, sistemi per la produzione industriale sono governati e gestiti dal software. Tutto lo sviluppo dei servizi su Internet è basato sul software, Basti pensare a quanto stanno facendo aziende come Google e Skype. O anche la trasformazione che sta avvenendo nel mondo dell'intrattenimento, con il successo clamoroso di Apple iTunes e l'avvento della TV digitale. Se quindi il software è uno dei motori dell'innovazione, allora *un numero sempre maggiore di professionisti e manager deve conoscere i problemi legati allo sviluppo del software.*

In sintesi, sentivamo il bisogno di un *testo agile*, che potesse servire alla *formazione* (e non solo all'*informazione*) di un *insieme di utenti abbastanza diversificato*. Nello scrivere il libro, abbiamo volutamente valorizzato alcuni temi a scapito di altri, e scelto di affrontare quelle tematiche che ci sembrano più critiche e importanti per un primo corso sull'ingegneria del software. L'obiettivo è quello di offrire uno strumento che rende possibile acquisire i concetti di base dell'ingegneria del software sia allo specialista che continuerà in altri corsi lo studio della materia, sia a coloro che vogliono capire il mondo dell'ingegneria del software, attraverso lo studio dei suoi passaggi e temi più rilevanti.

Struttura del libro

Essendo un testo per il primo corso sull'ingegneria del software, abbiamo deciso di strutturare il libro attorno alle due fasi più importanti del processo di sviluppo del software: lo studio del problema e la progettazione della soluzione. Spesso queste fasi sono chiamate con altri termini come "specifica dei requisiti" e "analisi e progettazione". Sono le più importanti perché costituiscono il cuore del problema. Un software serve in quanto risolve un problema: conoscere bene il problema è essenziale per comprendere come risolverlo. L'altro aspetto essenziale è la progettazione della soluzione. Lo sviluppo di un programma di qualità deve basarsi su un progetto solido. Nessun sistema complesso viene costruito senza che prima sia progettato. Lo stesso vale per il software.

Nell'affrontare i temi della descrizione del problema e della progettazione della soluzione abbiamo cercato di distinguere nettamente l'aspetto metodologico dall'uso dei linguaggi e in particolare di UML. Molti testi parlano di descrizione del problema e di progettazione della soluzione illustrando le caratteristiche dei linguaggi che in quelle fasi si utilizzano. È come se un docente di letteratura, invece di parlare di come, nel corso dei secoli, si sono succedute le diverse forme letterarie, spendesse il suo tempo a presentare la

grammatica dell’italiano e dell’inglese. Certamente, per scrivere un poema o un romanzo è necessario conoscere una lingua, ma non è solo quella conoscenza che fa diventare capace di costruire (o apprezzare) un’opera d’arte. Allo stesso modo, la conoscenza di UML è strumentale alla costruzione di descrizione e formalizzazione del problema e della soluzione, ma non aiuta in alcun modo a comprendere nel dettaglio le caratteristiche del problema, né a individuare l’architettura più efficace per risolverlo. Per questo abbiamo distinto chiaramente la presentazione di UML, linguaggio certamente largamente diffuso e quindi da conoscere e applicare, con la sostanza del processo di studio del problema e costruzione della soluzione. UML è lo strumento, non il fine. Peraltro, molti testi su UML presentano le diverse notazioni del linguaggio (la “sintassi”) senza fornire una qualche chiara guida che illustri come utilizzarle in modo coerente nelle diverse fasi del processo di sviluppo. Abbiamo cercato di ovviare offrendo alcune semplici linee guida su come utilizzare UML nelle diverse attività. Praticamente tutti gli schemi e diagrammi presenti nel testo sono redatti in UML. Che senso ha introdurre linguaggi di descrizione come UML se poi si continuano ad usare disegni informali per descrivere quanto si vuole trasmettere?

L’altro aspetto sul quale si è insistito nella costruzione del testo è l’approccio ingegneristico alla soluzione dei problemi. L’ingegnere lavora per schemi (non in modo schematico!), classificando e organizzando la conoscenza che acquisisce in modo che sia riutilizzabile e estendibile in nuovi progetti. È ciò che rende l’ingegneria diversa dall’artigianato. Lo sviluppo del software richiede una grande dose di creatività, intuito e genio. Ma queste qualità devono essere accompagnate e temprate da una solida cultura ingegneristica (il metodo!) che permetta di gestire in modo convincente ed economicamente conveniente i complessi progetti che sempre più spesso l’ingegnere del software si trova ad affrontare.

In sintesi, il testo è organizzato in 11 capitoli più un’appendice:

- Il Capitolo 1 introduce alcuni concetti di base e molte delle argomentazioni che sono state brevemente discusse in questa prefazione.
- Il Capitolo 2 fornisce un’introduzione a UML 2.0, linguaggio utilizzato nel resto del libro.
- Il Capitolo 3 illustra i principali concetti legati alla qualità del software. Essi costituiscono i parametri secondo i quali giudicare la bontà di una soluzione informatica.
- Il Capitolo 4 tratta il processo di descrizione di un problema e introduce i concetti chiave che lo contraddistinguono: dominio applicativo, requisiti, interfaccia.
- Il Capitolo 5 illustra gli schemi che si possono usare per strutturare un problema, i cosiddetti *problem frames*.
- Il Capitolo 6, analogamente a quanto previsto per il Capitolo 4, spiega come si progetta e descrive una soluzione informatica complessa.

XII Prefazione

- Il Capitolo 7, come il Capitolo 5, descrive gli schemi utilizzati per progettare una soluzione informatica: gli stili architettonici e i design pattern.
- Il Capitolo 8 discute gli aspetti di pianificazione e gestione del progetto di sviluppo.
- Il Capitolo 9 introduce alcuni concetti base relativi al controllo di qualità del prodotto (in particolare, i concetti chiave delle tecniche di testing) e di miglioramento del processo.
- Il Capitolo 10 fornisce una panoramica delle diverse tecnologie a disposizione dell'ingegnere del software durante le diverse fasi del processo e, in particolare, le moderne tecnologie di middleware, componente oggi essenziale di qualunque soluzione informatica minimamente complessa e significativa.
- Infine, il Capitolo 11 descrive alcuni concetti e tecniche che guidano il progettista nel passare dalla descrizione del problema all'identificazione della soluzione e al suo sviluppo.

L'appendice contiene un caso di studio che illustra in modo estensivo come applicare le diverse tecniche illustrate nel testo.

Prerequisiti e potenziali utenti del testo

I prerequisiti per poter fruire dei contenuti di questo testo sono di due tipi.

1. Conoscenza ed esperienze di utilizzo di un linguaggio di programmazione, preferibilmente Java, e dei concetti del paradigma object-oriented. Conoscenza dei principi della programmazione strutturata e delle tecniche di costruzione modulare del software.
2. Conoscenza del funzionamento di un sistema operativo (processi e thread). È auspicabile anche una conoscenza di massima dei principali concetti relativi alle basi di dati.

Con questi prerequisiti, il testo si pone l'obiettivo di fornire al lettore i principali concetti, metodi e tecniche che devono essere utilizzate nell'impostare e condurre un progetto di sviluppo di software. Per questo motivo, come già accennato in precedenza, il testo può essere fruito da diversi tipi di lettori.

- Studenti dei corsi di laurea di primo livello nel settore dell'ingegneria dell'informazione (ingegneria informatica, elettronica, automatica, delle telecomunicazioni) e in generale di corsi di laurea similari (per esempio, informatica). Per questi studenti, normalmente esiste un corso di ingegneria del software al secondo o terzo anno.
- Studenti dei corsi delle lauree specialistiche che non abbiano già fruito di un corso di ingegneria del software al primo livello.

- Studenti di corsi di specializzazione o dottorato o master, che provengano da indirizzi di studi non tecnologici o che comunque non hanno avuto modo di studiare ingegneria del software.
- Professionisti e ricercatori che vogliono comprendere i concetti e le tecniche di base utilizzate per condurre e gestire un progetto di sviluppo del software.

A valle di un corso basato su questo testo, il lettore potrà affrontare e approfondire una serie di tematiche di tipo più specialistico come metodi formali per l'ingegneria del software, testing e validazione del software, gestione dei progetti, middleware e tecnologie per sistemi distribuiti, requirement engineering, valutazione delle prestazioni.

Servizi di supporto

Ogni testo moderno deve essere oggi accompagnato da servizi aggiuntivi che aiutino il lettore nell'apprendere e approfondire le materie di studio. Nel nostro caso, da tempo esiste su web un *WebBook di ingegneria del software*, che contiene lucidi, materiali e link sulle tematiche di interesse del testo.

L'indirizzo è <http://webbook.cefriel.it>

Questo WebBook, sviluppato originariamente nell'ambito dei corsi tenuti dagli autori al Politecnico di Milano e al CEFRIEL, è in corso di estensione e arricchimento. Vogliamo continuamente inserire nuovo materiale a supporto del docente (slide) e dello studente (esercizi e altro materiale di ausilio allo studio).

Nella scrittura del libro abbiamo preferito il criterio del “bene subito” rispetto al “meglio poi”. Come qualunque opera minimamente complessa, ci sono diversi aspetti del testo che possono essere migliorati e arricchiti. Piuttosto che lavorare a una infinita tela di Penelope, abbiamo preferito rilasciare il nostro lavoro e metterlo alla prova nei nostri corsi, nell'interazione con gli studenti e nell'esame di tutti coloro che vorranno leggerlo. Speriamo che da ciò derivino suggerimenti, contributi e consigli che rendano il testo sempre più efficace e utile. Anche il sito vuole essere lo strumento per facilitare questo processo.

Ringraziamenti

Questo libro nasce a valle di esperienze di insegnamento e di utilizzo delle tecniche dell'ingegneria del software condotte nel corso degli ultimi anni presso il CEFRIEL (<http://www.cefriel.it>). Lo stile e il taglio del testo derivano proprio dal bisogno di coniugare la capacità di formare con il bisogno di concretezza e di collegamento con il mondo delle applicazioni. Ci hanno aiutato in questo cammino i tanti colleghi che hanno condiviso con noi le occasioni di lavoro di questi anni. Alcuni di loro sono stati particolarmente attivi nel fornirci indicazioni su come costruire il materiale presente nel te-

XIV Prefazione

sto e nel correggere anche alcuni errori e imprecisioni. Li vogliamo ringraziare esplicitamente: Maurizio Brioschi, Fabiano Cattaneo, Cesare Colombo, Massimiliano Colombo, Luigi Lavazza, Marco Mauri, Massimiliano Pianciamore, Gianluca Ripa, Giordano Sassaroli, Paolo Selvini, Emma Tracanella e Maurilio Zuccalà. Un ringraziamento particolare anche a Micaela Guerra e Alessandra Piccardo di Pearson Education per il loro continuo supporto.

A tutti i colleghi e amici un grazie per i tanti consigli e contributi forniti, a volte in modo anche inconsapevole, nelle molte giornate di lavoro e di discussione trascorse insieme. Qualunque cosa buona sia contenuta in questo libro nasce proprio dal continuo scambio di opinioni, idee e contributi che contraddistingue il CEFRIEL, un luogo di lavoro unico che ci permette tutti i giorni di crescere insieme umanamente e professionalmente.

Annalisa Binato

Alfonso Fuggetta

Laura Sfardini

Usmate e Como, gennaio 2006

Capitolo 1

Lo sviluppo del software: creatività e ingegneria

Lo sviluppo di una soluzione informatica costituisce una sfida ingegneristica particolarmente complessa. Come anni di ricerche e studi hanno dimostrato, il software è un'entità per certi versi unica.

In primo luogo, è un *bene immateriale*. Non è il risultato di un'attività produttiva di tipo classico. Il termine “fabbriche del software” spesso utilizzato in letteratura, in realtà mal si adatta a descrivere il processo attraverso il quale si crea un programma. Non esiste una fase di progettazione separata da quella della produzione in senso stretto. Non esiste il problema della costruzione delle diverse copie di uno stesso prodotto, come nel caso della catena di montaggio di un’auto. Una volta sviluppata la prima copia di un programma, è possibile soddisfare i bisogni di milioni di utenti semplicemente replicando per via informatica il codice stesso. Anzi, oggi il codice viene spesso distribuito attraverso Internet e non è quindi neanche necessario copiarlo usando mezzi classici quali i floppy disk o i CD-ROM. Se nel caso di un’auto ogni singola vettura che esce dalla catena di montaggio può avere difetti unici e diversi in quanto il risultato di una lavorazione individuale e distinta, nel caso del software ogni copia (di uno stesso programma, ovviamente) è uguale alle altre “per definizione”. Inoltre, il software ha un costo marginale pressoché nullo. Il *costo marginale* rappresenta la spesa necessaria per costruire una nuova copia di un certo bene, avendone già costruite altre. Ovviamente se si tratta di un’auto questo costo è molto significativo, in quanto si tratta di allocare in via esclusiva una quota di risorse umane, semilavorati e impianti per ogni singola vettura, mentre una nuova copia del software è il risultato di un click del mouse e di pochi secondi di elaborazione.

Un secondo aspetto particolarmente distintivo del software è la difficoltà che si incontra nel doverne *definire e valutare la qualità*. Esistono diversi parametri che possono essere utilizzati per questo scopo. Essi saranno discussi in dettaglio in un successivo capitolo. Certamente, la bontà di un programma dipende da una miriade di fattori che vanno dalla capacità di soddisfare in modo semplice e fedele i bisogni dell’utenza, all'affidabilità, efficienza e sicurezza. Inoltre occorre considerare la facilità secondo la quale un prodotto informatico può essere reso disponibile all’utente e modificato in base a nuove

2 Capitolo 1 Lo sviluppo del software: creatività e ingegneria

o mutate esigenze o semplicemente perché sono stati rilevati errori che devono essere corretti. Si tratta spesso di obiettivi contrastanti difficili da ottenere in modo completo. Per esempio, garantire piena sicurezza e affidabilità può richiedere un insieme di operazioni che rendono l'applicazione meno efficiente, usabile e distribuibile. Si tratta di scegliere di volta in volta il giusto compromesso tra le diverse esigenze, concependo la soluzione che meglio si adatta al contesto del problema considerato.

Il software è quindi il risultato di una *attività creativa*, di un'azione dell'ingegno più che di una produzione industriale. Per questo, esso è assimilato a opere quali romanzi e poesie, o ad altre creazioni artistiche come i film e i brani musicali. Sviluppare software richiede *talento, intuito, capacità analitiche e deduttive significative*, anche quando si deve modificare o riusare parti di codice preesistente. Allo stesso tempo, questa forte connotazione creativa non cancella il bisogno anche di una profonda *capacità ingegneristica*. Persino grandi compositori come Mozart, Beethoven o Mahler dovevano necessariamente saper coniugare il genio e l'intuito musicale con le conoscenze tecniche relative alla composizione, al contrappunto, all'arrangiamento e all'utilizzo efficace dell'orchestra nel suo complesso. Non basta aver orecchio e saper concepire melodie affascinanti, bisogna saperle tradurre in strutture musicali organiche e convincenti. La maestria nell'usare l'orchestra da parte di geni come Wagner e Mahler è da questo punto di vista esemplare. Allo stesso modo, costruire sistemi software complessi richiede la capacità di coniugare l'intuizione, la creatività e il genio del programmatore con la sistematicità e il rigore delle discipline ingegneristiche. Solo così si possono realmente costruire applicazioni e sistemi informatici in grado di soddisfare in modo sicuro e affidabile i bisogni sempre più complessi dell'utenza.

Qual è l'approccio dell'ingegnere? Lavorare con *metodo* e per *schemi*. Si badi bene, non in modo schematico, secondo un'accezione negativa del termine. Lavorare con metodo e per schemi vuol dire classificare e strutturare la conoscenza acquisita al fine di riutilizzarla in modo efficace nei nuovi progetti e poterla ulteriormente incrementare, raffinare e, in generale, arricchire con nuove esperienze, risultati e intuizioni. È un processo di accumulazione e di riuso creativo della conoscenza che ha permesso di raggiungere i risultati che chiunque può ammirare osservando un moderno ponte. Nel Golden Gate di San Francisco si riconosce la struttura (lo schema) dei ponti sospesi, che si ritrova anche nel Verrazzano Narrows Bridge di New York, costruito molti anni dopo sfruttando gli stessi principi ingegneristici e tutte le esperienze che nel frattempo erano state accumulate nella costruzione di quel tipo di ponti. E questa conoscenza verrà riusata e arricchita nella costruzione dei futuri ponti sospesi.

Nel caso del software, i metodi e gli "schemi" sono molteplici. Nei corsi di base di programmazione si studiano concetti quali la programmazione strutturata, l'information hiding, la modularità, la progettazione top-down. Quando si sviluppano sistemi software di una certa dimensione, a queste tecniche e metodi di base bisogna aggiungere gli "schemi" che permettono di strutturare soluzioni complesse quali i design pattern e le architetture software (discusse nel seguito). In generale, occorre saper *strutturare correttamente il problema che si vuol risolvere e la soluzione che si ritiene adatta allo scopo*.

1. Un sistema software deve servire a *risolvere un qualche problema*. Nel caso di un'azienda, il problema può essere la gestione efficace ed efficiente dei suoi processi

1.1 Linguaggi, descrizioni e qualità delle descrizioni 3

chiave: il magazzino e le scorte, le linee di produzione, i sistemi di distribuzione, la logistica e l'amministrazione. Nel caso del software di controllo di un aeroplano, il problema è controllare i dispositivi elettromeccanici del velivolo per far sì che possa decollare, volare e atterrare in modo sicuro e confortevole. Spesso, il problema è molto complesso e deve essere descritto con estrema cura e precisione perché sia possibile comprenderlo correttamente, discuterlo con il committente e trasmetterlo ai progettisti e programmati che dovranno scrivere il codice del programma.

2. *La soluzione è l'insieme dei componenti informatici che realizzano le funzioni richieste dall'utente per risolvere il problema.* Nel caso dell'azienda, è il sistema informatico che ne gestisce le diverse attività e reparti; nel caso dell'aereo, è il software dell'avionica che controlla i dispositivi di bordo. Anche la soluzione, come è ovvio che sia, è spesso molto complessa e deve essere a sua volta strutturata. Il programmatore già conosce termini e concetti quali moduli, tipi di dati astratti e procedure: tutti utilizzati per dare "struttura" a un programma. Nell'ambito di sistemi complessi, questi concetti, come osservato in precedenza, non sono più sufficienti ed è necessario estendere, anche in modo significativo, l'insieme degli strumenti che il progettista deve saper dominare per poter descrivere e valutare efficacemente una soluzione informatica.

Nei prossimi paragrafi sono introdotti e caratterizzati altri concetti di base che verranno largamente utilizzati nei successivi capitoli:

- linguaggi e descrizioni
- problema, soluzione e loro mutua relazione
- struttura dello spazio del problema e di quello della soluzione.

1.1 Linguaggi, descrizioni e qualità delle descrizioni

Nelle pagine precedenti in diversi punti sono stati usati termini quali "descrizione" e "descrivere". In particolare, si è detto che il problema e la soluzione devono essere "descritti" con precisione. Che cosa si intende con questo termine? Quali sono gli strumenti che si possono utilizzare allo scopo?

Lo sviluppo di un sistema informatico raramente può essere condotto da una singola persona. E quand'anche si trattasse di un singolo, non è pensabile che possa ricordare e tenere a mente l'insieme delle informazioni che caratterizzano il problema da risolvere e la soluzione da costruire. È necessario che queste informazioni siano consolidate in una qualche forma scritta, sia per facilitarne la comunicazione sia per conservarne memoria. "Descrivere" problema e soluzione è quindi l'azione attraverso la quale si *consolidano in forma scritta queste informazioni*. Il risultato di questo processo è un insieme di *descrizioni*.

4 Capitolo 1 Lo sviluppo del software: creatività e ingegneria

Come si produce una descrizione? Utilizzando un *linguaggio di descrizione*. I più ovvi linguaggi di descrizione sono le lingue parlate e scritte come per esempio l'italiano e l'inglese. In effetti, il modo più semplice e intuitivo di descrivere un qualche concetto è quello di usare testi scritti. Un altro meccanismo per produrre descrizioni è costituito da diagrammi e disegni. Molto spesso, per raccontare come "è fatto" un programma, il progettista traccia qualche schema su un foglio di carta o alla lavagna. Sono blocchi e frecce che rappresentano in modo informale diverse porzioni del codice che si scambiano dati o segnali di controllo.

Le descrizioni basate su testi e diagrammi informali presentano una serie di problemi. Sono in primo luogo imprecise e ambigue: diverse persone possono dare interpretazioni diverse della stessa descrizione. Spesso sono eccessivamente prolisse e poco sintetiche. Inoltre, sono di scarso ausilio nella creazione del codice. In poche parole, trasmettono poca informazione e spesso in modo impreciso e confuso. Per questo motivo, sono stati concepiti dei linguaggi che rendono possibile la creazione di descrizioni univoche e sintetiche. Se il linguaggio è basato su tecniche formali, è possibile anche costruire strumenti automatici che analizzano la descrizione prodotta al fine di valutarne la qualità e coerenza, e provarne le proprietà. In alcuni casi, partendo da una descrizione formale del sistema che deve essere sviluppato, è possibile generare in modo automatico porzioni di codice sorgente.

Il linguaggio di descrizione oggi forse più diffuso nel settore dell'ingegneria del software è UML (*Unified Modeling Language*), giunto recentemente alla seconda edizione. Questo linguaggio è in realtà una collezione di diverse notazioni grafiche: alcune di esse sono formali, hanno cioè una semantica definita in modo preciso; altre sono semiformali, hanno cioè una sintassi ben definita, ma mancano di una semantica precisa. UML può essere utilizzato sia per descrivere il problema che la soluzione, anche se originariamente è stato concepito soprattutto per la soluzione. Infatti, si basa sui concetti e sui costrutti classici della programmazione orientata agli oggetti: classi, oggetti, componenti. I principali concetti e diagrammi di UML 2.0 verranno spiegati più avanti e saranno utilizzati estensivamente nel resto del libro.

Qualunque sia il linguaggio di descrizione utilizzato, il progettista ha il problema di valutare la qualità della descrizione che ha costruito. Tre sono le tipologie di verifiche che devono essere condotte e che verranno discusse nel seguito.

1. *Verifica della struttura complessiva.* Una descrizione è per tutto e in tutto simile a un codice scritto utilizzando un linguaggio di programmazione. Deve essere leggibile, comprensibile, ben strutturata e organizzata in porzioni distinte e organicamente collegate tra loro (modularizzazione della descrizione). Alle descrizioni, come discusso nei prossimi capitoli, si applicano quindi molte delle nozioni utilizzate per il codice sorgente, inclusa l'idea di information hiding e di livelli di astrazione.
2. *Verifica di coerenza interna.* Una descrizione deve essere *internamente coerente*, deve cioè rappresentare in modo non contraddittorio o ambiguo le informazioni. Non può accadere che due diverse parti di una descrizione affermino un fatto e il suo contrario. Per esempio, non può accadere che descrivendo un'azienda, in una parte del documento si dica che i prodotti sono venduti solo presso punti vendita dell'azienda mentre in altre parti si affermi che è possibile utilizzare anche rivenditori terzi.

1.2 Lo spazio del problema 5

3. Una descrizione è valida quando rappresenta in modo coerente l'entità che si vuole descrivere. In modo più preciso, *la descrizione deve essere esternamente coerente con quanto si suppone rappresenti*, deve essere cioè una rappresentazione fedele della realtà studiata. Tale coerenza in generale non può essere provata in modo automatico. È l'ingegnere del software che deve valutare se e in quale misura la descrizione prodotta è una rappresentazione fedele di quanto osservato e studiato. Ciò è più critico e delicato nel caso della descrizione del problema, in quanto si tratta di distillare informazioni relative all'utente, ai suoi problemi ed esigenze. Si tratta quindi di raccogliere e consolidare informazioni spesso poco chiare all'utente stesso. La coerenza esterna di una descrizione deve essere valutata interagendo con l'utente e verificando se quanto viene consolidato è una corretta rappresentazione del suo pensiero.

Da quanto esposto, si evince che il processo di costruzione di una descrizione è complesso e articolato. Raramente procede in modo lineare e richiede, al contrario, interazioni continue tra i diversi attori: utente, progettista, programmatore, gestore del programma. Per capire meglio come si costruiscono le descrizioni del problema e della soluzione, è quindi necessario studiare con maggiore precisione la natura di queste due entità e le loro mutue relazioni.

1.2 Lo spazio del problema

Il rapporto tra problema e soluzione può essere sinteticamente rappresentato dallo schema della Figura 1.1.

Lo *spazio del problema* è innanzi tutto l'insieme di fenomeni che caratterizzano il dominio applicativo. Nel caso di un'azienda, il dominio applicativo è l'azienda stessa, le sue procedure operative, il suo modello di business. I fenomeni sono le persone, i fatti e gli eventi che caratterizzano il funzionamento dell'azienda: la procedura per l'emissione di una fattura o i controlli di carico e scarico dal magazzino. Lo *spazio della soluzione* è l'insieme degli elementi che caratterizza il software che deve essere sviluppato per risolvere il problema: codice sorgente, documentazione, casi di test, stub e driver (cioè i programmi usati per testare il codice). Lo spazio del problema e quello della soluzione hanno delle parti in comune: i *fenomeni condivisi*. Essi sono quei fenomeni che fanno parte

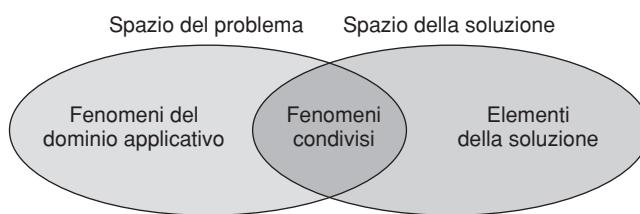


Figura 1.1 Spazio del problema e spazio della soluzione.

6 Capitolo 1 Lo sviluppo del software: creatività e ingegneria

sia dello spazio del problema che di quello della soluzione. Per esempio, il modulo che permette di compilare una fattura sarà presente sia come parte del dominio applicativo in quanto modulo cartaceo, sia come parte della soluzione in quanto implementato in una qualche maschera a video. In generale, i fenomeni condivisi sono gli elementi fondamentali dell'interfaccia utente, cioè di come il sistema informatico interagisce con il dominio applicativo e quindi il “mondo reale”.

L'attitudine del progettista muta profondamente nello studiare lo spazio del problema e quello della soluzione.

- Nel primo caso (lo spazio del problema) l'attitudine è quella di *ascoltare, capire e sintetizzare* in forma coerente le informazioni che caratterizzano il problema. Il progettista deve primariamente “ascoltare” l'utente.
- Nel secondo caso (lo spazio della soluzione) l'attitudine del progettista è quella di *creare, progettare, costruire*. Il progettista è “parte attiva” e attore principale nella costruzione della soluzione.

È importante distinguere queste diverse attitudini in quanto il metodo di giudizio e l'atteggiamento del progettista cambiano profondamente. È quindi fondamentale che le due attività vengano tenute, almeno concettualmente, distinte. I problemi sono dell'utente, non del progettista. Gli utenti non devono essere vincolati o forzati da quelli che sono i punti di vista del progettista. Egli potrà contribuire a chiarire aspetti ambigui o fornire suggerimenti e chiavi di lettura, ma sempre nell'ottica di aiutare l'utente a comprendere correttamente i propri bisogni. Al contrario, in fase di costruzione della soluzione, il progettista ha un ruolo propositivo e costruttivo.

In modo sintetico, si può affermare che *il problema esiste anche prima che qualcuno si metta nell'ottica di costruire una soluzione informatica che lo risolve*. Per esempio, l'azienda doveva gestire il magazzino anche in assenza di un sistema informatico (in effetti così si faceva fino agli anni '60). Nel costruire la descrizione del problema, quindi, è necessario saper ascoltare e capire. Da questo punto di vista, è essenziale il *ruolo della conoscenza di dominio* e di chi la possiede. Non è facile per un informatico conoscere i diversi domini applicativi nei quali può trovarsi a operare. Le regole che caratterizzano il funzionamento di un'azienda sono ovviamente diverse da quelle che regolano il volo di un aeroplano. Per questo motivo, il progettista informatico devo saper acquisire queste informazioni dagli *esperti di dominio*, cioè da coloro che conoscono i fenomeni del dominio applicativo. Nei casi presi in considerazione, si tratta degli ingegneri gestionali che seguono il funzionamento dell'azienda o degli ingegneri aerospaziali che hanno progettato il velivolo.

Un'altra osservazione importante sul rapporto tra spazio del problema e spazio della soluzione riguarda il ruolo del software. Istintivamente si è portati a pensare il software come parte della soluzione, non del problema. In realtà, *il software può essere sia parte del problema che della soluzione*. Se per esempio il problema è costruire un sistema di reportistica che estragga dati dal sistema informatico di gestione del magazzino di un'azienda, quest'ultimo (software) sarà parte del problema in quanto entità preesistente alla soluzione che il progettista deve costruire. Ovviamente, il software per la reportistica che deve essere sviluppato è invece la soluzione. Di volta in volta, quindi, è necessario delimitare

1.3 Lo spazio della soluzione 7

tare correttamente lo spazio del problema e tutti i fenomeni a esso collegati, senza farsi influenzare dalla semplice distinzione software/non software.

È essenziale a questo punto sottolineare che la distinzione tra spazio del problema e spazio della soluzione non vuole in alcun modo suggerire che lo sviluppo di software debba necessariamente procedere con un ciclo di vita a cascata, nel quale prima si descrive il problema e poi si concepisce e si sviluppa la soluzione. *La distinzione tra problema e soluzione è concettuale e prescinde dal ciclo di vita.* Può essere che la costruzione delle due descrizioni proceda per iterazioni successive o per incrementi, come suggerito da cicli di vita quali lo spiral model di Boehm (si veda il Capitolo 8). È peraltro essenziale che anche quando si adottano questi cicli di vita non lineari si tengano ben presenti le differenze concettuali e di attitudine discusse in precedenza.

Dopo questa disamina dei concetti di spazio del problema e di spazio della soluzione è ora possibile precisare maggiormente cosa si intende per descrizione del problema e della soluzione.

La descrizione del problema, come suggerito da Jackson (Jackson[2001]), si compone di tre parti principali.

- *Descrizione del dominio applicativo.* È la descrizione dei fenomeni che contraddistinguono il dominio per il quale si dovrà costruire la soluzione informatica.
- *Descrizione dei requisiti dell'utente.* È l'aspettativa dell'utente, cioè quello che vuole che accada una volta che il sistema informatico verrà messo in esercizio. È l'effetto atteso del sistema informatico sul dominio applicativo. Nel caso dell'azienda, l'utente si aspetta, per esempio, di poter derivare in modo automatico la giacenza di magazzino in base alle spedizioni fatte, al materiale grezzo ricevuto e alle attività produttive dell'azienda.
- *Descrizione dell'interfaccia del sistema (o specifica).* Si tratta della descrizione di come il sistema informatico che deve essere sviluppato dovrà presentarsi e interagire con l'utente. È l'anello che congiunge lo spazio del problema con quello della soluzione e per questo motivo si deve basare esclusivamente sui fenomeni condivisi. Per esempio, è la descrizione delle maschere di interazione tra utente e sistema e l'elenco delle funzioni presenti a menu.

Il Capitolo 4 tratterà nel dettaglio come studiare lo spazio del problema al fine di descriverlo in modo completo e coerente.

1.3 Lo spazio della soluzione

Una soluzione informatica è innanzi tutto costituita da un *codice o programma sorgente*, cioè un insieme organico di istruzioni scritte in un qualche linguaggio di programmazione. Il programma sorgente è normalmente organizzato in *moduli* che ne definiscono la struttura logica. In un linguaggio object-oriented un modulo è tipicamente una classe, mentre in un linguaggio tipo Pascal o C è solitamente un tipo di dato astratto costruito attraverso variabili e relative procedure di accesso. Ogni modulo viene memorizzato in uno o più file sor-

8 Capitolo 1 Lo sviluppo del software: creatività e ingegneria

genti. Per esempio, un tipo di dato astratto C è realizzato tipicamente utilizzando uno o più file “.h” e un file “.c”. In Java, una classe è un singolo file sorgente “.java”.

Il codice sorgente viene normalmente tradotto in *codice oggetto* o, nel caso di Java, in *codice intermedio (byte code)*. In particolare, in funzione del numero di file sorgenti e del linguaggio utilizzato si otterranno diversi tipi di risultati. Nel caso di Java, a ogni file sorgente “.java” corrisponde normalmente un file “.class” che contiene il codice intermedio. Nel caso di C, tipicamente a un file “.c” e a uno o più file “.h” corrisponde un file oggetto “.o”. Più file oggetto possono costituire una *libreria*. Diverse porzioni di codice oggetto, intermedio o librerie vanno a costituire i *programmi eseguibili* che si ottengono attraverso il linking. Tale attività può essere svolta in modo statico (come nel caso dei programmi C più semplici), utilizzando un linkage editor che produce un programma eseguibile (tipicamente memorizzato in un file “.exe”) e assemblando diversi file oggetto e librerie. Il codice così ottenuto è direttamente eseguibile dal computer e dal sistema operativo che lo controlla. Il linking può avvenire anche dinamicamente, cioè ogni qual volta durante l'esecuzione di un programma si richiede l'esecuzione di una qualche altra parte di codice non direttamente disponibile. Nel caso di Java e di altri linguaggi dello stesso tipo, il codice intermedio non viene tradotto in codice eseguibile, ma viene *interpretato da una virtual machine* che legge le singole istruzioni del codice intermedio, le traduce “al volo” e le esegue.

Questa breve descrizione ha introdotto i *componenti “grezzi” di un programma*, le diverse entità logiche (i moduli) e fisiche (i file) che costituiscono un sistema informatico. Ma ciò non è sufficiente a descrivere una soluzione software. In realtà, per caratterizzare un sistema informatico complesso è innanzitutto necessario studiare in maggior dettaglio come il codice sorgente viene organizzato e strutturato. Il concetto di *modulo* visto in precedenza permette di rappresentare il livello più semplice di strutturazione del codice sorgente ed è sufficiente per descrivere semplici programmi di qualche centinaia di linee di codice sorgente. Non appena la complessità del sistema aumenta, è necessario ricorrere ad altri concetti.

Un sistema informatico complesso è caratterizzato da un'architettura software che ne identifica le principali componenti e le modalità secondo le quali esse interagiscono. Come nel caso dei ponti stradali discussi in precedenza, i sistemi informatici sono costruiti sulla base di un insieme tutto sommato ridotto di schemi ricorrenti detti *stili architettonici*. Lo stile architettonico più noto è il client-server, nel quale un programma informatico è spezzato in una parte server, che normalmente contiene dati e servizi condivisi, mentre la parte client contiene il codice per interagire con l'utente. Il server risiede solitamente su una sola macchina mentre il client viene installato su tutte le macchine che vogliono poter accedere ai dati e servizi offerti dal server. L'interazione tra client e server è di tipo sincrono-pull: è sempre il client che avvia una richiesta di servizio (pull); inoltre il client resta in attesa fino a quando il server non ha risposto (sincrono). Questa semplice e schematica presentazione dello stile client-server illustra i principi di base secondo i quali costruire un'applicazione che si ispira a questo stile. È l'analogo dello stile “ponte sospeso”: il Golden Gate e il ponte Da Verrazzano hanno strutture diverse, ma si ispirano allo stesso stile. In modo simile, esistono tanti sistemi che hanno strutture diverse tutte ispirate allo stile client-server.

1.3 Lo spazio della soluzione 9

Esistono diversi stili architettonici che verranno discussi in un prossimo capitolo. Ognuno presenta vantaggi e svantaggi. Spesso, stili diversi vengono combinati per far sì che l'architettura risultante possa sfruttarne tutti i vantaggi. È quindi essenziale conoscere i differenti stili e capire come essi si possono combinare per costruire architetture complesse. Sono gli "schemi progettuali" di alto livello secondo i quali opera l'ingegnere del software.

Il concetto di architettura software è molto astratto e parecchio "distanse" dall'idea di moduli discussa in precedenza. Come si passa da un'architettura ai moduli che costituiscono il codice sorgente? È necessario raffinare l'architettura identificando i mattoni che la costituiscono. Nel fare questo, spesso si utilizzano i cosiddetti *design pattern*, cioè degli "schemi" secondo i quali costruire un singolo programma o una parte di esso. Per esempio, il pattern *model-view-controller* dice come costruire un programma che deve memorizzare delle informazioni e visualizzarle secondo diverse modalità all'utente, sulla base dei comandi invocati. È il caso di tanti programmi interattivi come gli editor e gli strumenti CAD. Un'altra forma di "schema" molto utile sono i *programming idiom*. Essi rappresentano sequenze di operazioni ricorrenti in un programma. Per esempio, la memorizzazione su file system di un oggetto Java richiede l'apertura di una stream di scrittura e la serializzazione dell'oggetto stesso. Altri esempi di programming idiom sono l'invocazione di un metodo remoto via RMI o la creazione di una dialog box.

Stili architettonici, design pattern e programming idiom sono gli schemi che un progettista informatico usa per costruire l'architettura della soluzione che, alla fine, sarà costituita da una serie di moduli sorgenti. La descrizione della soluzione è quindi innanzi tutto la descrizione dell'architettura software e degli "schemi" sui quali è basata. È la cosiddetta *documentazione di progetto*, che contiene tutta la conoscenza relativa alla struttura e organizzazione della soluzione.

In realtà, l'architettura software di un sistema informatico complesso non può essere descritta guardando solo al codice sorgente. Per capire la struttura del sistema è necessario capire come i diversi frammenti di codice saranno distribuiti sui diversi dispositivi che costituiscono l'infrastruttura run-time del sistema: computer in rete (si pensi ai sistemi client-server discussi in precedenza), ma anche palmari, cellulari e dispositivi specializzati come sensori o controllori specializzati. Un'architettura software deve quindi essere documentata prevedendo anche la descrizione delle modalità secondo le quali il sistema viene messo in esercizio o, utilizzando un'espressione inglese molto efficace, "deployed". Per fare ciò, è necessario identificare i diversi elementi che devono essere installati perché il sistema possa operare: file eseguibili, librerie, file di configurazione, file di dati.

Peraltro ciò non è ancora sufficiente. Un sistema informatico complesso deve essere descritto anche per quanto concerne le modalità secondo le quali opera a run-time. Un programma in esecuzione genera uno o più processi e thread di esecuzione. Per capire la struttura di una soluzione informatica (architettura), è essenziale fornire anche una descrizione del suo comportamento dinamico a run-time che illustri come i diversi processi e thread evolvono nel corso del tempo.

Infine, la soluzione non è costituita solo dal codice che va in esecuzione e dalla sua descrizione: occorre anche considerare tutto ciò che serve durante la fase di verifica del sistema informatico. Spesso infatti per testare un programma o una sua parte è necessario sviluppare *stub* e *driver* che permettono di provare il funzionamento di una sezione di

10 Capitolo 1 Lo sviluppo del software: creatività e ingegneria

codice “in isolation”. Inoltre, i test vengono condotti costruendo sequenze anche complesse di *casi di test*, cioè dati di prova che stimolano diverse parti del codice in modo da rendere possibile un controllo estensivo del suo comportamento. Stub, driver e casi di test sono anch’essi parte della soluzione e devono essere gelosamente conservati e documentati in quanto essenziali per provare il codice a fronte di cambiamenti ed estensioni che nel tempo si dovessero rendere necessari.

Un ultimo concetto importante che è essenziale ricordare in questa sede è il *componente*. Questo termine viene utilizzato per indicare nozioni spesso molto diverse tra loro. Secondo alcuni, un componente è un insieme di moduli (sorgenti) che realizzano una funzione ben identificabile e gestibile separatamente dal resto del sistema. In altri casi, con componente si intende un’unità di deployment, per esempio un file eseguibile o una libreria. Nel seguito del testo verrà privilegiata questa seconda accezione anche se, laddove ritenuto utile, verrà fatto riferimento anche alla prima.

Lo studio dello spazio della soluzione costituirà l’oggetto del Capitolo 6.

1.4 Riferimenti bibliografici

I temi e i problemi legati all’ingegneria del software sono discussi in numerosi libri, articoli e contributi di varia natura. Alcuni testi molto noti sull’ingegneria del software sono Ghezzi et al. [2003], Pressman [2005] e Sommerville [2004]. In realtà, lo studio del processo di sviluppo del software è iniziato alla fine degli anni ’60, con conferenze e lavori di ricerca che hanno iniziato a studiare e analizzare i problemi e le criticità emersi con il progressivo diffondersi delle applicazioni dell’informatica.

Un primo lavoro che ha segnato la nascita e lo sviluppo dell’ingegneria del software è l’articolo di Royce che introduce il ciclo di vita a cascata. Informazioni sul modello a cascata, sull’articolo originale di Royce che lo introduce e ulteriori informazioni sono reperibili su Wikipedia alla corrispondente voce (*Waterfall model*).

Un secondo contributo essenziale allo studio dell’ingegneria del software è il saggio *The mythical man-month* (Brooks [1995]). Originariamente pubblicato nel 1975, è stato esteso e arricchito nella versione del 1995. Brooks analizza le caratteristiche di complessi progetti di sviluppo di software nei quali ha operato e in particolare lo sviluppo del sistema operativo IBM OS/360. Le considerazioni di Brooks sono ancora oggi considerate dei capisaldi nello studio dello sviluppo di sistemi software.

L’approccio utilizzato in questo testo e centrato sullo studio degli spazi del problema e della soluzione ha origine nel lavoro di Jackson. Due sono i suoi contributi più importanti: un primo saggio nel quale ha iniziato ad affrontare i problemi legati al requirement engineering e la distinzione tra problema e soluzione (Jackson [1995]); un secondo volume che definisce in modo approfondito il processo di studio e descrizione del problema e degli schemi secondo i quali esso può essere analizzato e strutturato (Jackson [2001]).

Capitolo 2

UML: Unified Modeling Language

UML (*Unified Modeling Language*) è un insieme di diagrammi formali e semiformali usato per coadiuvare il progettista informatico nella descrizione del problema e della soluzione. Utilizza largamente i concetti e i principi del paradigma object-oriented. La prima versione di UML è stata sviluppata nel 1994 con lo scopo di fornire uno strumento unico di descrizione a supporto del lavoro dell'ingegnere del software. Prima del 1994 vi erano infatti diverse proposte di linguaggi di modellazione basate sul paradigma object-oriented, ma erano tra loro incongruenti. UML è continuamente aggiornato e arricchito: la versione più recente è la 2.0, ed è quella utilizzata in questo testo.

Questo capitolo presenta in modo sintetico le principali caratteristiche di UML. Lo stile della presentazione è indipendente dal fatto che si voglia descrivere un problema o una soluzione: per questo motivo, nel seguito, si userà l'espressione *ambito della descrizione* in modo volutamente ambiguo per caratterizzare ciò che viene descritto tramite UML. Alcuni diagrammi, peraltro, sono usati per descrivere un sistema informatico (sia esso parte del problema o della soluzione): in questi casi l'espressione *sistema informatico* sostituirà il generico *ambito della descrizione*.

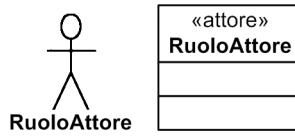
Si noti che il presente capitolo è pensato come un'introduzione alla versione 2.0 di UML ed è finalizzato alla comprensione degli esempi presenti nei capitoli successivi del libro. Non ha, quindi, la pretesa di essere una descrizione completa di UML.

2.1 Use case diagram

Lo *use case diagram* descrive le interazioni tra l'ambito della descrizione e le entità a esso esterne. Tipicamente l'ambito della descrizione è il sistema informatico da sviluppare e le funzioni che esso offre ai propri utenti; lo use case diagram può essere anche usato per caratterizzare una serie di processi e funzioni presenti in un'organizzazione, senza necessariamente che questi vengano gestiti tramite un sistema informatico (presente o da sviluppare). Gli elementi principali che costituiscono uno use case sono:

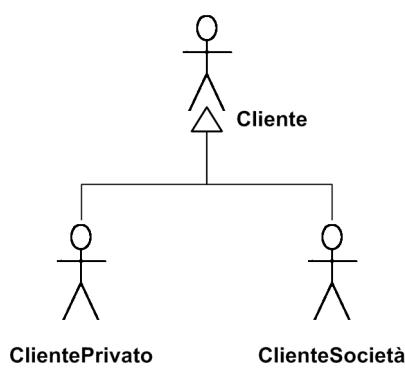
- attore
- caso d'uso (use case).

12 Capitolo 2 UML: Unified Modeling Language

**Figura 2.1** Rappresentazioni dell'entità Attore.

L'*attore* rappresenta un'entità che interagisce con l'ambito della descrizione e che è esterno a esso: può essere sia una persona (per esempio, gli utenti del sistema), sia un componente di un altro sistema. L'identificazione corretta degli attori in uno use case diagram è molto importante: in questo modo, infatti, si identifica chi fa che cosa. L'attore può essere rappresentato in due modi, come mostrato nella Figura 2.1.

I diversi attori che partecipano allo use case possono essere collegati tra loro attraverso la relazione di *generalizzazione*. Si ha una relazione di generalizzazione tra entità quando, partendo da una particolare entità, definita *padre*, si vogliono derivarne altre, definite *figli*, che mantengano le stesse proprietà del padre, oltre a possedere altre caratteristiche particolari. La relazione di generalizzazione è usata in quasi tutti i diagrammi UML. Nello use case diagram, in particolare, questa relazione permette di specificare, quando necessario, le caratteristiche degli attori dello use case e, di conseguenza, le loro interazioni con il sistema informatico. In particolare, la generalizzazione tra attori comporta che l'attore figlio erediti tutte le caratteristiche dell'attore padre e che possa interagire in tutti gli scenari in cui questi interagisce. La Figura 2.2 mostra come l'attore Cliente è specificato in due attori diversi: ClientePrivato, ClienteSocietà. Questo significa che sia una singola persona sia un rappresentante di una società può interagire con il sistema considerato in qualità di cliente. Entrambi questi attori usufruiranno delle stesse funzionalità di base, oltre a quelle specifiche definite per ciascuna *sottoclasse* (per esempio: una ditta effettua pagamenti in modo diverso rispetto a un privato).

**Figura 2.2** Generalizzazione di attori.

2.1 Use case diagram 13



Figura 2.3 Caso d'uso.

Un *caso d'uso* descrive una macro-funzionalità. Un caso d'uso è rappresentato con un'ellisse ed è identificato da un nome che identifica la macro-funzionalità fornita. La Figura 2.3 mostra un esempio.

Lo use case diagram è per sua natura un diagramma poco dettagliato. Nell'esempio della Figura 2.3, infatti, tutto ciò che riguarda l'archiviazione è contenuto nella macro-funzionalità definita "gestione archivio". Come nel caso degli attori, è possibile tuttavia aumentare il livello di dettaglio, definendo use case più specifici usando particolari relazioni. Le relazioni che intercorrono tra i diversi use case sono:

- generalizzazione
- relazione di inclusione (<<include>>)
- relazione di estensione (<<extend>>).

La generalizzazione tra use case comporta che lo use case figlio erediti tutte le caratteristiche e sia presente in tutti gli scenari dello use case padre. Lo use caso figlio, inoltre, presenta caratteristiche aggiuntive ed è richiamato in altri scenari rispetto al padre. L'esempio della Figura 2.4 mostra una generalizzazione di use case: la gestione archivio è specializzata nella gestione del personale e nella gestione degli ordini. Queste due gestioni presentano caratteristiche comuni, come la funzionalità di ricerca, e caratteristiche distinte, come la funzionalità di modifica dei dati del personale.

Le relazioni <<include>> e <<extend>> sono stereotipi della relazione di dipendenza. Uno *stereotipo* è un elemento di UML che viene qualificato in modo particolare al fine di evidenziarne proprietà o caratteristiche peculiari. Nel caso delle due relazioni considerate, l'elemento standard di UML è la *relazione di dipendenza*: lo stereotipo

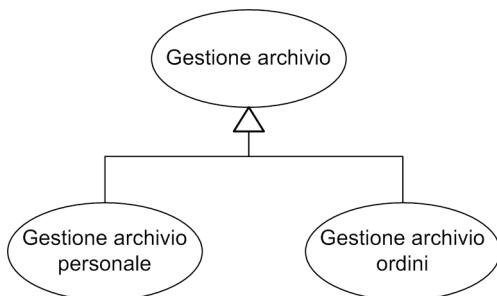
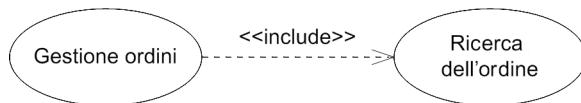


Figura 2.4 Generalizzazione di use case.

14 Capitolo 2 UML: Unified Modeling Language

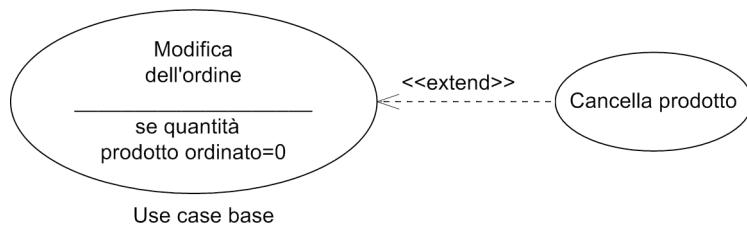
**Figura 2.5** Relazione <<include>>.

<<include>> specializza la dipendenza generica assegnandole un significato specifico. Uno stereotipo è rappresentato utilizzando l'elemento base al quale viene aggiunto il nome dello stereotipo incluso tra parentesi angolari. Lo stereotipo è molto usato in UML, tanto che alcuni di essi sono molto diffusi (come <<include>>): inoltre, è possibile creare ulteriori stereotipi secondo necessità.

La relazione <<include>> indica che, per la realizzazione completa dello use case che include, è necessario che sia realizzato lo use case incluso. In generale lo use case incluso indica un'attività che può essere realizzata indipendentemente dallo use case che include, e che è ricorrente in altri use case: in questo modo, quindi, è conveniente scorporare quest'attività e descriverla con uno use case separato. La relazione <<include>> è rappresentata con una freccia tratteggiata che parte dallo use case base e punta allo use case incluso. La Figura 2.5 mostra un esempio di relazione di <<include>>: per gestire gli ordini, è necessario trovare di volta in volta l'ordine su cui si deve operare.

La relazione <<extend>> indica che uno use case estende un secondo use case (detto use case base) quando descrive in modo più ampio e dettagliato una variante dell'attività dello use case base. Questa relazione è rappresentata come una freccia tratteggiata che parte dallo use case che estende la funzionalità e punta allo use case base. Una caratteristica rilevante della relazione <<extend>> è che l'esecuzione dello use case che estende lo use case base è opzionale, vale a dire che l'esecuzione è soggetta alla verifica di particolari vincoli espressi nel diagramma. I vincoli relativi all'esecuzione dello use case sono chiamati *extension point* e sono espressi nello use case base. Il diagramma della Figura 2.6 mostra che la funzionalità di modifica di un ordine può richiedere che venga eseguita anche la funzionalità di cancellazione di un prodotto dall'ordine considerato: questa funzionalità è eseguita nel momento in cui si verifica la condizione espressa nell'extension point, vale a dire se la quantità richiesta di un prodotto è stata portata a zero.

Spesso il significato e l'uso delle due relazioni <<include>> ed <<extend>> vengono tra loro confusi. In realtà la caratteristica principale che le due relazioni hanno in co-

**Figura 2.6** Relazione <<extend>> ed extension point.

2.1 Use case diagram 15

mune è che entrambe ampliano e arricchiscono lo use case base. Le caratteristiche che diversificano queste due relazioni sono le seguenti.

- Opzionalità dell'esecuzione: uno use case incluso è sempre eseguito, una volta che è eseguito lo use case di base. Al contrario, uno use case legato a un altro attraverso la relazione di estensione è eseguito solo in alcuni casi; in particolare, se si verificano particolari condizioni.
- Esecuzione dello use case: nel caso di inclusione, lo use case incluso è chiamato dallo use case che viene eseguito. Nel caso di <<extend>>, invece, è lo use case che estende che decide quando inserirsi nell'esecuzione dello use case di base: quest'ultimo è ignaro dell'esecuzione dello use case esteso.
- Rappresentazione delle relazioni: la relazione <<include>> è rappresentata con una freccia che parte dallo use case di base e punta a quello incluso. La relazione <<extend>>, invece, ha la direzione opposta, vale a dire che la freccia parte dallo use case che estende e punta a quello esteso.

La Figura 2.7 illustra un esempio completo di use case diagram. Si noti in particolare il riquadro che indica i confini di quanto descritto dal diagramma, cioè la porzione di un'organizzazione o processo per la quale si vuole costruire un sistema informatico di supporto.

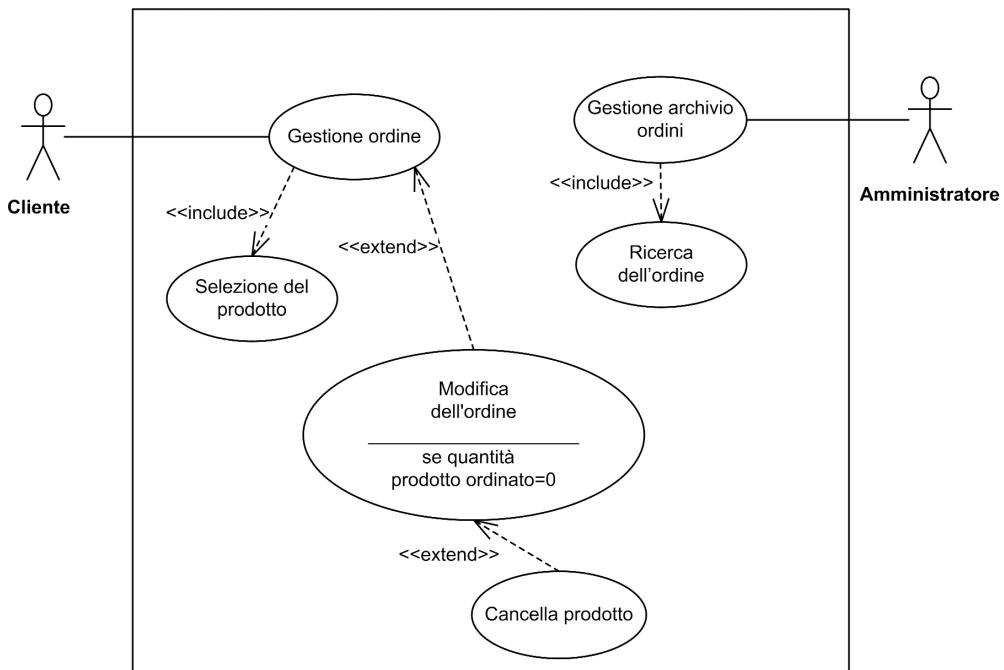


Figura 2.7 Un esempio completo di use case diagram.

2.2 Class diagram

Il *class diagram* illustra l'ambito di descrizione da un punto di vista *statico*, evidenziandone in particolare caratteristiche e mutue relazioni. Nel seguito sono descritti con maggior dettaglio gli elementi del class diagram: classi e relazioni.

Una *classe* descrive un insieme di entità dotate delle stesse caratteristiche e proprietà (detti *oggetti* o *istanze della classe*) ed è rappresentata graficamente con un rettangolo, come mostrato nella Figura 2.8: in funzione delle esigenze di descrizione, è possibile rappresentare una classe con un semplice rettangolo contenente solo il nome, oppure mostrando anche le informazioni di dettaglio. Nella descrizione estesa, per ogni classe si specifica il nome che la identifica, gli attributi che la descrivono e le operazioni che si possono effettuare sulle sue istanze. Il nome della classe è scritto con la lettera maiuscola ed è autodescrittivo.

Un *attributo* descrive una caratteristica propria di ogni istanza della classe. Le proprietà di un attributo sono le seguenti.

- *Visibilità*: indica se l'attributo può essere visibile ad altre classi o meno. I diversi tipi di visibilità sono:
 - *pubblico*: indicato con il simbolo “+” accanto al nome dell'attributo; indica che l'attributo considerato è visibile da tutte le classi;
 - *protetto*: indicato con il simbolo “#” accanto al nome dell'attributo; indica che l'attributo di una classe è visibile solo dalle classi che ereditano proprio da quest'ultima;
 - *privato*: indicato con il simbolo “-” accanto al nome dell'attributo; indica che l'attributo è visibile solo dalla classe in cui è definito.
- *Derivato*: la presenza del simbolo di derivazione (/) indica che l'attributo considerato è ottenuto derivandolo da altri attributi attraverso opportuni calcoli.
- *Nome*: identifica l'attributo. I nomi devono essere dati in modo che descrivano il significato dell'attributo stesso.

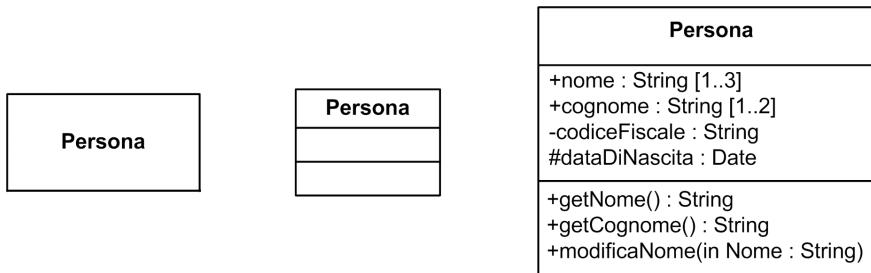


Figura 2.8 Rappresentazioni di una classe.

2.2 Class diagram 17

- **Molteplicità:** indica il numero di istanze dell'attributo che una classe può avere. Ad esempio, gli attributi `nome` e `cognome` hanno rispettivamente molteplicità da 1 a 3 e da 1 a 2: una persona può avere fino a tre nomi e fino a due cognomi.
- **Tipo:** indica il tipo di riferimento dell'attributo. Un attributo può essere, ad esempio, di tipo `Integer`, `String`, `Boolean`, o anche un'istanza di un'altra classe.
- **Valore di default:** indica il valore iniziale da assegnare a un attributo.

Il nome è l'unica proprietà obbligatoriamente presente per ogni attributo: le altre proprietà sono incluse o omesse in funzione del grado di dettaglio che si vuole ottenere.

La Figura 2.9 mostra due diversi tipi di rappresentazione degli attributi. Il primo tipo di rappresentazione mostra solo il nome degli attributi e la loro visibilità. La visibilità è indicata con i simboli a sinistra dei nomi: nell'esempio il nome e il cognome di una persona sono attributi pubblici, il codice fiscale è un attributo privato, mentre la data di nascita è un attributo protetto. Il secondo caso è più dettagliato: oltre al nome sono visibili anche la molteplicità dell'attributo e il tipo.

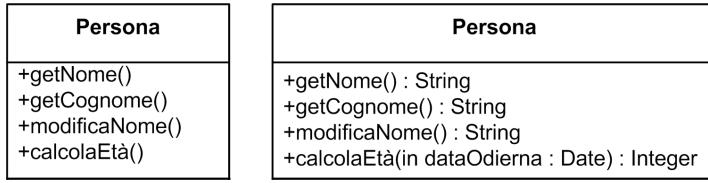
Un'*operazione* descrive un'azione eseguita sull'istanza di una classe ed è descritta dalle seguenti informazioni.

- **Visibilità:** indica se l'operazione è visibile da altre classi o meno. Il concetto e i tipi di visibilità possibili sono gli stessi utilizzati per la visibilità degli attributi.
- **Nome:** identifica l'operazione.
- **Parametri dell'operazione:** indicano i parametri di ingresso. Per ogni parametro si indica:
 - **direzione:** indica se il parametro è di ingresso (`in`), di uscita (`out`) o di ingresso uscita (`inout`); nel caso non si specifichi nulla il parametro è definito di ingresso (`in`);
 - **nome:** identifica e descrive il parametro;
 - **tipo:** indica di che tipo deve essere il parametro fornito (`Integer`, `String`, `Boolean`...);
 - **valore di default:** indica un valore iniziale predefinito da assegnare al parametro.
- **Tipo di ritorno:** indica il tipo del risultato che viene restituito dall'operazione. Si ricorda che un'operazione può non restituire alcun valore: in questi casi non si indica nulla o si utilizza la parola chiave `void`.

Persona	Persona
+nome +cognome -codiceFiscale #dataDiNascita	+nome : String [1..3] +cognome : String [1..2] -codiceFiscale : String #dataDiNascita : Date

Figura 2.9 Rappresentazioni degli attributi.

18 Capitolo 2 UML: Unified Modeling Language

**Figura 2.10** Rappresentazione delle operazioni di una classe.

Il nome dell'operazione, come quello degli attributi, è obbligatorio, mentre tutto il resto della descrizione può essere omesso, in funzione del grado di dettaglio che si vuole dare allo schema. Nella Figura 2.10 sono mostrate le operazioni che si possono eseguire sull'istanza della classe persona. In particolare, è utile soffermarsi sull'operazione `calcolaEtà`: ha un parametro in ingresso chiamato `dataOdierna`, che è di tipo `Date`, e restituisce come risultato l'età in anni della persona, parametro di tipo `Integer`.

Vi sono due ulteriori tipi di classi: le association class e le classi astratte.

Una *association class* è una classe legata a una particolare associazione. Contiene attributi significativi per l'associazione in sé e non singolarmente riconducibili a nessuna delle classi coinvolte nell'associazione. L'association class sarà descritta in dettaglio in seguito.

Una *classe astratta* è una classe che non può essere istanziata direttamente; contiene una serie di attributi e dichiarazioni di operazioni. L'implementazione delle operazioni dichiarate è propria delle classi che implementano la classe astratta. Una classe astratta è utile nel caso in cui ci siano degli elementi con alcuni attributi comuni, ma con altre caratteristiche.

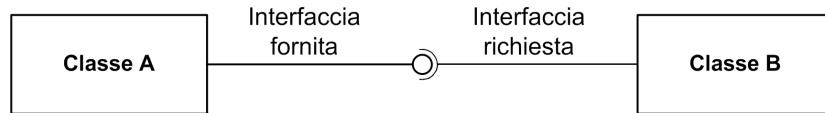
Un'*interfaccia* è una classe che non ha implementazione: presenta, infatti, solo dichiarazioni di operazioni. Le interfacce sono di due tipi (Figura 2.11).

1. *Interfaccia fornita*: la classe che espone questo tipo di interfaccia fornisce le operazioni che vi sono dichiarate.
2. *Interfaccia richiesta*: la classe che espone questo tipo di interfaccia ha bisogno delle operazioni in essa dichiarate per poter svolgere le proprie elaborazioni.

Nell'esempio della Figura 2.12, la Classe A mette a disposizione un'interfaccia, mentre la Classe B la richiede.

**Figura 2.11** Rappresentazioni di interfacce.

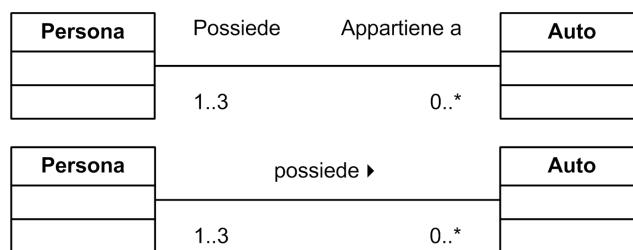
2.2 Class diagram 19

**Figura 2.12** Esempio di interfacce.

Un'*associazione* è una relazione statica che lega le classi tra di loro. Ogni associazione deve essere definita in modo univoco tramite un nome che descrive il significato dell'associazione stessa. Un'associazione è descritta, inoltre, dal ruolo ricoperto dalle classi coinvolte. Il ruolo giocato da una classe in un'associazione è rappresentato accanto a essa. Per esempio, nella Figura 2.13 le parole chiave *Possiede* e *Appartiene a* identificano i ruoli delle classi *Persona* e *Auto* nella relazione *possiede*. Il triangolo nero posto di fianco al nome dell'associazione indica come leggere e interpretare il nome dell'associazione stessa (“una persona *possiede* un’auto” e non viceversa).

Una caratteristica importante delle associazioni è il concetto di molteplicità. La *molteplicità* di un'associazione è riferita alle classi: indica il numero di istanze di una determinata classe coinvolte nella relazione. Le dichiarazioni di molteplicità possibili sono:

- 1: indica che nell'associazione è coinvolta una e una sola istanza della classe
- 0..1: indica che nell'associazione può essere coinvolta o meno un'istanza di una classe
- 0..*/1..*: indica che le istanze coinvolte possono essere molte. In questo caso non è indicato un numero massimo preciso; la presenza del limite inferiore uguale a uno indica che almeno un'istanza della classe è coinvolta
- 0..n/1..n: indica che le istanze coinvolte possono essere al massimo n
- n: indica che le istanze della classe considerata coinvolte nell'associazione sono esattamente n.

**Figura 2.13** Descrizioni alternative di un'associazione tra due classi.

20 Capitolo 2 UML: Unified Modeling Language

La molteplicità di un'associazione è indicata come illustrato nel diagramma della Figura 2.13, che descrive la relazione di proprietà che lega una persona a un'auto. I modi per leggere tale relazione sono due.

- Una persona possiede zero o più auto. In questo caso la molteplicità “0” indica che una persona può anche non possedere alcuna auto.
- Un'auto appartiene a un numero di persone che varia tra uno e tre.

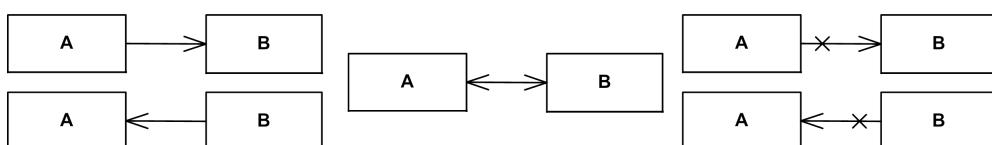
Un'associazione fornisce anche informazioni sulla navigabilità tra le classi considerate. La navigazione indica la direzione di percorrenza di un'associazione: in particolare, in presenza di quest'informazione si deduce che la classe sorgente ha un riferimento alla classe obiettivo della navigazione. La navigabilità è rappresentata con una freccia aggiunta all'associazione. I tipi di navigabilità sono i seguenti.

- *Unidirezionale*: indica che è specificata una particolare direzione dell'associazione, mentre l'altra non è specificata e rimane, quindi, indefinita. Negli esempi (a) della Figura 2.14 sono mostrate le due alternative unidirezionali.
- *Bidirezionale*: la navigabilità è specificata in entrambe le direzioni. Di conseguenza, l'associazione presenta la punta di una freccia in entrambe le estremità, come mostrato nell'esempio (b) della Figura 2.14.
- *Non navigabile*: la navigabilità è esplicitamente proibita verso una direzione o entrambe. In questo caso, l'associazione non presenta la punta di una freccia ma una croce. Nell'esempio (c) della Figura 2.14, l'associazione in alto, che va dalla classe A alla classe B non è navigabile nella direzione che va da B ad A.

La navigabilità è opzionale e, quindi, può anche non essere inserita in un diagramma.

Si noti che il concetto di navigabilità è diverso dal verso di lettura dell'associazione (il triangolo nero della Figura 2.13). La navigabilità chiarisce in quale direzione da un'istanza di una classe si può passare a istanze di altre classi seguendo una specifica associazione (corrisponde, cioè, al concetto di puntatore). Il verso di lettura serve solo a facilitare la comprensione del diagramma e a interpretare correttamente il nome dell'associazione.

L'associazione della Figura 2.13 coinvolge due classi ed è detta *associazione binaria*. Un'associazione tra più classi è definita *n-aria* ed è rappresentata nella Figura 2.15. Per l'associazione n-aria valgono le stesse considerazioni dell'associazione binaria. Ad esem-



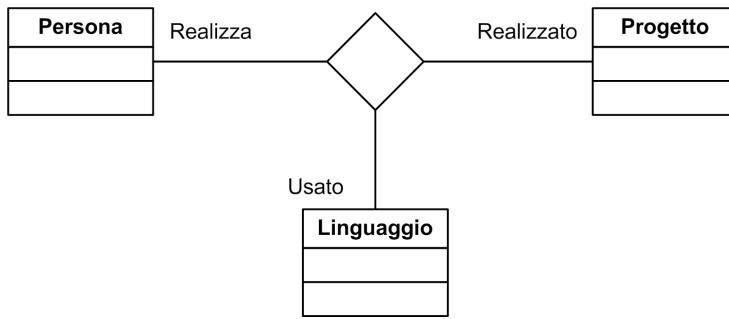
a) navigabilità unidirezionale

b) navigabilità bidirezionale

c) non navigabilità

Figura 2.14 Navigabilità di un'associazione.

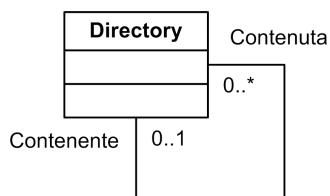
2.2 Class diagram 21

**Figura 2.15** Associazione n-aria.

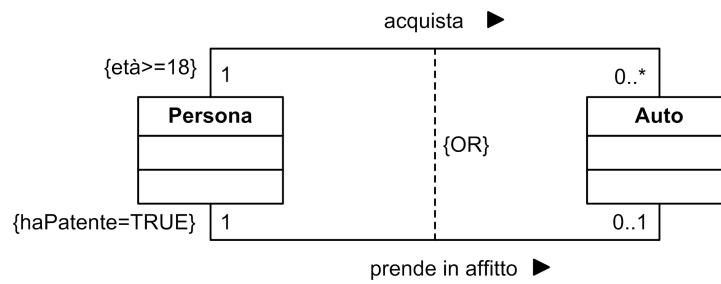
pio, anche in questo caso è possibile assegnare a ogni classe un ruolo, che indica in che modo la classe è coinvolta nell'associazione. Nella Figura 2.15, si descrive che una persona realizza un progetto in un particolare linguaggio tra quelli che conosce: questo fatto non è descrivibile con le associazioni binarie perché si perderebbero le informazioni che associano persona, progetto e linguaggio.

Infine, è possibile realizzare anche un'*associazione riflessiva*. Un'associazione è riflessiva se le istanze tra cui è applicata appartengono alla stessa classe. Nella Figura 2.16 si descrive che una directory può essere contenuta in un'altra e può a sua volta contenere zero o più directory.

Un'associazione può essere specificata da particolari vincoli. I *vincoli* sono delle condizioni espresse sull'associazione stessa o sulle classi coinvolte: tali condizioni devono essere verificate per realizzare l'associazione. I vincoli si esprimono tra parentesi graffe, in linguaggio naturale o tramite espressioni. Le associazioni della Figura 2.17 ne presentano diversi: il vincolo sull'associazione *acquista* indica che solo una persona che abbia compiuto diciotto anni può comprare un'auto, mentre il vincolo dell'associazione *prende in affitto* indica che solo chi è munito di patente può affittare un'auto (*età* e *haPatente* sono due ulteriori attributi di *Persona*). In UML è possibile anche vincolare tra loro due associazioni: nella Figura 2.17 il vincolo *{OR}* descrive che una persona può affittare un'auto oppure comprarne una, ma non entrambe le cose.

**Figura 2.16** Associazione riflessiva.

22 Capitolo 2 UML: Unified Modeling Language

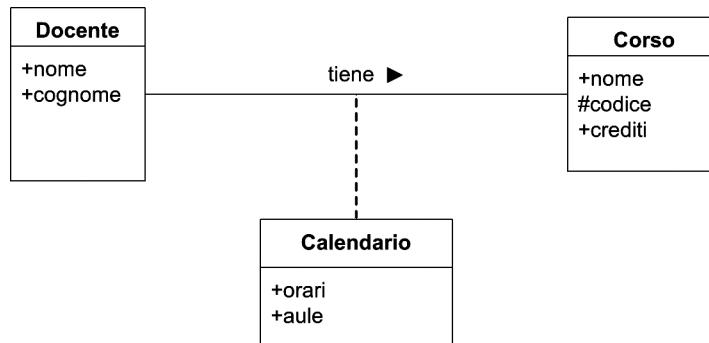
**Figura 2.17** Vincoli sulle associazioni.

Come già definito in precedenza, un'*association class* è una classe che descrive le informazioni proprie dell'associazione e non delle classi connesse a quest'ultima. Nella Figura 2.18 è mostrata l'*association class* **Calendario**: le informazioni sugli orari e sulle aule in cui un dato professore tiene un suo corso non sono proprie né dell'entità **Docente**, né dell'entità **Corso**, ma sono introdotte dall'associazione per descrivere il legame tra corso e docente.

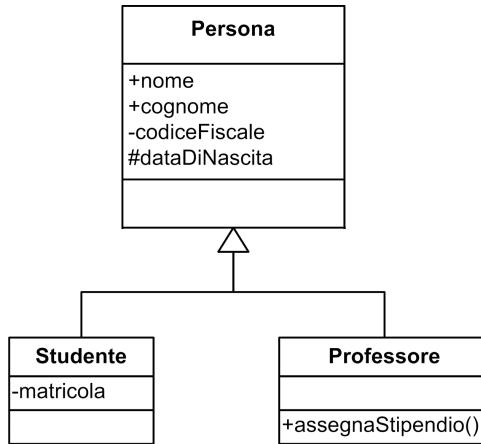
Oltre alle associazioni sopra descritte, le classi possono essere collegate anche da associazioni particolari:

- generalizzazione
- aggregazione o condivisione
- composizione.

Una *relazione di generalizzazione* si ha nel caso in cui una classe è specializzata in classi diverse che, oltre a mantenere le caratteristiche di fondo, ne posseggono di nuove e diverse. Nell'esempio della Figura 2.19, le classi **Studente** e **Professore** specializzano la classe **Persona**: questo significa che hanno tutti gli attributi e le operazioni descritte nella classe

**Figura 2.18** Esempio di association class.

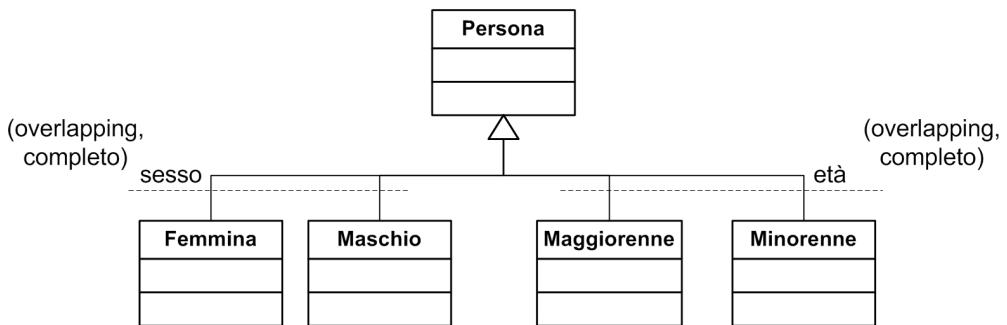
2.2 Class diagram 23

**Figura 2.19** Esempio di generalizzazione.

Persona. Inoltre, la classe Studente ha l'attributo matricola in più mentre la classe Professore ha l'operazione assegnaStipendio in più rispetto alla classe Persona. Un'istanza di Studente o Professore è automaticamente un'istanza di Persona. La classe Professore non ha l'attributo matricola, così come la classe Studente non ha l'operazione assegnaStipendio.

In alcuni casi una generalizzazione è realizzata attraverso più vincoli che guidano la partizione delle sottoclassi: tali vincoli sono espressi nel generalization set. Nella generalizzazione della Figura 2.20, si può specificare una persona in base al sesso (maschio/femmina) o all'età (minorenne/maggiorenne). Ogni generalizzazione può essere:

- overlapping: nel caso in cui un'istanza appartenga a due sottoclassi di due set distinti
- disgiunta: nel caso in cui un'istanza appartenga solo a un'unica sottoclasse

**Figura 2.20** Generalizzazione con più vincoli.

24 Capitolo 2 UML: Unified Modeling Language

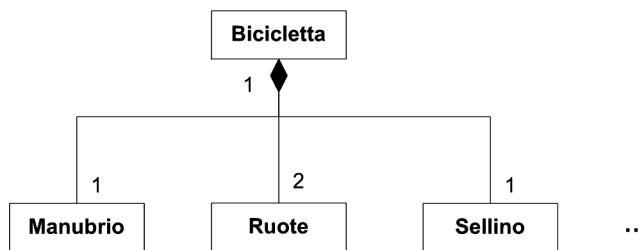
**Figura 2.21** Relazione di aggregazione.

- completa: nel caso in cui tutte le possibili istanze della superclasse appartengono a una delle sottoclassi definite
- incompleta: nel caso in cui almeno un'istanza della superclasse non appartenga ad almeno alcuna delle sottoclassi definite nella generalizzazione.

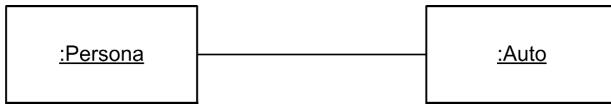
L'esempio della Figura 2.20 mostra una generalizzazione completa e overlapping. È overlapping perché, ad esempio, una ragazza di vent'anni rientra nelle due sottoclassi Femmina e Maggiorenne che appartengono a set diversi; è anche completa perché qualsiasi istanza di persona rientra sicuramente in due delle quattro sottoclassi definite. Il diagramma della Figura 2.19, invece, mostra una generalizzazione disgiunta e incompleta. È una associazione disgiunta perché una persona o è uno studente o è un professore, ma non può essere entrambi; è incompleta perché una persona, che non è né uno studente né un professore, non ricade in nessuna delle sottoclassi definite.

La relazione di *condivisione* o *aggregazione* indica che una classe è vista come un insieme delle istanze di altre classi, dette di aggregazione: in questo caso gli attributi della classe composta, così come le operazioni, sono diversi e indipendenti da quelli delle varie classi che la compongono. Nella Figura 2.21 la relazione di aggregazione descrive che un club è un insieme di più persone. In questa relazione un'istanza della classe Persona può esistere anche se non appartiene ad alcun club, inoltre può appartenere a più club contemporaneamente.

La relazione di *composizione* è un particolare tipo di aggregazione: nella composizione ogni istanza delle classi componenti, può appartenere a una sola istanza della classe composta. Nella Figura 2.22 la classe Bicicletta è formata, tra le altre cose, da un manubrio, un sellino e due ruote. Questo significa che l'istanza di un manubrio forma e

**Figura 2.22** Relazione di composizione.

2.2 Class diagram 25

**Figura 2.23** Oggetti e link.

appartiene una sola istanza di bicicletta; un manubrio, infatti, non può appartenere a due biciclette contemporaneamente.

Classi e associazioni rappresentano categorie di concetti (per esempio, il concetto di bicicletta e le relazioni con i suoi componenti). Spesso è necessario descrivere specifiche istanze di classi e associazioni. Per questo motivo UML utilizza i concetti di oggetto e link.

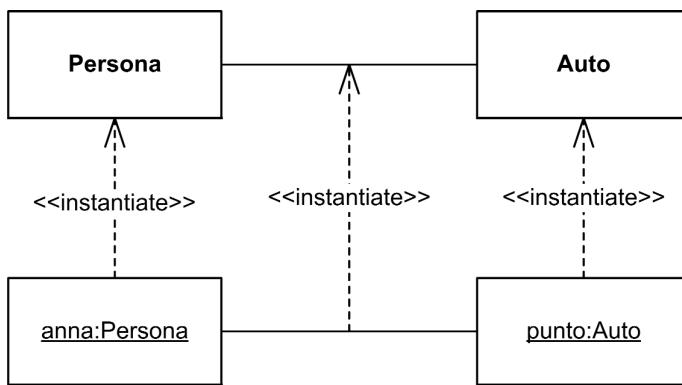
Un *oggetto* è un'istanza di una classe: è rappresentato con il simbolo della Figura 2.23. Un oggetto ha un identificatore composto da due parti:

nomeOggetto:nomeClasse (il nome dell'oggetto è opzionale).

Un oggetto è descritto dal suo identificatore e, facoltativamente, dal valore degli attributi, cioè dalle informazioni che caratterizzano quello specifico oggetto.

Un *link* è un'istanza di un'associazione. Viene utilizzato per rappresentare il legame tra oggetti, in coerenza con quanto previsto dal diagramma delle classi relativo.

Un diagramma che include oggetti e link viene denominato *object diagram* (o diagramma degli oggetti). La Figura 2.24 illustra un esempio di object diagram collegato alla relativa porzione di class diagram. La relazione <<instantiate>> lega oggetti e link alle classi e all'associazione dalle quali sono istanziati.

**Figura 2.24** Esempio di object diagram.

26 Capitolo 2 UML: Unified Modeling Language

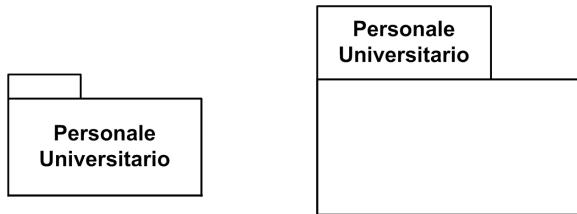


Figura 2.25 Rappresentazione di un package.

2.3 Package diagram

Il *package diagram* permette di raggruppare insieme più entità mostrandone le mutue relazioni. Grazie al meccanismo di raggruppamento, è possibile operare a un livello di astrazione superiore rispetto a diagrammi in cui si considera la singola entità. In particolare, il package diagram è spesso usato per raggruppare le classi definite in un class diagram.

L'elemento principale di questo tipo di diagramma è il package. Un *package* contiene un numero arbitrario di elementi UML e permette di trattarli come un'entità unica. Un package è identificato da un nome che descrive la caratteristica del gruppo che il package rappresenta e che è inserito al centro del package stesso oppure, nella maggior parte dei casi, sulla parte in alto. La Figura 2.25 mostra le due alternative di rappresentazione di un package.

La seconda rappresentazione della Figura 2.25, con il nome sulla parte superiore del simbolo, è usata nel caso in cui si voglia rappresentare anche il contenuto del package. È possibile rappresentare le classi contenute nel package sia come un elenco che graficamente, disegnandole nel box del package. In questo secondo caso, il dettaglio delle classi rappresentate può arrivare fino a includere gli attributi e le operazioni proprie di ogni singola classe, oltre al nome della stessa. La Figura 2.26 mostra le diverse modalità

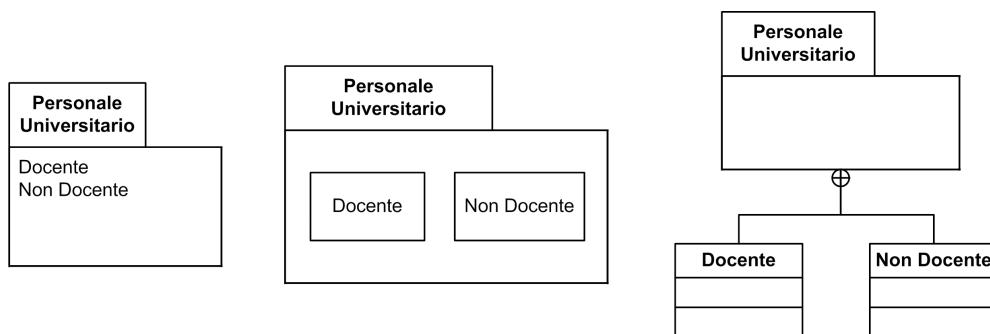


Figura 2.26 Package contenente classi.

2.3 Package diagram 27

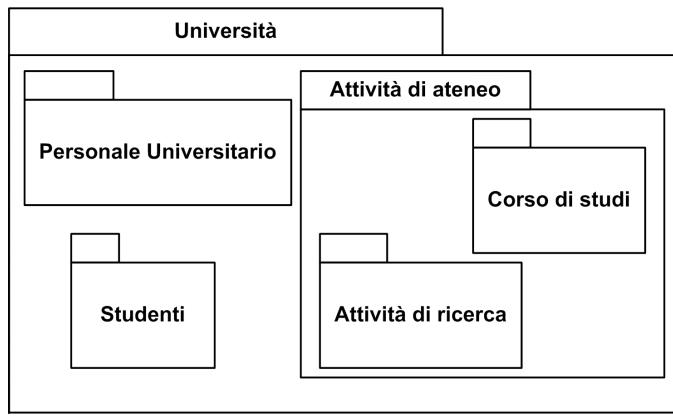


Figura 2.27 Gerarchia di package (o anche package annidati).

di rappresentazione di un package che contiene solo classi: le classi contenute possono essere legate da relazioni o associazioni che sono rappresentate nel package, nel caso siano di aiuto per una maggior comprensione del diagramma considerato.

Un package, oltre a contenere delle entità base come le classi, è in grado di contenere anche altri package formando così una gerarchia. La Figura 2.27 mostra la rappresentazione di package annidati, vale a dire di package che contengono altri package.

I package presenti in un diagramma sono legati tra loro tramite relazioni di dipendenza e in particolare quella specificata dallo stereotipo <<access>>. Questo stereotipo indica che all'interno di un package si utilizzano attributi e operazioni di un altro package. La Figura 2.28 mostra un esempio di dipendenza di tipo <<access>>: la classe Docente appartiene al package Personale Universitario, ma è utilizzata anche nel package Corso di studi per descrivere l'associazione insegna.

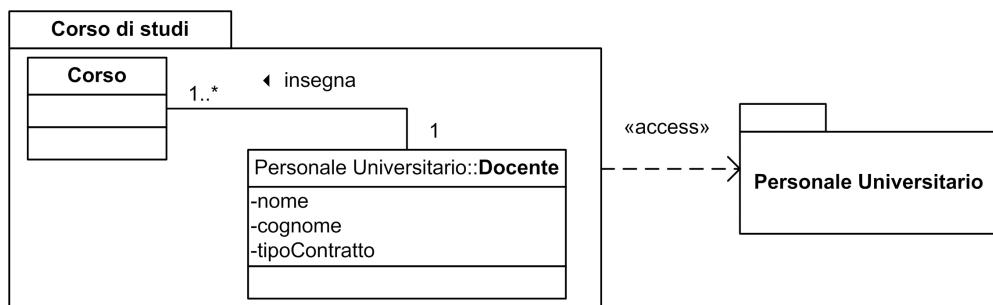


Figura 2.28 Dipendenza <<access>>.

2.4 Activity diagram

Un *activity diagram* descrive il flusso di azioni necessarie per eseguire una particolare procedura. Gli elementi principali sono:

- nodi azioni
- archi
- nodi di inizio e fine
- nodi di controllo
- nodi oggetto
- segnali
- partizione.

Un'azione è un'unità di lavoro atomica: essa è rappresentata nel diagramma con un rettangolo arrotondato ed è identificata con una voce verbale che descrive l'azione stessa. L'esempio della Figura 2.29 mostra un activity diagram molto semplice, formato da due nodi azione: Ricerca ordine ed Evadi ordine.

Ogni diagramma delle attività ha due nodi particolari: Inizio e Fine. Inizio è il punto di partenza del diagramma, indica la prima azione da eseguire ed è rappresentato

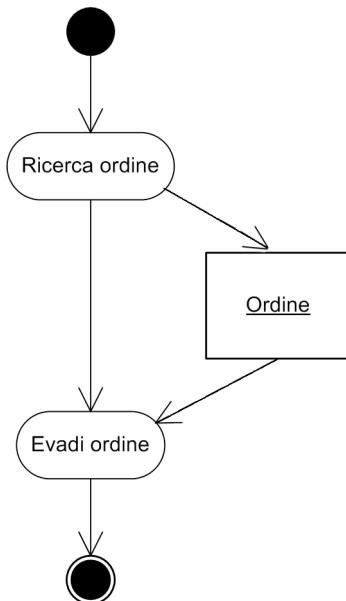


Figura 2.29 Esempio di activity diagram con passaggio di oggetto.

2.4 Activity diagram 29

da un cerchio con solo archi in uscita. Il nodo *Fine* indica la conclusione dello scenario descritto e ha solo archi in entrata. L'azione finale del diagramma deve puntare sempre al nodo finale. Nell'esempio della Figura 2.29 sono mostrati chiaramente sia il nodo iniziale, in alto, sia quello finale in basso.

Un *arco*, o *flusso*, collega tra loro i nodi. L'insieme degli archi del diagramma rappresenta il flusso di esecuzione complessivo. Un arco è rappresentato con una semplice freccia e indica che l'azione puntata dall'arco è eseguita al termine dell'azione da cui questa parte. Il flusso delle azioni progredisce solo nel momento in cui l'azione considerata è completata. La Figura 2.29 mostra un semplice flusso sequenziale.

L'esecuzione di un'azione può richiedere o produrre un oggetto. L'oggetto viene esplicitamente indicato nel caso in cui si voglia evidenziare il legame che si crea tra due azioni attraverso il passaggio di informazioni dalla prima alla seconda. L'oggetto è legato alle azioni attraverso archi che rappresentano in modo intuitivo la produzione dell'informazione e il suo consumo. Nell'esempio della Figura 2.29, la ricerca di un ordine produce un risultato che viene utilizzato dall'azione *Evadi ordine*.

Una funzionalità complessa non è costituita da una semplice successione di azioni in sequenza: può presentare, per esempio, azioni da eseguire contemporaneamente o sotto particolari condizioni. A questo scopo sono stati introdotti i *nodi di controllo*:

- Fork
- Join
- Decision
- Merge

Il nodo *fork* descrive l'esecuzione in parallelo di più azioni: quelle puntate dal nodo fork sono avviate contemporaneamente e in parallelo. Un fork è applicato quando le azioni considerate sono indipendenti tra loro e possono quindi essere eseguite contemporaneamente. La Figura 2.30 mostra la rappresentazione del nodo fork: una barra nera puntata da un solo arco e dalla quale partono due o più archi verso le azioni che sono eseguite in parallelo. La Figura 2.30 illustra l'evasione di un ordine: dopo aver trovato il prodotto presente nell'ordine, si calcola l'importo e in parallelo si preleva il prodotto desiderato dal magazzino. Le due azioni *Calcola importo ordine* e *Preleva prodotto da magazzino* sono indipendenti tra loro e possono essere svolte in parallelo.

Il nodo *join* è il duale del fork. Il join specifica che un'azione è eseguita solo nel momento in cui le azioni precedenti hanno terminato la propria esecuzione. Il nodo join è definito anche *sincronizzazione* perché sincronizza due rami svolti parallelamente, generando un unico flusso di esecuzione; è rappresentato da una barra nera, come il nodo fork. La Figura 2.30 mostra un caso di join: l'ordine soddisfatto può essere spedito solo quando le due azioni precedenti si sono concluse, vale a dire quando l'importo è stato calcolato e il prodotto risulta pronto per la spedizione.

Il nodo di controllo *decision* indica che l'esecuzione di un'azione dipende dal verificarsi di una determinata condizione chiamata *guard*. Questo nodo descrive il fatto che, al termine di una particolare azione, lo scenario può proseguire in modo diverso con azio-

30 Capitolo 2 UML: Unified Modeling Language

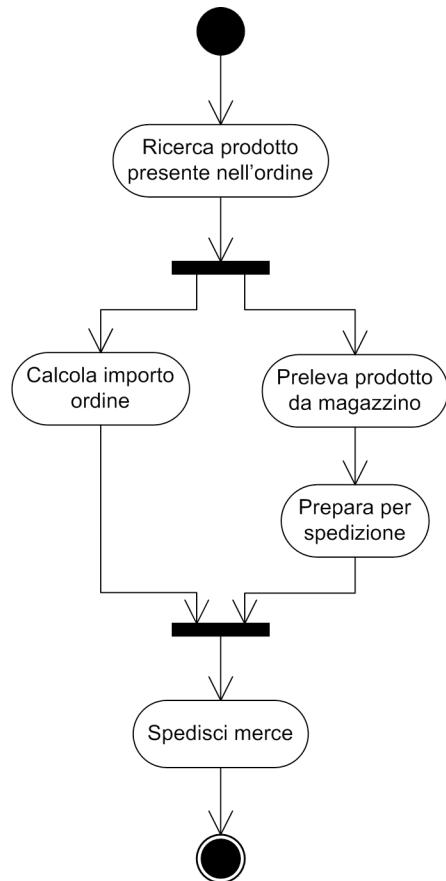


Figura 2.30 Descrizione di funzionalità con uso di nodi fork e join.

ni che dipendono da specifiche situazioni che si vengono a verificare. Questo nodo è chiamato anche *if* perché spesso la condizione espressa ha due sole possibilità: alternativa verificata o non verificata. Il nodo decision è rappresentato con un rombo vuoto puntato da un arco solo e dal quale partono almeno due archi. Sull'arco entrante si specifica la condizione da soddisfare e sugli archi uscenti si specificano le alternative della condizione. La Figura 2.31 mostra un esempio di decision applicato alla verifica che un ordine sia relativo a prodotti diversi. Il nodo decision posizionato immediatamente prima dell'azione *Spedisci merce* è usato per controllare la presenza di altri prodotti: se ci sono altri prodotti nell'ordine, si ripete l'iter partendo dall'inizio, altrimenti si procede alla spedizione dell'ordine.

Il nodo di controllo *merge* è il duale del nodo decision e descrive un punto del processo in cui si ricongiungono due o più flussi alternativi. A differenza del nodo join, non

2.4 Activity diagram 31

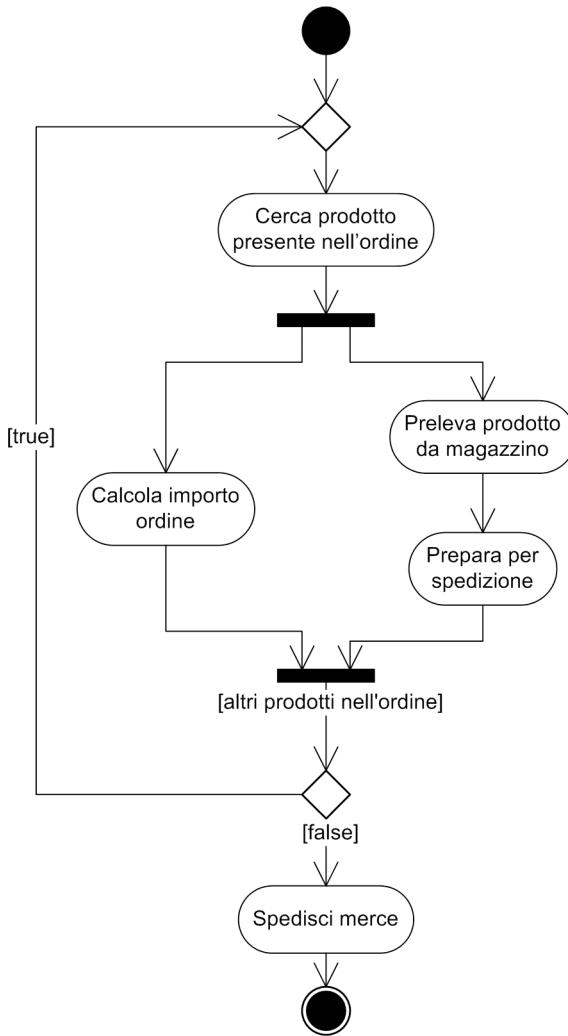


Figura 2.31 Esempio di activity con uso dei nodi decision e merge.

implica una sincronizzazione in quanto si è in presenza di un unico flusso di esecuzione (quello selezionato a fronte dell'esecuzione del nodo decision). Il merge è anch'esso descritto da un rombo. La Figura 2.31 mostra l'uso di un nodo merge (posizionato immediatamente dopo il nodo di inizio): gli archi che puntano al merge indicano che la ricerca del prodotto e il flusso che ne segue sono eseguiti per il primo prodotto dell'ordine e per ogni altro eventualmente presente nell'ordine stesso (ciclo do-while).

32 Capitolo 2 UML: Unified Modeling Language

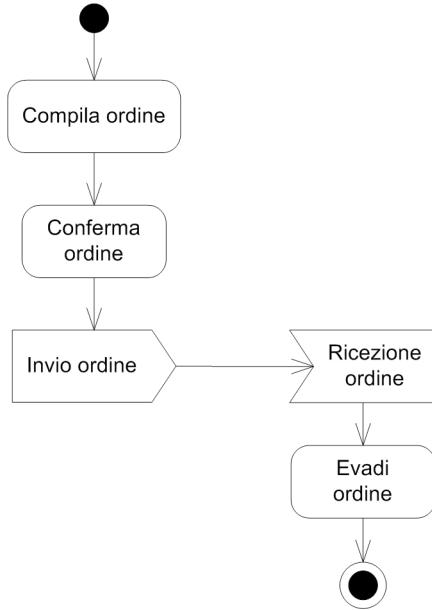


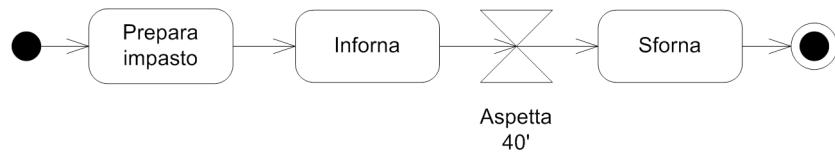
Figura 2.32 Esempio di segnale inviato e ricevuto.

Un *segnalet* è un elemento usato per specificare le relazioni tra azioni. In particolare, rappresenta un’azione che è lanciata o attesa per proseguire nell’esecuzione delle attività. I segnali sono di tre diversi tipi.

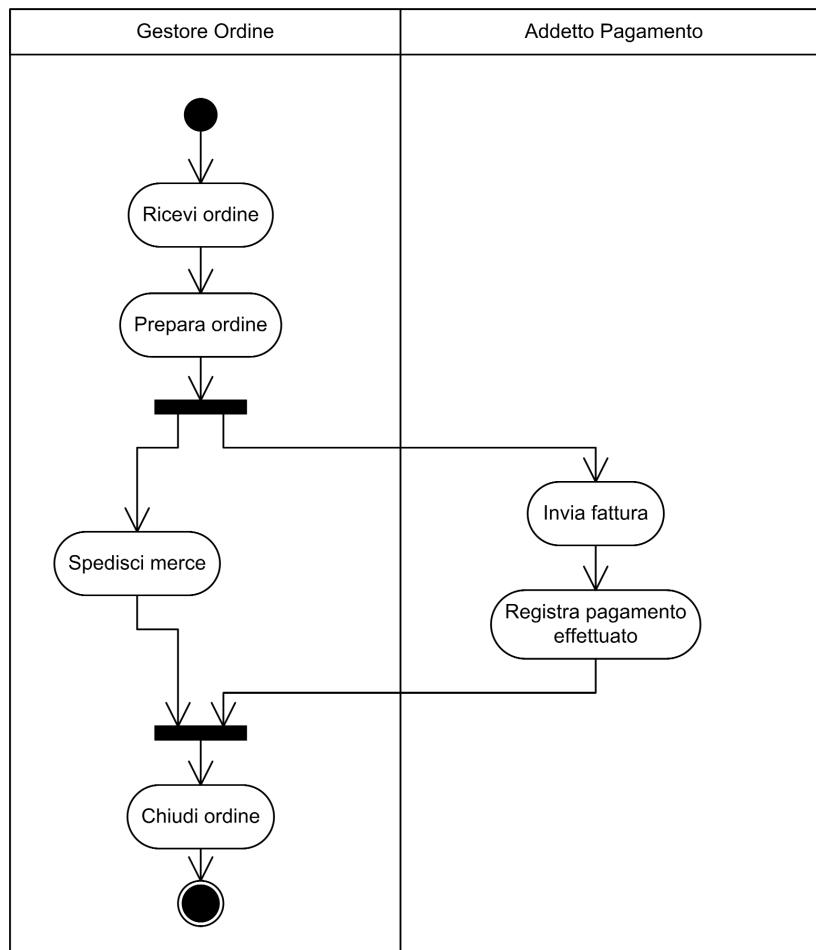
- Segnale inviato: indica un’azione lanciata alla fine di un’azione. È rappresentato da un rettangolo appuntito sul lato destro: l’esempio della Figura 2.32 mostra il segnale inviato *Invio ordine*.
- Segnale ricevuto: è quello che un’azione attende per poter esser eseguita. In mancanza del segnale, l’azione rimane in attesa. È rappresentato da un rettangolo con la punta rientrante sul lato sinistro: la Figura 2.32 mostra il segnale ricevuto *Ricezione ordine*.
- Segnale temporale: non è associato ad alcuna azione, ma a un preciso intervallo di tempo che vincola l’esecuzione dell’attività a valle. Un’azione in attesa di un segnale temporale è eseguita solo se il vincolo temporale descritto dal segnale è soddisfatto (si veda la Figura 2.33).

L’esecuzione di un’attività particolarmente complessa spesso comporta che le azioni descritte non siano eseguite dallo stesso attore. In questo caso il diagramma viene suddiviso in *partizioni* per raggruppare le diverse azioni di uno scenario in base agli attori che sono responsabili della loro esecuzione. Per esempio, si consideri ancora una volta la descrizione della gestione di un ordine commerciale, in questo caso distinguendo la parte

2.4 Activity diagram 33

**Figura 2.33** Esempio di segnale temporale.

relativa all'evasione dell'ordine e quella relativa al pagamento. Il gestore dell'ordine esegue il primo insieme di azioni, mentre l'addetto ai pagamenti si occupa del secondo insieme. La Figura 2.34 mostra chiaramente le due partizioni identificate con il nome del-

**Figura 2.34** Esempio di partizione in un activity diagram.

34 Capitolo 2 UML: Unified Modeling Language

l'attore responsabile delle azioni: l'invio della fattura al cliente è un'azione eseguita dall'addetto ai pagamenti, mentre la spedizione dell'ordine e la sua chiusura, ad esempio, sono azioni eseguite dal gestore dell'ordine.

I processi descritti negli activity diagram possono essere molto complessi: questo può portare alla creazione di diagrammi poco leggibili e, di conseguenza, poco comprensibili. Per ovviare al problema si può *modularizzare il diagramma* ricorrendo all'uso di attività che richiamano altre attività: nel diagramma principale si inserisce un'azione che rimanda a un altro diagramma all'interno del quale è descritto con maggior dettaglio l'insieme di azioni da svolgere. È lo stesso meccanismo utilizzato nel caso dei sottoprogrammi di un linguaggio di programmazione tradizionale. L'attività invocata è rappresentata come un insieme di azioni racchiuse in un rettangolo arrotondato, identificato dal nome dell'azione che lo richiama. Nel diagramma possono essere specificati anche parametri di input e di output, rappresentati da rettangoli come nel caso degli oggetti. Infine, l'azione che nel diagramma principale richiama il diagramma di dettaglio presenta al suo interno un simbolo illustrato nella Figura 2.35 per indicare che l'articolazione dell'azione *Prepara ordine* è illustrata nel diagramma relativo (illustrato a destra).

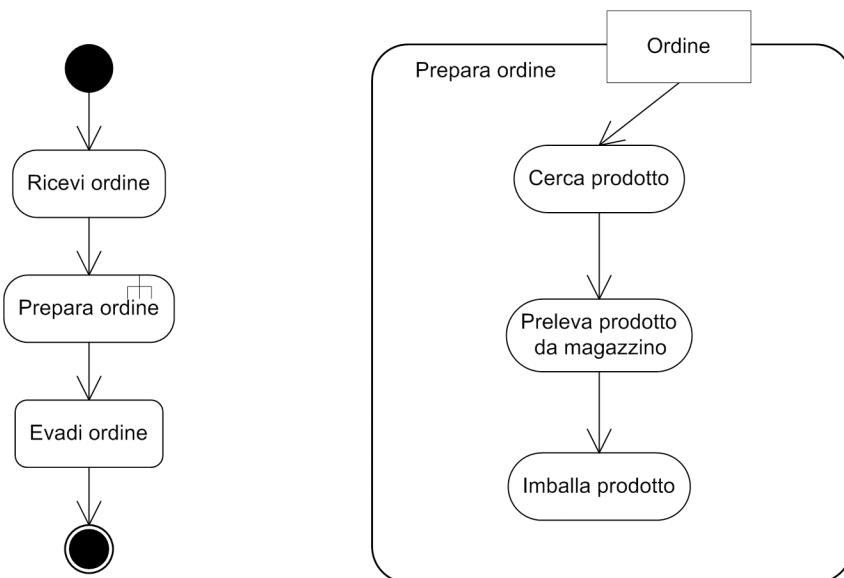


Figura 2.35 Esempio di attività dettagliata in un secondo activity diagram.

2.5 State machine diagram

Gli *state machine diagram* rappresentano le caratteristiche di una qualche entità attraverso la descrizione degli stati in cui può venirsi a trovare e dei relativi passaggi di stato.

L'elemento principale del diagramma è lo stato. Uno *stato* descrive una possibile situazione/configurazione in cui si trova l'ambito della descrizione o un suo elemento. È rappresentato con un rettangolo dagli spigoli arrotondati come mostrato nella Figura 2.36.

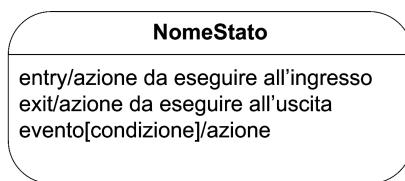


Figura 2.36 Rappresentazione di uno stato.

Uno stato è descritto dalle seguenti informazioni.

- *Nome*: identifica lo stato e lo descrive, indicando la condizione in cui si trova l'oggetto considerato.
- *Attività interne*: sono attività o azioni eseguite quando l'oggetto si trova nello stato considerato. Tali azioni non comportano il cambiamento di stato e possono essere eseguite al verificarsi di particolari eventi e secondo determinate condizioni.

Le attività interne sono rappresentate con la stringa: *evento[condizione]/azione*. In particolare, un *evento* indica una data occorrenza che può verificarsi in un determinato istante. Un'*azione* è eseguita solo quando l'evento associato si verifica. Accanto all'evento può essere espressa una *condizione*, vale a dire un'espressione booleana che vincola l'esecuzione dell'azione. L'azione associata, quindi, è eseguita solo nel caso in cui accade l'evento specificato e si verifica la condizione espressa.

L'esempio della Figura 2.36 mostra alcuni tipi di attività interne e, nello specifico, due attività interne particolari.

- Attività di ingresso: è un'attività eseguita quando l'oggetto entra nello stato. È rappresentata dalla stringa *entry/nome attività da eseguire*.
- Attività di uscita: è un'attività eseguita quando l'oggetto abbandona lo stato considerato. È rappresentata, all'interno dello stato con la stringa *exit/nome attività da eseguire*.

Gli stati sono collegati tra loro tramite transizioni che cambiano lo stato corrente. La *transizione* tra due stati è caratterizzata da un'attività eseguita al verificarsi di un particolare evento e sotto determinate condizioni. In particolare, anche le transizioni sono rappresentate nella forma *evento[condizione]/azione*: valgono, pertanto, le considerazioni fatte per le azioni interne.

36 Capitolo 2 UML: Unified Modeling Language

Allo stato semplice si affiancano altri tipi di stato:

- superstato
- stato concorrente
- pseudostato.

Un *superstato* è uno stato composito che contiene altri stati. Si possono raggruppare alcuni stati in un superstato quando questi ultimi hanno delle transizioni e dei comportamenti simili: così facendo questi elementi in comune sono riferiti al superstato. Ad esempio, il diagramma degli stati della Figura 2.37 descrive l'evoluzione dei processi di un sistema operativo e illustra il concetto di superstato. Una volta creato, un processo può trovarsi in uno tra tre principali superstati: in esecuzione (*Running*), sospeso in memoria (*Waiting*) e sospeso su memoria di massa (*Swapped*). In realtà questi superstati sono poi descritti nel dettaglio da altri stati. Un processo in esecuzione può operare in modalità utente o, a fronte di una chiamata al sistema operativo o di un interrupt, in modalità kernel. Un processo sospeso in memoria può essere nello stato *Asleep* (cioè in attesa di un qualche evento come il completamento di un'operazione di input/output) oppure *Ready* (cioè pronto per andare in esecuzione) oppure *Preempted* (cioè interrotto per assegnare tempo di calcolo ad altri processi). Quando il sistema operativo ha bisogno di rilasciare memoria centrale per nuovi processi, un processo può venire "swappato" cioè copiato su disco (*Swap out*). Prima di poter tornare in esecuzione deve essere ricaricato in memoria (*Swap in*). Un processo termina normalmente eseguendo il comando *Exit*. Qualunque sia lo stato, può venire "ucciso" attraverso il comando *Kill*. Quest'ultima situazione è descritta con la transizione di stato che coinvolge il superstato, intendendo con questo il fatto che tale transizione si applica qualunque sia il reale sottostato del superstato in cui si trova il processo.

Uno *stato concorrente* è uno stato che contiene altri stati, validi contemporaneamente in modo concorrente. L'esempio della Figura 2.38 illustra l'utilizzo di uno stato concorrente. Si supponga di voler descrivere un server che è controllato da una console di comando. Lo stato complessivo del sistema è l'unione dello stato del server e di quello della console. Al momento dell'attivazione, i due sottosistemi sono posti negli stati indicati dalle *transizioni di default*. Quando la console riceve la richiesta dell'utente di effettuare lo shutdown del sistema, esegue una transizione di stato portandosi nello stato finale e genera un evento che causa la transizione di stato anche del server. La terminazione dei due sottosistemi provoca la transizione finale dello stato concorrente.

Gli *pseudostati* sono stati che non hanno alcuna attività interna associata, ma servono a completare la descrizione del comportamento dell'oggetto descritto.

- *Pseudostato iniziale*: indica lo stato in cui l'ambito della descrizione si trova inizialmente; è rappresentato da un cerchio pieno nero.
- *Pseudostato finale*: indica lo stato finale dell'entità descritta dal diagramma; è rappresentato da un cerchio vuoto che ne racchiude uno nero pieno.

2.6 Sequence diagram 37

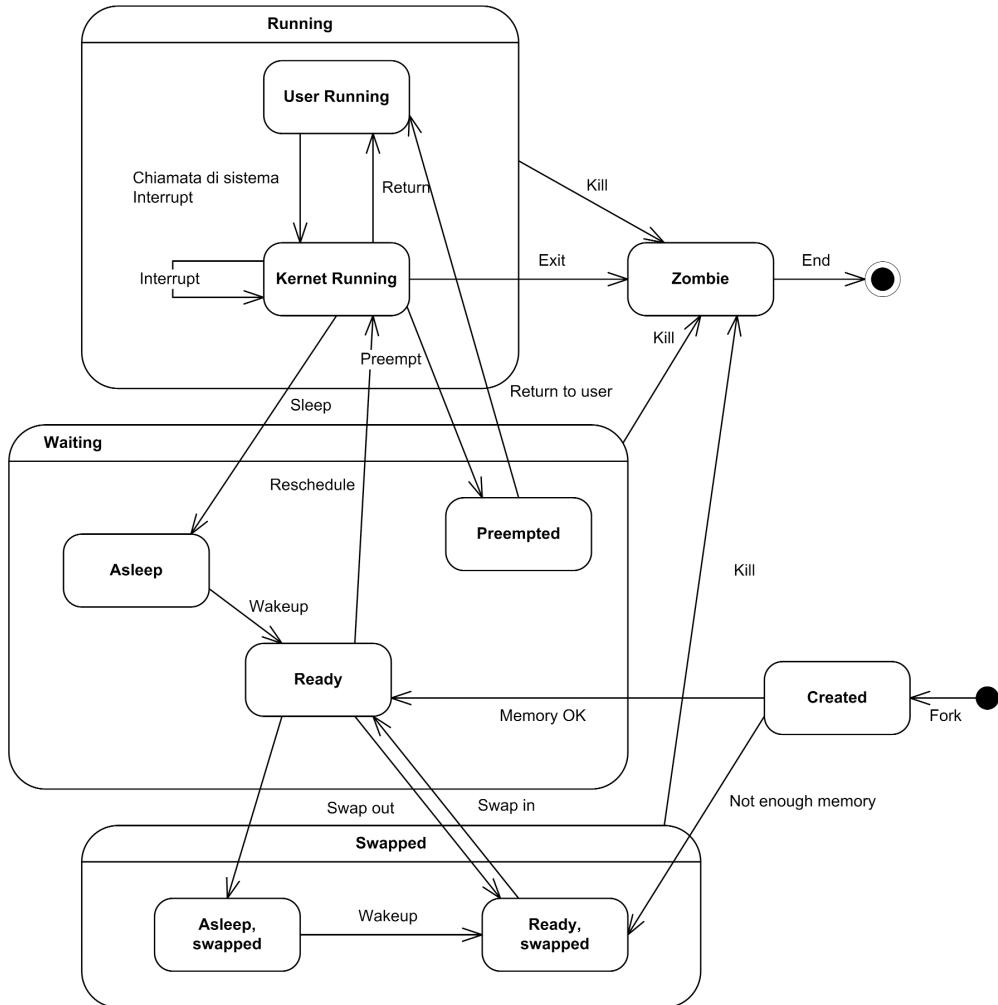


Figura 2.37 Esempio di state diagram con superstati.

2.6 Sequence diagram

Il *sequence diagram* descrive l'interazione tra elementi come una sequenza temporale di azioni ed eventi. Gli elementi principali di un sequence diagram sono gli oggetti. Ogni oggetto ha una propria linea di vita rappresentata da una linea tratteggiata che parte dal rettangolo: essa descrive la sua “vita”. Per indicare che un oggetto è attivo, si usa la barra di attivazione, rappresentata con una barra posizionata lungo la linea di vita. Un oggetto

38 Capitolo 2 UML: Unified Modeling Language

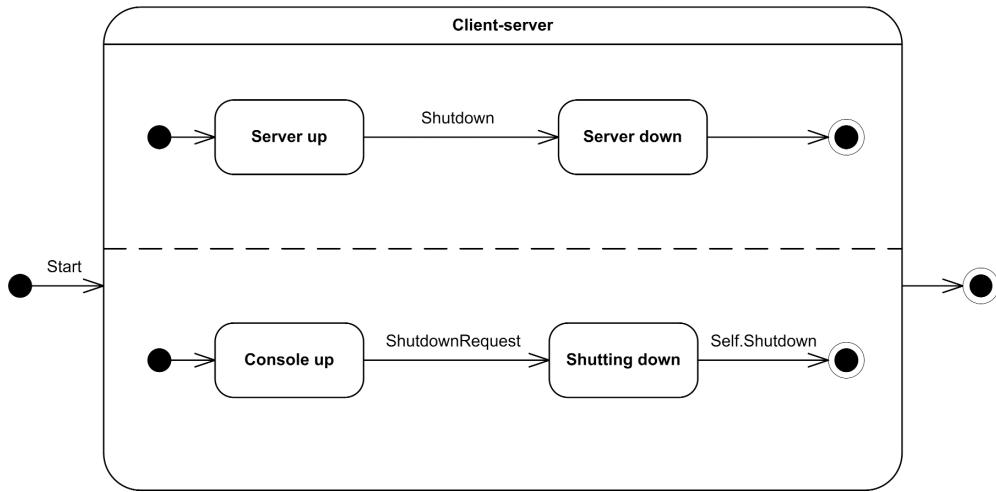


Figura 2.38 Esempio di stato concorrente.

può essere creato o distrutto da un altro oggetto che chiama i metodi opportuni. La Figura 2.39 mostra la creazione e la cancellazione di un oggetto.

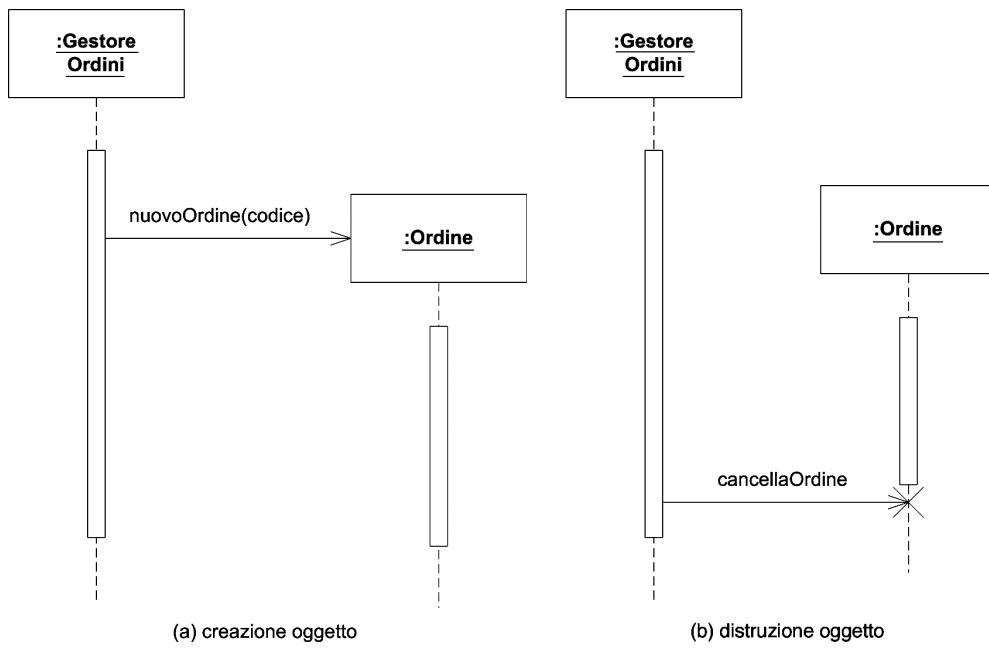
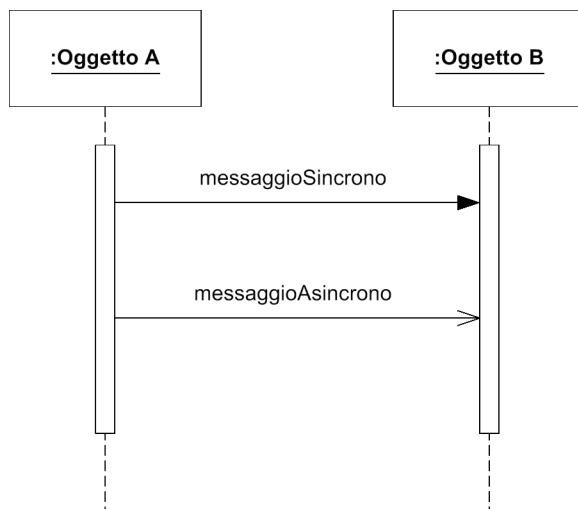
La sequenza di interazioni tra oggetti è realizzata tramite messaggi. Un *messaggio* è rappresentato da una freccia che parte dall'oggetto che invia il messaggio e punta all'oggetto destinatario. I messaggi di richiesta sono identificati da un nome che indica la richiesta effettuata. I messaggi inviati sono di due tipi (si veda la Figura 2.40).

- *Sincroni*: l'oggetto che invia il messaggio attende la risposta dall'oggetto interpellato, interrompendo la propria esecuzione. Un messaggio sincrono è rappresentato con una freccia dal tratto continuo e con la punta piena.
- *Asincroni*: l'oggetto che invia il messaggio non attende la risposta prima di continuare le interazioni dello scenario. Un messaggio asincrono è rappresentato con una freccia.

Un messaggio può rappresentare l'invio di un segnale, l'invocazione di un'operazione, la creazione o distruzione di un'istanza. Nel caso in cui il messaggio descrive una chiamata di metodo il nome del messaggio coincide con il nome del metodo chiamato. È possibile, quindi, introdurre nel messaggio inviato i parametri relativi. Nell'esempio della Figura 2.39, il parametro inviato è il codice dell'ordine.

Oltre ai messaggi di richiesta tra due oggetti diversi, il sequence diagram permette di descrivere chiamate interne. Una chiamata interna è un messaggio che un oggetto invia a se stesso: per esempio, è un metodo eseguito dall'oggetto stesso. Nell'esempio della Figura 2.41, il metodo `calcolaPrezzo` è chiamato ed eseguito dall'oggetto `:Ordine`.

2.6 Sequence diagram 39

**Figura 2.39** Esempi di creazione e distruzione di un oggetto.**Figura 2.40** Messaggio sincrono e asincrono.

40 Capitolo 2 UML: Unified Modeling Language

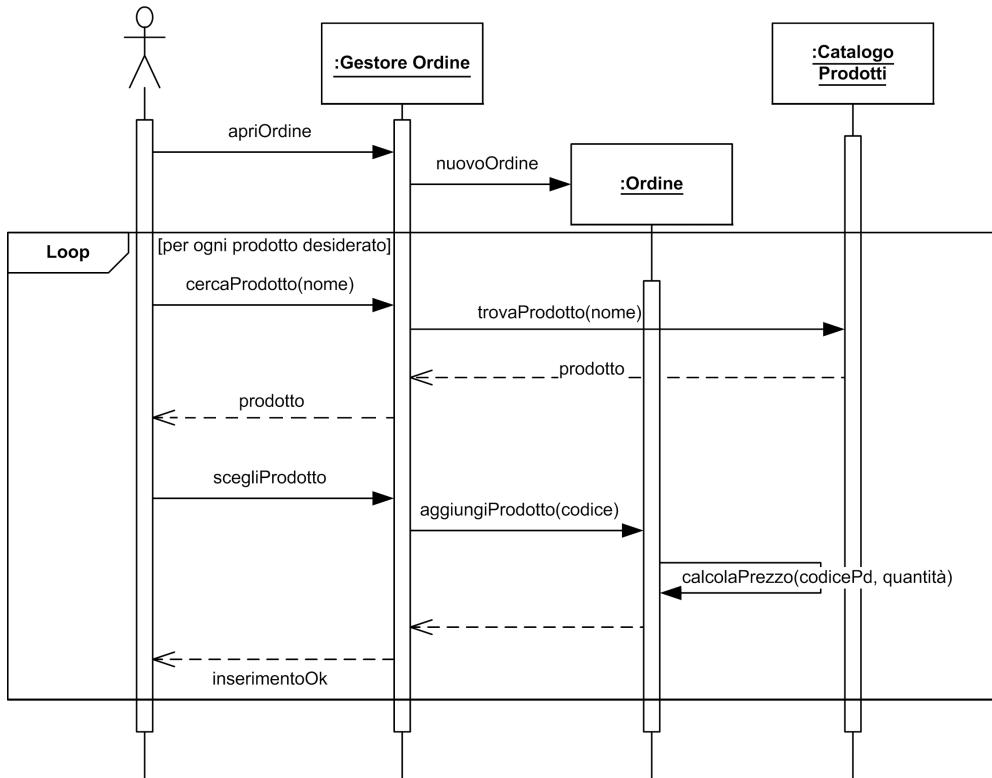


Figura 2.41 Esempio di selezione di prodotti per un ordine.

Un messaggio sincrono fa sì che chi lo ha emesso resti in attesa della risposta, detta messaggio di ritorno. Un *messaggio di ritorno* è rappresentato con una freccia tratteggiata che va dall'oggetto chiamato al chiamante ed è descritto dal contenuto della risposta. In particolare, il messaggio di risposta è identificato dal tipo di oggetto ritornato, nel caso in cui il messaggio di richiesta sia la chiamata di un metodo. È utile inserire i messaggi di ritorno nel diagramma solo nel caso in cui apportino nuove informazioni utili.

I sequence diagram della nuova notazione UML 2.0 introducono un ulteriore elemento: il frame. Un *frame* raggruppa un insieme di messaggi. È descritto da un rettangolo che racchiude i messaggi considerati ed è caratterizzato da un operatore che definisce il tipo di interazione da applicare sui messaggi inclusi. Inoltre, viene specificata una *guard* (guardia) che descrive una particolare condizione che deve essere verificata perché i messaggi possano essere eseguiti.

2.6 Sequence diagram 41

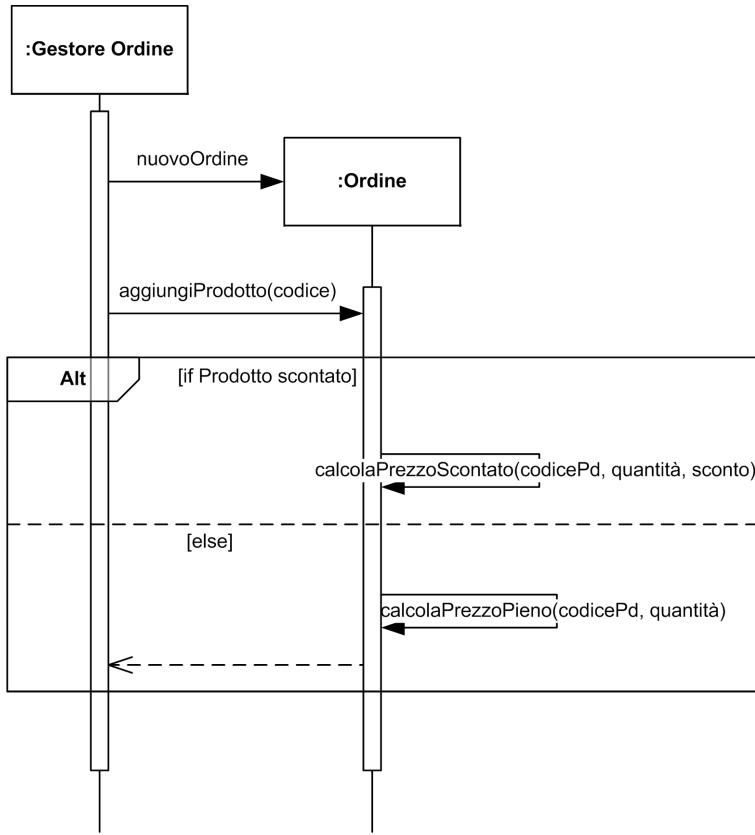


Figura 2.42 Esempio di frame.

Gli operatori più interessanti sono i seguenti.

- *Loop*: indica che i messaggi racchiusi dal frame sono eseguiti più volte all'interno di un ciclo. In questo caso la guardia descrive la condizione di ciclo (Figura 2.41).
- *Alt*: indica che i messaggi evidenziati dal frame sono eseguiti in alternativa tra loro. Le due alternative sono divise nel frame da una linea orizzontale tratteggiata (Figura 2.42). La guardia descrive la condizione che deve essere verificata per eseguire i primi messaggi, altrimenti si passa all'esecuzione delle interazioni evidenziate nella seconda alternativa.
- *Opt*: indica che i messaggi racchiusi nel frame sono eseguiti optionalmente. La guardia è la condizione che deve essere verificata per eseguire i messaggi inclusi nel frame; se la guardia non è verificata, nessun messaggio del frame viene inviato né ricevuto.

2.7 Component diagram

Il *component diagram*, o diagramma dei componenti, descrive un'entità complessa scomponendola in parti distinte in modo da evidenziarne struttura e legami. L'elemento principale del component diagram è il componente. Un *componente* è definito come una parte modulare di un sistema che incapsula il suo contenuto, la cui manifestazione è sostituibile all'interno del suo ambiente. In realtà esistono diverse interpretazioni del termine *componente*, come discusso estensivamente nel Capitolo 1: in questa sede, il termine “*componente*” è utilizzato in modo volutamente ambiguo e generico.

Un componente è identificato da un nome ed è rappresentato come un rettangolo che può presentare alternativamente il simbolo del componente sull'angolo in alto a destra del rettangolo o lo stereotipo <<component>> sopra il nome. La Figura 2.43 mostra le due diverse rappresentazioni di un componente.

Un componente può essere utilizzato in una molteplicità di situazioni: ad esempio, un componente può coincidere con un sottosistema, vale a dire una parte ben definita e relativamente autonoma di un'applicazione informatica. In questo caso esso è descritto con lo stereotipo <<subsystem>>. Spesso si definiscono stereotipi di componenti particolari, come appunto <<subsystem>>, per indicare la natura di quel particolare elemento. I più comuni sono <<database>>, <<infrastructure>>, <<process>> e <<thread>>.

Il comportamento di un componente è completamente definito dalle interfacce che offre o richiede. Le interfacce tra componenti sono realizzate tramite le funzionalità interne dei singoli componenti e sono descritte tramite i port.

Il concetto di *port* costituisce una delle principali innovazioni di UML 2.0. Esso viene utilizzato per rappresentare le modalità secondo le quali un elemento complesso come una classe o un componente interagisce con altri elementi. La Figura 2.44 mostra un componente la cui struttura interna è formata da due classi che forniscono o richiedono delle interfacce. La relazione <<delegate>> descrive il legame tra i port, cioè le interfacce esterne del componente (i due port *Interfaccia richiesta* e *Interfaccia offerta*), e quelle interne delle singole parti da cui il componente è composto.

Da un punto di vista formale, UML prevede che i componenti siano un sottotipo di classe. Come le classi, i componenti possono essere quindi istanziati. Ciò ha senso quando in un diagramma si vogliono evidenziare diverse istanze di uno stesso componente che giocano ruoli diversi oppure per descrivere i componenti presenti in un altro componente. Per identificare un'istanza di componente, in analogia a quanto previsto per gli oggetti, si utilizza la notazione

nomeIstanza:nomeComponente

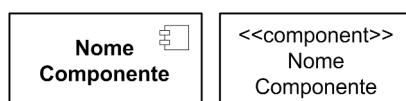


Figura 2.43 Rappresentazioni alternative di un componente.

2.7 Component diagram 43

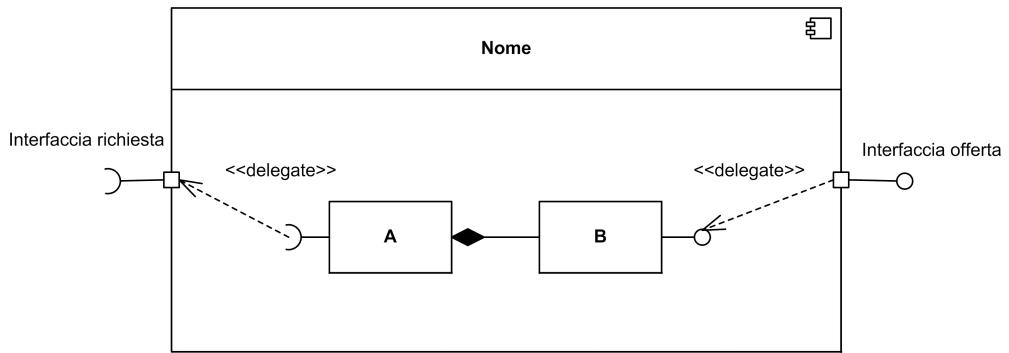


Figura 2.44 Classi utilizzate per realizzare un componente.

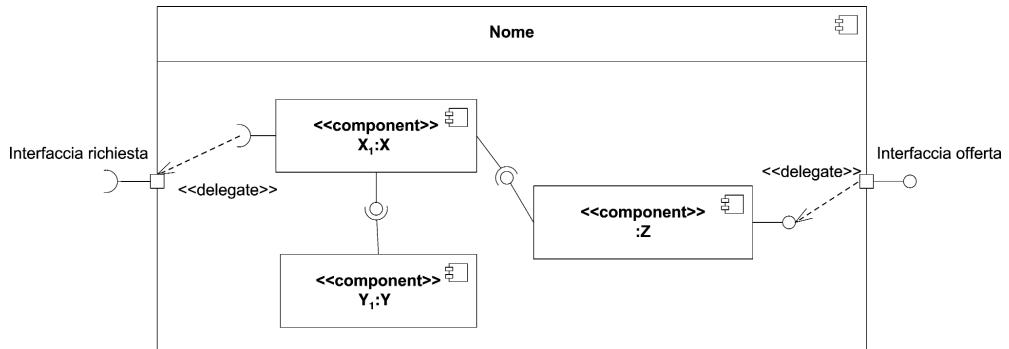


Figura 2.45 Componente che contiene istanze di altri componenti.

(dove *nomeIstanza* è opzionale), come illustrato nella Figura 2.45.

Il secondo elemento presente in un component diagram è l'artifact. Un *artifact* rappresenta la realizzazione concreta di un qualche elemento; esempi tipici sono i file sorgenti, gli eseguibili, le librerie. Un artifact è rappresentato come un rettangolo con un'icona sull'angolo in alto a destra, come mostrato nella Figura 2.46.

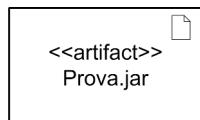


Figura 2.46 Rappresentazione di un artifact.

44 Capitolo 2 UML: Unified Modeling Language

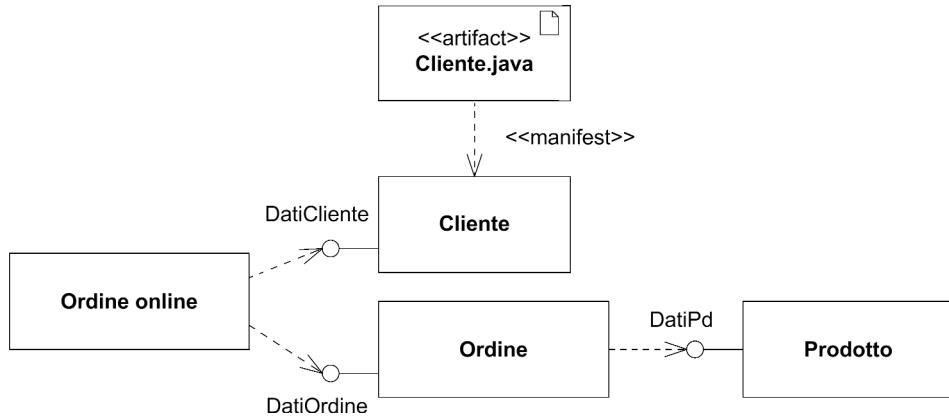


Figura 2.47 Esempio di relazioni tra componenti.

Un component diagram descrive, oltre ai componenti, le relazioni di dipendenza che sussistono tra questi e gli artifact. In particolare, la relazione <<manifest>> mette in relazione i componenti con gli artifact. In particolare, indica che gli artifact sono le rappresentazioni fisiche dei componenti considerati. Questa relazione può legare più componenti a un unico artifact. La Figura 2.47 mostra un esempio della dipendenza <<manifest>>: il file Cliente.java è l'implementazione del componente Cliente. Si noti la notazione alternativa relativa alle interfacce, dove l'uso di un'interfaccia è rappresentato da una relazione di dipendenza. Tale notazione è equivalente a quella introdotta nella Figura 2.12.

Un esempio più complesso di uso della relazione <<manifest>> è illustrato nella Figura 2.48. Le classi sulla sinistra sono tutte implementate in un file .java separato, al quale corrisponde il relativo file .class. Insiemi di file .class costituiscono i file .jar che verranno effettivamente installati sui computer.

Il diagramma della Figura 2.48 può essere anche integrato da quello della Figura 2.49, dove è evidenziata la relazione tra il componente e l'artifact che lo rende “manifesto” (per semplicità non sono state evidenziate le relazioni interne al componente).

2.8 Composite structure diagram

Il *composite structure diagram*, o diagramma di struttura composita, permette di descrivere la struttura interna di elementi complessi, evidenziandone gli elementi costitutivi. Si consideri come esempio il diagramma delle classi della Figura 2.50.

Il diagramma descrive i seguenti concetti.

- Un'auto ha un motore.
- Un'auto ha due ruote posteriori (Posteriore è il ruolo che giocano le due ruote previste dalla relazione).

2.8 Composite structure diagram 45

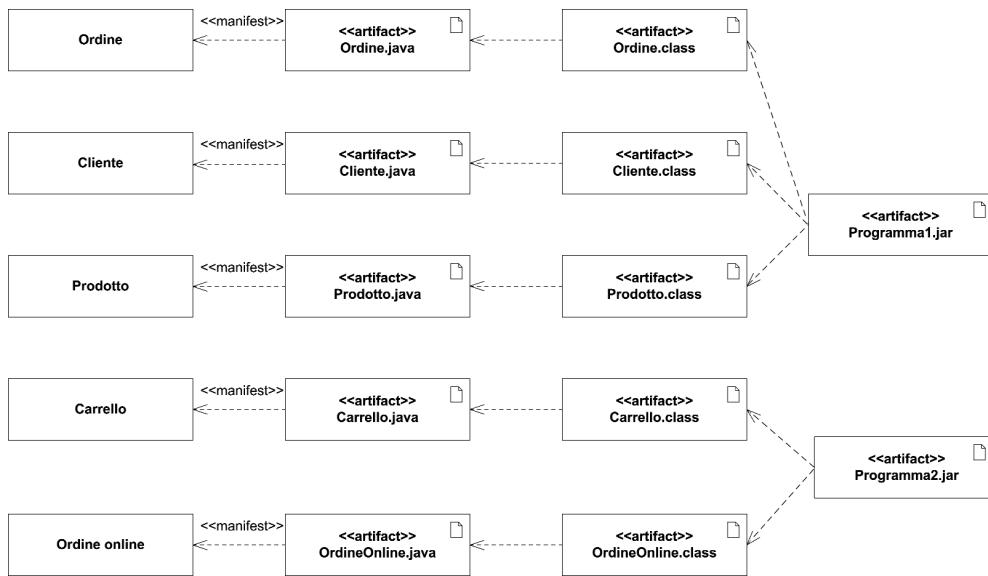


Figura 2.48 Dalle classi ai file .jar.

- Un motore può essere connesso con più ruote e viceversa. Ovviamente, questo non è normalmente vero nel caso dell'auto. Ma, supponendo che il diagramma sia parte di una schema più ampio, potrebbe essere vero per altri tipi di mezzo di trasporto (si pensi a una motrice o a un trattore).

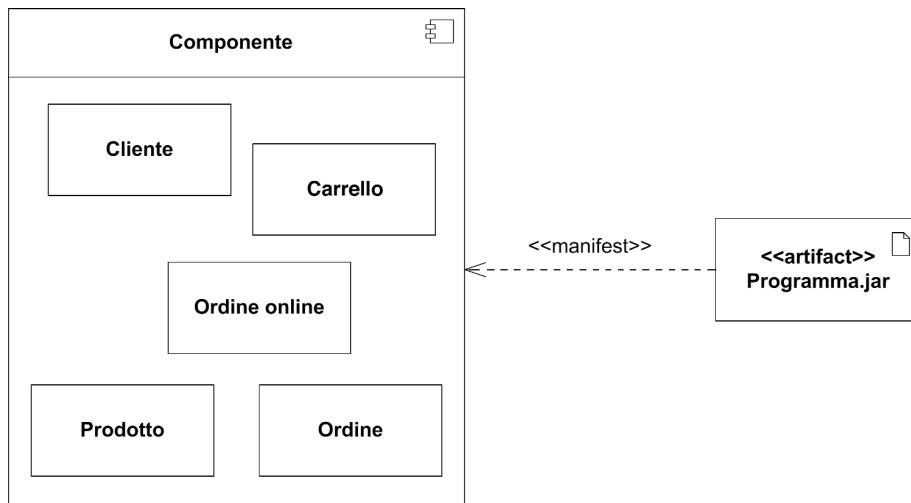
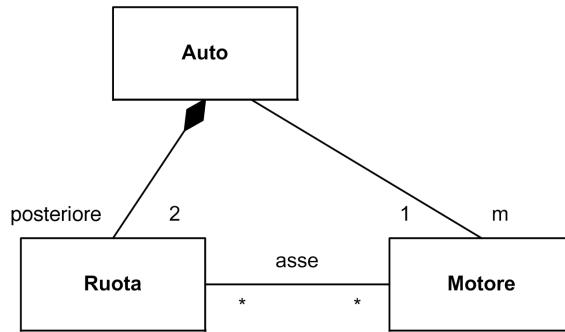


Figura 2.49 Relazione tra artifact e componente.

46 Capitolo 2 UML: Unified Modeling Language

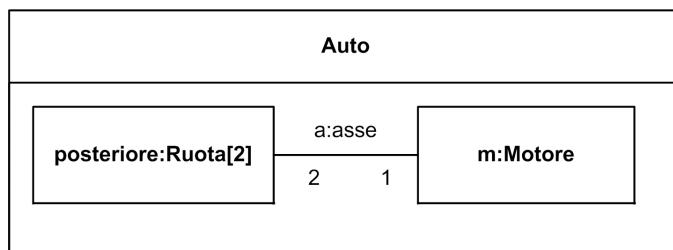
**Figura 2.50** Diagramma di esempio.

Il diagramma della Figura 2.50 esprime la relazione tra le classi, ma non dice nulla sulla struttura di un'auto: cosa c'è dentro ogni istanza della classe `Auto`? Quali sono i vincoli che devono essere considerati tenendo conto che si tratta di un'auto e non di un mezzo generico di trasporto? A questa domanda si può rispondere utilizzando il diagramma di struttura composita. Esso ha diverse possibili forme: per questo motivo, nel seguito verranno illustrati diversi diagrammi, ciascuno dei quali illustra alcune delle funzionalità disponibili. Il lettore potrà poi comporre o riusare le diverse funzionalità secondo le proprie necessità e preferenze.

Il diagramma della Figura 2.51 mostra la struttura interna di una classe: in particolare, il fatto che, nel caso di un'auto, il motore è connesso con le sole due ruote posteriori. I simboli presenti all'interno di `Auto` sono denominati parti e connettori.

Una *parte* è un sottoinsieme di un'entità composita ed è identificata da una stringa che ha uno dei seguenti formati.

- *nomeRuolo*. La parte è identificata solo dal *ruolo* che gioca nella struttura che la contiene. Non viene specificato il tipo degli elementi che compongono la parte.

**Figura 2.51** Le parti di un'auto.

2.8 Composite structure diagram 47

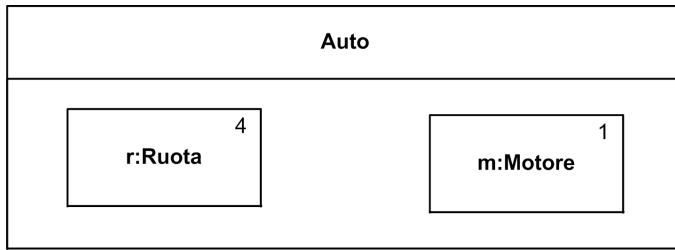


Figura 2.52 Notazione alternativa per indicare la numerosità delle parti.

- `nomeRuolo:nomeTipo`. La parte è identificata dal ruolo che gioca nella struttura e dal tipo degli elementi che la compongono.
- `:nomeTipo`. Il ruolo della parte non è specificato, ma viene indicato il tipo degli elementi che la compongono.

Una parte è caratterizzata anche dalla *numerosità* degli elementi che la costituiscono. Per esempio, nel caso della Figura 2.51 la parte sulla sinistra ha come ruolo `posteriore` ed è composta da due oggetti di tipo `Ruota`. Al contrario, la parte con ruolo `m` è una singola istanza della classe `Motore`. Una parte non è quindi direttamente assimilabile a un singolo oggetto. La Figura 2.52 illustra una notazione alternativa per indicare la numerosità delle parti.

Un *connettore* esprime un legame tra parti. Potrebbe essere un'associazione oppure un altro tipo di informazione che in un qualche modo collega le parti (per esempio, gli oggetti che costituiscono una parte sono passati come parametri di un qualche metodo che li rende visibili all'altra parte).

L'identificativo utilizzato per descrivere un connettore ha una delle seguenti forme.

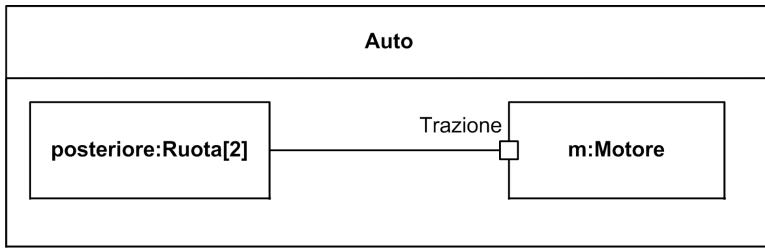
- `nomeConnettore:nomeAssociazione`. Il connettore `nomeConnettore` è di tipo `nomeAssociazione`.
- `nomeConnettore`. Il connettore `nomeConnettore` è di tipo anonimo, non esiste cioè una specifica associazione in base alla quale esso viene definito.
- `:nomeAssociazione`. Il connettore non ha nome ma è di tipo `nomeAssociazione`.

Nel caso della Figura 2.51, le due parti caratterizzate dai ruoli `posteriore` e `m` sono connesse da un connettore che è di tipo `asse`, ma il cui vincolo di molteplicità è 2-1 e non molti-a-molti (si veda la Figura 2.50).

Le informazioni contenute nei diagrammi delle Figure 2.51 e 2.52 rappresentano la reale struttura della classe `Auto`, cioè di tutte le sue istanze. Tali informazioni non sono ricavabili dal diagramma delle classi della Figura 2.50.

Il diagramma della Figura 2.53 illustra un diagramma di struttura composita con le due parti già viste in precedenza. Al posto dell'associazione, tuttavia, in questo caso vie-

48 Capitolo 2 UML: Unified Modeling Language

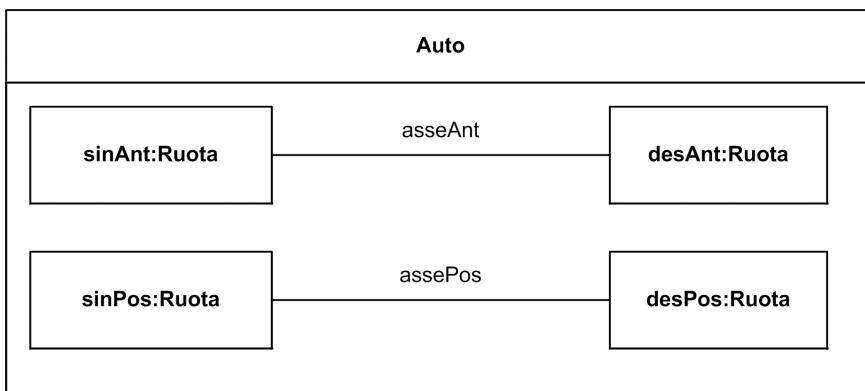
**Figura 2.53** Parti e interfacce.

ne specificato che la parte con ruolo posteriore utilizza l'interfaccia Trazione della parte con ruolo *m*. In questo caso, quindi, è descritto come le parti interagiscono e non il loro legame logico.

Il diagramma della Figura 2.54 mostra una descrizione più precisa della classe Auto. Essa contiene quattro parti, ciascuna delle quali è composta da una sola ruota e gioca un ruolo differente. I due connettori (non tipizzati) illustrano il collegamento tra le ruote anteriori e tra quelle posteriori.

I diagrammi esaminati fino questo punto illustrano sempre la struttura interna di una classe. È possibile utilizzare i diagrammi di struttura composita anche per descrivere istanze di classi, cioè particolari oggetti. In questo caso, la notazione si arricchisce e modifica in modo da poter descrivere le istanze delle diverse informazioni: il diagramma della Figura 2.55 illustra un'istanza della classe Auto, cioè una particolare auto.

Ogni elemento che costituisce una parte è descritto singolarmente ed è identificato da un nome (per esempio *s1*) e dal nome della parte a cui appartiene (in questo caso *sinAnt : Ruota*). Inoltre, ogni elemento è descritto tramite le informazioni che lo caratterizzano. I connettori diventano link, cioè istanze di associazioni.

**Figura 2.54** Il tipo auto.

2.9 Deployment diagram 49

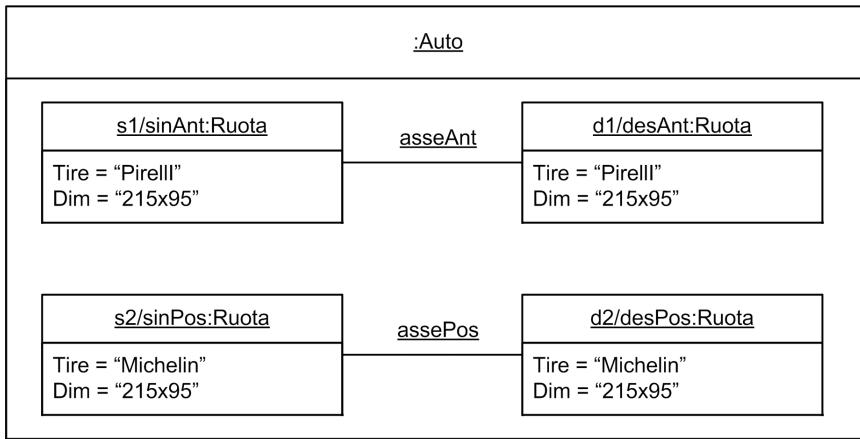


Figura 2.55 Un'istanza di auto.

2.9 Deployment diagram

Il *deployment diagram* illustra l'organizzazione e strutturazione degli elementi di un sistema informatico dal punto di vista fisico. In particolare, un deployment diagram è molto efficace nel descrivere come i diversi elementi di un sistema informatico sono distribuiti sui diversi nodi elaborativi utilizzati. Questo tipo di diagramma, quindi, è particolarmente indicato per rappresentare sistemi informatici distribuiti: in questo caso, infatti, la distribuzione in diversi elementi fisici è una caratteristica fondamentale del sistema e, quindi, la sua descrizione acquista un'importanza rilevante.

L'elemento principale di questo diagramma è il nodo (si veda la Figura 2.56). Un *nodo* rappresenta un'entità che ha la capacità di eseguire software, e può essere:

- un *dispositivo*: rappresenta l'hardware necessario per mandare in esecuzione il software; si descrive con questo tipo di nodo, ad esempio, un PC o una stampante;
- un *ambiente di esecuzione*: rappresenta un software che lancia altre parti di software; un esempio di ambiente di esecuzione è l'ambiente Java 2 Enterprise Edition.

Un nodo è rappresentato con un cubo identificato dal suo nome. Come nel caso degli oggetti, i nodi possono essere istanziati nel caso si voglia indicare la presenza di nodi dello stesso tipo che giocano ruoli diversi. Conseguentemente, il nome di un'istanza di nodo ha la struttura *nomeIstanza:nomeNodo*, dove *nomeIstanza* è opzionale. Per distinguere i due tipi di nodi presentati si usano due stereotipi: <<device>> se si tratta di un dispositivo; <<execution environment>> se si tratta di un ambiente di esecuzione.

Un nodo può presentare anche delle etichette. Un'*etichetta* è una stringa racchiusa da parentesi graffe e fornisce informazioni aggiuntive sui nodi. Ad esempio, nella Figura 2.56 è indicato il tipo di sistema operativo utilizzato.

50 Capitolo 2 UML: Unified Modeling Language

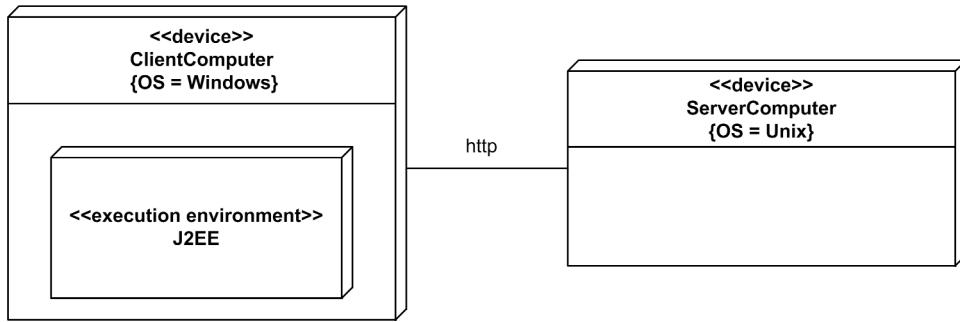


Figura 2.56 Esempi di rappresentazione dei nodi.

I nodi sono collegati tra loro da path di comunicazione. I *path di comunicazione* indicano che i dispositivi sono collegati tra loro così come gli ambienti di esecuzione; sono rappresentati come dei semplici segmenti che congiungono due nodi. È possibile specificare il tipo di comunicazione adottata dai nodi considerati indicando il protocollo di comunicazione con un'etichetta posta accanto al segmento: ad esempio, si può specificare il protocollo di comunicazione usato tra due nodi. La Figura 2.56 descrive la comunicazione client-server via protocollo http.

Sui nodi sono collocati gli artifact che costituiscono il sistema informatico.

In particolare, la Figura 2.57 illustra le diverse modalità grafiche secondo le quali è possibile operare. Nel caso (a) la collocazione dei due artifact *Applicazione.jar* e *Console.jar* viene illustrata attraverso la relazione di dipendenza <<deploy>>; nel caso (b) i due artifact vengono direttamente collocati all'interno del nodo e, infine, nel caso (c) si segue lo stesso approccio, utilizzando, tuttavia, una indicazione puramente testuale. Le tre notazioni sono equivalenti: il progettista sceglierà di volta in volta quella che risulta più comoda in funzione delle informazioni da descrivere.

Considerazione di carattere generale

Come già accennato nell'introduzione, questo capitolo non vuole essere una guida completa a UML, ma solo uno strumento per comprendere al meglio i diagrammi descritti nei capitoli seguenti. In particolare, proprio perché non usati in questo testo, non sono stati descritti alcuni diagrammi presenti in UML 2.0: il timing diagram, che permette di descrivere informazioni e vincoli temporali relativi all'ambito della descrizione considerata, e i communication e interaction diagram che ne rappresentano il comportamento dinamico. Questi ultimi non sono stati inseriti perché si è scelto di descrivere tale comportamento con altri diagrammi, come ad esempio i sequence e activity diagram. Il lettore che desideri studiare anche questi diagrammi è invitato a consultare la bibliografia consigliata.

2.10 Riferimenti bibliografici 51

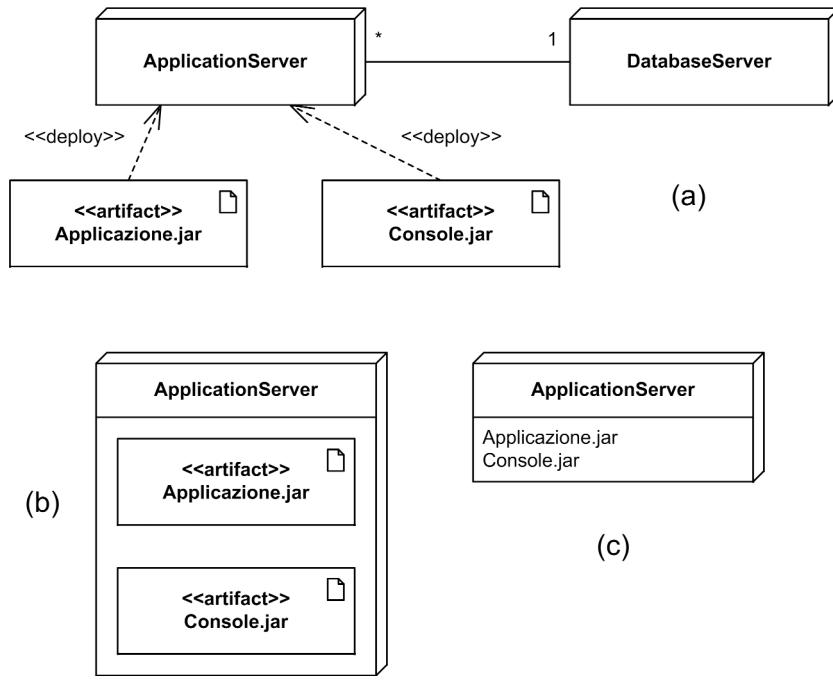


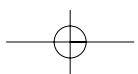
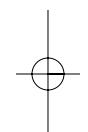
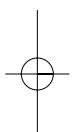
Figura 2.57 Collocazione degli artifact sui nodi.

La Figura 2.57 illustra anche l'utilizzo di un'associazione tra tipi di nodi (*Application Server* e *DatabaseServer*). Tale associazione specifica che nel sistema informatico descritto vi sono più nodi *ApplicationServer* connessi a un unico *DatabaseServer*.

2.10 Riferimenti bibliografici

UML è stato originariamente descritto in una serie di libri scritti dai tre principali autori: Booch, Rumbaugh e Jacobson. UML è poi divenuto oggetto di standardizzazione da parte dello Object Management Group (OMG). Sul sito di questa organizzazione sono disponibili sia la definizione standard del linguaggio, sia informazioni, tutorial e collegamenti ad altre informazioni utili per studiare e applicare nella pratica UML.

Vi sono anche numerose pubblicazioni e articoli che discutono diversi aspetti di UML. Due testi tra i più recenti sono Arlow e Neudstadt [2005] e Eriksson et al. [2004]. Il primo contiene anche una descrizione dello Unified Process (discusso nel Capitolo 8) e della relazione tra questo ciclo di vita e UML. Un testo molto sintetico di introduzione a UML è Fowler [2004], giunto alla terza edizione. Tutti questi testi fanno riferimento alla versione 2.0 di UML.



Capitolo 3

Qualità del software

Un prodotto software è costituito generalmente da una *soluzione software* e dall'insieme di tutta la *documentazione di prodotto*. La soluzione software consiste nel codice (in particolare, il programma sorgente). La documentazione di prodotto, invece, comprende tutte le descrizioni realizzate durante le varie fasi di cui si compone il processo software: descrizione del problema, specifica dei requisiti, manuale utente, manuale tecnico, manuale d'installazione ecc. Inoltre va considerata tutta la *documentazione di supporto*, per esempio i documenti di pianificazione del progetto e la documentazione dei casi di test.

La qualità della soluzione software può essere intesa sia in termini di proprietà intrinseche del software, sia in termini di ciò che è percepito dal committente. Per chiarire meglio questa distinzione, si può pensare che la qualità intesa in termini di proprietà intrinseche del software riguarda tutti gli aspetti che rispondono alla domanda “il programma è giusto?”, mentre la qualità percepita dal committente è relativa a tutti gli aspetti che rispondono alla domanda “è il giusto programma?”.

Tipicamente le domande considerate per valutare se il programma “è giusto” sono le seguenti.

- Il software è affidabile?
- Quanto è efficiente?
- Quanto è semplice modificarlo?
- Quanto costa trasferire il software da un ambiente a un altro?

Per valutare invece se è “il giusto programma”, ci si pone le seguenti domande.

- Le funzionalità offerte dal software soddisfano i fabbisogni del committente?
- Il committente riconosce e comprende le funzionalità del software?

In generale, la qualità di un prodotto software dipende non solo dalla qualità della soluzione software – come erroneamente si potrebbe essere portati a credere – ma anche dalla qualità di tutti gli altri elementi di cui si compone il prodotto. Per esempio, a molti è sicuramente capitato di utilizzare applicazioni software ottime dal punto di vista funzionale, per le prestazioni e la ricchezza delle funzionalità offerte, ma prive di un esaustivo

54 Capitolo 3 Qualità del software

manuale utente o help online. In questo caso l'assenza di qualità della manualistica disponibile ha un effetto negativo sulla qualità totale del software: l'utente come può infatti percepire e quindi apprezzare appieno le funzionalità di un'applicazione software se non viene messo in condizione di apprendere come possono essere utilizzate? Si ottiene in questo caso una percezione di qualità complessiva piuttosto bassa: quanto maggiore è la ricchezza delle funzionalità e scadente la documentazione di corredo, tanto maggiore sarà la percezione dell'utente di essere di fronte a un'applicazione estremamente complessa, poco usabile e, in generale, non soddisfacente.

La qualità del prodotto software è anche intrinsecamente legata alla qualità del processo di sviluppo e delle fasi di cui si compone. Poiché un prodotto si ottiene seguendo un determinato processo, la modalità con cui quest'ultimo viene eseguito influenza la qualità del risultato che si ottiene. La qualità del processo software è strettamente correlata anche alla sua capacità di essere *predicibile* e *controllabile*. Un processo predicibile permette di effettuare, in fase di pianificazione del progetto, stime realistiche; un processo controllabile consente di svolgere verifiche sullo stato di avanzamento del processo e di completamento del prodotto rispetto alla relativa pianificazione. Pertanto se il processo non viene eseguito e controllato con un adeguato livello di qualità, può capitare ad esempio che vengano introdotti errori o ritardi che si ripercuotono in "non qualità" del prodotto finale.

Si può parlare di qualità anche per le fasi di cui si compone il processo software: facendo riferimento alla fase di descrizione del problema, sicuramente assume un ruolo importante la qualità dell'analisi dei requisiti. Tale qualità dipende strettamente dal livello di comprensione del dominio applicativo, dalla chiarezza con cui viene descritto il problema, dalla completezza e correttezza della specifica dei requisiti, nonché dall'esperienza delle persone deputate a svolgere tale fase. Con riferimento invece alla fase di progettazione della soluzione, la qualità dipende dalla modalità con cui vengono svolte la progettazione architettonica e di dettaglio.

Il seguito di questo capitolo si occuperà esclusivamente della qualità di un prodotto software. Come è possibile valutare la qualità del prodotto software? Esistono definizioni univoche di qualità? Quali caratteristiche devono essere misurate per valutare la qualità?

Esiste uno standard, denominato *ISO/IEC 9126*, che delinea un modello di qualità del software, definendo alcune specifiche metriche che si possono utilizzare per misurare le caratteristiche di qualità definite all'interno dello standard stesso.

Lo standard ISO/IEC 9126 è stato definito dagli organismi ISO e IEC. ISO (*International Organization for Standardization*) è un'organizzazione non governativa che definisce standard internazionali ed è costituita da una federazione di organismi di standardizzazione nazionali appartenenti a 149 nazioni. Collabora con altri enti internazionali, tra cui IEC, specialmente nel settore delle tecnologie di informazione e comunicazione. IEC (*International Electrotechnical Commission*) è un'organizzazione internazionale che definisce standard relativi a tutte le tecnologie elettriche ed elettroniche. Gli standard definiti da ISO e IEC specificano i requisiti per prodotti, servizi, processi, materiali, sistemi, assessment di conformità, prassi gestionali e manageriali. Tra gli altri, gli standard della famiglia ISO 9000 sono relativi alla qualità in senso lato e ai sistemi di gestione della qualità.

In particolare, ISO/IEC 9126 definisce la qualità del software secondo tre differenti viste: *qualità interna*, *qualità esterna* e *qualità in uso*. La qualità interna si riferisce alle

3.1 Il modello di qualità interna ed esterna 55

caratteristiche interne del prodotto software, cioè a tutte quelle proprietà che sono rilevanti durante le fasi di sviluppo e di manutenzione del software stesso. La qualità esterna descrive invece tutte le proprietà del software che lo caratterizzano durante la sua esecuzione. La qualità in uso, infine, descrive le proprietà del prodotto software in uno specifico ambiente di esecuzione o contesto d'uso (quindi dal punto di vista di chi utilizza il software). Tali viste di qualità sono applicabili a qualsiasi tipologia di software. Per descrivere le tre viste di qualità, lo standard ISO/IEC 9126 fornisce due *modelli*, uno per le qualità interna ed esterna e uno per quella in uso. Entrambi i modelli definiscono la qualità in termini di un insieme di caratteristiche organizzate secondo una specifica struttura. In particolare, il modello per quella interna ed esterna del software fornisce un insieme di 6 *caratteristiche* e di 27 *sottocaratteristiche*, mentre il modello per la qualità in uso definisce altre 4 caratteristiche. La Tabella 3.1 illustra sinteticamente i due modelli e le rispettive caratteristiche/sottocaratteristiche.

3.1 Il modello di qualità interna ed esterna

Questo modello stabilisce che la qualità del software dipende dalla presenza delle seguenti sei caratteristiche di qualità: funzionalità, usabilità, affidabilità, efficienza, manutenibilità e portabilità. Per ciascuna di esse viene definito un insieme di sottocaratteristiche che meglio descrivono la caratteristica di qualità.

3.1.1 Funzionalità (functionality)

Per *funzionalità* di un prodotto software si intende la sua capacità di svolgere compiti che abbiano una qualche utilità in relazione a bisogni espressi o impliciti degli utenti. Per esempio, nel caso di un word processor le funzionalità sono quelle relative alla capacità di formattare il testo, di impaginarlo, di farne il controllo ortografico, di applicarci dei template o di creare l'anteprima di stampa.

Per meglio definire che cosa si intende per funzionalità del software, lo standard indica cinque sottocaratteristiche: adeguatezza, accuratezza, interoperabilità, aderenza agli standard e sicurezza.

■ Adeguatezza (*suitability*)

Per *adeguatezza* del prodotto software si intende la sua capacità di svolgere un insieme di funzioni che risultino appropriate per un compito o un obiettivo specifico. Si consideri per esempio un'applicazione per il calcolo delle tasse e si ipotizzi che essa recepisca le regole fiscali vigenti in Italia. Tale applicazione risulterà adeguata se utilizzata per calcolare le tasse da pagare in Italia, ma del tutto inadeguata se utilizzata per calcolare le tasse in paesi che adottano regole fiscali diverse da quelle italiane.

■ Accuratezza (*accuracy*)

L'*accuratezza* esprime la capacità del prodotto software di fornire i risultati e gli effetti corretti e attesi. Per esempio, il software che calcola le tasse da versare allo stato deve essere ac-

56 Capitolo 3 Qualità del software

Modello	Caratteristica	Sottocaratteristica
Qualità interna ed esterna	Funzionalità	Adeguatezza Accuratezza Interoperabilità Aderenza standard (Conformità) Sicurezza
	Usabilità	Comprensibilità Apprendibilità Operabilità Attrattività Conformità
	Affidabilità	Maturità Tolleranza ai guasti Ripristinabilità Conformità
	Efficienza	Comportamento temporale Consumo di risorse Conformità
	Manutenibilità	Analizzabilità Modificabilità Stabilità Testabilità Conformità
	Portabilità	Adattabilità Installabilità Coesistenza Sostituibilità Conformità
Qualità in uso	Efficacia Produttività Sicurezza Soddisfazione	

Tabella 3.1 Le caratteristiche di qualità del software nello standard ISO/IEC 9126.

curato in quanto deve calcolare con precisione assoluta l'importo dovuto. In altri casi, l'accuratezza può essere espressa con il margine di errore tollerato nel risultato fornito.

- **Interoperabilità (*interoperability*)**

L'*interoperabilità* riguarda la capacità del prodotto software di interagire con altri sistemi. Un esempio è fornito dalla clipboard di Windows, che consente di scambiare dati tra due qualsiasi applicazioni, permettendo anche la scelta del formato. Nel mondo dei programmi UNIX, un esempio di interoperabilità è dato dalla possibilità di interconnettere

3.1 Il modello di qualità interna ed esterna 57

diverse applicazioni attraverso il meccanismo del *pipe&filter*, ridirigendo gli input e gli output. Gli esempi di tecnologie più moderne che hanno lo scopo di garantire e favorire lo sviluppo di software interoperabile sono XML e i web service.

- Aderenza agli standard (*compliance*)

L'aderenza agli standard si riferisce alla capacità del software di rispettare protocolli, leggi, convenzioni, prescrizioni e standard. Per esempio, vi sono applicazioni Java sviluppate secondo il modello architettonico J2EE (*Java 2 Enterprise Edition*): tale modello indica quali siano gli standard che devono essere adottati dalle applicazioni web-based, dalle applicazioni remote, dalle applicazioni che accedono a dati mantenuti in database ecc.

È importante evidenziare il forte nesso tra l'aderenza agli standard e l'interoperabilità: molti degli standard esistenti, in particolare quelli relativi alla comunicazione, sono proprio finalizzati a facilitare l'interoperabilità.

- Sicurezza (*security*)

La sicurezza del prodotto software esprime la sua capacità di prevenire accessi non autorizzati e di resistere ad attacchi intenzionali mirati ad accedere a informazioni confidenziali, a modificare dati o il software stesso, a fornire qualche vantaggio a chi svolge gli attacchi e a impedire che i legittimi utenti possano utilizzare il prodotto software.

È quasi superfluo ricordare l'attenzione riposta da tutti i produttori di software nel garantire che i propri prodotti esibiscano questa caratteristica.

3.1.2 Usabilità (usability)

Un prodotto software si dice *usabile* se viene compreso e appreso dall'utente senza difficoltà e viene quindi utilizzato in modo naturale e immediato. Tale caratteristica risulta particolarmente critica perché spesso il successo di un prodotto software dipende più dalla percezione che ne ha l'utente piuttosto che da altre caratteristiche quali ad esempio la ricchezza delle funzionalità. Accade spesso, infatti, che gli utenti preferiscono un prodotto software che offre poche funzionalità che risultano semplici e intuitive da utilizzare, piuttosto che applicazioni software con tante funzionalità la cui complessità d'uso le rende di fatto inutilizzabili.

Nello standard ISO 9126, l'usabilità viene scomposta in cinque sottocaratteristiche: comprensibilità, apprendibilità, operabilità, attrattività e conformità.

- Comprensibilità (*understandability*)

Il software si dice *comprendibile* quando un utente è in grado di capire se è adatto per le proprie esigenze e se è in grado di utilizzarlo per svolgere determinati compiti. Tutte le applicazioni software che si adeguano alle linee guida dei sistemi grafici a finestre per l'interfaccia utente (ad esempio quelli di Windows, Mac, OS/2 ecc.) sono un ottimo esempio di applicazione software comprendibile: infatti in tutte le applicazioni le funzionalità di gestione del file si trovano in un menù a tendina che occupa sempre la stessa posizione; per chiudere il programma è messa a disposizione sempre la stessa icona; in tutte le applicazioni è sempre presente un'icona che attiva il salvataggio del file ecc.

58 Capitolo 3 Qualità del software

■ Apprendibilità (*learnability*)

Il livello di *apprendibilità* di un prodotto software indica lo sforzo necessario perché gli utenti ne apprendano il corretto utilizzo. Esistono applicazioni software la cui modalità d'utilizzo risulta particolarmente intuitiva grazie al fatto che le funzionalità offerte sono alquanto semplici. Ciò non significa che l'applicazione sia apprendibile: l'intuitività non va infatti confusa con l'apprendibilità, che serve proprio a comprendere quanto sia facile imparare l'uso di funzionalità che non siano di intuitiva comprensione.

■ Operabilità (*operability*)

L'*operabilità* del prodotto software indica lo sforzo necessario agli utenti per eseguirne e controllarne le varie operazioni. Confrontando tra di loro due applicazioni software che offrono le stesse funzionalità (ad esempio due differenti fogli di calcolo), risulta più operabile l'applicazione che consente di svolgere le operazioni con il minor sforzo per l'utente (minor numero di click del mouse, minor numero di pressioni di tasti ecc.).

■ Attrattività (*attractiveness*)

L'*attrattività* esprime la capacità di un prodotto software di risultare piacevole all'uso per l'utente.

■ Conformità (*compliance*)

Un prodotto software risulta *conforme* per quanto concerne l'usabilità se rispetta standard, linee guida, convenzioni o regole di usabilità. Un esempio classico è fornito dai portali sviluppati per la Pubblica Amministrazione: perché siano conformi (e quindi approvati dal committente) devono essere realizzati rispettando gli standard di navigabilità per le persone ipovedenti.

3.1.3 Affidabilità (*reliability*)

L'*affidabilità* di un prodotto software esprime la sua capacità di mantenere un livello adeguato di prestazioni sotto determinate condizioni e per un periodo di tempo fissato. Si dettaglia nelle seguenti sottocaratteristiche di qualità: maturità, tolleranza ai guasti, ripristinabilità e conformità.

■ Maturità (*maturity*)

Un prodotto software si dice *maturo* se la presenza di guasti¹ non si manifesta come malfunzionamento del prodotto stesso. Quindi, più un prodotto è maturo, meno frequentemente l'utente riscontra dei malfunzionamenti.

¹ Si fa qui riferimento alle definizioni IEEE, utilizzate anche dall'ISO, dei seguenti termini:

- malfunzionamento (*failure*): manifestazione visibile dall'utente di un guasto (ad esempio, la terminazione imprevista dell'applicazione);
- guasto (*fault*): condizione di funzionamento anomalo interno al programma (ad esempio, un valore scorretto di un indice di un array);
- errore (*error*): ciò che causa la presenza di un guasto nel programma (ad esempio, scrivere nel testo l'identificatore "i" anziché "j").

3.1 Il modello di qualità interna ed esterna 59

- Tolleranza ai guasti (*fault tolerance*)

La *tolleranza ai guasti* esprime la capacità del prodotto software di mantenere un determinato livello di prestazioni in presenza di guasti nel software o dell'uso scorretto delle interfacce. Come esempio si pensi a quei programmi che sono in grado, anziché interrompersi bruscamente, di rilevare un proprio malfunzionamento e di continuare a operare anche se con funzionalità ridotte.

- Ripristinabilità (*recoverability*)

La *ripristinabilità* indica la capacità del prodotto software di ristabilire il livello delle prestazioni e di recuperare i dati interessanti antecedenti a un malfunzionamento, nonché il tempo e lo sforzo necessario per tali operazioni. Un esempio di applicazione ripristinabile è relativo a quegli applicativi che offrono la funzionalità di autoriparazione. Tale funzionalità permette infatti di controllare se i file di programma, le librerie e le informazioni di configurazione non si siano corrotti nel tempo e, se necessario, è in grado di ripristinare i file originali.

- Conformità (*compliance*)

Un prodotto software risulta *conforme* per quanto concerne l'affidabilità se rispetta standard, convenzioni o regole di affidabilità.

3.1.4 Efficienza (efficiency)

L'*efficienza* misura il rapporto tra il livello delle prestazioni del prodotto e la quantità di risorse che devono essere impiegate per raggiungere tali prestazioni. Per valutare l'efficienza del software è quindi necessario analizzare da un lato il comportamento temporale e dall'altro il consumo di risorse.

- Comportamento temporale (*time behavior*)

Il *comportamento temporale* assunto dal software si riferisce ai tempi di risposta e di elaborazione, nonché al tempo necessario per smaltire le richieste di esecuzione delle sue funzionalità.

- Consumo di risorse (*resource utilization*)

Il *consumo di risorse* indica la quantità, il tipo di risorse utilizzate e la durata del loro impiego necessari per l'esecuzione delle funzionalità di un prodotto software.

- Conformità (*compliance*)

Un prodotto software risulta *conforme* per quanto concerne l'efficienza se rispetta standard e convenzioni relative all'efficienza.

3.1.5 Manutenibilità (maintainability)

La *manutenibilità* del prodotto software indica la capacità del software di subire modifiche come correzioni, miglioramenti o adattamenti a seguito di cambiamenti dell'ambiente, dei requisiti o delle specifiche funzionali. Si mostrerà nel seguito del capitolo co-

60 Capitolo 3 Qualità del software

me sia possibile adottare un approccio alla progettazione tale da rendere un prodotto più facilmente manutenibile.

La manutenibilità può essere scomposta nelle seguenti sottocaratteristiche: analizzabilità, modificabilità, stabilità, testabilità e conformità.

- **Analizzabilità (*analyzability*)**

Un prodotto software si dice *analizzabile* se consente da un lato di diagnosticare le inadeguatezze e le cause dei malfunzionamenti e, dall'altro, di identificare le porzioni di prodotto su cui è necessario intervenire per apportare una modifica.

- **Modificabilità (*changeability*)**

Il livello di *modificabilità* è un'indicazione dello sforzo necessario per apportare una determinata modifica al prodotto software.

- **Stabilità (*stability*)**

Un prodotto software si dice *stabile* se, apportandovi delle modifiche, non si riscontrano o sono ridotti al minimo gli effetti inattesi dovuti alle stesse.

- **Testabilità (*testability*)**

La *testabilità* del prodotto software esprime lo sforzo necessario per validare il prodotto dopo le modifiche.

- **Conformità (*compliance*)**

Un prodotto software risulta *conforme* per quanto concerne la manutenibilità se rispetta standard e convenzioni di manutenibilità.

3.1.6 Portabilità (portability)

Un prodotto software si dice *portabile* quando può essere facilmente trasferito da un ambiente a un altro (per ambiente si può intendere, per esempio, il sistema operativo, il software di sistema, il middleware, l'ambiente grafico). Per comprendere quanto un prodotto software sia portatile occorre valutare in che misura il software risulta essere adattabile, installabile, sostituibile, conforme e fino a che punto sia in grado di convivere con altre applicazioni. Tali proprietà corrispondono infatti alle sottocaratteristiche di qualità nelle quali è possibile suddividere la portabilità.

- **Adattabilità (*adaptability*)**

L'*adattabilità* di un prodotto software esprime la capacità di adattarsi ad ambienti diversi da quello di origine senza che sia necessario applicare azioni o strumenti oltre a quelli previsti e messi appositamente a disposizione dal software stesso. Un esempio classico di adattabilità è dato dalle applicazioni sviluppate in Java™ e più in generale, dalle applicazioni sviluppate su una *virtual machine* disponibile su ambienti operativi diversi.

3.2 Il modello di qualità in uso 61

- **Installabilità (*installability*)**

L'*installabilità* indica lo sforzo necessario per installare il software in uno specifico ambiente diverso da quello di origine. È ormai irrinunciabile per un prodotto software la disponibilità di un *wizard* (o di meccanismi analoghi) che, attraverso una procedura guida, determina tutti i parametri di installazione e personalizzazione del prodotto, ed evita all'utente di dover effettuare manualmente molte operazioni (copie di file, modifica di file di configurazione ecc.).

- **Coesistenza (*co-existence*)**

La *coesistenza* si riferisce a quanto un prodotto software sia in grado di coesistere in uno stesso ambiente con altre applicazioni indipendenti e di condividere con esse delle risorse comuni. Un esempio in cui tale qualità è assente è fornito da quelle applicazioni che smettono inspiegabilmente di funzionare correttamente a seguito dell'installazione di un altro pacchetto apparentemente totalmente scorrelato. Per quanto riguarda invece la condivisione delle risorse comuni, tutti i moderni sistemi operativi danno la possibilità alle applicazioni software di condividere ad esempio le librerie dinamiche (consentendo quindi di poterle installare una volta sola).

- **Sostituibilità (*replaceability*)**

Un prodotto software si dice *sostituibile* quando è possibile utilizzarlo al posto di un altro specifico prodotto che abbia lo stesso scopo e si riferisca allo stesso ambiente. Un esempio tipico è dato dal proliferare delle versioni open source di pacchetti di office automation che consentono di sostituire quasi completamente i corrispondenti pacchetti commerciali.

- **Conformità (*compliance*)**

Un prodotto software risulta *conforme* per quanto concerne la portabilità se rispetta standard e convenzioni relative alla portabilità.

3.2 Il modello di qualità in uso

Il modello di qualità in uso descrive le caratteristiche di qualità di un prodotto software dal punto di vista dell'utente che lo utilizza in uno specifico contesto. In particolare, esso ne definisce quattro: l'efficacia, la produttività, la sicurezza e la soddisfazione. Secondo lo standard, ciascuna di esse va valutata per uno specifico contesto d'uso.

- **Efficacia (*effectiveness*)**

Un prodotto software è *efficace* se permette a un utente di raggiungere i propri obiettivi completamente e con accuratezza. Questa proprietà è strettamente legata al fatto che i bisogni dell'utente siano soddisfatti, cioè che l'utente ritrovi tra le funzionalità offerte dal prodotto quelle che gli permettono di svolgere il compito che si è prefissato.

62 Capitolo 3 Qualità del software

- **Produttività (*productivity*)**

Un prodotto software si dice *produttivo* se gli utenti, per utilizzare efficacemente il prodotto, devono spendere una limitata quantità di risorse (economiche, tempo di utilizzo, potenza di calcolo, memoria ecc.).

- **Sicurezza (*safety*)**

La *sicurezza* si riferisce alla capacità del prodotto software di garantire accettabili livelli di rischio di danni alle persone, al software, al business, alle proprietà e all'ambiente. Senza ricorrere ad esempi estremi quali i noti fallimenti in ambito aerospaziale, basta pensare a tutte le applicazioni che governano le transazioni bancarie, piuttosto che al firmware che governa gli elettrodomestici più evoluti (ad esempio il firmware di una lavatrice deve mantenere bloccata la porta dell'oblò finché il cestello non si è completamente svuotato d'acqua).

- **Soddisfazione (*satisfaction*)**

La *soddisfazione* indica la capacità del prodotto software di soddisfare gli utenti. Una misura indiretta della soddisfazione di un utente è data ad esempio dall'assiduità con cui l'utente utilizza il prodotto.

3.3 Le metriche del software

Le metriche sono uno strumento utilizzato nell'ambito dell'ingegneria del software, analogamente a quanto avviene in altri settori dell'ingegneria, per misurare le caratteristiche dei prodotti software e dei processi di sviluppo.

Lo standard ISO/IEC 9126 indica alcune metriche che possono essere utilizzate per misurare la presenza delle caratteristiche e sottocaratteristiche di qualità in un determinato prodotto software.

- **Numero di linee di codice (LOC, *Lines Of Code*)**

La proprietà che viene misurata da questa metrica è la dimensione del software. Tale misura non è univoca, poiché esistono diversi modi di determinarla: si può per esempio contare le linee di codice includendo o meno i commenti, considerando solo il codice rilasciato o tutto quello prodotto ecc. Tale metrica consente ad esempio di valutare la produttività del programmatore in termini di linee di codice scritte in una giornata di attività.

- **Numero di classi e interfacce**

Costituisce un'altra possibile misura della dimensione del software, utilizzabile, in particolare, nel caso di codice scritto in un linguaggio object-oriented. Si noti che questa metrica e il numero di linee di codice (e in generale tutte le metriche di questo tipo) richiedono che il codice sia scritto o formattato seguendo regole o linee guida specifiche per poter dare delle misure confrontabili fra software diversi (ad esempio, per il numero di linee di codice, quando "andare a capo" nella scrittura di uno statement, o, per il numero di classi e inter-

3.4 Principi di progettazione come strumenti di qualità 63

facce, quale può essere il numero minimo e massimo di metodi per classe e interfaccia e quale può essere la lunghezza minima e massima del codice di ciascun metodo).

- Function Point (FP)

I Function Point calcolano il numero di funzionalità a partire dalle specifiche dettagliate del prodotto da realizzare, utilizzando un insieme di regole e linee guida definite in un apposito manuale di conteggio. Sono stati definiti per riuscire a misurare, prima che un prodotto software venga realizzato, la “funzionalità” che esso offrirà all’utente. Anche i Function Point possono essere utilizzati per valutare la produttività di un programmatore (numero di Function Point realizzati in un mese). Sono anche comunemente utilizzati per definire il corrispettivo da richiedere al committente per lo sviluppo di un certo numero di funzionalità.

- Numero di errori per linea di codice

Tale metrica misura, sulla base dei risultati dei test o a seguito della messa in esercizio del prodotto software, il rapporto tra il numero complessivo di errori riscontrati e la dimensione del software. Può essere utilizzato come un elemento per valutare l’affidabilità del software.

Esistono inoltre numerose altre metriche definite in letteratura ed è possibile definirne ulteriori per misurazioni specifiche. Sono anche stati definiti in letteratura degli approcci strutturati che consentono di scegliere e stabilire opportunamente le metriche per ciascun obiettivo specifico di misurazione (si veda per esempio il paradigma Goal/Question/Metric discusso nel Capitolo 9).

3.4 Principi di progettazione come strumenti di qualità

Le scelte che si intraprendono durante tutte le fasi della costruzione di un prodotto software per realizzare tutti i semilavorati (descrizione del problema, descrizione architettonica, codice ecc.) influenzano il livello di qualità del prodotto che si otterrà. Infatti esistono delle forti correlazioni tra la modalità con cui si programma il codice e si redigono le descrizioni e le caratteristiche di qualità introdotte in precedenza.

Considerando ad esempio la progettazione, principi quali la modularità, l’information hiding, la coesione e il disaccoppiamento influenzano il conseguimento di caratteristiche di qualità del codice come l’affidabilità, la comprensibilità, la manutenibilità e la portabilità. In realtà, *gli stessi principi devono essere utilizzati e applicati in tutte le situazioni nelle quali si debbano produrre descrizioni complesse*. In particolare, durante la fase di studio del problema devono essere prodotte descrizioni, per esempio, del dominio applicativo. Queste possono essere particolarmente complesse e articolate. Anche nella produzione di queste descrizioni, quindi, devono essere applicati i principi che verranno ora introdotti, così che sia possibile avere una rappresentazione del problema che goda delle proprietà di cui si è parlato in precedenza (in particolare, leggibilità e modificabilità).

3.4.1 Modularità

Un prodotto software, un documento di specifica dei requisiti, un documento di progetto si dicono modulari quando sono composti da porzioni di descrizione sufficientemente indipendenti ancorché integrate, chiamate moduli. Ad esempio, nel caso di codice, il modulo raccoglie un insieme di elementi del programma (funzioni, metodi, classi); nel caso di un documento, è costituito da un insieme di paragrafi o sezioni; nel caso di un diagramma, il modulo è una porzione ben identificata del diagramma complessivo.

L'applicazione del principio di modularità consente di semplificare la gestione di strutture complesse, perché ci si riconduce a gestire un insieme di singole parti più semplici. Per ottenere una buona modularità si devono realizzare i vari moduli in modo tale che essi risultino il più possibile indipendenti gli uni dagli altri. Applicando in modo opportuno i criteri di information hiding, disaccoppiamento e coesione, che verranno descritti nei paragrafi seguenti, si riesce a ottenere un buon livello di indipendenza dei moduli.

3.4.2 Information hiding

Il principio di information hiding si applica ai moduli di un prodotto software (in particolare al codice, alle specifiche formali e alle specifiche di progetto espresse in qualche linguaggio almeno semiformale) e si basa sull'idea di non far trasparire all'esterno del modulo quelle scelte raccolte nel modulo stesso che siano soggette a cambiamenti durante la vita del prodotto software. Per esempio, nello sviluppo di moduli scritti in codice Java™ un buon livello di information hiding si ottiene definendo un insieme ben circoscritto di metodi pubblici, in modo da permettere l'accesso alle funzionalità messe a disposizione dalla classe senza dover necessariamente conoscerne le strutture dati interne. In questo modo la modifica delle modalità secondo le quali è implementato un metodo (ad esempio il cambio di una struttura dati per aumentarne l'efficienza) non ha alcun effetto sull'interfaccia, che è la parte della classe utilizzata da altre classi.

3.4.3 Coesione

Un modulo si dice coeso quando gli elementi che contiene sono tutti strettamente correlati tra di loro. Considerando un documento di progetto, un capitolo che tratti argomenti relativi tutti allo stesso tema risulterà fortemente coeso. Con riferimento invece al codice sorgente, la coesione indica ad esempio quanto il codice definito all'interno di un modulo implementi un unico tipo di dato astratto oppure uno specifico sottoinsieme di funzionalità. Nel contesto di un programma software, si consideri un modulo che racchiuda al suo interno tutta la logica di accesso ai dati persistenti. Tale modulo è fortemente coeso se tutte le operazioni di accesso ai database saranno offerte solo da esso; inoltre all'interno di tale modulo non ci dovranno essere funzioni non correlate con lo scopo del modulo, quali ad esempio la strutturazione e la visualizzazione dei dati estratti da un database.

3.4.4 Disaccoppiamento

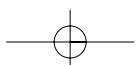
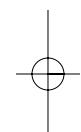
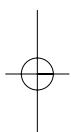
Due moduli si dicono fortemente disaccoppiati quando l'uno accede in modo limitato alle risorse offerte dall'altro. Ad esempio in un'applicazione object-oriented due moduli sono fortemente disaccoppiati se i metodi del primo hanno un numero minimo di riferimenti agli elementi (classi, attributi, metodi) del secondo.

Non c'è ovviamente nessuna utilità a ottenere un disaccoppiamento assoluto, perché ciò significherebbe l'assenza di interazioni tra i vari moduli. Pertanto, un alto livello di disaccoppiamento consente di ottenere che gli eventuali cambiamenti da apportare al software impattino su poche parti del prodotto.

3.5 Riferimenti bibliografici

I problemi legati alla qualità del software sono discussi in moltissimi contributi. A livello tecnico-scientifico, è possibile consultare testi di ingegneria del software quali Ghezzi et al. [2003] o articoli e pubblicazioni specifiche sulle diverse tecniche di verifica e validazione. Lo standard ISO 9126, nella sua più recente versione, è descritto nelle pubblicazioni relative dell'ISO: ISO [2001], ISO [2003] e ISO [2004].

I principi della modularità e della qualità del codice sono stati introdotti negli anni '70 da Dijkstra (Dijkstra [1976]) e Parnas. Sono molti i lavori di Parnas relativi a questo argomento: per un primo approfondimento può essere utile riferirsi a Parnas [1972] (concetto di modularità), Parnas [1976] ("program families") e Parnas [1979] (discussione generale sullo sviluppo di software di qualità).



Capitolo 4

Descrivere il problema

Lo sviluppo di un prodotto software ha come obiettivo la realizzazione di una soluzione che, basandosi su un certo insieme di funzionalità e caratteristiche di qualità, sia in grado di risolvere un determinato problema dell'utente. Perché la soluzione software risulti "adeguata", è di fondamentale importanza concentrarsi dapprima sul problema, delineandone tutti gli aspetti, al fine di poterlo comprendere nel modo più corretto possibile. Quanto maggiormente saranno chiari gli elementi del problema che devono essere risolti, tanto più adeguata risulterà la soluzione software. Lo stesso Einstein affermava che *occorre il 95% del tempo per definire il problema e il 5% per trovare le soluzioni*. Il problema tipicamente si trova all'interno del mondo reale, in uno specifico contesto (dominio applicativo) e comprende un insieme di esigenze, bisogni e desiderata (requisiti) che l'utente esprime. La soluzione software risiede nella "macchina", ovvero nel computer, e dev'essere in grado di produrre degli effetti sul mondo reale tali da soddisfare i requisiti dell'utente. Il collegamento tra la soluzione e lo spazio del problema è costituito dall'interfaccia della soluzione software: l'interfaccia si basa sui fenomeni condivisi tra la soluzione e lo spazio del problema e tramite essa l'utente interagisce con il software (Figura 4.1).

Per comprendere il problema nel modo migliore, poiché esso si riferisce a uno specifico contesto del mondo reale e ai requisiti che in tale contesto l'utente esprime in relazione al problema, è necessario svolgere le seguenti attività.

- Un'attenta analisi del contesto di riferimento mirata a comprenderne tutte le caratteristiche rilevanti (dominio applicativo).
- Un'indagine accurata presso l'utente al fine di evidenziare e far emergere tutti i suoi bisogni ed esigenze (requisiti utente).
- La definizione dell'interfaccia della soluzione software che, basandosi sui fenomeni condivisi tra lo spazio del problema e lo spazio della soluzione, descrive come l'utente potrà interagire con la soluzione software (specifiche dell'interfaccia).

I risultati di tali attività devono essere consolidati all'interno di un documento di descrizione del problema, che per essere compreso in maniera univoca dev'essere compilato con la massima chiarezza.

68 Capitolo 4 Descrivere il problema

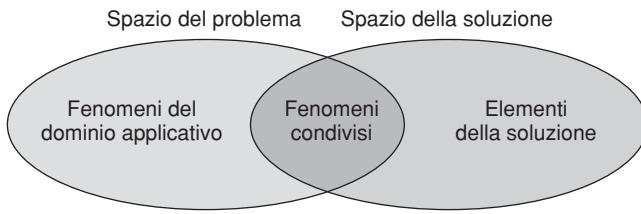


Figura 4.1 Posizionamento del problema e della soluzione.

Un altro aspetto importante da considerare nell'analisi dello spazio del problema riguarda il fatto che non bisogna in alcun modo anticipare gli elementi della soluzione: si deve osservare, comprendere, rilevare nel modo più oggettivo possibile aspetti, proprietà, fenomeni del mondo reale, requisiti dell'utente e caratteristiche dell'interfaccia del sistema, evitando di introdurre aspetti tipici della progettazione. Invece capita spesso che fin dall'inizio, durante la descrizione del problema, vengano involontariamente anticipate delle scelte progettuali, laddove si dovrebbero descrivere tutti e soli gli aspetti del problema. Un esempio tipico di questo fenomeno si verifica quando è necessario memorizzare dei dati: anziché limitarsi a delineare come elemento del problema un generico contenitore di dati, chi svolge l'analisi spesso introduce una specifica soluzione, quale ad esempio un particolare database management system.

I prossimi paragrafi illustrano in dettaglio gli aspetti che descrivono lo spazio del problema. A supporto della spiegazione teorica verrà presentato anche un esempio, relativo a un sistema di commercio elettronico, che consente agli utenti di acquistare online libri e dischi. Per continuità e completezza, tale esempio verrà portato avanti fino all'identificazione della soluzione, descritta nel Capitolo 6. Le descrizioni verranno realizzate tramite l'uso di diagrammi UML e diagrammi introdotti da Jackson [2001].

4.1 Il dominio applicativo

Per identificare e caratterizzare bene il problema, il primo passo da compiere riguarda la comprensione del dominio applicativo. Infatti, esso rappresenta la parte del mondo reale nella quale il software da realizzare dovrà produrre i suoi effetti: esprime l'insieme di tutti i fatti, proprietà e dati relativi allo specifico contesto che si sta considerando. Ogni dominio applicativo è caratterizzato da un insieme di entità, con i loro attributi e relazioni, eventi e leggi causali. Si faccia riferimento al dominio applicativo relativo alla compravendita di automobili: esso è identificato da entità come le automobili e le persone, da attributi come il colore e il costo, da relazioni tra entità quali il fatto che un'auto è posseduta da una persona, da eventi quali il fatto che un'auto viene acquistata da una persona, da leggi causali come la condizione per cui il motore può essere spento se viene girata la chiave in posizione off. In questa fase del processo di sviluppo software, il com-

4.1 Il dominio applicativo 69

pito del progettista (d'ora in avanti indicato come analista) è quello di individuare le caratteristiche del dominio applicativo svolgendo delle analisi nel modo più esaustivo possibile. Un buon approccio da seguire è quello di procedere per raffinamenti successivi: dapprima si identificano le macrocaratteristiche rilevanti del dominio applicativo e poi si dettaglia l'analisi sempre di più, tramite iterazioni successive. Tipicamente il dominio applicativo da analizzare presenta un livello di complessità tale per cui l'analista non è in grado, lavorando da solo, di coglierne tutti gli aspetti rilevanti. È necessario pertanto coinvolgere gli esperti di dominio, persone cioè che conoscono bene come si caratterizza il contesto di riferimento da analizzare.

Comprendere correttamente il dominio applicativo non è sufficiente: occorre descriverlo in modo tale che la caratterizzazione fornita risulti comprensibile in maniera univoca da chiunque la legga. Poiché a valle della fase di studio del problema inizia la progettazione, che tipicamente implica diverse attività che vengono svolte da differenti gruppi di lavoro, è fondamentale che tutte le persone coinvolte abbiano una visione condivisa del problema da risolvere. Si noti che questa considerazione vale anche per la descrizione delle altre parti che compongono il problema, ovvero i requisiti utente e la specifica dell'interfaccia.

Tra i diversi modi con cui è possibile descrivere il dominio applicativo, esistono dei diagrammi che consentono di costruire descrizioni molto chiare. Si tratta dei *context diagram*. Tali diagrammi, definiti da Jackson, sono pensati appositamente per descrivere la relazione tra lo spazio del problema e quello della soluzione. Essi consentono, infatti, di rappresentare:

- il dominio applicativo, come un insieme di sottodomini che caratterizzano il contesto del problema
- la soluzione software
- le interfacce tra i sottodomini e la soluzione software.

In particolare, tali diagrammi distinguono tra due macrotipologie di sottodomini: un *machine domain* e un insieme di *problem domain*. Il machine domain rappresenta il computer e la soluzione software da realizzare; i problem domain rappresentano tutte quelle parti del mondo reale esterno alla macchina rilevanti nel contesto del problema. La Figura 4.2 illustra come viene rappresentato un context diagram: ogni dominio è raffigurato con un rettangolo e, più precisamente, il dominio macchina è un rettangolo con due strisce verticali. Ogni context diagram contiene sempre un unico machine domain. Tutti gli altri rettangoli rappresentano problem domain. Poiché tra le caratteristiche di un dominio applicativo vi è anche il livello di libertà con cui è possibile intervenire sulle sue proprietà in fase di progettazione della soluzione, i context diagram consentono di indicare tale grado di libertà suddividendo i problem domain in *designed domain* e in *given domain*. I domini di tipo designed esprimono la possibilità, in fase di realizzazione della soluzione, di progettarne le caratteristiche (in termini di fatti, entità, attributi, struttura dei dati ecc.) oppure la struttura dei fenomeni con cui il dominio macchina interagisce e vengono raffigurati con un rettangolo con una striscia verticale. Al contrario, quando non è possibile intervenire su nessuna caratteristica di un dominio, esso viene indicato come dominio di tipo given e viene raffigurato con un rettangolo semplice. Quando due

70 Capitolo 4 Descrivere il problema

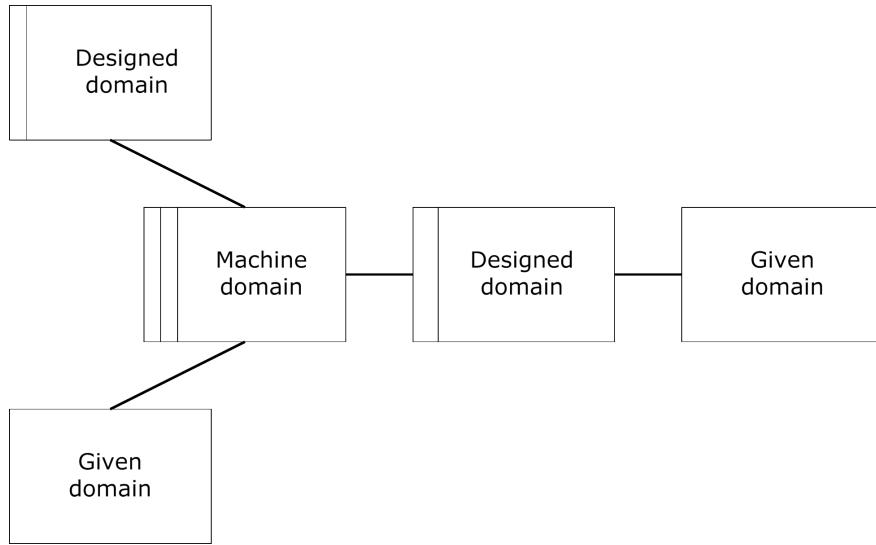


Figura 4.2 Rappresentazione di un context diagram.

domini condividono fenomeni (che possono essere eventi, stati, valori), i rettangoli che li rappresentano vengono uniti tramite una linea di collegamento.

Si supponga di voler descrivere il seguente problema. Una società multimediale vuole estendere la sua rete di vendita al mercato Internet, realizzando un sistema di commercio elettronico che consenta ai suoi clienti di effettuare acquisti online. Di tutti i beni che produce, l'azienda inizialmente vuole destinare al commercio elettronico soltanto libri e dischi. Per tali beni esistono differenti cataloghi (ognuno composto da un determinato insieme di libri e dischi) curati da uno specifico addetto dell'azienda che li mantiene aggiornati inserendo ed eliminando beni. Per gestire gli ordini effettuati online l'azienda vuole mantenere una gestione separata da quella relativa agli ordini effettuati sul canale tradizionale. Tale gestione deve consentire all'addetto ordini aziendale di ricevere le ordinazioni provenienti da Internet, preparare l'ordine internamente all'azienda, spedire la merce e chiudere l'ordine. Per la fatturazione dev'essere utilizzato il sistema di fatturazione già in uso presso l'azienda, che consente di registrare i pagamenti effettuati e di inviare le fatture. Tale servizio di commercio elettronico dev'essere rivolto non solo ai clienti già acquisiti dall'azienda sul canale tradizionale, ma anche a tutti quelli nuovi che vorranno registrarsi "solo" sul canale Internet.

Per comprendere correttamente il problema, il primo passo da compiere riguarda l'identificazione delle macrocaratteristiche rilevanti. I clienti dell'azienda rappresentano sicuramente un'entità di interesse, così come i beni che possono acquistare. Anche i cataloghi attraverso i quali i clienti possono selezionare i beni devono essere contemplati come parte rilevante del problema, così come la gestione degli acquisti (cioè gli ordini e la relativa fatturazione).

4.1 Il dominio applicativo 71

Volendo rappresentare le informazioni raccolte tramite un context diagram si ottiene la descrizione rappresentata nella Figura 4.3. La soluzione software da realizzare (machine domain) è stata chiamata *Sistema di commercio elettronico*. I clienti e i beni sono stati raffigurati come given domain in quanto non sono ovviamente in alcun modo progettabili dalla soluzione software. Il catalogo e gli ordini sono stati rappresentati con un dominio di tipo designed, poiché in fase di progettazione della soluzione possono essere liberamente strutturati. Infine, la fatturazione è rappresentata come un dominio di tipo given poiché dovrà essere utilizzato il sistema di fatturazione già in uso presso l'azienda, pertanto il dominio *Fatturazione* deriva da esso e quindi la sua struttura è imposta. Esiste, per ciascun problem domain, un collegamento con il machine domain: tali collegamenti esprimono lo scambio di fenomeni tra i vari domini coinvolti. Ad esempio, l'insieme di fenomeni scambiati tra il dominio *Clienti* e il machine domain è costituito dalle informazioni che essi si scambiano tramite l'interfaccia utente (nome e cognome del cliente, selezione di un elemento da un elenco ecc.); il collegamento tra il dominio *Catalogo* e il machine domain indica che il sistema di commercio elettronico interagisce con il catalogo scambiandosi le informazioni a esso relative (aggiunta di un bene, selezione dei beni, eliminazione di un bene, selezione dei cataloghi ecc.); il collegamento tra il *Sistema di commercio elettronico* e il dominio *Ordini* indica la condivisione di fenomeni relativi agli ordini effettuati e, indirettamente, alle informazioni sui beni contenuti negli ordini. Non esiste un collegamento diretto tra il dominio dei beni e il *Sistema di commercio elettronico* da realizzare, in quanto tale sistema non gestisce direttamente i beni, bensì gli ordini a essi relativi.

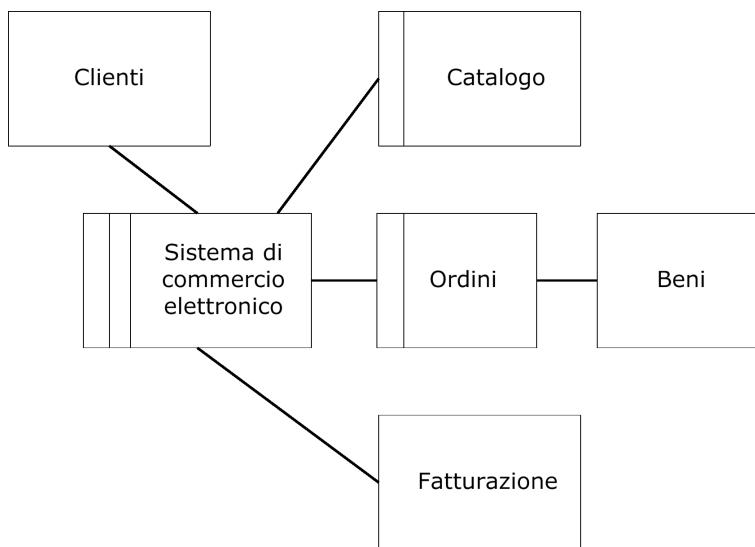


Figura 4.3 Context diagram per un sistema di commercio elettronico.

72 Capitolo 4 Descrivere il problema

La descrizione fornita con il context diagram illustrato nella Figura 4.3 mostra una rappresentazione ad alto livello del contesto di riferimento del sistema di commercio elettronico, ma può essere ulteriormente raffinata. Potrebbe, infatti, essere utile evidenziare i differenti tipi di clienti possibili, quelli già acquisiti e quelli nuovi, così come la tipologia di beni acquistabili. Per chiarire tutti questi aspetti può essere impiegato con successo un diagramma UML delle classi, come quello mostrato nella Figura 4.4. Tale diagramma chiarisce meglio le relazioni esistenti tra le varie entità che costituiscono il dominio applicativo per il sistema di commercio elettronico: da esso si evince infatti che esistono due sottoclassi di Cliente, il nuovo e quello già acquisito, e che tali clienti possono consultare cataloghi, effettuare ordini e richiedere l'emissione delle fatture a essi relativi.

Dopo aver delineato con chiarezza il dominio applicativo del problema, il successivo passo da compiere riguarda l'identificazione di tutti i requisiti utente.

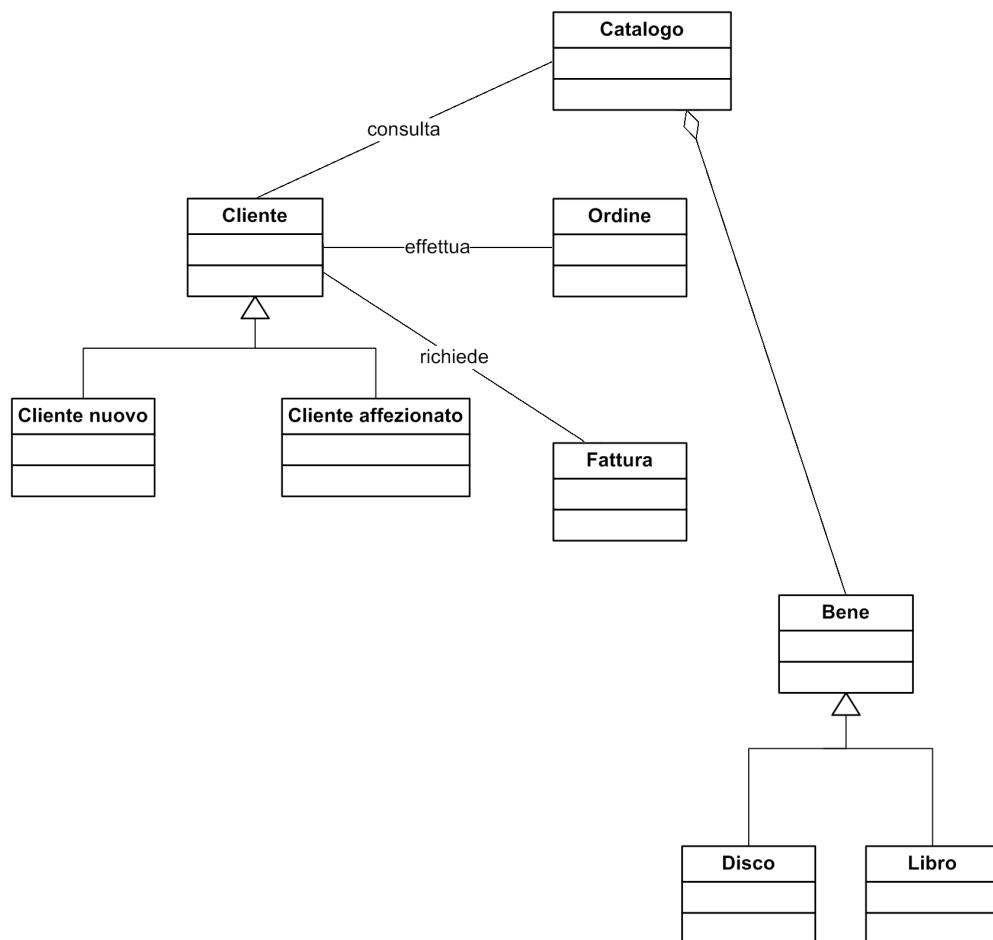


Figura 4.4 Class diagram per un sistema di commercio elettronico.

4.2 I requisiti utente

Una volta noto il dominio applicativo del problema, è necessario individuare tutti i requisiti utente che dovranno essere soddisfatti tramite l'implementazione della soluzione software. Ma cosa si intende per requisito utente? È la capacità, caratteristica fisica o fattore qualitativo che un prodotto o processo deve soddisfare per dare una soluzione al problema; il requisito esprime quindi una condizione o una prestazione necessaria a un utente per risolvere un problema o raggiungere un obiettivo, oppure una condizione o “capacità di fare” che deve essere conseguita o posseduta da un sistema, o da un suo componente, per adempiere a un contratto, uno standard, una specifica o altro documento formalmente imposto.

È importante però capire anche cosa si intende per utente: durante la fase di raccolta dei suoi requisiti infatti ci si deve rivolgere a tutti gli *stakeholder di progetto*. Generalmente si considera stakeholder chi ha un qualche interesse nel sistema software da realizzare.

- *Committente*: chi commissiona il sistema software da realizzare. Tipicamente è costituito dal top management oppure dai manager intermedi dell'azienda committente, che definiscono gli obiettivi di alto livello e i fattori critici di successo che si vogliono ottenere con l'introduzione del sistema software. Nel sistema di esempio di commercio elettronico il committente è la direzione dell'azienda che decide di rivolgersi al canale Internet per estendere il proprio mercato di vendita.
- *Utenti*: coloro che utilizzano il sistema. Tra questi vanno distinti i referenti di processo dagli utenti di riferimento. I primi esprimono i requisiti gestionali e funzionali e sono le fonti primarie dalle quali attingere informazioni sul processo che dovrà essere implementato nel sistema software. Gli utenti di riferimento sono “utenti chiave” che forniscono informazioni di supporto a quelle indicate dai referenti di processo. Nell'esempio del sistema di commercio elettronico, si prenda in considerazione ad esempio il modulo di gestione degli ordini online. Per comprenderne i requisiti gestionali e funzionali occorre intervistare come referente di processo chi si occupa della gestione degli ordini tradizionale del sistema, come utenti chiave gli impiegati dell'ufficio vendite che usano quotidianamente il modulo di gestione degli ordini.
- *Direzione dei sistemi informativi*: fornisce informazioni sugli attuali sistemi informativi (architetture hardware, applicazioni, tecnologie) in uso dal committente. Tipicamente esprime requisiti (o più spesso vincoli¹) tecnologici, di interfaccia verso i sistemi presenti in azienda, di gestione dei dati. Nell'esempio, relativamente all'esigenza di installare il sistema di commercio elettronico, il responsabile dei sistemi

¹ Per vincolo s'intende una limitazione o un requisito隐含的 che vincola il disegno della soluzione o l'implementazione del sistema, che non può essere modificato o negoziato.

74 Capitolo 4 Descrivere il problema

informativi potrebbe opporsi alla possibilità di insediare un'applicazione che dev'essere invocata via Internet all'interno della rete LAN aziendale.

- *Analisti di mercato:* quando il prodotto da sviluppare non è specifico per un committente ma è destinato al mercato, tipicamente la struttura commerciale dell'azienda che lo produce ne definisce i requisiti. Si faccia ad esempio riferimento a un'azienda che produce applicativi per l'ufficio: per definire i requisiti dei vari moduli (foglio di calcolo, word processor ecc.) la struttura commerciale svolge delle indagini di mercato per capire le tendenze e le esigenze di mercato.

Definire requisiti significa rispondere a un insieme di domande, che dipendono dalla natura del semilavorato da realizzare, e che consentano di individuare tutte le esigenze del caso e di tradurle in requisiti. Per fare ciò è sicuramente d'aiuto conoscere quali possono essere i differenti tipi di requisiti in cui far rientrare le esigenze e i fabbisogni degli stakeholder: una loro possibile classificazione si ottiene strutturandoli secondo il seguente schema.

- *Requisiti funzionali:* descrivono in sostanza cosa il sistema deve fare (le sue funzionalità), in termini delle azioni che il prodotto deve svolgere e di quali dati deve trattare. Richiamando il modello di qualità definito dallo standard ISO/IEC 9126 (descritto nel Capitolo 3) i requisiti funzionali possono essere ulteriormente dettagliati facendoli corrispondere con le sottocaratteristiche relative alla caratteristica "funzionalità". Si ottengono pertanto requisiti funzionali di adeguatezza, accuratezza, interoperabilità, aderenza agli standard, sicurezza.
- *Requisiti di usabilità:* descrivono gli appropriati livelli di impiego della soluzione software, quali ad esempio la facilità d'uso e di apprendimento. Per avere l'elenco completo di tali requisiti si può farli corrispondere alle sottocaratteristiche di "usabilità" del modello di qualità ISO/IEC 9126 (comprensibilità, apprendibilità, operabilità, attrattività, conformità).
- *Requisiti di performance:* identificano "quanto bene" il sistema debba funzionare. Con riferimento al modello di qualità ISO/IEC 9126, si possono definire tante famiglie di requisiti di performance quante sono le caratteristiche residue, ottenendo quindi requisiti di affidabilità, efficienza, manutenibilità, portabilità. Analogamente a quanto visto sopra, si possono dettagliare ulteriormente tali famiglie di requisiti secondo lo schema delle relative sottocaratteristiche.
- *Requisiti di interfaccia:* sono i requisiti relativi a tutte le interfacce esterne del sistema da realizzare, quali interfacce hardware, interfacce software, interfacce di comunicazione e interfacce utente.
- *Requisiti fisici:* si riferiscono a tutte le caratteristiche fisiche, come quelle elettriche, o relative ai materiali, alle dimensioni, agli aspetti meccanici per dispositivi "embedded" in altri prodotti (per esempio, un computer che deve operare come sistema di controllo a bordo di un aereo).
- *Requisiti tecnologici:* sono relativi alle tecnologie da utilizzare e possono avere un impatto sia sull'infrastruttura che sull'ambiente.

4.2 I requisiti utente 75

- *Requisiti relativi a licenze/brevetti*: da utilizzare (o meno) o da produrre in concomitanza con lo svolgimento del progetto.
- *Vincoli*: sono costituiti da obiettivi, regole generali o limitazioni, generalmente definiti a livello alto (relativi cioè al sistema o sottosistema). Esempi di vincoli sono standard da rispettare, limitazioni hardware, costi imposti per la realizzazione del prodotto.

Il risultato della fase di analisi dei requisiti deve produrre l'elenco di tutti i requisiti del sistema. Dev'essere redatto un documento che raccoglie tutti i requisiti, catalogandoli e strutturandoli. Tale documento di descrizione dei requisiti deve risultare il più possibile non ambiguo, leggibile, completo e i requisiti devono essere coerenti gli uni con gli altri: infatti, poiché gli stakeholder esprimono punti di vista diversi, a volte hanno esigenze contrastanti dalle quali scaturiscono requisiti conflittuali che devono ovviamente essere risolti.

Per ottenere tale documento si può procedere per passi, analizzando i requisiti a un livello di dettaglio sempre maggiore, tramite iterazioni successive (come già visto per il dominio applicativo). Alla fine si deve ottenere una tabella che, per ciascun requisito, raccoglie le seguenti informazioni.

- Nome del requisito.
- Identificatore unico.
- Tipo di requisito: funzionalità, usabilità, performance, interfaccia, fisico, tecnologico, licenze, vincolo.
- Priorità: ordine di priorità con cui dovranno essere soddisfatti i requisiti.
- Criticità (alta, media, bassa): indica quanto un requisito è fondamentale.
- Livello di rischio (alto, medio, basso): esprime la presenza di fattori esterni che possono influenzare il raggiungimento del requisito.
- Data di rilevazione.
- Breve descrizione: consente di spiegare il requisito ad alto livello.

Per raccogliere queste informazioni può essere impiegata una matrice come quella indicata nella Tabella 4.1. È necessario evidenziare il fatto che, poiché per ogni requisito nella matrice viene fornita una descrizione a livello generale, in qualche altro documento complementare dev'essere fornita anche una molto particolareggiata.

Ma come si può procedere per fornire una descrizione che, partendo da una panoramica generale, divenga sempre più particolareggiata fino a raggiungere il livello voluto? Per fornire una visione generale di alto livello dei requisiti del sistema, possono ad esempio essere impiegati i problem diagram definiti da Michael Jackson, unitamente a diagrammi UML.

I problem diagram estendono i context diagram, introducendo i requisiti e mostrando come essi si correlano ai vari sottodomini che compongono il dominio applicativo. Inoltre consentono di indicare esplicitamente, tramite delle annotazioni sulle interfacce, il tipo di fenomeno condiviso tra due domini. La Figura 4.5 illustra come

76 Capitolo 4 Descrivere il problema

Tabella 4.1 Una possibile matrice dei requisiti.

rappresentare un generico problem diagram: i requisiti vengono indicati attraverso un ovale tratteggiato; le relazioni esistenti tra i requisiti e i sottodomini del problema vengono rappresentate tramite frecce o linee tratteggiate a seconda che i requisiti impongano oppure no dei vincoli sul dominio cui si riferiscono; le annotazioni sulle interfacce, indi-

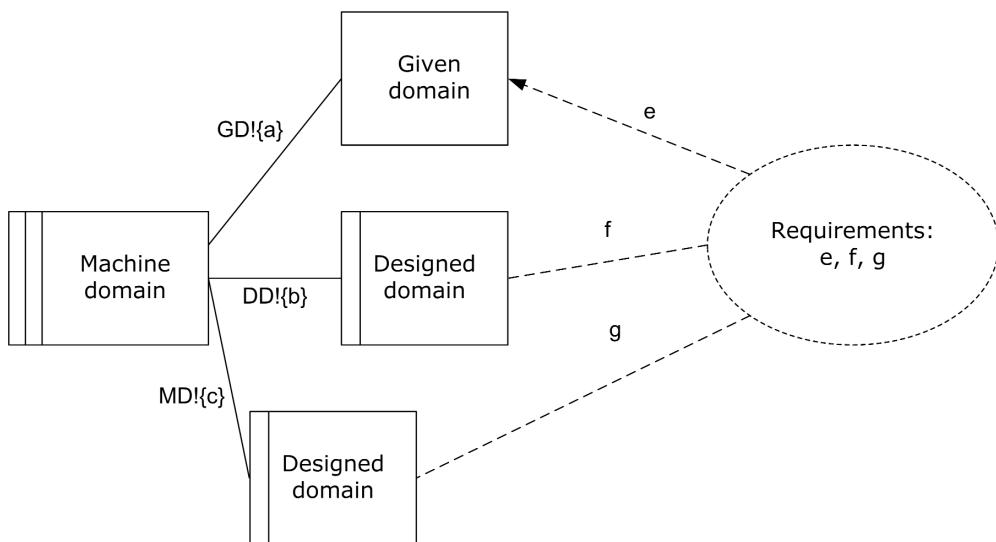


Figura 4.5 Rappresentazione di un problem diagram.

4.2 I requisiti utente 77

cate tramite una sintassi del tipo `dominio!{eventi}`, forniscono informazioni relative all'insieme di eventi condivisi tra due domini (indicati con `{eventi}`) mostrando quale parte del sistema controlla i suddetti eventi (indicato con `dominio!`).

Riprendendo l'esempio del sistema di commercio elettronico, si illustrerà come sia possibile fornire delle descrizioni dei requisiti, considerando il sistema dapprima da un punto di vista specifico, quello dei clienti, e poi da un punto di vista complessivo. I requisiti funzionali relativi ai clienti sono riconducibili alle seguenti funzionalità: il sistema deve permettere ai nuovi clienti che accedono al sistema online di registrarsi; deve offrire a tutti i clienti la possibilità di consultare i cataloghi dei prodotti acquistabili; deve consentire ai clienti di comporre degli ordini e di richiedere l'emissione della fattura per gli ordini effettuati. È possibile descrivere graficamente tali requisiti, tramite il problem diagram mostrato nella Figura 4.6. L'ovale tratteggiato racchiude al suo interno l'elenco di tutti i requisiti individuati: in particolare, il requisito relativo alla registrazione degli utenti (indicato con `a` sulla relazione verso il dominio corrispondente) si riferisce al sottodomino dei clienti e la relazione tra di essi è indicata con una freccia poiché la registrazione di un nuovo cliente modifica lo stato del dominio dei clienti, in quanto ne aggiunge uno nuovo. Il requisito di consultazione del catalogo è relativo al dominio Catalogo, e la relazione tra di essi (indicata con la lettera `b`) è mostrata con una linea in quanto la consultazione non modifica lo stato del dominio Catalogo. Entrambi i requisiti di Composizione

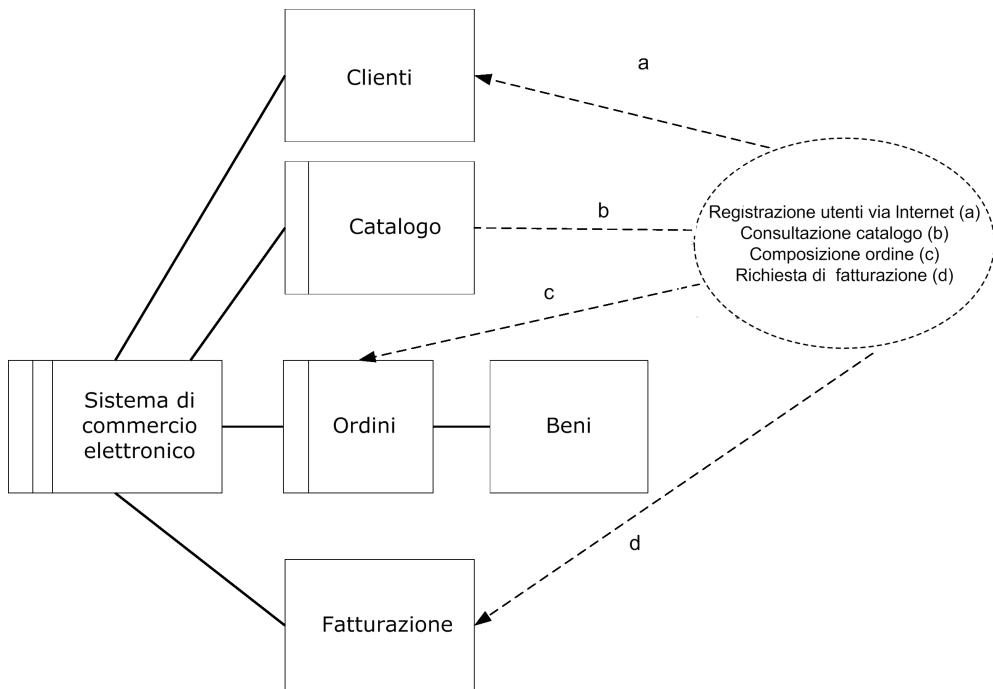


Figura 4.6 Problem diagram per il sistema di commercio elettronico.

78 Capitolo 4 Descrivere il problema

ordine (c) e Richiesta di fatturazione (d) sono collegati ai domini ai quali si riferiscono (Ordini e Fatturazione, rispettivamente) con una freccia, poiché modificano lo stato dei domini stessi (in entrambi viene aggiunto un elemento nuovo).

Dopo aver descritto i requisiti funzionali del cliente a livello generale, è necessario specificare in modo più dettagliato le entità e le funzionalità coinvolte.

Si supponga di svolgere un'analisi più approfondita presso l'azienda committente, al fine di comprendere meglio le relazioni/associazioni che esistono tra le varie entità di cui si compone il dominio applicativo, come ad esempio la relazione che lega tra di loro clienti, cataloghi e beni, piuttosto che i clienti, ordini e fatture. Dall'analisi emerge che ogni cliente può consultare più di un catalogo, e ogni catalogo può essere consultato da più clienti; ogni catalogo inoltre contiene un insieme di beni variabile da uno a più elementi, e uno stesso bene può essere contenuto in cataloghi differenti. I clienti possono effettuare da zero a un numero illimitato di ordini e ogni ordine può essere emesso da un unico cliente. Inoltre ci dev'essere una corrispondenza univoca tra ordini e fatture a essi relative. Per rappresentare tutte queste informazioni può essere impiegato il diagramma delle classi mostrato in precedenza e opportunamente arricchito, come illustrato nella Figura 4.7.

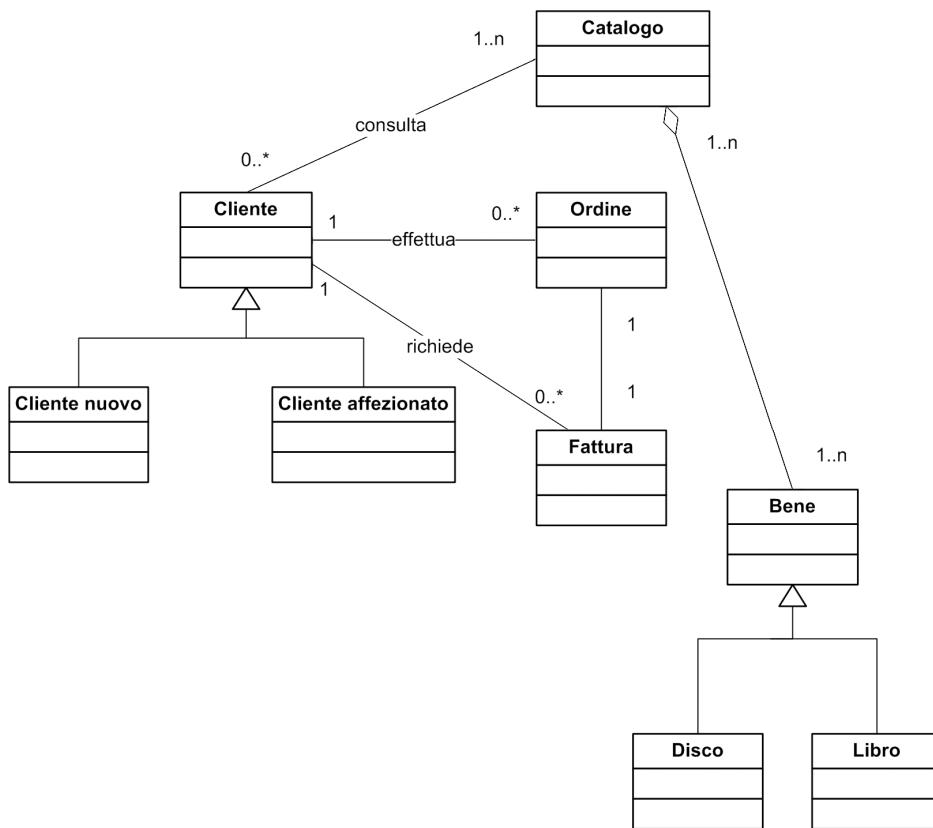


Figura 4.7 Associazioni tra le classi del sistema di commercio elettronico.

4.2 I requisiti utente 79

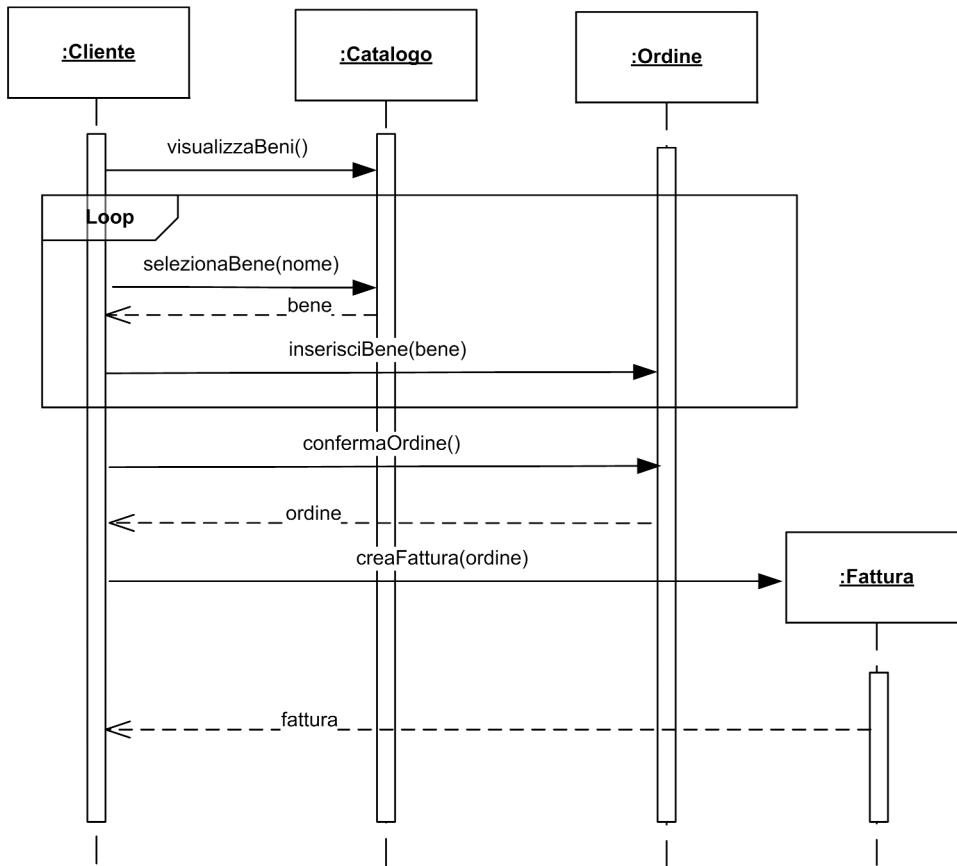


Figura 4.8 Descrizione dinamica della composizione di un ordine.

Volendo introdurre un livello di dettaglio ulteriore nella descrizione dei requisiti funzionali del cliente, ci si può ad esempio concentrare sulle attività che compongono i vari requisiti. Ad esempio facendo riferimento al requisito di composizione degli ordini, può essere utile fornire una descrizione dinamica delle attività che devono essere svolte dal cliente per comporre un ordine. A tale scopo può essere impiegato il sequence diagram mostrato nella Figura 4.8, che descrive le azioni che il cliente svolge per comporre un ordine e richiederne la relativa fattura.

Un'altra informazione attinente alla descrizione dinamica è costituita dall'illustrazione degli stati possibili entro i quali si può trovare un ordine e le azioni che fanno passare l'ordine da uno stato all'altro, come descritto nello state diagram della Figura 4.9.

80 Capitolo 4 Descrivere il problema

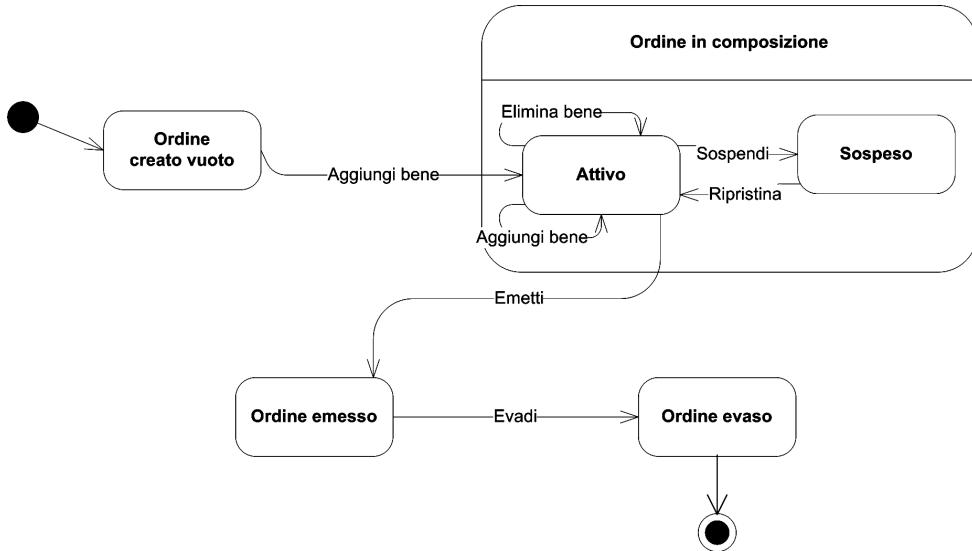


Figura 4.9 State diagram dell'ordine.

A partire da una vista particolare sul problema, e cioè quella relativa soltanto ai requisiti funzionali del cliente, è possibile fornire delle descrizioni che partono da un livello di astrazione elevato per poi dettagliarsi a passi successivi. Ortopogonalmente a tale vista può essere invece fornita una panoramica generale del sistema, in cui vengono descritti tutti i requisiti funzionali del sistema. Per fare ciò si può ricorrere all'utilizzo di use case diagram e activity diagram, come riportato nella Figura 4.10 e nella Figura 4.11.

Lo use case mostrato nella Figura 4.10 individua i quattro attori Cliente, Addetto catalogo, Gestore ordini e Addetto pagamenti e le relative funzionalità. Per descrivere le funzionalità relative all'attore Cliente è stata utilizzata la relazione di tipo `<<extend>>`, per mostrare il fatto che per acquistare un bene il cliente nuovo rispetto a quello vecchio deve svolgere una funzionalità in più: la registrazione. Per descrivere le funzionalità degli altri attori sono state utilizzate relazioni di tipo `<<include>>`. L'Addetto catalogo gestisce il catalogo tramite le operazioni di inserimento ed eliminazione dei beni; il Gestore degli ordini invece deve ricevere l'ordine, prepararlo, spedire la merce e chiudere l'ordine. L'Addetto ai pagamenti si occupa della generazione delle fatture registrando i pagamenti effettuati e inviando le fatture.

Per descrivere come avvengano dinamicamente le correlazioni tra le funzionalità svolte, ad esempio, tra gli attori Gestore ordini e Addetto ai pagamenti, può essere impiegato con successo un activity diagram, come quello descritto nel Capitolo 2 e qui riportato per comodità di consultazione nella Figura 4.11.

4.2 I requisiti utente 81

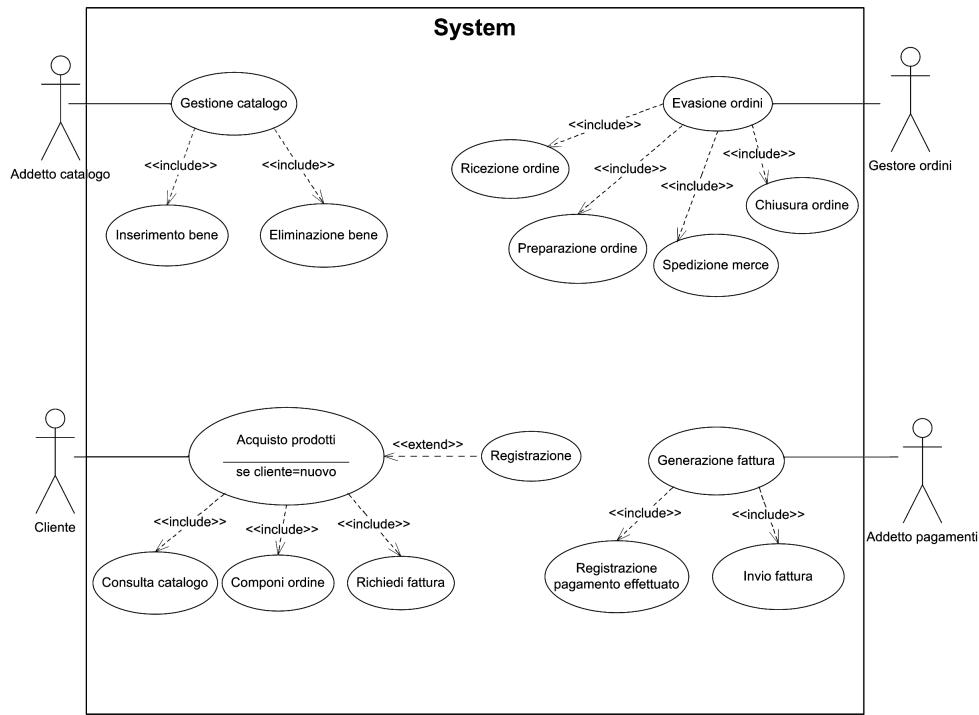


Figura 4.10 Use case diagram per il sistema di commercio elettronico.

La descrizione dei requisiti costituisce un documento molto importante perché è una sorta di contratto tra il committente e il progettista: nei confronti del progettista lo tutela perché indica tutti e soli i fabbisogni, le esigenze e i desiderata che il sistema dovrà implementare; tutela il committente perché indica tutti i requisiti che il committente avrà il diritto di vedere soddisfatti nel sistema realizzato. È importante quindi che sia scritto in modo molto comprensibile (anche dal committente) e univocamente interpretabile e verificabile. Perché sia verificabile è opportuno identificare, per ciascun requisito, i collegamenti che esso ha con le fonti/documenti che l'hanno originato, con i componenti che lo implementeranno e con eventuali altri requisiti. Tale operazione, detta *tracciatura dei requisiti*, consente infatti di seguire l'evoluzione di ogni requisito, a partire dalla fonte che l'ha espresso (tracciatura verso l'alto) fino a giungere alla sua implementazione in una specifica funzionalità o componente del prodotto finale (tracciatura verso il basso). La tracciatura orizzontale consente infine di mantenere la traccia di come un requisito sia collegato ad altri (ad esempio quando si deriva un requisito figlio da uno padre). Da notare che la tracciatura verso il basso non può ovviamente essere svolta durante la fase di

82 Capitolo 4 Descrivere il problema

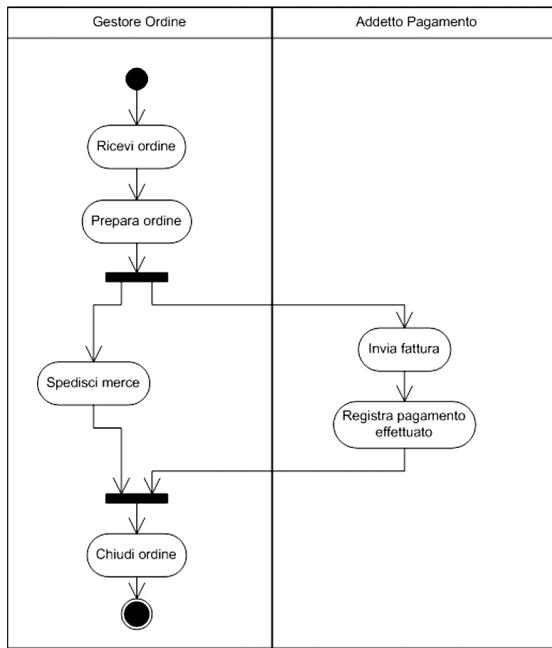


Figura 4.11 Activity diagram per il sistema di commercio elettronico.

analisi dei requisiti, ma solo dopo che questi saranno stati implementati nella soluzione software. La tracciatura dei requisiti può essere effettuata ad esempio utilizzando la tabella citata in precedenza, in cui si aggiungono delle colonne per indicare le fonti da cui il requisito è stato originato, i moduli implementativi attraverso i quali è stato soddisfatto e le relazioni con gli altri requisiti.

In conclusione, per vedere come sia possibile giungere a una descrizione dettagliata dei requisiti funzionali, si riprende l'esempio del sistema di commercio elettronico e si costruisce la matrice dei requisiti, che, rispetto alla matrice precedentemente mostrata nella Tabella 4.1, presenta anche le colonne relative alla tracciabilità. Il risultato è illustrato nella Tabella 4.2: a ogni riga corrisponde uno specifico requisito funzionale. La tabella riporta solo i requisiti relativi al cliente e all'addetto catalogo. Si è ipotizzato che le informazioni relative ai requisiti siano state ricavate durante riunioni svolte presso il committente. Il rischio associato al requisito relativo alla registrazione dei clienti è alto, poiché durante la registrazione vengono inviati dati personali, che quindi vanno gestiti in modo sicuro per evitare intrusioni e conseguenti letture/manomissioni degli stessi.

4.2 I requisiti utente 83

nome requisito	id requisito	tipo	priorità	criticità	rischio	data	descrizione	fonte	implemen- tazione	requisito padre	requisiti figli
acquisto prodotti	1	funzionale	1 alta	basso	22/06/2005	deve permettere al cliente di fare acquisti online	verbale riunione 22/06/05			2,3,4,5	
registrazione	2	funzionale	1 alta	alto	22/06/2005	deve permettere al cliente nuovo di registrarsi	verbale riunione 22/06/05			1	
consulta catalogo	3	funzionale	1 alta	basso	22/06/2005	deve consentire al cliente di consultare i cataloghi disponibili dei prodotti	verbale riunione 22/06/05			1	
componi ordine	4	funzionale	1 alta	basso	22/06/2005	deve permettere al cliente la composizione degli ordini, selezionando i prodotti dai cataloghi	verbale riunione 22/06/05			1	
richiedi fattura	5	funzionale	1 alta	basso	22/06/2005	deve consentire al cliente di richiedere l'emissione della fattura	verbale riunione 22/06/05			1	
gestione catalogo	6	funzionale	2 bassa	basso	01/07/2005	deve consentire al gestore catalogo di aggiornare i cataloghi	verbale intervista 01/07/05			7,8	
inserimento bene	7	funzionale	2 bassa	basso	01/07/2005	deve permettere al gestore catalogo di inserire un bene in un catalogo	verbale intervista 01/07/05			6	
eliminazione bene	8	funzionale	2 bassa	basso	01/07/2005	deve permettere al gestore catalogo di eliminare un bene dal catalogo	verbale intervista 01/07/05			6	

Tabella 4.2 La matrice dei requisiti per il sistema di commercio elettronico.

4.2.1 La comunicazione con gli stakeholder

L'obiettivo della raccolta dei requisiti consiste nel catturare tutti i fabbisogni ed esigenze manifestate dagli stakeholder e formalizzarli in requisiti. Questa fase è fortemente influenzata anche dalla modalità con cui viene gestita la comunicazione con gli stakeholder. Esistono diverse tecniche di comunicazione che possono essere impiegate durante lo svolgimento della raccolta dei requisiti, come per esempio le seguenti.

- **Intervista:** interazione tra intervistato e intervistatore, avente scopo cognitivo, guida dall'intervistatore sulla base di uno schema di interrogazione (traccia) definito a priori e rivolta a un numero di soggetti sulla base di un piano di interazione. Le

84 Capitolo 4 Descrivere il problema

interviste possono essere semistrutturate o strutturate, a seconda della struttura che viene data alla traccia (questionari con domande aperte oppure chiuse) e alla modalità di conduzione dell'intervista.

- *Brainstorming*: discussione di gruppo, in cui un animatore illustra l'oggetto della discussione e regola gli interventi con i quali i singoli esprimono le proprie opinioni. A partire dalle idee espresse dai singoli si arriva a formulare un'idea condivisa migliore delle singole proposte. Poiché tale approccio si basa sul concetto di collaborazione, funziona molto bene in tutti quei casi in cui i partecipanti percepiscono di essere tutti allo stesso livello.
- *Focus group*: intervista di gruppo focalizzata, in cui un moderatore pone delle domande e i singoli individui rispondono alla presenza di tutti. Si tratta di discussioni di gruppo, in cui c'è un buon livello di interattività e attraverso le quali si riescono a ottenere diversi punti di vista dello stesso aspetto. Tale approccio è particolarmente indicato nelle prime fasi di raccolta dei requisiti.
- *Condivisione di prototipo*: al fine di verificare la corretta comprensione dei requisiti, può essere utile ricorrere alla creazione di un prototipo che simuli le funzionalità di prodotto richieste. La raccolta dei requisiti si basa sui feedback raccolti dalla condivisione del prototipo con gli stakeholder. Questo approccio risulta applicabile non solo per condividere i requisiti di un nuovo prodotto, ma anche per gestire le richieste di modifiche dei requisiti di un prodotto che si trova già nelle fasi del processo di sviluppo a valle della prototipazione (di laboratorio, alpha o beta).
- *Interazioni indirette*: alcune informazioni o presupposti per la corretta definizione dei requisiti possono essere contenute in fonti informative già esistenti, come ad esempio la documentazione contrattuale (capitolati tecnici, allegati tecnici, contratti ecc.), i verbali di riunioni, la documentazione della situazione attuale del committente (flussi fisici e logici dei processi interessati dallo sviluppo del software, architettura tecnica dell'IT, architettura applicativa dei sistemi informativi in uso ecc.).

4.3 Specifica dell'interfaccia

Un generico problema software può essere definito come l'esigenza di configurare un computer e il suo software (sistema informatico) in modo tale da produrre determinati effetti in una o più parti del mondo reale. L'analisi del dominio applicativo consente di individuare quali parti del mondo reale sono rilevanti nell'ambito del problema. Attraverso l'analisi dei requisiti utente tali parti vengono caratterizzate sulla base degli effetti che devono essere prodotti: tali effetti sono infatti determinati dalle specifiche esigenze, fabbisogni e richieste che gli utenti vogliono vedere soddisfatti. Per completare la descri-

4.3 Specifica dell'interfaccia 85

zione del problema occorre quindi comprendere in che modo il sistema informatico debba o possa "interagire" con il mondo reale: tale attività prende il nome di specifica dell'interfaccia del sistema informatico.

In generale le interazioni tra due sistemi avvengono attraverso le interfacce: nel caso in esame occorre pertanto individuare tutte le interfacce esistenti tra il mondo reale (dominio applicativo) e il sistema informatico. Si è visto in precedenza che lo spazio del problema e quello della soluzione condividono alcuni fenomeni. La specifica dell'interfaccia del sistema software deve descrivere attraverso quali fenomeni il sistema software si interfaccia e interagisce con tutte le varie parti del mondo reale di cui si compone il dominio applicativo. Con riferimento ai context diagram, definire l'interfaccia del sistema informatico significa definire le interfacce con tutti i sottodomini, sia di tipo designed che di tipo given.

Le interfacce del sistema si riferiscono principalmente a due famiglie di interazioni: quelle con gli utenti del sistema informatico e quelle che vengono svolte con i sistemi che fanno parte del dominio applicativo. Le prime vengono indicate come interfacce utente, le altre come interfacce hardware e software.

Come si può redigere la specifica dell'interfaccia? Essa può essere costruita fornendo due descrizioni complementari delle interfacce.

- Descrizione statica: descrive gli elementi che costituiscono l'interfaccia. Ad esempio, gli elementi di cui si compone l'interfaccia utente sono dati dai menu (per esempio di selezione delle funzionalità offerte all'utente), dalle maschere (come le finestre di dialogo), dai pulsanti, dalle caselle di testo, e così via. Nel caso invece di interfaccia software verso un sistema che fa parte del dominio applicativo (nell'esempio del sistema di commercio elettronico il sistema di fatturazione) gli elementi che la costituiscono sono riconducibili ai metodi che possono essere invocati sul sistema.
- Descrizione dinamica: mostra come deve evolvere nel tempo l'interazione con gli elementi dell'interfaccia. Nell'interfaccia utente la descrizione dinamica spiega ad esempio che premendo il pulsante X appare il menù Y, viene visualizzata la maschera A e viene generato l'evento C. Nel caso dell'interfaccia software, la descrizione dinamica mostra l'ordine con cui devono essere invocati i metodi del sistema con il quale la soluzione software si deve interfacciare.

Riprendendo l'esempio del sistema di commercio elettronico, ci si concentrerà su due interfacce: l'interfaccia utente e l'interfaccia software verso il sistema di fatturazione. Un modo molto chiaro ed esaustivo di descrivere l'interfaccia utente, sia dal punto di vista statico che dinamico, può essere quello di fornire un prototipo grafico che mostri l'insieme di tutti gli elementi che compongono l'interfaccia e le interazioni previste per ciascuno di essi. Solitamente tale prototipo è indicato col nome di *mock up* grafico. A titolo d'esempio vengono mostrati tre screen shot di un possibile mock up grafico del sistema di commercio elettronico. Il primo, illustrato nella Figura 4.12, è relativo alle seguenti funzionalità:

86 Capitolo 4 Descrivere il problema

The screenshot shows a web page for 'Azienda XY Sistema di e-commerce'. At the top, there is a navigation bar with five links: 'Informazioni sul servizio', 'Guida online', 'Chi siamo', 'Come contattarci', and a blank space. Below the navigation bar, there is a large, light-gray area containing two rounded rectangular boxes. The left box is labeled 'Utente registrato' and contains fields for 'Username' and 'Password', followed by a gray 'Enter' button. The right box is labeled 'Nuova registrazione'.

Figura 4.12 Login e registrazione nell'interfaccia utente.

- informazioni varie sul servizio e sull'azienda
- login per i clienti già registrati
- registrazione per i nuovi clienti.

Quando un cliente registrato inserisce username e password, il sistema deve visualizzare l'elenco dei cataloghi che possono essere consultati. La Figura 4.13 mostra uno screen shot in cui nella parte sinistra della schermata vengono elencati tutti i cataloghi selezionabili, mentre nella parte destra vengono mostrati alcuni cataloghi particolari. Quando il cliente ne seleziona uno, il sistema deve mostrare quali beni fanno parte di tale catalogo e deve fornire al cliente la possibilità di selezionare i beni per inserirli nell'ordine. Si supponga che il cliente abbia selezionato il catalogo A: lo screen shot della Figura 4.14 illustra i beni contenuti e per ciascuno consente al cliente di inserirlo nell'ordine selezionandone anche la quantità. A questo punto della sua interazione con il sistema al cliente vengono offerte delle funzionalità di gestione dell'ordine, come la visualizzazione, lo svuotamento e l'emissione. Da notare che il mock up grafico non dev'essere inteso come il risultato della progettazione della veste grafica: il suo scopo principale non è infatti quello di stabilire forma, posizione e colore degli elementi grafici, quanto piuttosto quello di verificare con il committente che l'interfaccia utente del sistema offre tutte le funzionalità richieste e che tali funzionalità si svolgano correttamente e vengano presentate con un susseguirsi corretto di pagine.

4.3 Specifica dell'interfaccia 87

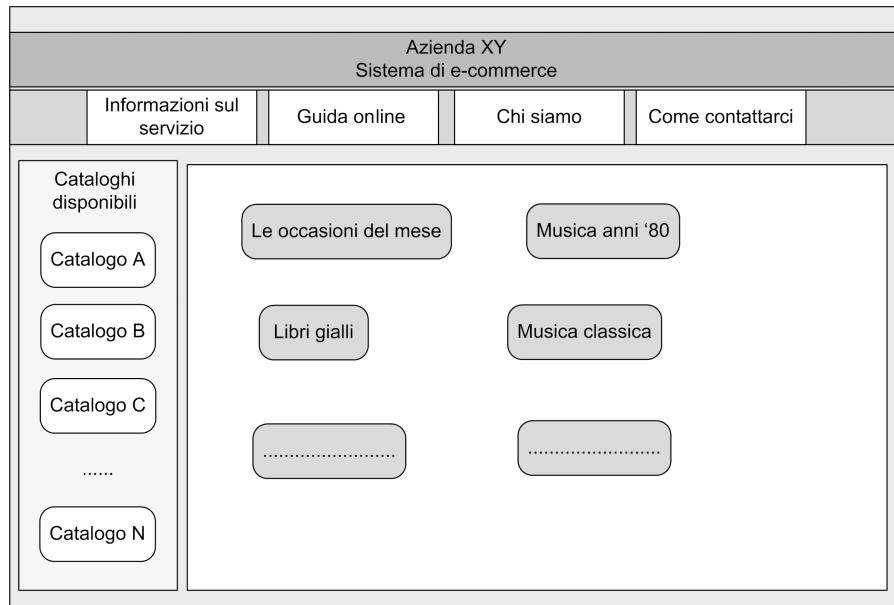


Figura 4.13 Selezione di un catalogo nell'interfaccia utente.

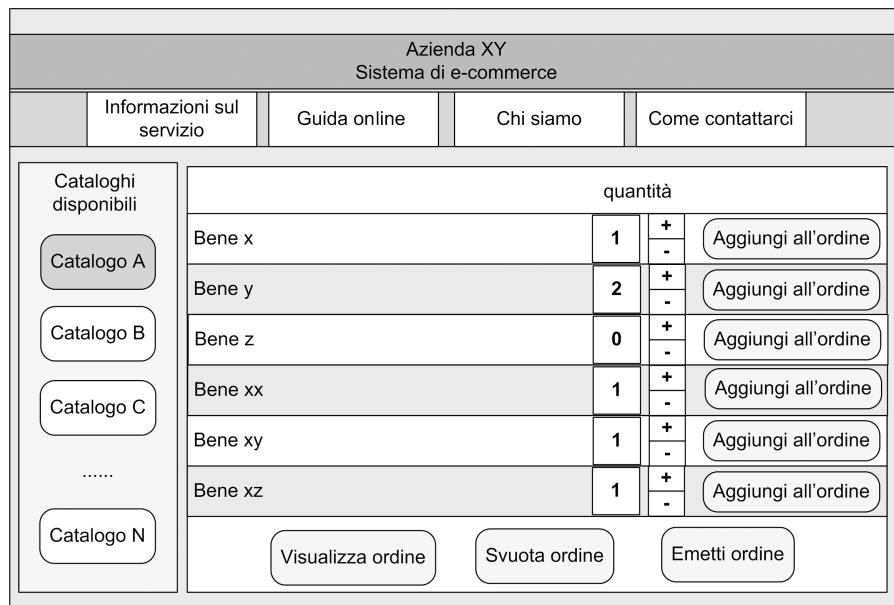


Figura 4.14 Aggiunta dei beni all'ordine nell'interfaccia utente.

88 Capitolo 4 Descrivere il problema

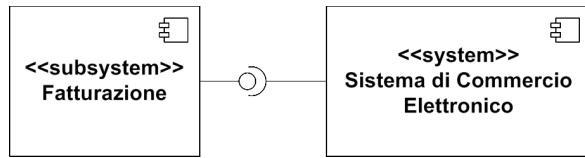


Figura 4.15 Descrizione statica dell'interfaccia software verso il sistema di fatturazione.

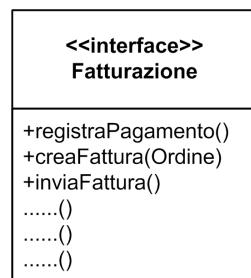


Figura 4.16 Rappresentazione dell'interfaccia di fatturazione.

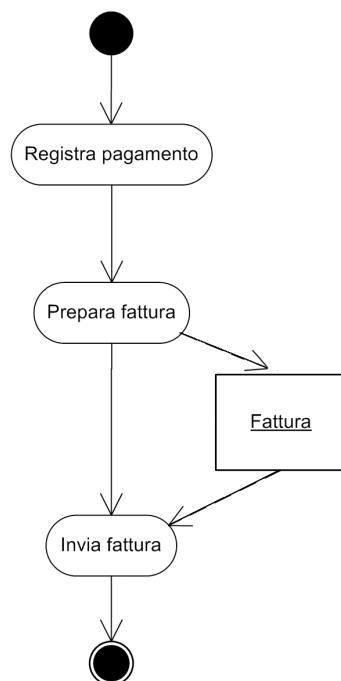


Figura 4.17 Descrizione dinamica dell'interfaccia fatturazione.

4.4 Alcune considerazioni relative a UML 89

Per descrivere staticamente l'interfaccia verso il sistema di fatturazione si può ad esempio ricorrere a un component diagram (si veda la Figura 4.15) che illustra l'esigenza del sistema soluzione (il sistema di commercio elettronico) di invocare l'interfaccia offerta dal sottosistema di fatturazione.

Per descrivere i metodi che fanno parte dell'interfaccia di fatturazione si può ricorrere a un class diagram, come indicato nella Figura 4.16.

La descrizione dinamica dell'interfaccia di fatturazione può ad esempio essere fornita tramite l'activity diagram rappresentato nella Figura 4.17.

Poiché la specifica dell'interfaccia fornisce una descrizione dettagliata del sistema software che è necessario sviluppare, in termini del suo comportamento atteso, essa costituisce un documento molto importante perché è una sorta di contratto fra gli utilizzatori finali del sistema informatico e chi dovrà progettare e implementare il sistema stesso.

4.4 Alcune considerazioni relative a UML

Quanto visto non costituisce l'unica modalità secondo la quale utilizzare UML per descrivere da un punto di vista logico-funzionale un problema. Piuttosto, è rappresentativo di un metodo di lavoro: innanzi tutto *sono state identificate le informazioni che si sono ritenute utili e significative per descrivere il problema*; quindi *sono stati scelti e utilizzati quei diagrammi UML che risultano maggiormente adatti a descrivere tali informazioni*. Ovviamente, sarebbe conveniente e utile utilizzare un unico diagramma, ma ciò è spesso impossibile a causa della molteplicità di informazioni che devono essere rappresentate e delle limitazioni expressive delle singole notazioni. Peraltra, l'utilizzo di diverse notazioni e diagrammi rende necessaria una fase di controllo della coerenza tra le diverse descrizioni sviluppate attraverso UML: per esempio, dev'esserci una corrispondenza tra i nomi dei diversi elementi (e il relativo significato) presenti nei diagrammi.

Nel costruire le descrizioni del problema viste in precedenza sono stati utilizzati diversi diagrammi UML. Da notare che, in funzione della tipologia di sistema da descrivere, delle preferenze dell'analista e della criticità e complessità delle informazioni da rappresentare, il numero e tipo di diagrammi utilizzato potrebbe variare. Per esempio, si potrebbe aggiungere uno state diagram per descrivere gli stati attraverso i quali evolve una singola classe. Ovviamente ciò ha senso se si tratta di un'informazione di natura complessa e che non può essere facilmente e direttamente dedotta dalle descrizioni già esistenti.

Si noti altresì che la stessa informazione può essere, quanto meno parzialmente, replicata in diversi diagrammi. Benché si debba sempre cercare di non introdurre informazioni ridondanti (e quindi potenzialmente incoerenti), è per certi versi ineludibile il fatto che differenti diagrammi possano descrivere le stesse nozioni in forma diversa.

Queste considerazioni sull'uso di UML sono di carattere generale e si applicano non solo allo spazio del problema, ma anche allo spazio della soluzione, in relazione a tutte le descrizioni architettoniche che possono essere prodotte e che verranno discusse nel prossimo capitolo.

4.5 Riferimenti bibliografici

Lo studio dello spazio del problema e la sua caratterizzazione rispetto allo spazio della soluzione hanno trovato un punto di snodo e svolta nel lavoro di Jackson (Jackson [1995] e Jackson [2001]).

Per quanto riguarda l'uso di linguaggi di descrizione, tutti i testi su UML hanno sezioni dedicate all'uso del linguaggio in questa fase del processo di sviluppo. Vi sono molti altri linguaggi utilizzati per descrivere lo spazio del problema. Un linguaggio molto noto e basato su un approccio alquanto diverso da UML è Z (Woodcock e Davies [1996]).

Infine, è importante segnalare lo standard emesso dall'IEEE relativo alle attività di gestione dei requisiti (IEEE [1998]).

Capitolo 5

I problem frame

Il fine ultimo della fase di analisi di un problema software dev'essere quello di fornire descrizioni chiare, complete e univocamente comprensibili. Per descrivere la struttura di un problema software si è visto nel Capitolo 4 che possono, tra gli altri, essere impiegati i problem diagram. Tali diagrammi consentono, attraverso una precisa sintassi, di delineare il problema software in termini di dominio applicativo, requisiti e interfacce del sistema da realizzare.

Generalmente i problemi che un ingegnere del software si trova ad affrontare sono sempre diversi gli uni dagli altri e di una complessità tale per cui risulta abbastanza difficile riuscire a identificare completamente tutte le relazioni esistenti tra i vari domini o le caratteristiche di requisiti e interfacce. Per agevolare la comprensione/descrizione del problema può essere conveniente scomporlo in un insieme di sottoproblemi. Così facendo si ottiene una ristretta rosa di problemi elementari "tipici". In questo modo ogni problema software, pur mantenendo la sua unicità, può essere visto come un'aggregazione di sottoproblemi, la cui struttura è nota.

Michael Jackson ha identificato una libreria di classi di problemi standard (*problem frame*) distinte in base ai requisiti e alle caratteristiche dei domini. In questo capitolo vengono presentati i principali frame, relativi alle seguenti cinque classi di problemi.

- *Required behaviour*
- *Commanded behaviour*
- *Information display*
- *Simple workpieces*
- *Transformation*

Ogni frame viene dapprima introdotto con una descrizione generale e successivamente applicato a un esempio (principalmente a quello relativo al sistema di commercio elettronico presentato nel Capitolo 4). Ogni problem frame è descritto da esempi di diagrammi UML. La scelta dei singoli diagrammi è puramente esemplificativa e non esaurisce in alcun modo le possibili forme di uso di UML. Per una trattazione completa dei problem frame si rimanda il lettore a Jackson [2001].

5.1 Required behaviour

Il *required behaviour frame* esprime tutti quei casi in cui è necessario controllare il comportamento di una qualche parte del mondo reale affinché vengano soddisfatte determinate condizioni. Nella fattispecie, il problema software è relativo alla costruzione di una macchina in grado di imporre un controllo sul suddetto comportamento. Per descrivere un problema di tipo required behaviour, Michael Jackson ha definito un apposito frame che, come illustrato nella Figura 5.1, è composto dai seguenti elementi.

- *Control machine*: tale dominio si riferisce alla macchina (computer e software) da realizzare, che dovrà controllare il comportamento del mondo reale.
- *Controlled domain*: dominio che esprime la parte del mondo reale il cui comportamento dev'essere controllato.
- *Required behaviour*: indica l'insieme dei requisiti, che in questo caso sono costituiti dall'insieme delle condizioni che devono essere soddisfatte dal comportamento della parte del mondo reale controllata.
- *Fenomeni condivisi*: esprimono l'insieme dei fenomeni condivisi tra i domini. In particolare, secondo la sintassi dei problem diagram introdotta nel Capitolo 4, CM!1 indica i fenomeni 1 generati dal dominio macchina (control machine) e scambiati con il dominio da controllare; CD!2 indica i fenomeni 2 generati dal dominio controllato (controlled domain) e scambiati con il dominio macchina; 3 indica le regole di comportamento che il dominio controllato deve seguire. Gli eventi 3 modificano lo stato del controlled domain, da cui la freccia tratteggiata per indicare il loro impatto sul dominio.

Riprendendo come esempio il sistema di commercio elettronico e, in particolare, la gestione dei pagamenti relativi alle fatture emesse, l'azienda vuole poter inviare solleciti ai clienti qualora, a fronte di fatture emesse da più di un mese, non sia ancora pervenuto il relativo pagamento. Per descrivere tale problema si può utilizzare un required behaviour frame, come illustrato nella Figura 5.2. Il dominio macchina è costituito dal modulo di gestione pagamenti, mentre il dominio fatture, che è di tipo given, esprime il dominio da controllare. I requisiti indicano le regole che devono essere applicate per stabilire quando sia necessario generare un sollecito. I fenomeni condivisi tra i due domini modulo di gestione pagamenti e fatture sono indicati con la lettera a: da un lato il modulo di gestione pagamenti modifica lo stato delle fatture portandole nella condizione di sol-

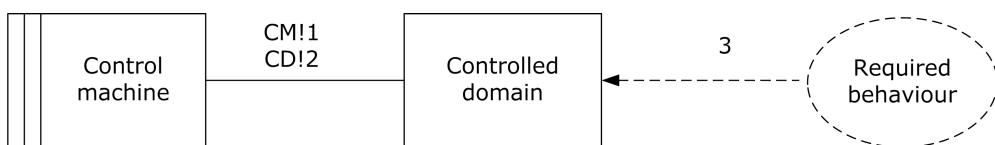


Figura 5.1 Required behaviour frame.

5.2 Commanded behaviour 93

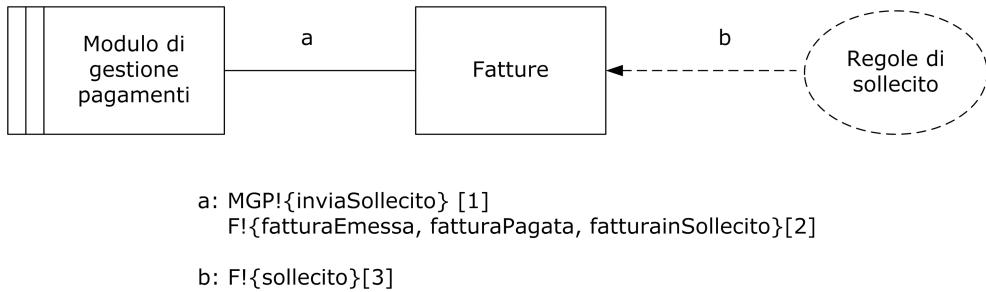


Figura 5.2 Required behaviour frame per il sistema di fatturazione.

lecito (`inviaSollecito`), dall'altro il dominio `Fatture` mette a disposizione del modulo di gestione pagamenti informazioni relative allo stato in cui si trovano le fatture (`fatturaEmessa`, `fatturaPagata`, `fatturainSollecito`). La lettera `b`, infine, indica lo stato (`sollecito`) che dev'essere applicato al dominio `Fatture`. Si è detto che le regole prevedono che venga inviato un sollecito quando, passato un mese dall'emissione della fattura, non sia ancora pervenuto il relativo pagamento.

5.2 Commanded behaviour

Tale frame estende il required behaviour in quanto introduce l'esigenza di controllare il comportamento di una parte del mondo reale prendendo in considerazione anche i comandi imposti da un operatore. Il problema software che ne deriva è pertanto relativo alla costruzione di una macchina che sia in grado di accettare comandi da parte di un operatore e di eseguire in accordo con essi un controllo sul mondo reale. Il diagramma che consente di rappresentare tale problema software è mostrato nella Figura 5.3: poiché il commanded behaviour frame estende il required behaviour, da esso vengono ripresi i due domini control machine e controlled domain e i relativi fenomeni condivisi. Gli elementi aggiuntivi che costituiscono tale diagramma sono dati dal dominio `Operator`, dai requisiti commanded behaviour e dai fenomeni che essi condividono con gli altri domini.

Per facilitare il lettore, di seguito vengono descritti tutti gli elementi che costituiscono il diagramma, nonostante alcuni siano del tutto analoghi a quelli presentati nel required behaviour frame.

- *Control machine*: tale dominio si riferisce alla macchina (computer e software) da realizzare, che dovrà controllare il comportamento del mondo reale.
- *Controlled domain*: dominio che esprime la parte del mondo reale il cui comportamento dev'essere controllato. Tale dominio è di tipo causale.
- *Operator*: tale dominio si riferisce all'operatore umano.

94 Capitolo 5 I problem frame

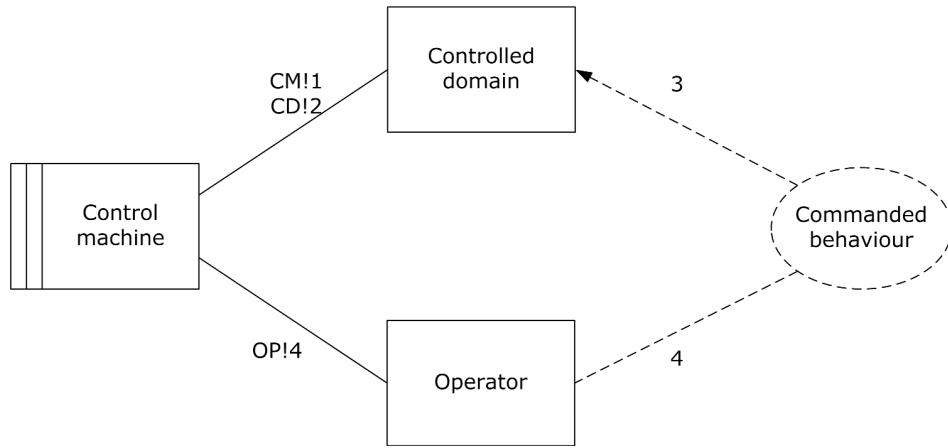


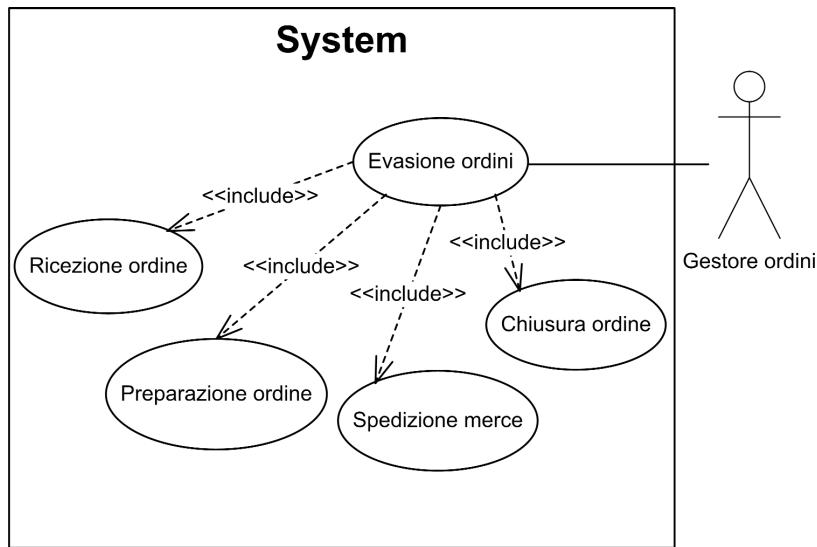
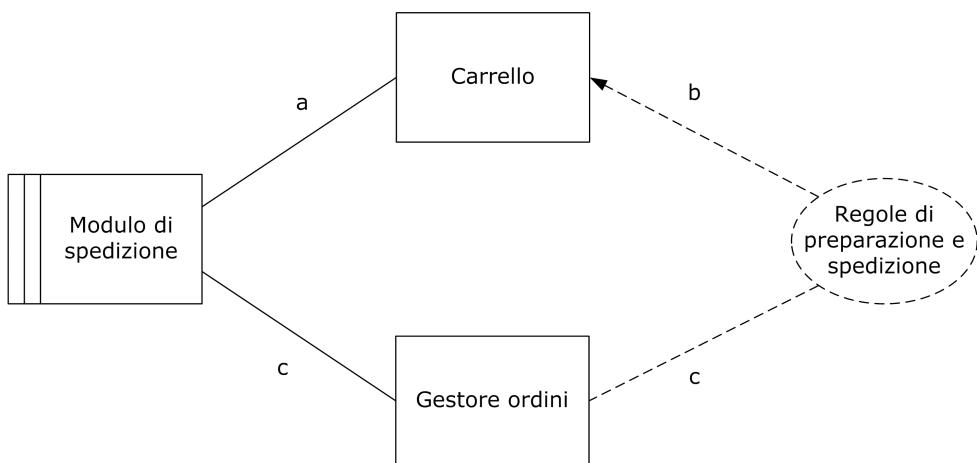
Figura 5.3 Commanded behaviour frame.

- *Commanded behaviour*: esprime l'insieme dei requisiti, che sono relativi da un lato alle regole che il comportamento del dominio controllato deve seguire e dall'altro ai comandi che possono essere eseguiti dall'operatore umano e alla modalità con cui il dominio macchina deve tradurre tali comandi impartiti dall'operatore in uno specifico controllo sul controlled domain.
- *Fenomeni condivisi*: esprimono l'insieme dei fenomeni condivisi tra i domini. 4 si riferisce all'insieme completo di comandi che l'operatore ha a disposizione; analogamente $OP!4$ indica l'insieme di comandi che l'operatore impedisce alla macchina; $CM!1$ indica i fenomeni 1 che il dominio macchina (control machine) genera in corrispondenza degli eventi 4 ricevuti dall'operatore e scambia con il dominio da controllare; $CD!2$ indica i fenomeni 2 generati dal dominio controllato (controlled domain) e scambiati con il dominio macchina; i fenomeni 3 indicano le regole che devono essere seguite dal comportamento del dominio controllato.

Come evidenziato nella Figura 5.4, per evadere un ordine il gestore degli ordini svolge le seguenti attività: a fronte della ricezione di un ordine inviato da un cliente, il gestore prepara l'ordine, invia la merce al cliente e chiude l'ordine. Ma quali attività devono essere svolte per preparare l'ordine e per spedire la merce? La merce è conservata nel magazzino e viene prelevata per mezzo di carrelli automatici che vengono pilotati a distanza dal Gestore ordini. I carrelli prelevano dal magazzino la merce e la depositano in una zona dedicata alle spedizioni. Quindi, per preparare e spedire la merce contenuta in un ordine, il gestore deve mandare dei comandi al carrello.

Per descrivere più in dettaglio questo aspetto del problema, può essere impiegato un commanded behaviour frame diagram come quello illustrato nella Figura 5.5. In tale diagramma il dominio macchina è costituito dal modulo di spedizione, che viene utiliz-

5.2 Commanded behaviour 95

**Figura 5.4** Attività svolte dal gestore ordini.

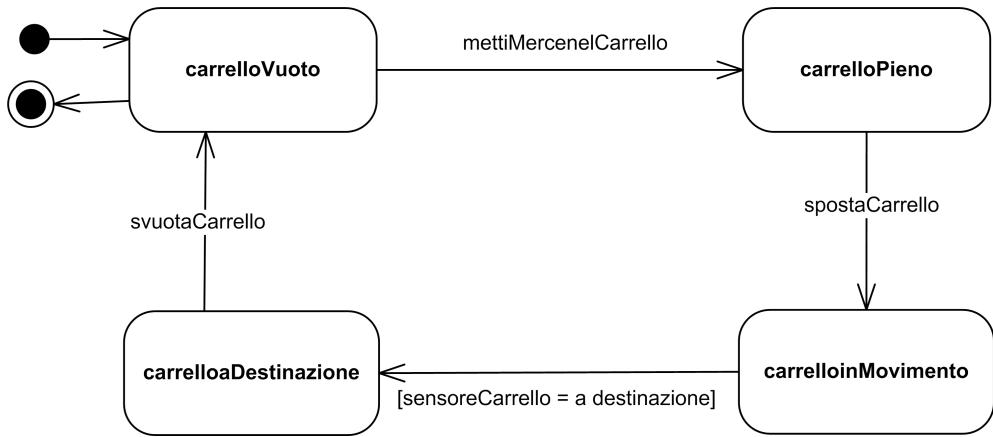
a: MS! {mettiMercenelCarrello, spostaCarrello, svuotaCarrello} [1]
C!{sensoreCarrello}[2]

b: C! {carrelloVuoto, carrelloPieno, carrelloinMovimento, carrelloaDestinazione} [3]

c: GO! {prelevaMerce, spedisciMerce} [4]

Figura 5.5 Commanded behaviour frame per il sistema di spedizione merce.

96 Capitolo 5 I problem frame

**Figura 5.6** State diagram del carrello.

zato dal gestore degli ordini per prelevare la merce di cui si compone l'ordine e per spedirla. Il carrello rappresenta il dominio che dev'essere controllato. Il modulo di spedizione, a fronte delle richieste provenienti dal gestore ordini, agisce pertanto sul carrello pilotandolo attraverso le operazioni di `mettiMercenelCarrello`, `spostaCarrello` e `svuotaCarrello` e recupera informazioni sullo stato del carrello attraverso l'interfaccia `sensoreCarrello` (contrassegnati dalla lettera a). I requisiti richiedono che, a fronte della ricezione di un ordine inviato da un cliente, il gestore prepari l'ordine, prelevando dal magazzino la merce richiesta, e predisponga l'invio della stessa. I comandi che il gestore ordini ha a disposizione per svolgere tali attività sono pertanto costituiti da `prelevaMerce` e `spedisciMerce` (contrassegnati dalla lettera c). I requisiti si riflettono anche sul dominio Carrello attraverso i fenomeni relativi allo stato del carrello (indicati con b): lo stato vuoto, pieno, in movimento o a destinazione traduce infatti la condizione di merce prelevata/in movimento/depositata.

Per chiarire ulteriormente i requisiti relativi al carrello può essere impiegato, tra gli altri, uno state diagram UML che mostri gli stati possibili e le transizioni da uno stato all'altro, come quello rappresentato nella Figura 5.6.

5.3 Information display

L'*information display frame* comprende tutti i casi in cui si manifesta l'esigenza di disporre di informazioni sullo stato e sul comportamento di una parte del mondo reale. Il problema software è relativo alla costruzione di una macchina che sia in grado di recuperare tali informazioni dal mondo reale e di presentarle secondo uno specifico formato richiesto. La Figura 5.7 illustra il diagramma corrispondente, che si compone dei seguenti elementi.

5.3 Information display 97

- *Information machine*: tale dominio indica la macchina (computer e software) da realizzare, che deve recuperare dal mondo reale determinate informazioni e presentarle in un apposito dominio.
- *Real world*: rappresenta il dominio al cui interno si trovano le informazioni richieste. Il dominio si caratterizza per essere attivo e autonomo, in quanto genera spontaneamente gli eventi, i fatti e gli stati cui si riferiscono le informazioni richieste.
- *Display*: indica il dominio all'interno del quale vengono rappresentate le informazioni richieste.
- *Display real world*: sono i requisiti del problema e stabiliscono una corrispondenza tra i fenomeni del mondo reale sui quali si vogliono avere informazioni e il formato con cui devono essere presentate.
- *Fenomeni condivisi*: i fenomeni 3 esprimono l'insieme delle informazioni che devono essere ricavate dal mondo reale; 4 indica l'insieme dei fenomeni con cui rappresentare le informazioni richieste; $RW!1$ indica l'insieme dei fenomeni che il mondo reale mette a disposizione sulla sua interfaccia verso il dominio macchina. Quest'ultimo deve pertanto identificare i fenomeni 3 a partire dai fenomeni 1. Infine $IM!2$ esprime l'insieme di eventi attraverso i quali il dominio macchina dovrà far visualizzare sul dominio *Display* le informazioni richieste.

Come esempio di applicazione dell'information display frame si consideri il caso di un frigorifero combinato, dotato cioè di un vano frigo e di un vano freezer. Esso dispone di un display sul quale dev'essere visualizzata la temperatura dei due vani. Se la temperatura supera i valori soglia, il display deve lampeggiare ed emettere un suono di allarme.

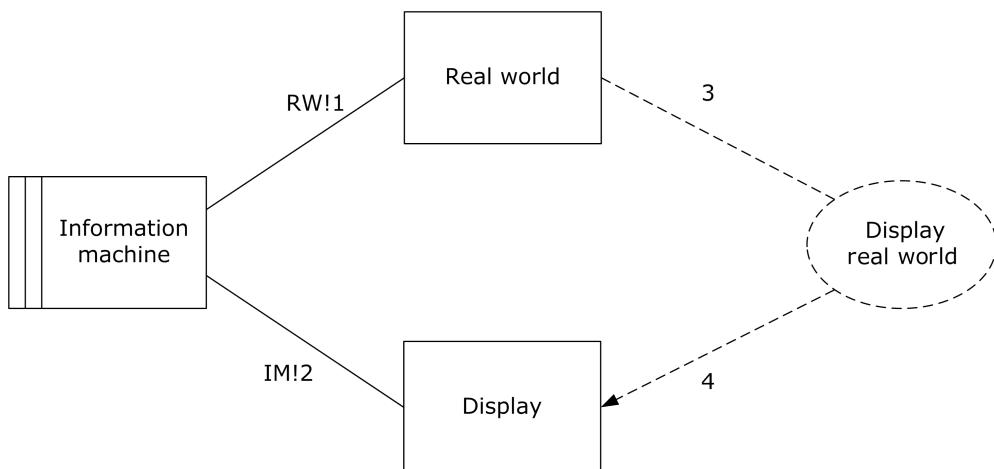


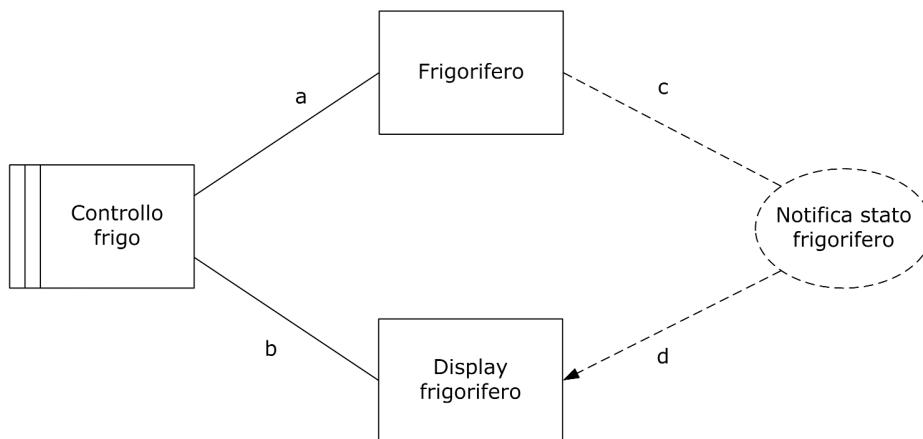
Figura 5.7 Information display frame.

98 Capitolo 5 I problem frame

L'allarme deve suonare anche nel caso in cui le porte del frigorifero rimangano aperte per più di cinque minuti.

La Figura 5.8 illustra una possibile descrizione del problema tramite l'uso di un information display frame: i requisiti relativi al display richiedono che su di esso vengano visualizzati i valori di temperatura dei due vani e che vengano emessi degli allarmi in caso le porte rimangano aperte o le temperature vadano fuori soglia (tali requisiti sono contrassegnati dalla lettera d). I requisiti relativi al dominio Frigorifero riguardano la temperatura dei due vani, nonché l'apertura delle porte dei due vani (indicati con c). Il dominio Controllo Frigo indica il dominio macchina. Esso accede al dominio Frigorifero attraverso i fenomeni (a) che quest'ultimo gli mette a disposizione: per ciascun vano sono costituiti dalle tensioni dei sensori di temperatura e dei sensori delle porte. Il dominio macchina visualizza sul display i valori di temperatura (char) e fa emettere il suono di allarme passando al display un valore booleano (value).

Per dettagliare ulteriormente la struttura del frigorifero, o meglio i componenti di esso interessati dal problema, può essere utile aggiungere all'information display frame il composite structure diagram UML raffigurato nella Figura 5.9: esso descrive il frigorifero in termini dei due vani, delle due porte a essi relativi e del display. Inoltre nella Figura 5.8 i requisiti sono indicati a livello generale dall'ovale tratteggiato: per delineare in modo più approfondito gli stati possibili del frigorifero e le regole da seguire per notificare gli allarmi sul display può essere affiancato all'information display frame un dia-



- a: F! {segnaliSensoriTemperatura, segnaliSensoriPorta} [1]
- b: CF! {char, value} [2]
- c: F! {temperaturaVanoSuperiore, temperaturaVanoInferiore, aperturaPortaFreezer, aperturaPortaFrigo, chiusuraPortaFreezer, chiusuraPortaFrigo} [3]
- d: DF! {valoriTemperaturaVani, allarmeFrigo, allarmeFreezer} [4]

Figura 5.8 Information display frame per un frigorifero.

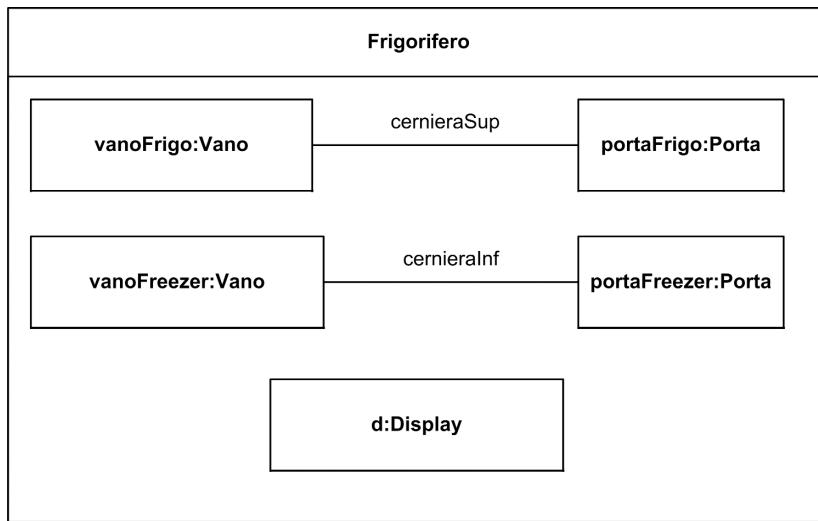


Figura 5.9 Composite structure diagram del sistema frigorifero.

gramma UML degli stati, come illustrato nella Figura 5.10. Tale diagramma indica gli stati in cui si possono trovare i tre sottosistemi frigo, freezer e display: in particolare i primi due, qualora rimangano nello stato aperto per più di cinque minuti, scatenano l'evento `alarmeFrigo` e `alarmeFreezer` sul sottosistema display.

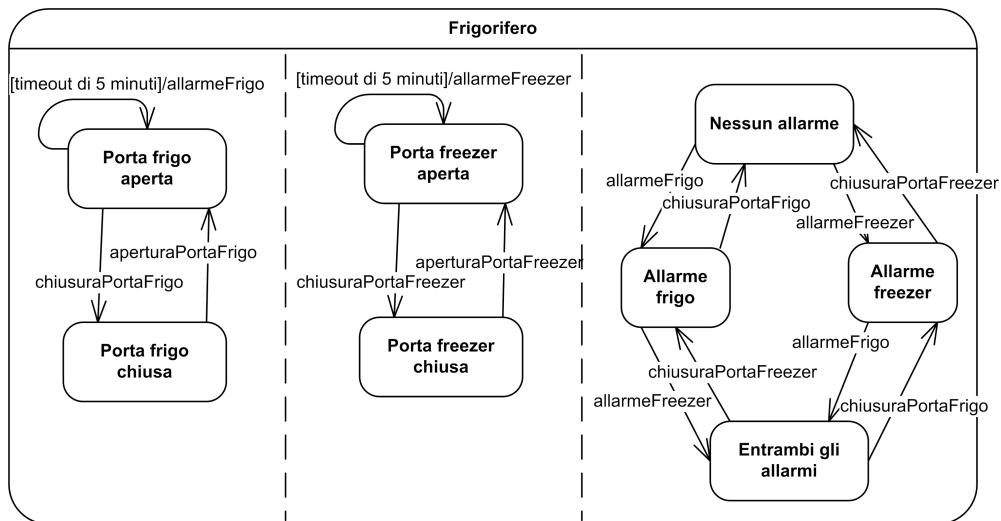


Figura 5.10 State diagram del sistema frigorifero.

5.4 Simple workpiece

Questo frame si riferisce a tutti quei problemi in cui è presente l'esigenza di fornire uno strumento che consenta a un utente di creare e modificare entità (testuali, grafiche ecc.), in modo tale che gli oggetti così creati possano poi essere successivamente manipolati (copiati, stampati, analizzati ecc.). Il problema software riguarda la costruzione di tale strumento. Il corrispondente diagramma, rappresentato nella Figura 5.11, è costituito dai seguenti elementi.

- *Editing tool*: tale dominio si riferisce alla macchina (computer e software) da realizzare, che dovrà svolgere delle operazioni sugli oggetti contenuti nel dominio Workpieces, sulla base di comandi eseguiti dall'utente.
- *Workpieces*: dominio che rappresenta le entità che possono essere create ed editate attraverso un computer, quali ad esempio file di testo, diagrammi, filmati.
- *User*: tale dominio si riferisce all'utente del tool. L'utente impedisce i comandi che il tool deve eseguire sugli oggetti contenuti nel dominio Workpieces.
- *Command effects*: sono i requisiti del problema. Stabiliscono, per ogni comando eseguito dall'utente, quali effetti debbano essere prodotti dall'editing tool sugli oggetti (relativamente ai loro valori e stati) che fanno parte del dominio Workpieces.
- *Fenomeni condivisi*: 3 si riferisce all'insieme completo di comandi (eventi) con cui l'utente può interagire con il tool; US!3 rappresenta l'interfaccia che l'utente condivide con il dominio macchina, ed è data dall'insieme di comandi 3, che sono controllati dall'utente; 4 indica l'insieme di valori e stati degli oggetti sui quali dovranno avere effetto i comandi 3; gli eventi ET!1 rappresentano le operazioni che sono

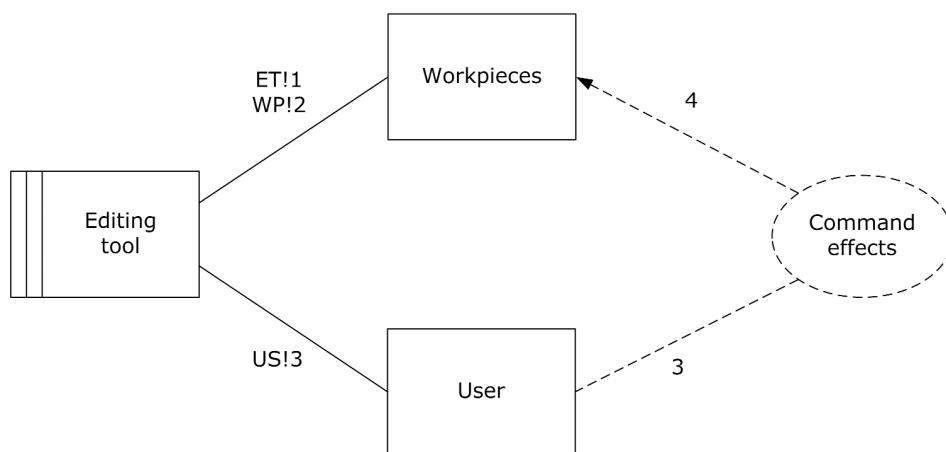


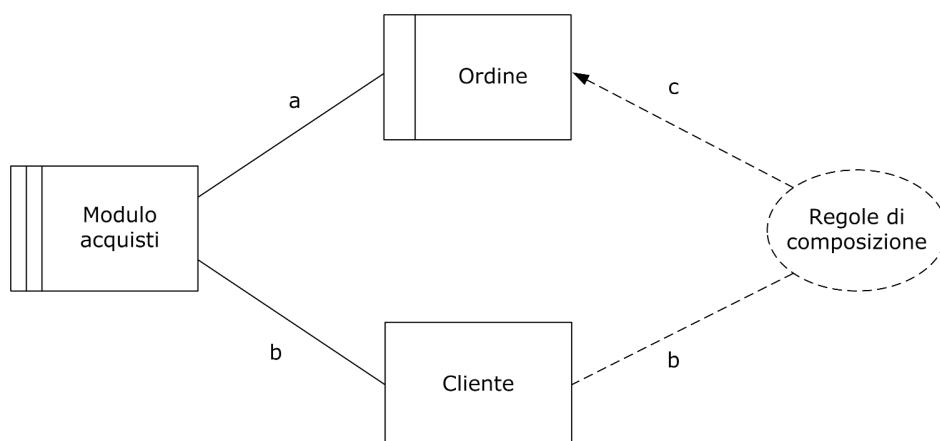
Figura 5.11 Simple workpiece frame.

5.4 Simple workpiece 101

eseguite dal dominio macchina sugli oggetti del dominio Workpieces; WP!2 sono i fenomeni che il dominio Workpieces espone come interfaccia verso il dominio macchina e attraverso i quali l'Editing tool accede ai valori e agli stati degli oggetti che fanno parte del dominio Workpieces.

Riprendendo l'esempio relativo al sistema di commercio elettronico si faccia riferimento a quella parte che deve consentire al cliente di comporre il proprio ordine. Per svolgere tale attività il cliente deve interagire con uno specifico modulo dedicato agli acquisti. Il problema può essere schematizzato attraverso il simple workpiece frame descritto nella Figura 5.12: i requisiti, che sono costituiti dalle regole di composizione, da un lato sono relativi al dominio Cliente e riguardano l'insieme di comandi che un cliente ha a disposizione per comporre un ordine di acquisto (contrassegnati dalla lettera b), dall'altro sono relativi al dominio Ordine e sono costituiti dagli effetti che si devono ottenere su un ordine in seguito ai comandi eseguiti dal cliente (contrassegnati dalla lettera c). Il Modulo acquisti rappresenta il dominio macchina: sulla base dei comandi ricevuti da parte del cliente esso consulta gli stati e i valori del dominio Ordine ed esegue sull'ordine delle operazioni (come indicato attraverso la lettera a).

Per descrivere in modo più approfondito il problema può essere utile affiancare al simple workpiece frame un class diagram UML, che illustri le funzionalità offerte dai va-



- a: MA! {operazioni} [1]
O! {stati e valori dell'ordine} [2]
- b: C! {comandi di composizione dell'ordine} [3]
- c: O! {effetti da produrre sull'ordine} [4]

Figura 5.12 Simple workpiece frame per il sistema di commercio elettronico.

102 Capitolo 5 I problem frame

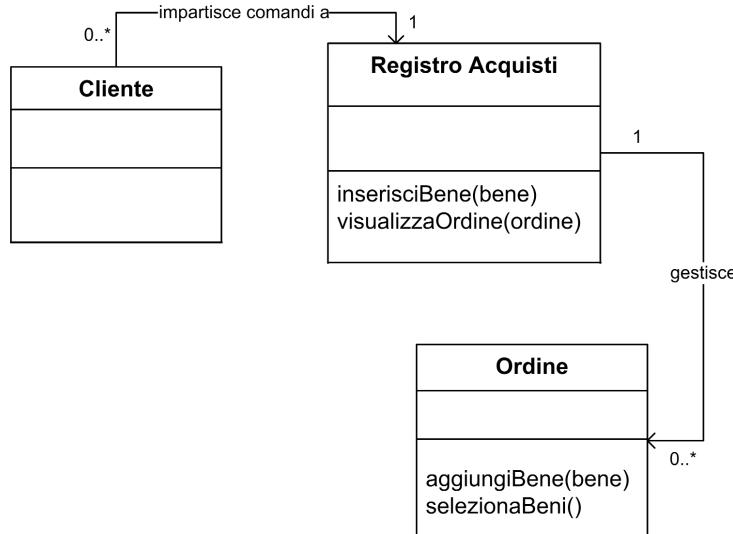


Figura 5.13 Descrizione statica della composizione di un ordine.

ri componenti e le relazioni tra di essi, e un sequence diagram UML, che mostri, per ogni comando eseguito dal cliente, quali operazioni devono essere svolte sull'Ordine da parte del Registro Acquisti. A titolo d'esempio si limiti la descrizione ai seguenti comandi: inserimento dei beni in un nuovo ordine e visualizzazione dei prodotti contenuti nell'ordine. Il class diagram è illustrato nella Figura 5.13, mentre la Figura 5.14 descrive dinamicamente le relazioni tra i comandi eseguiti dal cliente e le operazioni che il registro acquisti svolge sull'ordine.

5.5 Transformation

Il *transformation frame* esprime tutti quei casi in cui, avendo a disposizione alcuni dati di input in un determinato formato, si manifesta l'esigenza di trasformarli applicando determinate regole, per ottenere output che rispettino un formato richiesto. Il problema software è relativo alla costruzione di una macchina in grado di trasformare i dati di input che riceve nell'output richiesto. La Figura 5.15 illustra il frame diagram definito per descrivere questa classe di problemi. In particolare, esso si compone dei seguenti elementi.

- *Transformation machine*: si riferisce alla macchina (computer e software) da realizzare, che dovrà svolgere le trasformazioni per ottenere i dati di output a partire dai dati di input.
- *Inputs*: indica l'insieme degli input forniti, sui quali dovranno essere operate le trasformazioni.

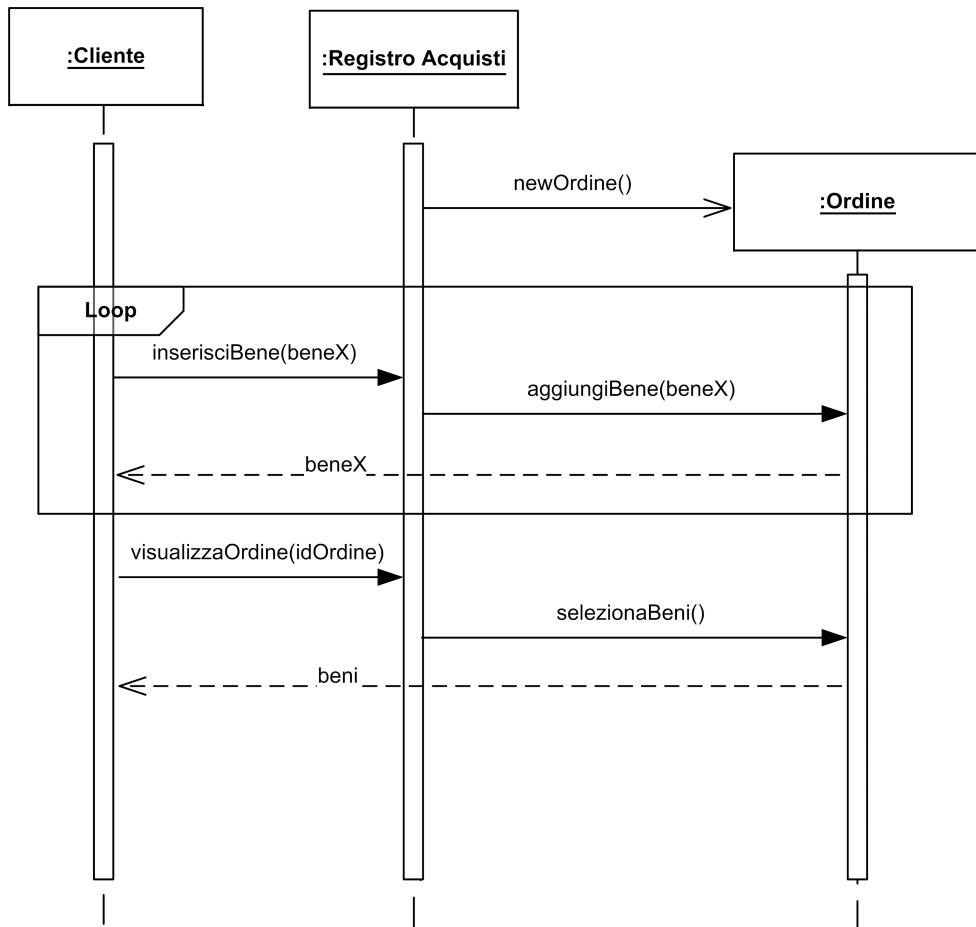
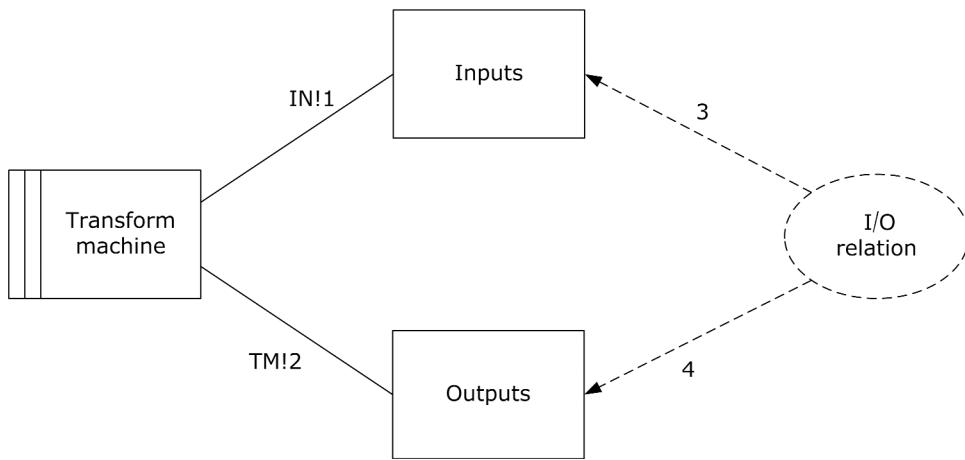


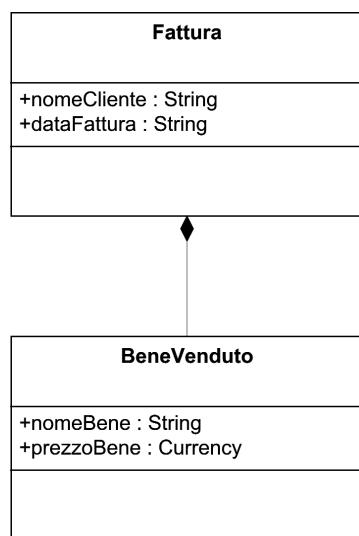
Figura 5.14 Sequence diagram relativo alla composizione di un ordine.

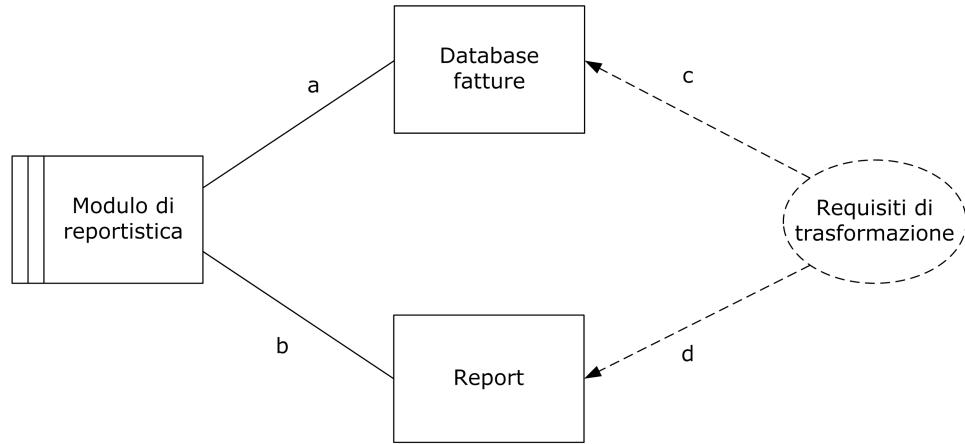
- *Outputs*: si riferisce all'insieme dei dati prodotti dalla Transformation machine.
- *I/O relation*: esprime i requisiti del sistema. I requisiti concernono le relazioni che devono esistere tra i fenomeni che costituiscono gli input e i fenomeni che caratterizzano gli output.
- *Fenomeni condivisi*: 3 e 4 esprimono i valori, i fatti, gli stati che caratterizzano, rispettivamente, i fenomeni di cui si compongono i domini di input e di output; $IN!1$ esprime l'insieme di fenomeni che costituiscono l'interfaccia offerta al dominio macchina dal dominio di input e da quest'ultimo controllati; analogamente $TM!2$ esprime l'insieme di fenomeni controllati dal dominio macchina e condivisi con il dominio di output.

104 Capitolo 5 I problem frame

**Figura 5.15** Transformation frame.

Il seguente esempio illustra un'applicazione del trasformation frame. Si supponga che nel sistema di commercio elettronico sia necessario produrre dei report mensili che, a partire dai dati contenuti nelle fatture, mostrino, per ciascun bene, il fatturato prodotto nel periodo di riferimento, la quantità venduta e il numero di clienti che l'hanno acquistato. Va precisato che le fatture vengono prodotte utilizzando il modulo di fatturazione aziendale già esistente che memorizza le fatture in un database secondo una precisa struttura. Le informazioni associate a ogni fattura sono indicate nel class diagram della Figura 5.16.

**Figura 5.16** Class diagram della fattura.



- a: DF! {databaseTable, tableView, tableField} [1]
- b: MR! {header, paragrafi, linee, caratteri} [2]
- c: DF! {cliente, data, beni, prezzo} [3]
- d: RI! {bene, quantitàBeneVenduta, numeroClienti, periododiRiferimento } [4]

Figura 5.17 Transformation frame per il sistema di commercio elettronico.

Pertanto le fatture e la relativa struttura con cui sono memorizzate nel database fanno parte del problema poiché costituiscono l'input a partire dal quale è necessario produrre un determinato output e costituiscono un dominio di tipo given.

La Figura 5.17 mostra il diagramma che si ottiene descrivendo il problema attraverso un transformation frame: il Modulo di reportistica consulta i valori delle fatture accedendo al database, alle tabelle che lo compongono, alle row e ai field delle tabelle (indicati con la lettera a). I fenomeni che il Modulo di reportistica scambia con il dominio Report (indicati con b) sono costituiti dagli elementi che compongono le varie linee del report, quali header, paragrafi, linee, caratteri. Per quanto riguarda i requisiti, i fenomeni del dominio Database fatture cui si rivolgono i requisiti sono indicati con la lettera c e sono costituiti dai valori delle varie fatture relativi al cliente, alla data di emissione, ai beni acquistati e al prezzo; i requisiti relativi al report, contrassegnati dalla lettera d, riguardano invece le informazioni che devono essere contenute nei report.

Per descrivere più in dettaglio i requisiti di trasformazione (nel nostro esempio, i requisiti del report) può essere introdotto un diagramma delle classi, come quello rappresentato nella Figura 5.18: ogni linea del report si riferisce a un bene diverso, e per ogni bene dev'essere indicato il fatturato complessivo che esso ha prodotto, la quantità complessiva venduta, il numero di clienti che ha acquistato il bene e il periodo di riferimento.

106 Capitolo 5 I problem frame

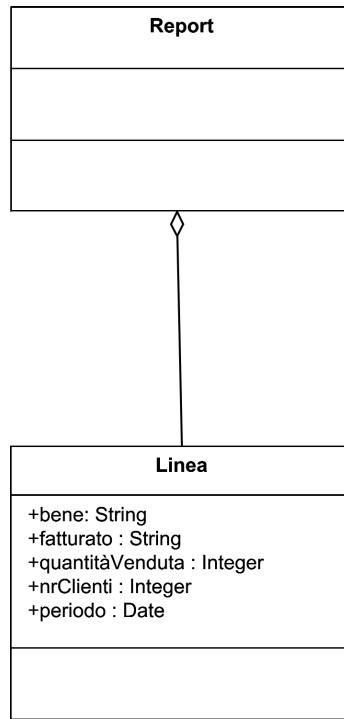


Figura 5.18 Descrizione del report tramite class diagram.

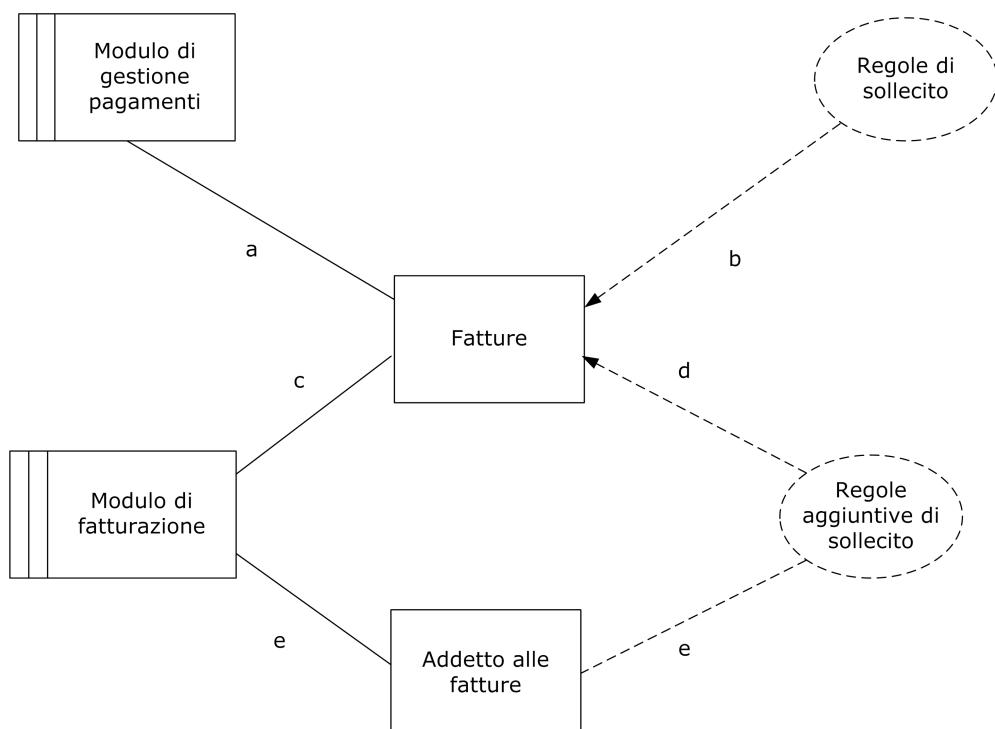
5.6 Combinazione di frame

Nei paragrafi precedenti si è visto come sia possibile scomporre un problema complesso in un insieme di sottoproblemi e come ogni sottoproblema possa essere delineato utilizzando i problem frame. Negli esempi citati ciascun sottoproblema è stato descritto utilizzando uno dei cinque problem frame elementari. In realtà i problemi del mondo reale che devono essere affrontati presentano una struttura la cui complessità fa sì che non sia sempre possibile riuscire a raffigurare ogni sottoproblema ricorrendo a un unico problem frame: quello che capita più spesso è che ogni sottoproblema venga descritto attraverso una struttura mista, basata cioè su una combinazione di problem frame elementari (*composite frame*). Questo accade ad esempio quando per esprimere in modo esauriente un certo requisito è necessario far riferimento a due sottoproblemi diversi, in quanto essi interagiscono l'uno con l'altro. Si riprenda l'esempio relativo all'invio dei solleciti per le fatture emesse e si supponga che i requisiti prevedano che l'addetto alle fatture possa decidere di inviare un sollecito anche prima che sia trascorso un mese dall'emissione di

5.6 Combinazione di frame 107

una fattura non ancora pagata, oppure di non inviarlo mai. Il problema si compone pertanto del required behaviour frame al quale si aggiunge un commanded behaviour frame per descrivere il comportamento del sistema legato ai comandi impartiti dall'addetto alle fatture. La descrizione che si ottiene è quella rappresentata nella Figura 5.19.

In generale, la descrizione di un problema complesso si otterrà scomponendolo in un numero di sottoproblemi, ciascuno dei quali più o meno direttamente riconducibile ad uno dei problem frame discussi in precedenza. In realtà, i problem frame sono uti-



- a: MGP!{inviaSollecito}
F!{fatturaEmessa, fatturaPagata, fatturainSollecito}
- b: F!{sollecito}
- c: MF!{inviaSollecito, sospendiSollecito}
F!{fatturaEmessa, fatturaPagata, fatturainSollecito}
- d: F!{sollecito, sollecitoSospeso}
- e: AF! {inviaSollecito, sospendiSollecito}

Figura 5.19 Esempio di composite frame.

li solo per comprendere la natura del (sotto)problema e le sue principali caratteristiche. Per giungere ad una descrizione dettagliata del problema, ai problem frame è necessario affiancare un numero cospicuo di descrizioni che ne specifichino peculiarità e proprietà. Se il linguaggio di descrizione utilizzato è UML, l'ingegnere del software dispone di diversi tipi di diagrammi. Tipicamente, si utilizzano uno o più diagrammi di tipo use case, class, sequence, activity, state. Nel caso si debba descrivere come parte del problema anche un sistema informatico esistente, sono anche particolarmente utili i component e deployment diagram. L'Appendice fornisce ulteriori esempi relativi a queste problematiche.

5.7 Riferimenti bibliografici

L'approccio di Jackson è stato (ed è) studiato e approfondito in altri studi come, ad esempio, Laney et al. [2004]. Inoltre, altri autori hanno cercato di rappresentare e studiare schemi per lo spazio della soluzione, secondo approcci alternativi. Un lavoro molto noto e che costituisce tuttora un contributo particolarmente interessante è Fowler [1996].

Capitolo 6

Progettare la soluzione

Il principale compito dell'ingegnere del software è quello di sviluppare e/o assemblare il codice che costituisce la soluzione al problema dell'utente. Come discusso estensivamente nel Capitolo 1, costruire una soluzione informatica non banale richiede necessariamente un'accurata fase di progettazione della struttura e organizzazione del software.

La progettazione del software è tradizionalmente articolata in due fasi: *progettazione architettonica* e *progettazione di dettaglio*. La fase di progettazione architettonica identifica i principali elementi di una soluzione informatica e lo schema di massima secondo il quale tali elementi interagiscono ed evolvono nel tempo. Per esempio, la fase di progettazione architettonica potrebbe portare alla scelta di un'architettura client-server (introdotta brevemente nel Capitolo 1 e discussa nei particolari nel seguito). La fase di progettazione di dettaglio scomponete gli elementi dell'architettura prescelta nei moduli software veri e propri (classi, tipi di dati astratti ecc.).

In realtà, l'attività di progettazione segue percorsi e fasi diversi in funzione del ciclo di vita e del processo di sviluppo utilizzato. Come discusso nel Capitolo 8, vi sono approcci che definiscono la soluzione partendo da componenti preesistenti attraverso quello che viene chiamato *processo di sviluppo per componenti*. In altri approcci evolutivi, all'inizio del processo l'architettura software è definita solo nelle sue caratteristiche essenziali, con attività di progettazione di dettaglio e di sviluppo vere e proprie che si alternano portando a uno sviluppo incrementale della soluzione. Una stretta sequenzialità tra progettazione architettonica e di dettaglio è tipica del *ciclo di vita a cascata*, che, seppur ancora molto diffuso e certamente conveniente per specifiche tipologie di progetti, ha mostrato limiti concettuali e pratici tali da renderlo sconsigliabile come modello universale. Ciononostante, la distinzione tra i due tipi di progettazione è molto utile per caratterizzare le attività del progettista di software. Infatti, indipendentemente dal ciclo di vita prescelto e dall'articolazione temporale delle attività svolte, l'ingegnere del software dovrà effettuare scelte e valutazioni a livello di architettura piuttosto che di singolo elemento funzionale.

Per procedere all'analisi dei passi e delle attività nelle quali si articolano la progettazione architettonica e di dettaglio, è innanzitutto necessario chiarire in quale modo la struttura di un sistema informatico può essere caratterizzata e descritta. Prima ancora di scegliere uno specifico linguaggio di descrizione, è necessario *capire cosa si vuole descrivere*.

re. Analogamente a quanto accade nel caso dello studio del problema, compito del progettista non è semplicemente quello di conoscere bene UML (o qualunque altro linguaggio di descrizione): egli deve saper valutare, scegliere e costruire una soluzione informatica efficiente, sapendola poi descrivere in modo appropriato nel linguaggio più adatto. Si tratta quindi di spostare il punto di vista dalle “technicalities” del linguaggio di descrizione utilizzato, allo studio delle informazioni che si vogliono descrivere. Il procedimento è analogo a quello di uno scrittore che pur dovendo certamente conoscere bene grammatica e sintassi della lingua nella quale vuole scrivere un romanzo, deve concentrarsi prima di tutto sulla trama, la scelta dei personaggi, lo sviluppo della storia. In questo, la lingua scritta ha un ruolo strumentale: una sua conoscenza approfondita non è sufficiente per bilanciare una scarsa o insufficiente vena creativa dello scrittore.

Per questi motivi, così come nel caso dello studio del problema, scopo principale di questo capitolo è illustrare la natura e la tipologia delle informazioni che dovranno essere raccolte, sintetizzate e descritte al fine di giungere alla completa caratterizzazione architettonale della soluzione software. Il Capitolo 1 ha già introdotto una serie di concetti e “mattoni” di base. Per classificare e organizzare in modo coerente e strutturato le informazioni che identificano una soluzione informatica complessa, è necessario ora introdurre il concetto di *viste architetturali (architectural view)*, cioè i differenti punti di vista secondo i quali è possibile (e necessario) studiare la struttura di una soluzione informatica.

6.1 Viste architetturali

Una soluzione informatica può essere studiata, caratterizzata e descritta da (almeno) quattro punti di vista.

- Struttura logico-funzionale (*logical-functional view*).
- Organizzazione del codice (*module view*).
- Distribuzione del software (*deployment view*).
- Struttura di esecuzione (*run-time view*).

Queste diverse viste sono tra loro complementari e forniscono nel loro insieme una descrizione esaustiva di un sistema software complesso. Le viste saranno ora presentate utilizzando un esempio di applicazione client-server (Paragrafo 7.1) che realizza un semplice servizio di commercio elettronico (acquisto online di beni come libri o dischi). L'esempio è volutamente semplificato al fine di mettere in luce gli aspetti di metodo e di processo. Si ipotizza la presenza di un programma (client) eseguito sulla macchina dell'utente. Il client interagisce con il sistema che gestisce il catalogo dei beni acquistabili e le procedure di acquisto e spedizione della merce.

Si noti che l'ordine di presentazione delle diverse viste non vuole in nessun modo suggerire che nella costruzione della descrizione della soluzione si deve procedere sempre e rigidamente secondo la sequenza proposta: è possibile che vi siano diverse iterazioni e ricicli non solo tra progettazione e altre fasi, ma anche nell'elaborare le diverse viste architetturali.

6.1.1 Logical-functional view

La *logical-functional view* mette in evidenza la struttura complessiva di un sistema informatico, i suoi elementi principali e le modalità secondo le quali sono interconnessi e interagiscono.

Nell'esempio prescelto, si suppone che l'applicazione di commercio elettronico sia basata su un'architettura client-server. Il client offre i servizi all'utente finale ed è installato sul suo personal computer. Esso è connesso tramite Internet con il sistema informatico che implementa le procedure di acquisto vere e proprie e gestisce l'accesso alla base di dati contenente tutti i dati relativi ai prodotti in vendita e agli utenti. Queste informazioni possono essere sintetizzate attraverso un component diagram UML come quello della Figura 6.1.

Il diagramma contiene tre componenti caratterizzati con lo stereotipo `<<subsystem>>`. Questa descrizione dice già molto del sistema in oggetto. In particolare, il diagramma indica che il sistema informatico è composto da tre elementi principali.

- Client.
- ApplicationLogic, cioè l'elemento che implementa le procedure relative agli acquisti.
- DatabaseManagement, cioè il gestore della base di dati (l'archivio) dove sono memorizzati i dati.

Il diagramma è ancora largamente insufficiente, tuttavia, per rappresentare la logica complessiva di funzionamento del sistema. Non dice quali sono, almeno da un punto di vista generale, le funzioni e come sono invocate. Ciò può essere rappresentato utilizzando un semplice activity diagram come quello della Figura 6.2. Il Client riceve le richieste dell'utente (`Get User Request`) e richiede ad ApplicationLogic l'esecuzione di una qualche operazione come, ad esempio, l'acquisto di una copia di un libro. Tale richiesta viene analizzata (`Analyse Request`) e processata inviando tutte le informazioni necessarie a DatabaseManagement. Questo accede alla base di dati (`Access Database`) e restituisce i dati ad ApplicationLogic che, a sua volta, confeziona la risposta per il client (`Prepare Results`). Non appena Client riceve i risultati, li mostra all'utente (`Display Results`) e torna a essere disponibile per nuove richieste. Il diagramma mostra che il sistema opera in modo totalmente sincrono: ciascun elemento attende la fine di una richiesta prima di procedere con altre operazioni. ApplicationLogic e DatabaseManagement sono quindi *due server* che rispondono contemporaneamente alle richieste che vengono loro inviate: da Client verso ApplicationLogic e da questo a DatabaseManagement.

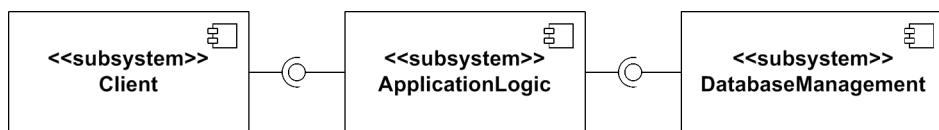


Figura 6.1 Struttura complessiva dell'applicazione.

112 Capitolo 6 Progettare la soluzione

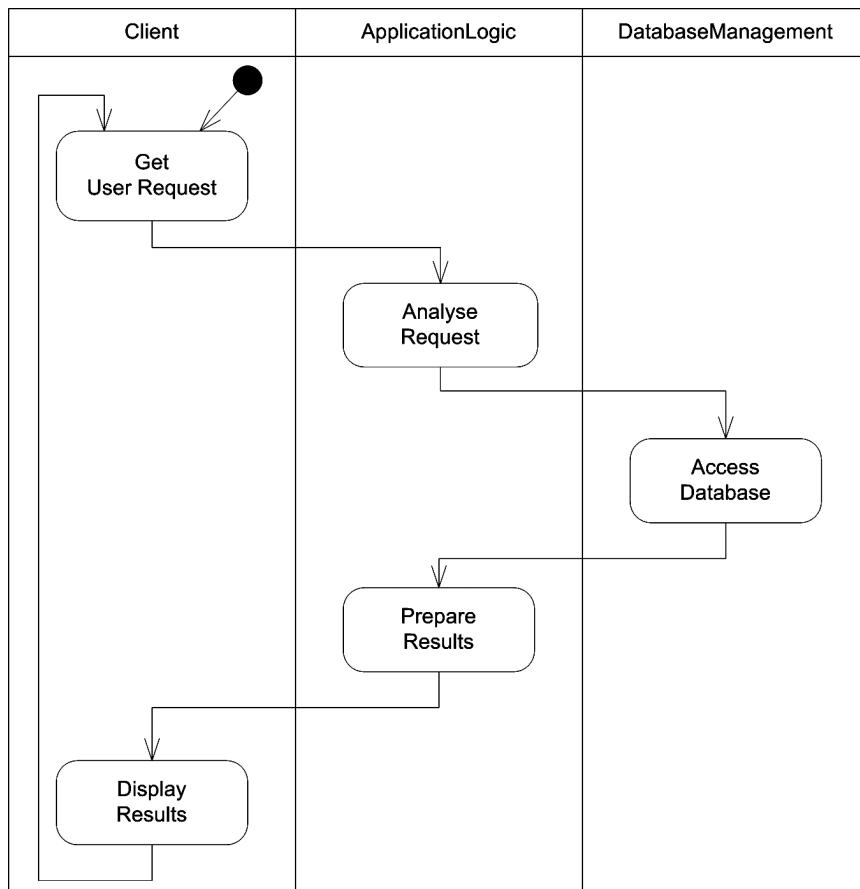


Figura 6.2 Funzionamento di massima dell'applicazione.

Un'informazione che non traspare dai diagrammi presentati in precedenza concerne la reale distribuzione fisica dei diversi elementi dell'architettura. In particolare, se è abbastanza ovvio che il client risiede sulla macchina dell'utente, non è per nulla chiaro dove siano collocati i due server ApplicationLogic e DatabaseManagement. Ci sono due possibilità: 1) i due server risiedono sulla stessa macchina; 2) ciascun server risiede su una macchina diversa. Ancora più importante, requisiti di scalabilità o di affidabilità potrebbero richiedere la replicazione di uno o di entrambi i server. Per esempio, potrebbero esistere più server applicativi tutti equivalenti tra loro e sui quali sono uniformemente distribuite le richieste che giungono dai client. Questo tipo di informazioni sono per lo più collegate all'aspetto implementativo vero e proprio ed è quindi tipico della *deployment view* che sarà esaminata nel seguito. A livello logico-funzionale, tuttavia, è possibile che la replicazione introduca un elemento funzionale importante: il front-end che gestisce il

6.1 Viste architetturali 113

reindirizzamento di una richiesta verso uno dei server applicativi. Il front-end potrebbe essere omesso a questo livello di descrizione, in quanto elemento che non contribuisce alle funzionalità vere e proprie dell'applicazione, ma solo a requisiti di tipo non funzionale. In realtà, a livello di prima descrizione complessiva dell'applicazione, è importante evidenziare fin da subito il fatto che sono previsti meccanismi di replicazione dei server.

La Figura 6.3 introduce il front-end nel component diagram. Il diagramma indica che ogni server `ApplicationLogic` si registra presso il front-end (utilizzando un servizio da esso offerto). La Figura 6.4 illustra le modalità secondo le quali il front-end interagisce con gli altri elementi dell'architettura.

L'introduzione del front-end evidenzia una nuova esigenza. Come rappresentare il fatto che `ApplicationLogic` esiste in istanze multiple? Esistono più istanze anche di `DatabaseManagement`? Si ipotizzi di voler rappresentare la seguente situazione:

- ogni client interagisce con un unico front-end;
- il front-end mette in contatto il client con un server `ApplicationLogic` scelto tra quelli disponibili;
- esiste un unico server `DatabaseManagement`.

Tale situazione può essere descritta tramite il component diagram della Figura 6.5.

Nel diagramma, i componenti sono correlati non attraverso le loro interfacce (come nei precedenti diagrammi), quanto attraverso associazioni che esplicitano le relazioni e le molteplicità tra componenti. L'associazione `first access` indica che esiste un front-end al quale effettuano un primo accesso zero o più client. Il front-end (l'unico presente) seleziona uno tra i server `ApplicationLogic` presenti. Ogni server `ApplicationLogic` interagisce con uno o più client che il front-end gli ha rediretto e con l'unico server `DatabaseManagement` presente. Il client interagisce con il server `ApplicationLogic` che gli viene indicato dal front-end.

Un'altra informazione importante per capire la logica di funzionamento complessivo dell'applicazione concerne il criterio secondo il quale i due server reagiscono alle ri-

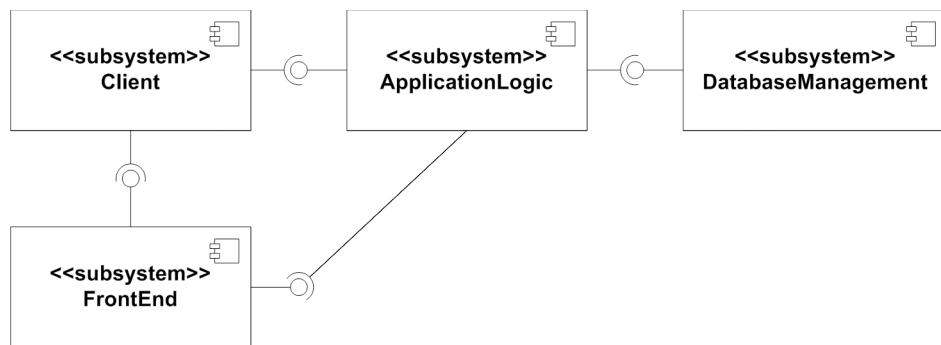
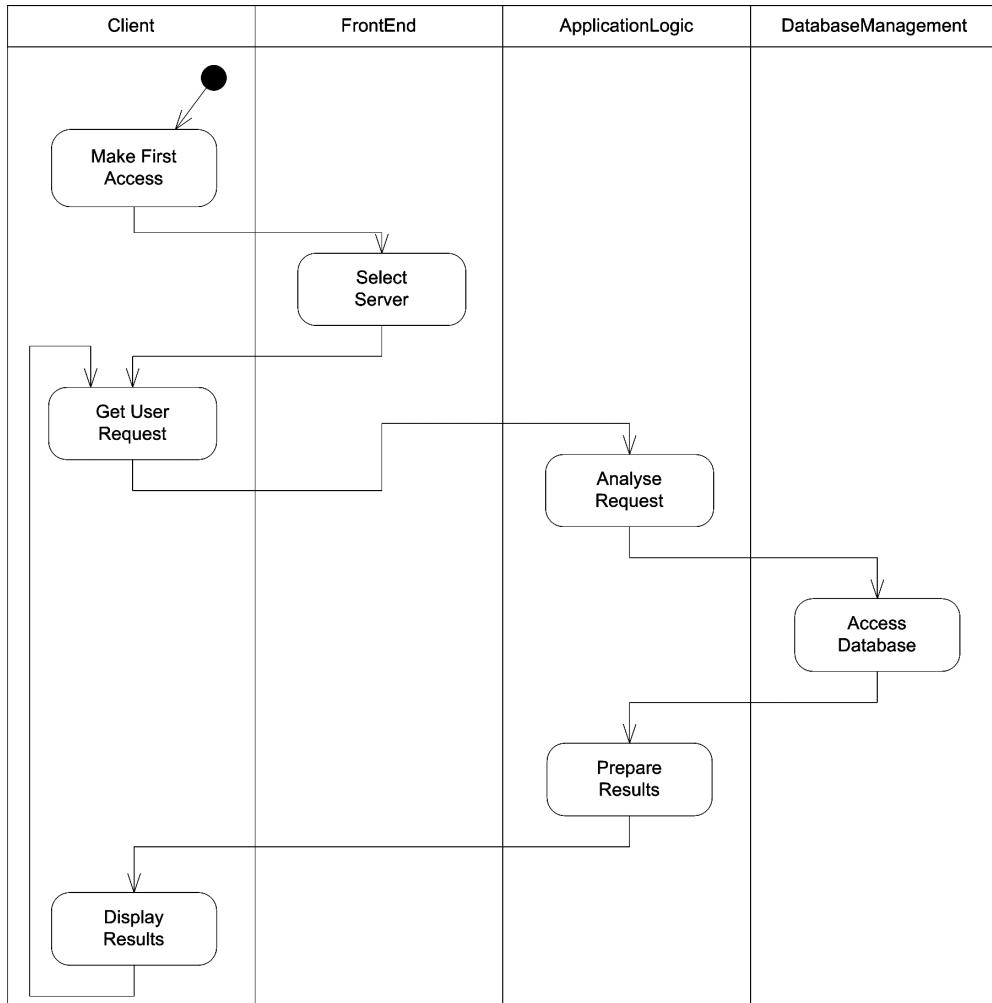


Figura 6.3 Architettura dell'applicazione con il front-end.

114 Capitolo 6 Progettare la soluzione

**Figura 6.4** Activity diagram con il front-end.

chieste di servizio. In generale, è noto che un server può essere single-thread o multi-thread. La distinzione ha una notevole rilevanza: cambiano sia le regole di gestione delle operazioni relative a diversi client (concorrenza) sia l'impatto sulle prestazioni e la scalabilità complessiva del sistema. Nel caso di un sistema single-thread, il server completa una richiesta prima di passare alla successiva. Nel caso multi-thread, il server attiva un thread per ciascuna richiesta (o anche per ciascun client). Queste due alternative devono essere prese in considerazione nell'ambito dell'*execution view* e, a livello di codice sorgente, nella *module view*.

6.1 Viste architettoniche 115

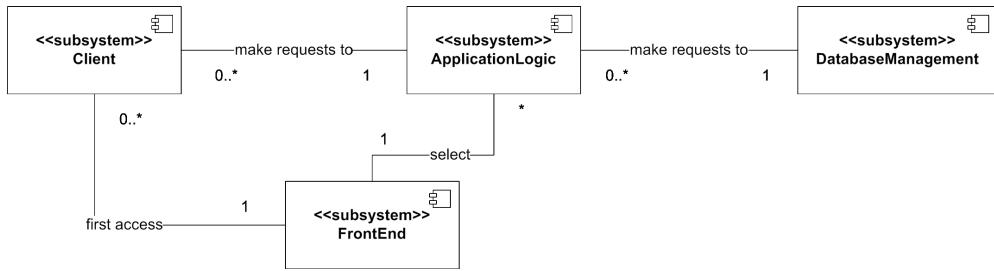


Figura 6.5 Numerosità degli elementi essenziali dell'applicazione.

6.1.2 Module view

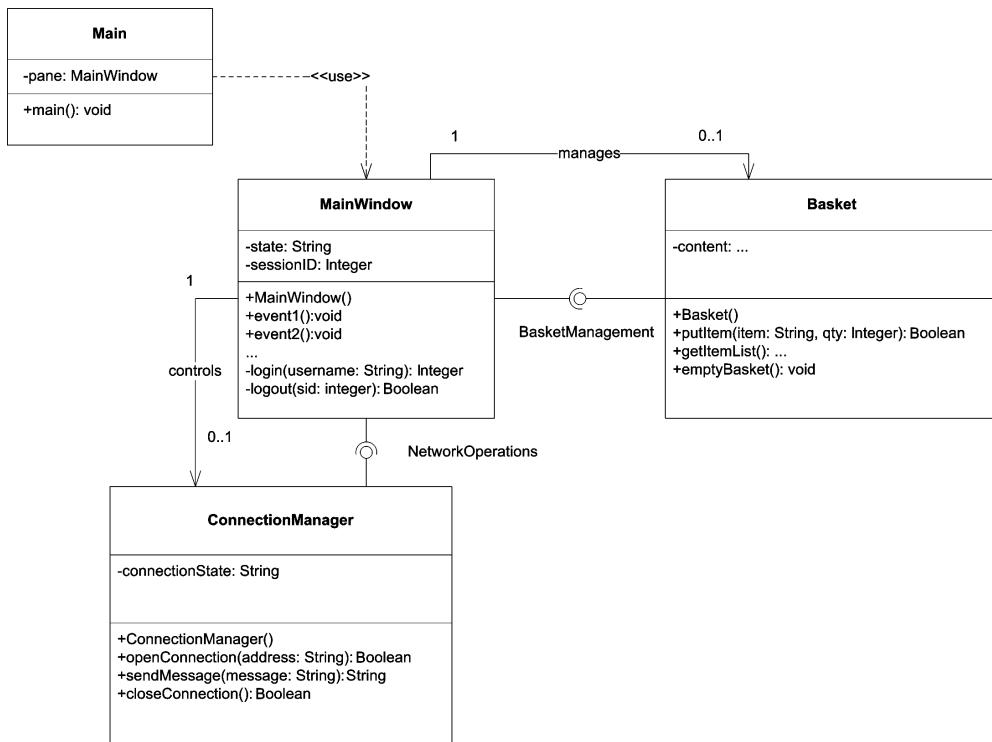
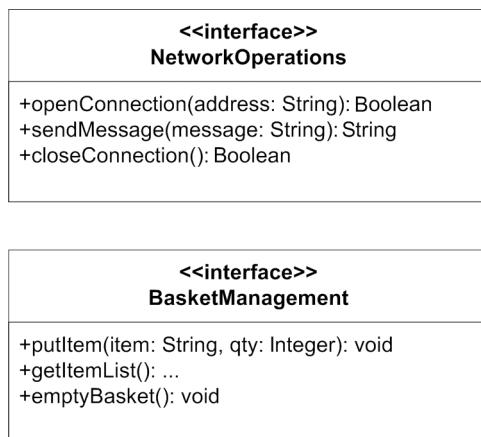
La *module view* illustra la struttura vera e propria del codice sorgente. Se per esempio si sta utilizzando un linguaggio come Java, nella module view sono descritte le classi che costituiscono il programma, come interagiscono e come si combinano per dare origine agli elementi dell'architettura descritta nella logical-functional view. In termini generali, se la logical-funzional view rappresenta un primo risultato della progettazione architettonica, la module view rappresenta la progettazione di dettaglio, anche se questa caratterizzazione è puramente indicativa e non deve essere interpretata in modo rigido.

Per descrivere i moduli presenti in un'applicazione informatica, è necessario innanzi tutto identificare le *caratteristiche statiche*: la loro identità, le informazioni che conservano e le funzionalità che offrono. Queste informazioni possono essere efficacemente rappresentate da un diagramma delle classi. Per esempio la Figura 6.6 illustra il diagramma delle classi utilizzate per il client dell'esempio discusso in precedenza.

Il diagramma delle classi contiene molte informazioni importanti. La classe `Main` contiene il programma principale dell'applicazione (metodo `main`) che crea il pannello (la finestra) principale (`pane:MainWindow`) attraverso il quale verrà gestita l'interazione con l'utente. Il pannello è un'istanza della classe `MainWindow`. Tale classe offre una serie di funzioni pubbliche che sono invocate a fronte di eventi generati dal sistema operativo. In particolare, quando l'utente preme un pulsante o attiva una voce di un menù, viene invocato il metodo di `MainWindow` che è stato associato a tale evento (metodi `eventn()`), permettendo quindi al programma di decidere come reagire alla richiesta dell'utente. `MainWindow` contiene un “cesto della spesa” (`Basket`) che viene gestito attraverso le funzioni dell'interfaccia `BasketManagement`. Il cesto della spesa permette di memorizzare i prodotti da acquistare. `MainWindow` utilizza anche la classe `ConnectionManager` che offre l'interfaccia `NetworkOperations`. Tale interfaccia permette di inviare messaggi al server per richiedere l'esecuzione di operazioni di accesso alla base di dati. La descrizione delle interfacce è illustrata nella Figura 6.7.

Il diagramma delle classi della Figura 6.6 contiene un'altra informazione molto importante. Attraverso le associazioni `controls` e `manages` si indica che `MainWindow` può accedere (“navigare”) agli oggetti istanze di `ConnectionManager` e di `Basket`, ma non viceversa. Inoltre, attraverso la molteplicità delle associazioni, si indica che `MainWindow`

116 Capitolo 6 Progettare la soluzione

**Figura 6.6** Classi che costituiscono il client.**Figura 6.7** Interfacce di classi.

6.1 Viste architetturali 117

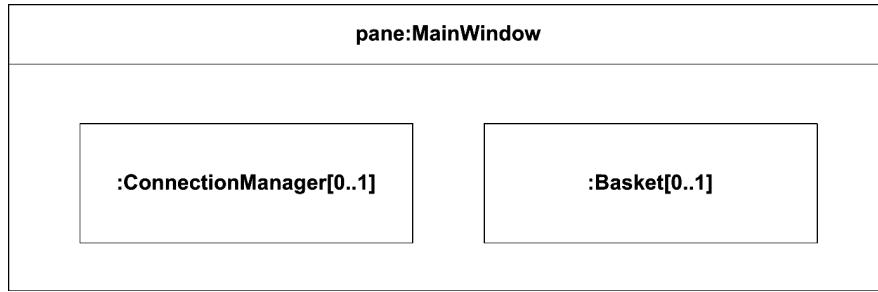


Figura 6.8 Composite structure diagram.

contiene al più un oggetto `ConnectionManager` e al più un oggetto `Basket`. Quest'informazione, che descrive come le classi sono instanziate in oggetti, può essere descritta molto efficacemente utilizzando un *composite structure diagram* come quello della Figura 6.8. Il diagramma rappresenta il fatto che esiste un oggetto `pane`, istanza di `MainWindow` (istanziato nella classe `Main`), che contiene le due istanze di `ConnectionManager` e `Basket`.

Il class diagram che descrive il server `ApplicationLogic` è illustrato nella Figura 6.9. Il server include un `Handler` che riceve i messaggi del client e invoca le funzioni corrispondenti della classe `Application`. Quest'ultima implementa la logica applicativa vera e propria del sistema, per esempio fornendo funzioni per aggiungere prodotti al basket ed effettuare il check-out finale. I metodi della classe `Application` utilizzano il linguag-

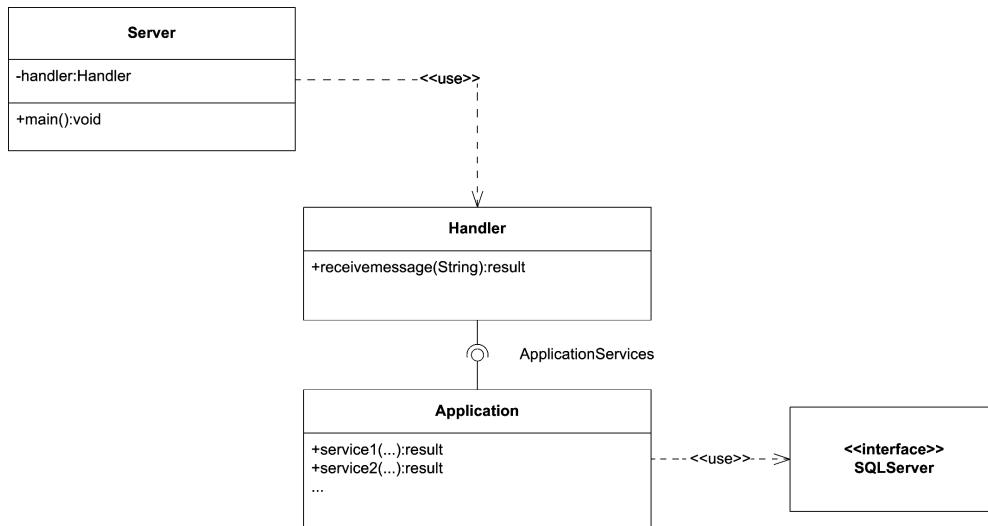


Figura 6.9 Class diagram del server.

118 Capitolo 6 Progettare la soluzione

gio SQL per accedere a una base di dati relazionale. Ciò è rappresentato attraverso la relazione di dipendenza <<use>> che connette Application con l'interfaccia SQLServer.

I due diagrammi delle classi visti finora non esplcitano le modalità secondo le quali il client è in grado di accedere ai servizi del server. Questo aspetto può essere descritto definendo un'interfaccia HandlerInterface esportata dalla classe Handler e utilizzata dalla classe ConnectionManager del client. In effetti, includendo questo tipo di informazioni diviene possibile completare la functional-logical view dell'applicazione introdotta nella Figura 6.1. Ogni componente viene descritto non solo dal proprio nome, ma anche dall'indicazione delle principali classi e interfacce che lo compongono. Il component diagram completo è quello della Figura 6.10. Questo esempio evidenzia il fatto che

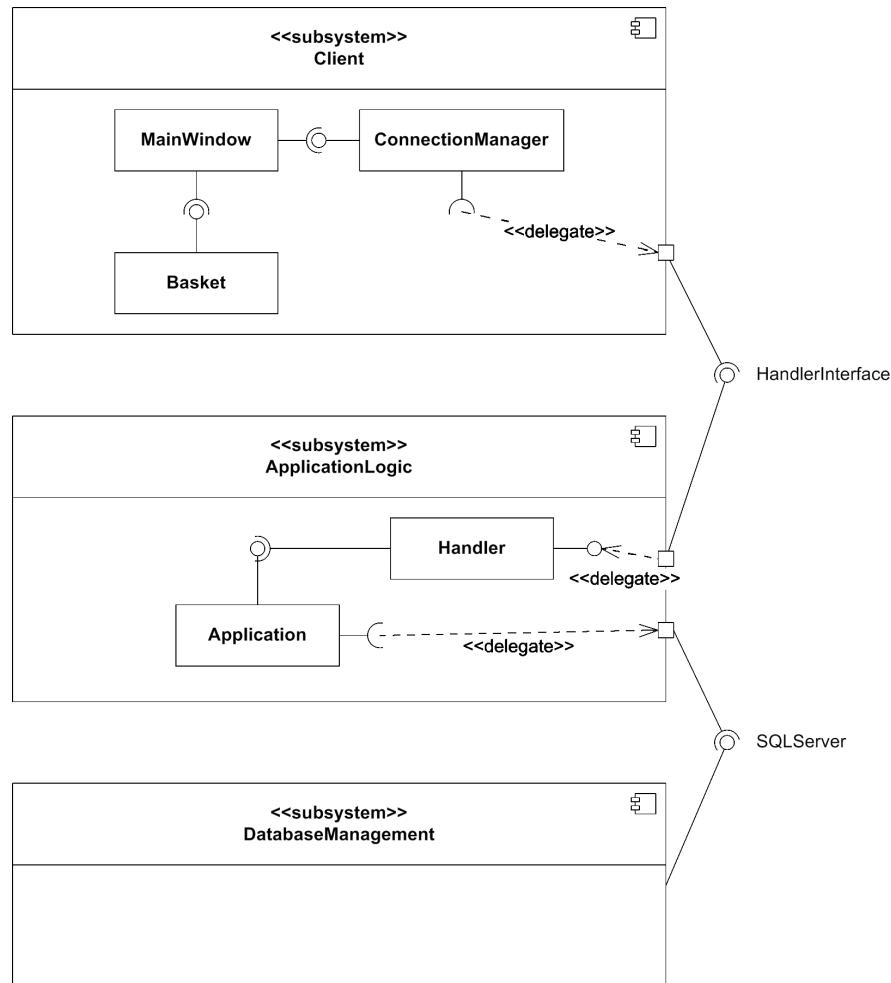


Figura 6.10 Component diagram completo.

6.1 Viste architettoniche 119

la produzione delle descrizioni relative alle diverse view non deve necessariamente seguire una rigida struttura temporale. Anzi, è attraverso i ricicli e le iterazioni tra diverse view che le descrizioni si arricchiscono di nuove e utili informazioni.

I diagrammi illustrati in precedenza mostrano la struttura interna del client da un punto di vista statico; tuttavia non dicono nulla della dinamica, cioè di come e quando gli oggetti vengono creati e di quali siano le loro interazioni. In sintesi, non viene realmente specificato cosa fanno (o dovranno fare) le istruzioni del codice presente nelle classi. La dinamica di un insieme di classi può essere descritta in diversi modi, utilizzando le notazioni presenti in UML. In particolare, il *sequence diagram* si combina in modo molto intuitivo con i diagrammi delle classi visti finora. In particolare, il sequence diagram della Figura 6.11 illustra la sequenza di operazioni eseguite a fronte di una richiesta dell'utente. I metodi invocati sono quelli offerti dagli oggetti istanziati dalle classi che costituiscono i diversi elementi del sistema. Nella costruzione di questo diagramma occorre assicurare la massima coerenza con gli altri diagrammi già introdotti, per esempio, verificando che oggetti e metodi relativi siano gli stessi presenti nei class diagram e nel composite structure diagram.

Il diagramma ora introdotto in realtà descrive solo una delle operazioni che l'utente è in grado di invocare. In generale, il progettista dovrà costruire tanti diagrammi quante sono le operazioni invocabili dall'utente. In questo senso, esiste un collegamento diretto tra questi diagrammi e quelli sviluppati in fase di descrizione del problema, laddove si è descritta l'interfaccia utente, cioè le modalità secondo le quali l'utente può interagire con il sistema informatico. In particolare, gli use case diagram e i sequence diagram prodotti in fasi di descrizione dell'interfaccia devono essere analizzati per determinare quali diagrammi sviluppare in fase di progettazione.

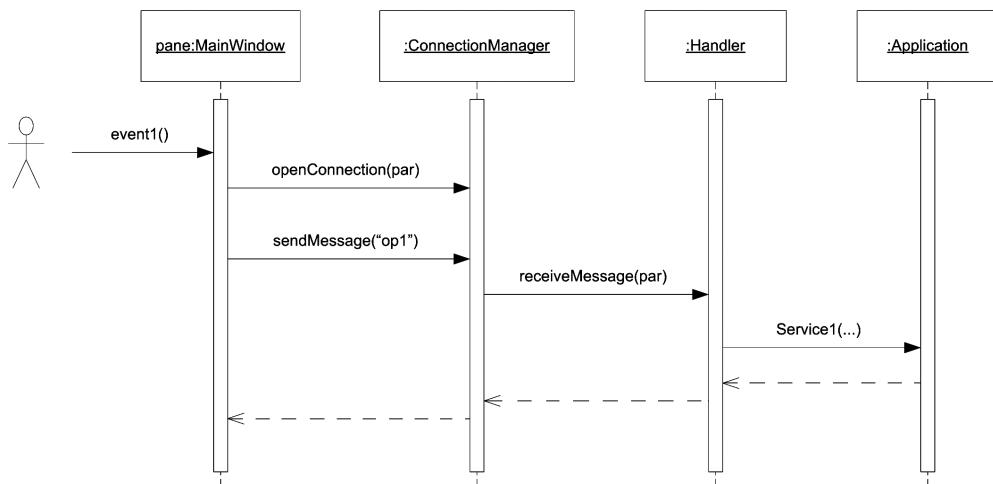
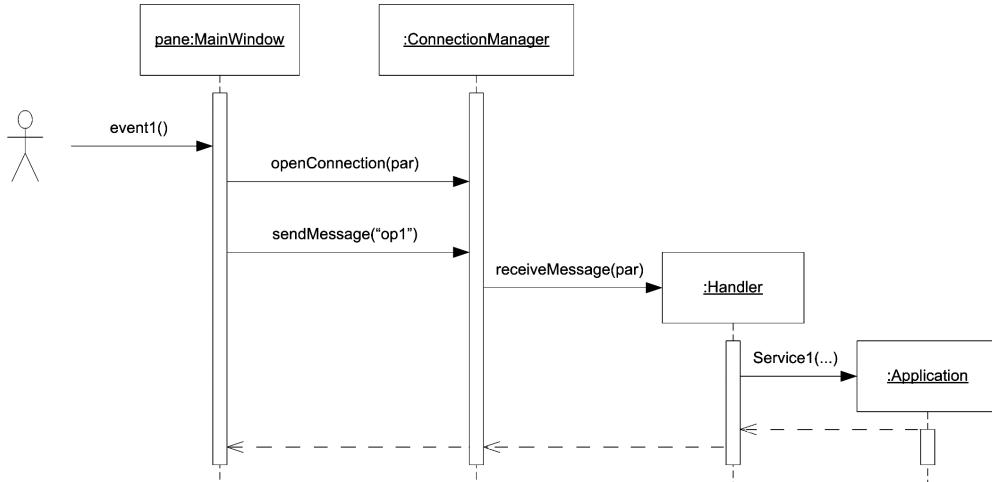


Figura 6.11 Comportamento dinamico.

120 Capitolo 6 Progettare la soluzione

**Figura 6.12** Creazione di thread.

La Figura 6.12 descrive la situazione nella quale a ogni richiesta dell'utente il server crea un thread di esecuzione separato. Un thread in Java viene realizzato definendo una classe attiva e instanziandone un oggetto. La Figura 6.12 mostra che a fronte di ogni messaggio ricevuto dal server, viene creato un nuovo oggetto `Handler`. Se la classe corrispondente è attiva, l'oggetto rappresenta un nuovo thread creato per rispondere a una singola richiesta.

6.1.3 Deployment view

Le viste precedentemente introdotte rendono possibile descrivere la struttura logica e l'organizzazione di dettaglio del codice. La *deployment view* illustra come il codice viene realmente distribuito e installato sui computer dell'utente (o degli utenti). Nel caso in questione, la deployment view deve spiegare come sono distribuiti client e server. In questa sede è necessario anche affrontare il tema di come gli elementi dell'architettura sono rappresentati e memorizzati. Queste informazioni sono descritte utilizzando due diagrammi. Il primo è un component diagram che utilizza gli stereotype `<<artifact>>` e `<<manifest>>`. Un *artifact* è un particolare tipo di componente utilizzato per rappresentare un file. La relazione `<<manifest>>` indica che una certa classe (o un intero elemento dell'architettura) si "manifesta" in uno o più file. La Figura 6.13 illustra la struttura del client dell'esempio considerato nelle precedenti view, supponendo che i diversi elementi siano realizzati in Java. La figura mostra che l'*artifact* `Client.jar` è la manifestazione fisica del componente `Client`. Il file `.jar` è usato per contenere in forma compatta il codice intermedio (file `.class`) generato a partire dalle classi Java sorgenti (file `.java`) ed è ciò che viene normalmente installato ("deployed") sui computer dove l'applicazione verrà eseguita. La Figura 6.13 mostra anche che `Client.jar` dipende dai file `.class` (codice

6.1 Viste architettoniche 121

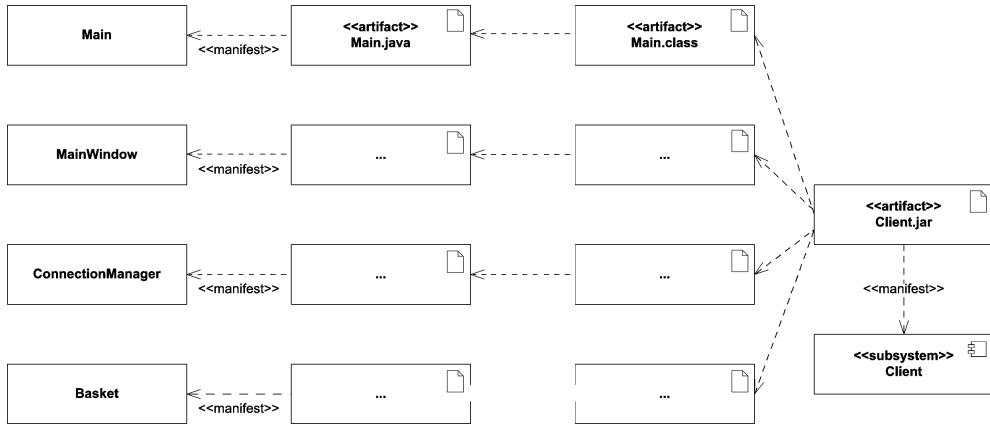


Figura 6.13 Struttura del client.

intermedio), che a loro volta dipendono dai file sorgenti .java. Questi ultimi sono la manifestazione delle relative classi.

La Figura 6.14 illustra il secondo diagramma utilizzato per descrivere la deployment view. Si tratta di un deployment diagram che identifica i nodi fisici (computer), gli ambienti di esecuzione (JVM, Java Virtual Machine) e la collocazione vera e propria degli artifact corrispondenti ai componenti identificati nella Figura 6.10 (si suppone che Client si manifesti in Client.jar, ApplicationLogic in ApplicationLogic.jar e che il componente DatabaseManagement sia effettivamente installato come tale). Come si può notare, vi sono tre nodi fisici: il nodo client, il nodo per il server ApplicationLogic e il nodo per il server DatabaseManagement. In questo caso, a ogni livello logico dell'applicazione corrisponde un elemento di deployment indipendente, localizzato su computer diversi.

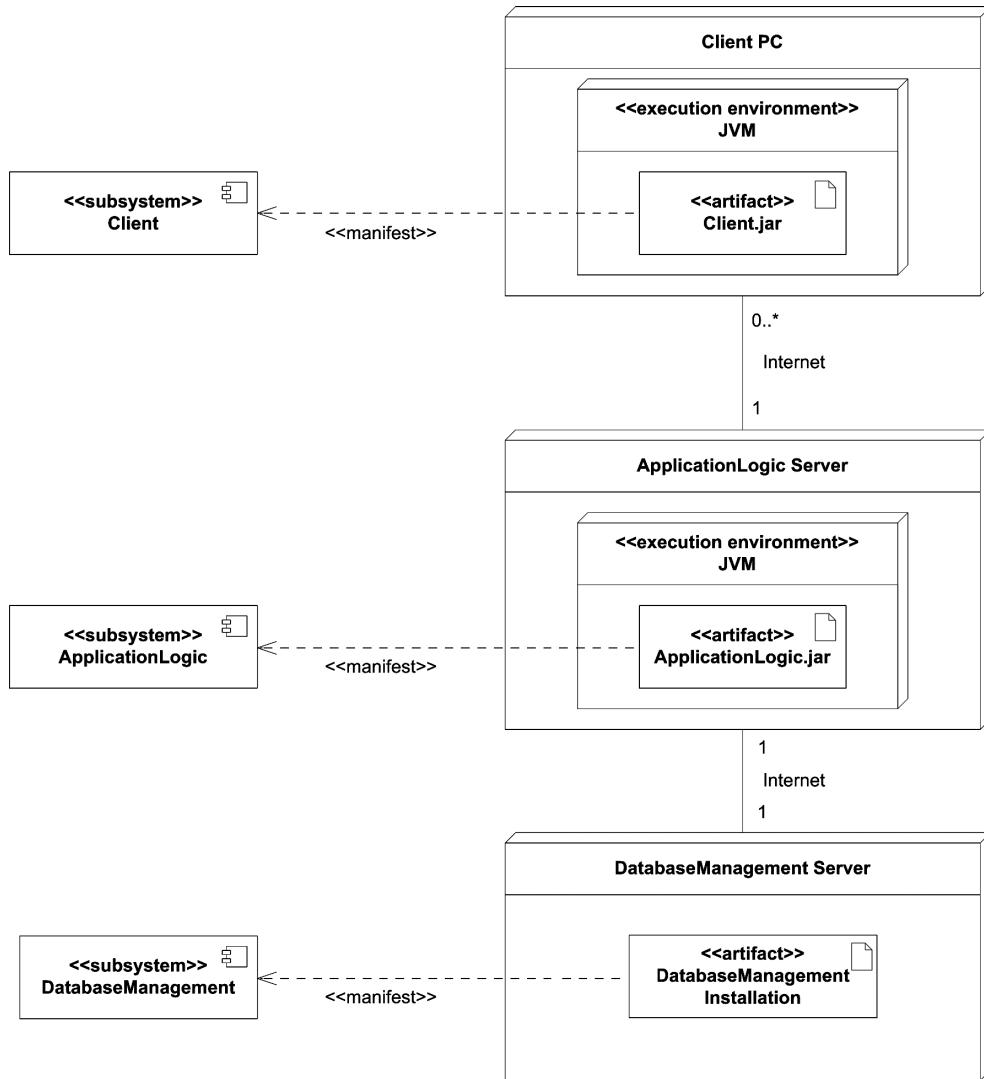
Una soluzione alternativa è quella rappresentata nella Figura 6.15. In questo caso, i due server risiedono sullo stesso computer.

Ovviamente, i diagrammi delle Figure 6.14 e 6.15 possono essere ulteriormente modificati e/o arricchiti in funzione della struttura del sistema. Per esempio, se si volesse rappresentare la situazione discussa in precedenza, nella quale esistono più server ApplicationLogic e un front-end, i diagrammi dovrebbero essere significativamente estesi per rappresentare sia il front-end che la replicazione dei server.

6.1.4 Execution view

Uno degli aspetti più critici, e per certi versi meno studiati in letteratura, è quello relativo alla descrizione run-time (“in fase di esecuzione”) di un sistema software. Nei casi più semplici, si può ragionevolmente ipotizzare che a ogni elemento di deployment corrisponda un processo in esecuzione. Tipicamente, nel caso di semplici programmi C o Pascal, l’unità di deployment è un file eseguibile .exe. La richiesta di esecuzione del file

122 Capitolo 6 Progettare la soluzione

**Figura 6.14** Deployment diagram (con due server separati).

eseguibile causa la creazione di un processo del sistema operativo. In questi casi, l'execution view è molto semplice ed è pressocché identica (o meglio analoga) alla deployment view: a ogni elemento della deployment view corrisponde un processo in esecuzione.

Questi casi, tuttavia, sono poco significativi. In primo luogo accade sempre più spesso che in fase di esecuzione siano creati molteplici processi e thread distribuiti su più

6.1 Viste architetturali 123

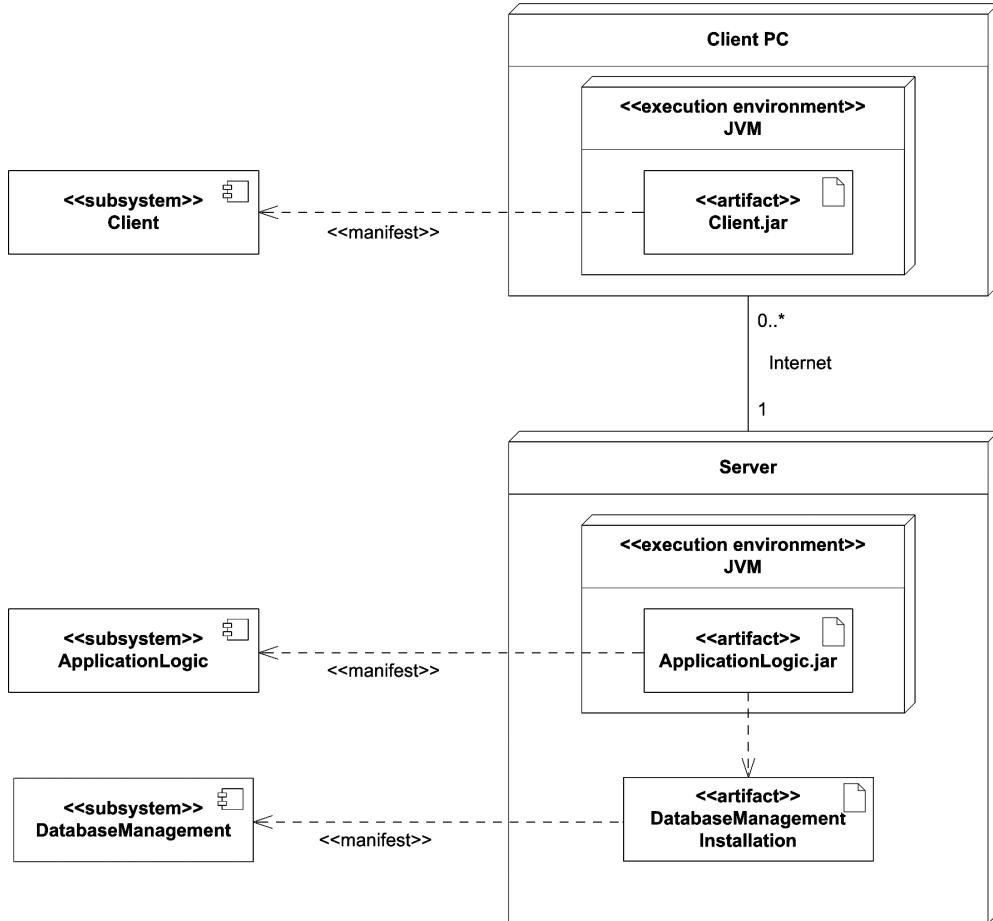


Figura 6.15 Un unico computer per i due server.

computer. In secondo luogo, il numero, la natura e la distribuzione di tali processi e thread può variare nel corso del tempo in funzione dello stato di avanzamento dell’elaborazione. Per questi motivi, non è sufficiente disporre della deployment view. È necessario descrivere in modo esauriente come il sistema evolve nel tempo e quali elementi sono di volta in volta in esecuzione.

Per comprendere come rappresentare adeguatamente l’evoluzione a run-time di un sistema software complesso è innanzitutto importante definire quali siano le diverse “situazioni”, o *configurazioni run-time*, nelle quali il sistema può venirsi a trovare. Il diagramma degli stati della Figura 6.16 illustra la sequenza di configurazioni run-time del sistema considerato in precedenza.

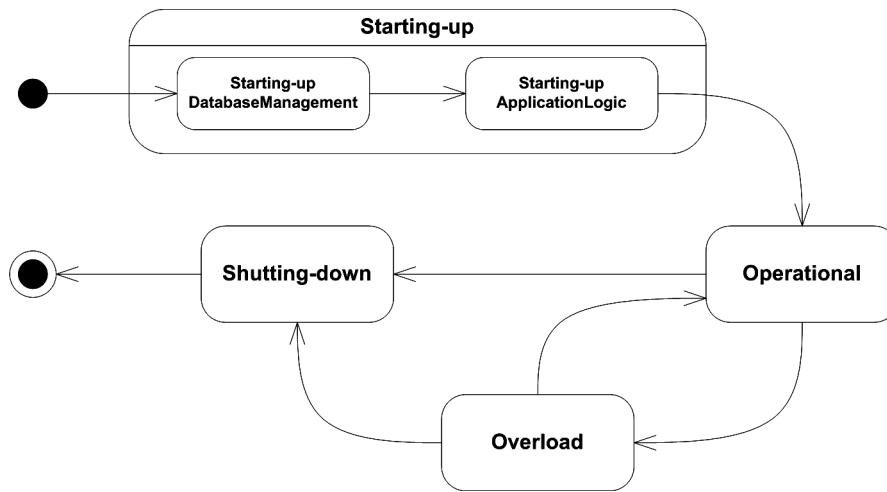


Figura 6.16 Configurazioni run-time del sistema.

Durante la fase di avvio (Starting-up), il sistema compie una serie di operazioni per predisporsi a ricevere le richieste dei client. Viene avviato per primo il server DatabaseManagement e, successivamente, il server ApplicationLogic.

Nella prima fase (Starting-up DatabaseManagement), l'unico processo attivo è il demone DbDaemon (cioè il processo in attesa di richieste) che gestisce l'accesso alla base di dati (Figura 6.17). Tale demone è ottenuto mandando in esecuzione il server DatabaseManagement (come illustrato dalla relazione di dipendenza <>derive>>).

Il secondo passaggio della base di start-up è l'avvio del server `ApplicationLogic`. Al termine di questa fase, il sistema si viene a trovare nella configurazione run-time descritta nella Figura 6.18. Il sistema entra quindi nello stato `Operational`, dove è possibile per i client connettersi e ottenere le informazioni e i servizi richiesti (si veda la Figura 6.19).

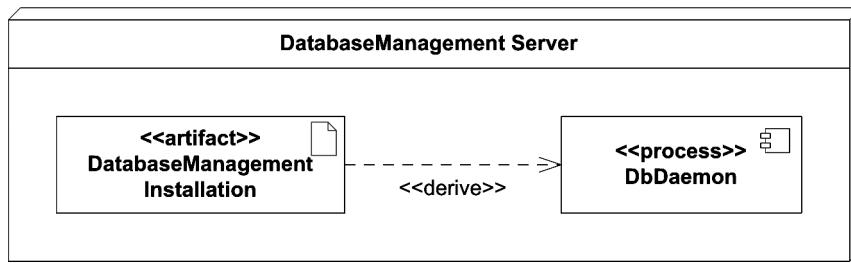


Figura 6.17 Avvio del server DatabaseManagement.

6.1 Viste architettoniche 125

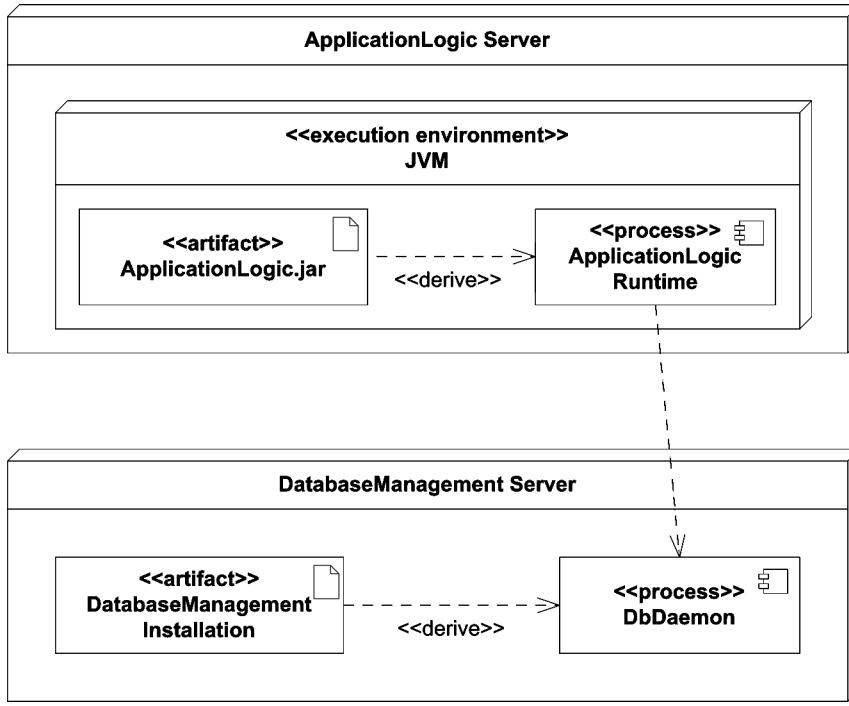


Figura 6.18 Avvio del server ApplicationLogic.

In tale configurazione, alla luce di quanto descritto in precedenza, tutti i processi client interagiscono con un unico server fisico (quest'informazione può essere derivata dal deployment diagram della Figura 6.14).

Rimane da capire se per ogni client viene utilizzato un thread di esecuzione indipendente sul server o se le richieste dei diversi client sono eseguite in modo strettamente sequenziale da un unico processo/thread. La Figura 6.20 illustra un diagramma di struttura composita che specifica, in questo caso, che ogni processo client interagisce con un diverso thread Handler (l'associazione che li unisce è 1..1).

In generale, partendo dal diagramma degli stati (si veda la Figura 6.16) che identifica le diverse configurazioni run-time del sistema, è possibile rappresentarle utilizzando diversi tipi di diagrammi UML. In alcuni casi, può essere sufficiente anche una semplice descrizione testuale. Per esempio, lo stato `Overload` può essere molto semplicemente caratterizzato dicendo che è simile allo stato `Operational`, con l'aggiunta che ulteriori richieste di accesso da parte di nuovi client non vengono accettate in quanto il sistema è sovraccarico.

126 Capitolo 6 Progettare la soluzione

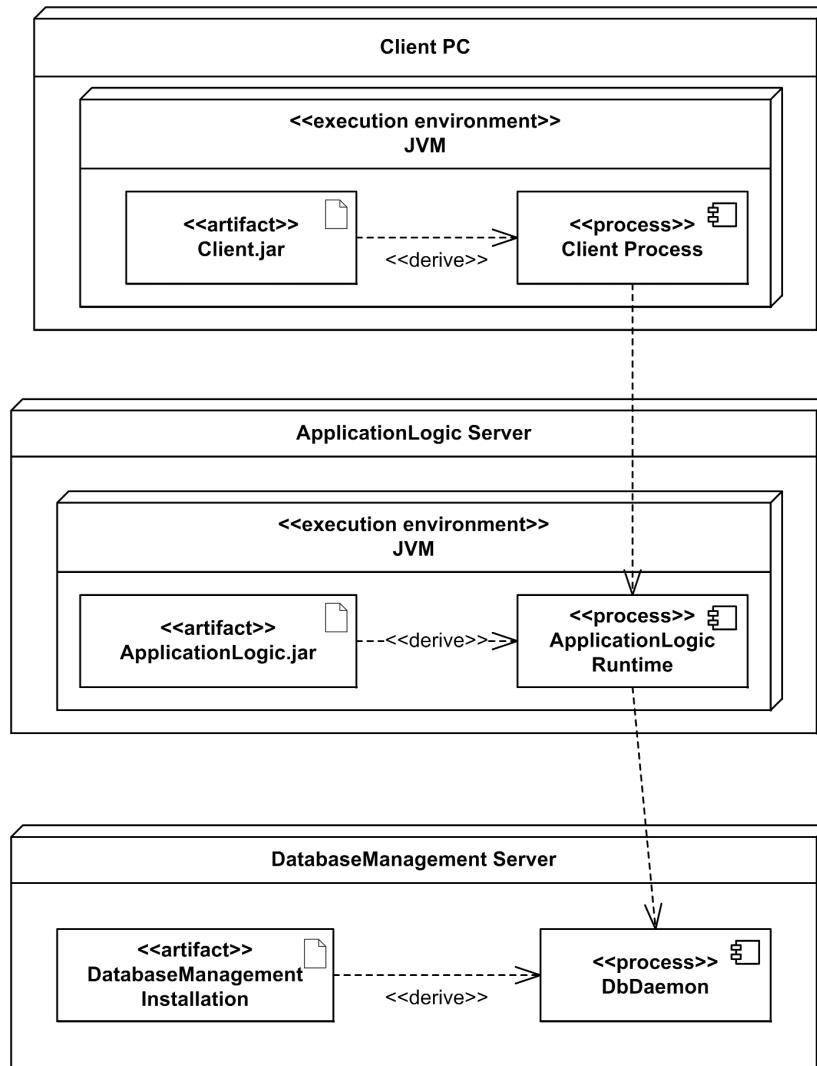


Figura 6.19 Configurazione run-time durante il funzionamento a regime.

Considerazione di carattere generale

Quanto visto costituisce lo schema logico che deve essere utilizzato per descrivere e caratterizzare in modo esaustivo l'architettura informatica di un sistema software. Le osservazioni e gli esempi che sono stati presentati sono maggiormente orientati alla fase di progettazione architettonica. Tuttavia, dal punto di vista delle tecniche di descrizione, lo stesso tipo di considerazioni possono essere applicate anche alla progettazione di dettaglio.

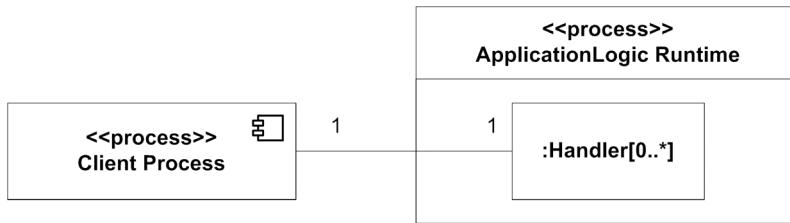


Figura 6.20 Processi e thread.

6.2 Cos'è un componente?

Il termine “componente” è stato sinora diffusamente utilizzato in due diversi ambiti. In primo luogo, è un elemento sintattico (e un tipo di diagramma) di UML. Vi è quindi un’interpretazione del termine che è strettamente collegata alla terminologia e alla struttura di UML.

Ma il termine “componente” viene utilizzato in modo molto più generale (e generico) per identificare una “parte” di un sistema informatico complesso. La letteratura più recente relativa alle architetture software indica nei “componenti” gli elementi di base di un’architettura informatica: in realtà, il significato della parola non è univocamente definito.

- In una prima accezione, è *un’unità di deployment*: cioè un’unità che può essere installata e localizzata su un qualche computer.
- Una seconda accezione lo identifica come una *unità di riuso*: un componente software è una porzione di codice che svolge una ben precisa funzione e che è strutturata e concepita in modo tale da semplificarne il riuso in nuove e diverse applicazioni.

La differenza non è marginale in quanto si tratta di concetti certamente tra loro collegati ma distinti, sia dal punto di vista concettuale che da quello pratico. Spesso il termine “componente” è usato in modo assolutamente informale; nel resto del libro sarà utilizzato per identificare un’unità di riuso; nel caso sia necessario o opportuno utilizzarlo nell’altra accezione, ciò verrà segnalato in modo esplicito.

6.3 Architetture hardware e software

Al termine di questo capitolo è opportuno precisare il significato e l’interpretazione che viene spesso data alla parola “architettura”. Infatti, anche per questo termine, così come nel caso del concetto di componente, esistono almeno due diverse interpretazioni che vengono spesso confuse o usate in modo ambiguo. Per esempio, l’espressione “architettura client-server” viene utilizzata sia per descrivere un PC connesso a un minicomputer, sia un programma client che interagisce con un server (come nell’esempio visto in questo capitolo).

In realtà, esistono (almeno) due tipi di architetture: architetture hardware e architetture software. Un'architettura *hardware* identifica i dispositivi di elaborazione dati e le modalità secondo le quali sono organizzati. Negli esempi visti in precedenza, sono state identificate due architetture hardware di tipo client-server a due livelli (Figura 6.15) e a tre livelli (Figura 6.14). Un'architettura *software* identifica i componenti software che realizzano un sistema informatico. Nell'esempio visto, l'architettura software del sistema è a tre livelli (client, application logic e database management), indipendentemente dal fatto che nella Figura 6.15 due elementi risiedano sulla stessa unità di elaborazione: si tratta in questo caso di un sistema software client-server a tre livelli, operante su un'architettura hardware client-server a due livelli.

Nel seguito il termine "architettura" sarà sempre utilizzato per indicare l'architettura software di un sistema informatico. Anche in questo caso, verranno esplicitamente indicati i contesti nei quali è necessario utilizzare il termine nell'altra accezione.

6.4 Riferimenti bibliografici

Il tema delle architetture software è una diretta evoluzione degli studi iniziati fin dagli anni '70 da Parnas sulla struttura modulare dei programmi. Con la crescita delle dimensioni e complessità dei progetti di sviluppo del software, i ricercatori si sono posti il problema di come caratterizzare sistemi informatici distribuiti e di dimensioni significative. Perry e Wolf [1992] introducono alcuni concetti e idee fondanti di quella che sarebbe divenuta l'area di ricerca sulle architetture software. Negli anni successivi si sono susseguiti una serie di studi e ricerche relativi alle modalità secondo le quali descrivere un'architettura software e alle diverse tipologie di schemi e stili disponibili. Un testo che discute il tema delle architetture software e dell'uso di UML come strumento di descrizione è Hofmeister et al. [2000].

Capitolo 7

Stili architetturali e design pattern

Lo sviluppo di soluzioni informatiche complesse ha reso possibile nel corso degli anni il consolidamento di un patrimonio di conoscenze ed esperienze estremamente significativo. In particolare, sono stati identificati una serie di schemi e stili ricorrenti nella progettazione e costruzione di sistemi complessi. Termini come “architetture client-server” sono entrati nel gergo quotidiano di tutti gli addetti e utenti di tecnologie informatiche. Così come in altri settori industriali e merceologici, l'utilizzo di schemi e stili permette di identificare in modo veloce e sintetico le caratteristiche peculiari di un bene. Allo stesso tempo, il progettista è in grado di associare in modo immediato caratteristiche funzionali (e non) a una particolare struttura informatica, identificando nel contempo i passi e le scelte progettuali implicati. Come discusso nel Capitolo 1, il concetto di schema è quindi uno dei principali strumenti di lavoro dell'ingegnere.

Nel caso della produzione del software, sono stati sviluppati diversi tipi di schemi e stili, in funzione del livello di astrazione al quale ci si pone nello studiare un sistema informatico complesso.

- *Stili architetturali*: descrivono la struttura complessiva di un sistema informatico complesso (tipicamente distribuito). Gli elementi che li caratterizzano sono normalmente i macrocomponenti dell'applicazione (componenti e relativi processi) e la descrizione delle interazioni che occorrono tra di essi. Il già citato client-server è un esempio classico di stile architettonico.
- *Design pattern*: sono schemi che descrivono porzioni ridotte di codice. Spesso sono utilizzati per strutturare un singolo programma o anche una sua parte significativa. Esempi classici di design pattern sono il model-view-controller, utilizzato per strutturare un programma interattivo, o il concetto di “factory”, utilizzato per gestire la creazione di oggetti complessi.
- *Programming idiom*: sono sequenze di istruzioni di frequente utilizzo (per esempio, la sequenza di operazioni necessarie per serializzare un oggetto complesso Java o per creare sullo schermo una finestra grafica).

130 Capitolo 7 Stili architetturali e design pattern

Stili architetturali, design pattern e programming idiom sono utilizzati in forme e modalità diverse. Certamente, va tenuto ben presente che non è sufficiente far riferimento a uno stile o a un design pattern per identificare in modo univoco o preciso la struttura di un'applicazione informatica. Così come non basta dire “ponte sospeso” per identificare tutte le caratteristiche del Golden Gate di San Francisco e client-server o peer-to-peer per descrivere un sistema informatico. Schemi e stili identificano i *tratti essenziali di una soluzione*, ma non sostituiscono in alcun modo la sua descrizione puntuale. Quindi, nel descrivere una particolare architettura software sarà utile menzionare e spiegare il legame che ha con specifici stili architetturali. Ma i due livelli rimangono concettualmente e operativamente distinti. Considerando un altro settore industriale, il mondo dell'auto, se certamente il termine “spider” caratterizza in modo distintivo le peculiarità di quel tipo di mezzi, è altrettanto indubbio che uno specifico prodotto (l'equivalente di una particolare architettura informatica) ha proprie caratteristiche distinte. Una Ferrari spider è una speciale “incarnazione” del concetto di spider. Una BMW spider, pur avendo tratti comuni con la Ferrari (lo “schema di riferimento di una spider”), sarà nel dettaglio costruita e “strutturata” in modo differente.

La progettazione e la descrizione di una soluzione informatica dovrà quindi basarsi sull'uso di stili, design pattern e idiom. Nel seguito del capitolo, l'enfasi della discussione sarà posta soprattutto sul tema degli stili architetturali. Per una discussione dettagliata di design pattern e idiom, il lettore è invitato a consultare i testi specializzati sulla programmazione object-oriented citati in bibliografia.

7.1 Client-server

Nel momento in cui si abbandona l'idea di un sistema informatico costituito da un unico programma monolitico, è naturale ricorrere allo stile client-server. Con questo termine si intende una configurazione nella quale un programma (server) offre una serie di servizi sincroni ad altri programmi (client). Lo schema di un sistema client-server è descritto dai component diagram delle Figure 7.1 e 7.2. In particolare, la Figura 7.2 illustra la tipica configurazione di un sistema client-server a due livelli: una molteplicità di client utilizza i servizi sincroni offerti da un server.

Gli aspetti essenziali dello stile client-server possono essere riassunti come segue.

1. Il server mette a fattore comune dati e servizi (cioè conoscenza) che possono essere riutilizzati da una molteplicità di utenti, ciascuno dotato di un proprio programma client.
2. In questo modo, si sfrutta la capacità di elaborazione che ciascun utente ha a disposizione.
3. L'utilizzo di servizi sincroni fa sì che ogni client, effettuata una richiesta di servizio al server, attenda una risposta prima di procedere.

In realtà, lo stile client-server va ben al di là dell'esempio presentato nella Figura 7.1: è possibile infatti averne diverse varianti in funzione della distribuzione delle funzionalità

7.1 Client-server 131

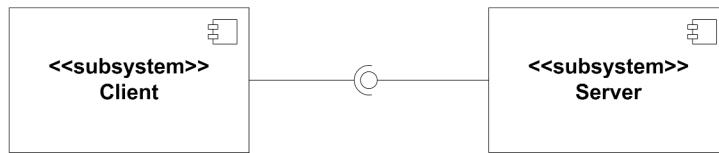


Figura 7.1 Component diagram di un sistema client-server.

applicative tra client e server. In generale, un'applicazione informatica è costituita da tre elementi (si veda la Figura 7.3.a): un primo elemento che gestisce l'interazione con l'utente (*presentation*); un secondo che contiene gli algoritmi applicativi veri e propri (*application logic*); un terzo che gestisce l'accesso ai dati e servizi condivisi (*data management*). Se è evidente che in buona misura le parti di presentation e di data management risiedono rispettivamente su client e server, è indubbio che esistono dei margini di scelta per quanto riguarda la configurazione complessiva del sistema.

Nella Figura 7.3 sono illustrate alcune varianti di un sistema client-server a due livelli. La variante (b) prevede sul client sia la gestione dell'interazione con l'utente che

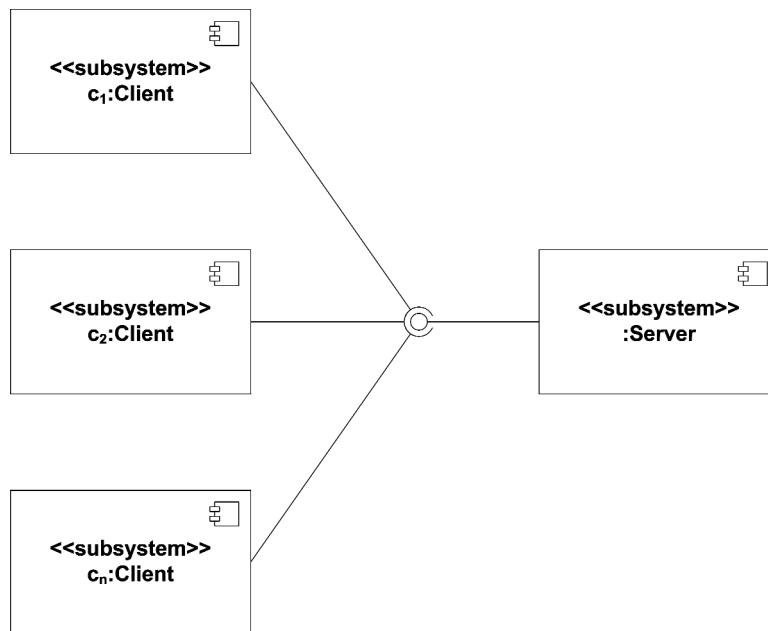


Figura 7.2 Istanze di un sistema client-server.

132 Capitolo 7 Stili architettonici e design pattern

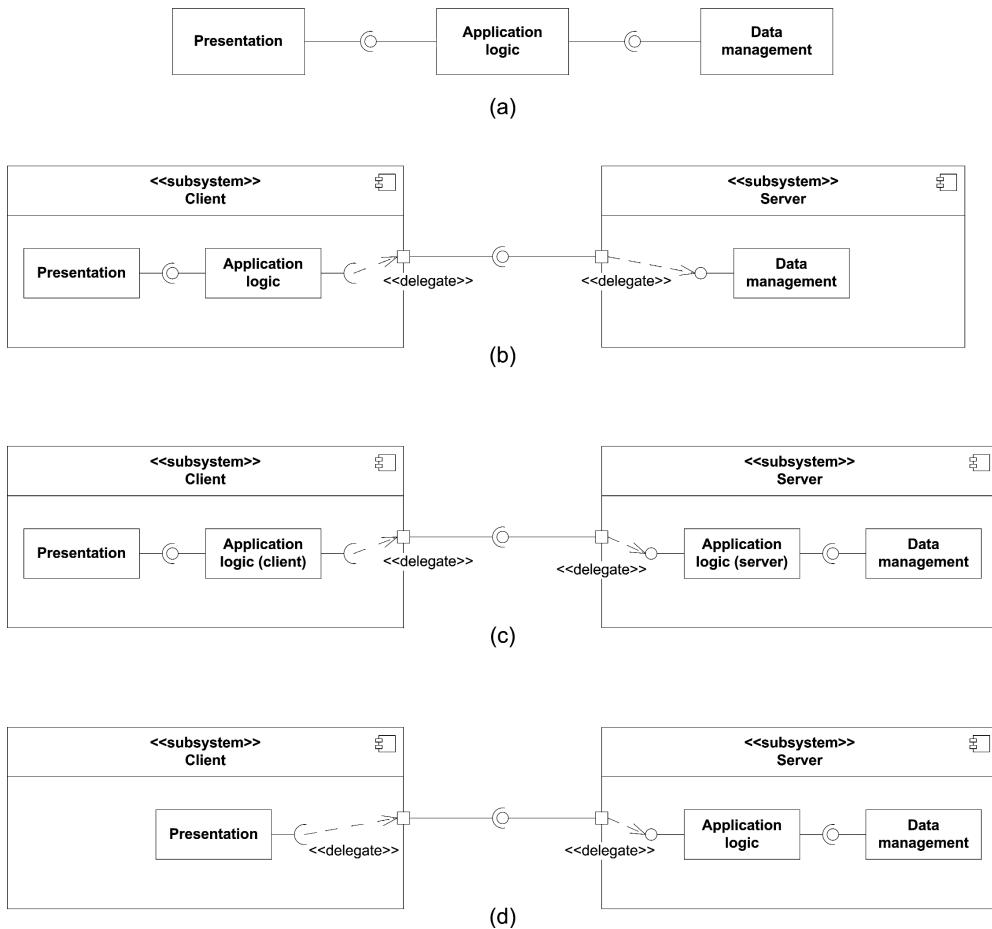


Figura 7.3 Varianti di un sistema client-server a due livelli.

gli algoritmi applicativi veri e propri. La variante (c) prevede che una parte della logica applicativa risieda sul client e un'altra parte sul server. La variante (d) è caratterizzata dalla presenza sul client delle sole funzionalità relative all'interazione con l'utente, mentre tutti gli algoritmi e la gestione di dati e servizi risiedono sul server. Si noti altresì che sono possibili altre forme di distribuzione dei tre elementi introdotti in precedenza. Per esempio, quando si sviluppano applicazioni utilizzando X Window System (tipico di molti ambienti Unix e di Linux), una parte delle funzioni di presentation si trovano sul client, mentre un'altra parte è residente sul server. Analogamente, anche la gestione dei dati e servizi può essere suddivisa tra client e server. In questo modo si introducono due

ulteriori varianti rispetto a quelle presentate nella Figura 7.3. In generale, l'ingegnere del software ha la possibilità di decidere come "ritagliare" i diversi elementi che dovranno essere collocati su client e server. Quali sono i criteri per effettuare questo taglio? Esiste una configurazione migliore delle altre oppure di volta in volta è necessario scegliere?

In realtà esistono pro e contro che derivano dal collocare le diverse funzioni sul client o sul server. In particolare, due sono i fattori che devono essere tenuti in considerazione.

- Se si collocano molte funzioni sul server, si ottiene che il client può essere meno potente e di più facile gestione. Non è necessario preoccuparsi di installare (molto) software sul client. Si semplifica quindi la gestione del lato client. Tale aspetto è molto importante specie in presenza di un numero alto di client distribuiti in organizzazioni complesse, magari disperse sul territorio. D'altro canto, in questo modo si appesantisce il compito del server, che deve accollarsi un carico elaborativo maggiore (i "conti" per tutti i client si fanno sul server).
- Se al contrario si "alleggerisce" il server, si ottiene che svolga un carico di lavoro minore. Spostando funzioni sui client, si può sfruttare al meglio la capacità di calcolo della macchina in dotazione al singolo utente. Allo stesso tempo, però, è necessario prevedere macchine client tendenzialmente più potenti e, inoltre, gestire la distribuzione e manutenzione delle componenti applicative distribuite sui client stessi.

Come si vede, quindi, non esiste una soluzione ottima. Di volta in volta, si dovrà scegliere quella "più conveniente" in funzione dei requisiti, delle caratteristiche del sistema da sviluppare e delle tecnologie a disposizione dell'ingegnere del software: per esempio, le piattaforme per la gestione centralizzata di applicativi distribuiti (si veda il Capitolo 10).

Proprio per venire incontro ai problemi che derivano da soluzioni client-server classiche a due livelli, nel corso di questi anni sono state sviluppate delle estensioni di questo modello, con l'introduzione delle architetture *multi-tier*. Esse prevedono la presenza di più livelli specializzati e dimensionati in funzione delle caratteristiche a cui si è fatto cenno in precedenza (requisiti funzionali, vincoli tecnologici, prestazioni richieste ecc.). In un sistema multi-tier, esistono diversi livelli di interazione client-server. Per esempio, la Figura 7.4 illustra lo schema e un esempio di sistema multi-tier a quattro livelli. Nel diagramma delle istanze di componenti si può notare che gli elementi di un livello di tier possono interagire con uno o più elementi degli altri livelli. Tale flessibilità permette di distribuire le funzionalità applicative in modo coerente con i requisiti e i vincoli di scenari applicativi a priori molto complessi.

Nel Capitolo 10 verrà presentato un ulteriore esempio di sistema client-server a più livelli utilizzato sempre più frequentemente nel caso di applicazioni web-based: l'architettura J2EE (*Java 2 Enterprise Edition*).

134 Capitolo 7 Stili architettonurali e design pattern

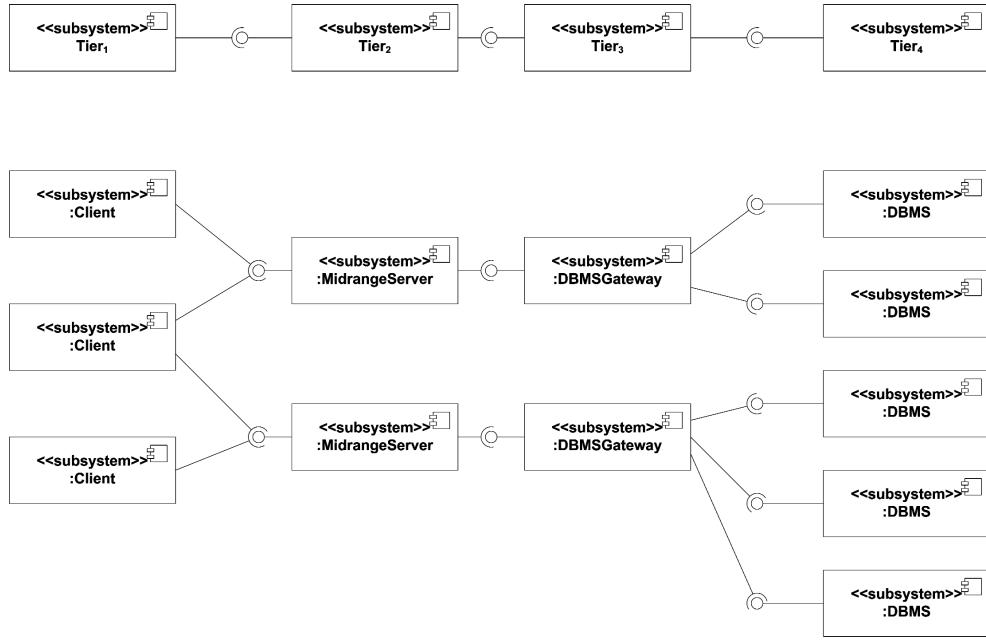


Figura 7.4 Sistema multi-tier a quattro livelli.

7.2 Peer-to-peer

Nello stile client-server, pur nelle sue molteplici varianti, vi sono alcuni programmi che offrono servizi e altri che ne fruiscono. I ruoli sono distinti e “fissi”. Questo approccio, ancorché molto utile e largamente utilizzato in una miriade di applicazioni e sistemi, ha peraltro una serie di limiti.

- Nel caso il server (o alcuni server) non sia disponibile in quanto irraggiungibile o inattivo, il client resta sostanzialmente bloccato: manca l’interlocutore in grado di offrire i servizi e/o i dati necessari per procedere nelle elaborazioni.
- Se un client volesse rendere disponibili alcuni propri dati o servizi, deve necessariamente copiarli sul server.
- La scalabilità del sistema, cioè la capacità di operare in modo efficiente al crescere dei client, dipende dalla potenza di calcolo del server. È certamente possibile ipotizzare una replica dei server (come discusso a proposito delle applicazioni client-server multi-tier). Ma è indubbio che tale scelta richiede una attento studio per dimensionare opportunamente il numero e la disposizione dei diversi server.

Lo stile architettonurale *peer-to-peer* (anche indicato con il termine “p2p”) nasce proprio per affrontare e risolvere questo tipo di problemi. In un sistema p2p, ogni elemento (detto *peer* o *nodo*) può operare da client e da server. È cioè in grado sia di richiedere dati e servizi che di offrirne. Per questo motivo, ogni peer offre al resto del sistema un’insieme di servizi. Inoltre, esistono dei meccanismi che permettono a ogni peer di identificare gli altri elementi del sistema che sono in grado di fornirgli i dati e/o i servizi.

È possibile identificare due paradigmi di funzionamento di riferimento: p2p ibrido e p2p puro.

L’esempio più noto di sistema basato sull’approccio p2p ibrido è Napster, che ha lanciato alcuni anni fa i servizi di scambio di file (in particolare musicali). In Napster, ogni peer conserva una serie di brani musicali che rende disponibili agli altri membri della comunità. La ricerca di un brano viene resa possibile dalla presenza di un indice centrale (detto anche *directory*) presso il quale ogni peer registra l’elenco di cui dispone e il proprio indirizzo di rete (si veda la Figura 7.5). Chi cerca un pezzo, invia una richiesta all’indice che risponde indicando l’elenco degli indirizzi di rete dei peer che ne hanno una copia. Il nodo richiedente può a quel punto interagire direttamente (“peer-to-peer”) con uno dei nodi indicati per recuperare il brano ricercato.

Si noti che l’interfaccia *Retrieve* è sia offerta che richiesta da ogni singolo peer: ognuno infatti, per quanto detto, è in grado di richiedere informazioni ad altri peer o di soddisfare le loro richieste.

La Figura 7.6 illustra una particolare istanza di architettura p2p ibrida. Per semplificità non sono state disegnate tutte le diverse interazioni tra i peer presenti. In particolare, l’interazione tra peer è indicata solo per le richieste che p_1 fa a p_2 (per completezza, andrebbero inserite analoghe indicazioni per tutte le possibili coppie ordinate di peer).

In un approccio p2p puro, non esiste il ruolo di *directory*. Esistono diversi tipi di algoritmi utilizzati per interconnettere i peer e propagare le richieste di informazioni. Per esempio, in Gnutella ogni peer conosce alcuni vicini. Le richieste vengono propagate nella comunità sfruttando proprio questa relazione di vicinanza. Per evitare che le richieste continuino a rigirare in circolo, normalmente viene indicato un numero massimo di “salti” (detti “hop”) che una richiesta può fare sui peer contigui a partire dal nodo che l’ha generata.

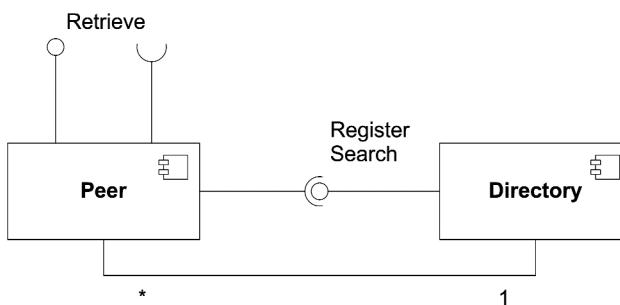


Figura 7.5 Componenti di un sistema p2p ibrido.

136 Capitolo 7 Stili architettonici e design pattern

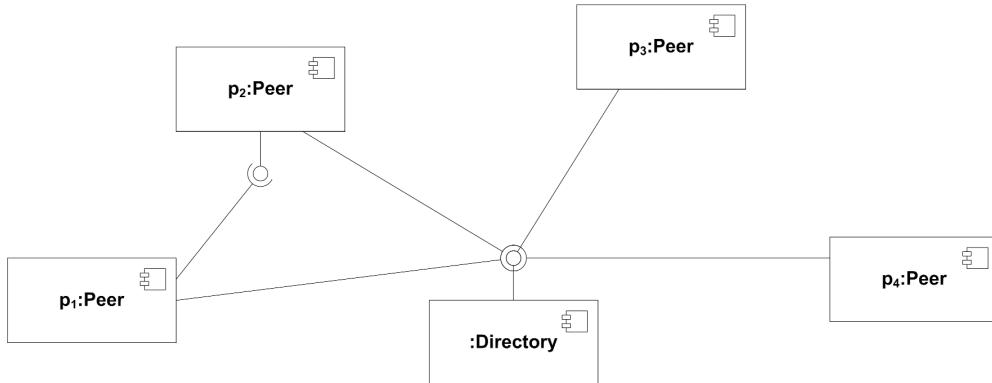


Figura 7.6 Un esempio di sistema p2p ibrido.

Ovviamente, esistono forme intermedie e più complesse di p2p che non sono direttamente riconducibili ai due paradigmi puro e ibrido. Questi approcci cercano di identificare un trade-off tra facilità di gestione, efficienza nei processi di ricerca e aggiornamento dei dati e disaccoppiamento funzionale tra elementi dell'architettura. L'argomento costituisce un tema di ricerca particolarmente interessante e darà probabilmente origine a nuove varianti dei due stili di base discussi in precedenza.

In generale, l'approccio p2p è nato per facilitare lo scambio di dati in community dove non si vuole avere un unico repository centrale di informazioni. In effetti, anche laddove si utilizzi un approccio ibrido, il paradigma p2p offre una serie di vantaggi rispetto al modello client-server classico. In particolare, permette di *federare sistemi autonomi* dotati di propri dati e servizi (il termine “federare” è utilizzato per indicare un processo di integrazione a posteriori di sistemi autonomi e spesso eterogenei). Questa situazione si ripresenta in molti ambiti applicativi. Nel campo dell'e-government, per esempio, ogni comune dispone di una propria anagrafe. È peraltro essenziale che le anagrafi di comuni diversi siano tra loro integrate per permettere operazioni quali la ricerca sul territorio nazionale di un cittadino o l'automazione delle procedure per il trasferimento di residenza. Per l'integrazione tra anagrafi (autonome e spesso preesistenti al processo di integrazione), viene utilizzato un modello p2p ibrido che prevede un indice centrale con l'elenco dei codici fiscali di tutti i cittadini e, per ciascuno di essi, l'indicazione della particolare anagrafe che ne mantiene le informazioni di dettaglio.

In generale, l'avvento dello stile p2p ha permesso di realizzare un numero molto vasto di nuove applicazioni. Il caso più recente di successo di un sistema basato sullo stile p2p (ibrido) è certamente Skype, il sistema che permette di realizzare chiamate audio sulla rete Internet. Anche in Skype, ogni nodo (Skype client) si registra presso una directory centrale che fa da snodo e da “centralino” tra tutti coloro che vogliono avviare chiamate vocali.

7.3 Publish-subscribe

I sistemi client-server si basano su un meccanismo di interazione punto a punto di tipo sostanzialmente sincrono: quando un client richiede un servizio a un server, rimane in attesa di una qualche risposta. Questo tipo di approccio è molto efficace in svariate situazioni e scenari applicativi; in altri casi, invece, presenta una serie di problemi e limiti. Si consideri ad esempio la sala contrattazione titoli di una banca. Il tipico modo di operare dei broker che acquistano e vendono titoli non è quello di chiedere esplicitamente a ciascun potenziale acquirente/venditore una dichiarazione di interesse a effettuare una certa transazione. Al contrario, il broker “pubblica un’informazione” circa i propri interessi: “voglio vendere X” oppure “vorrei comprare Y”. In linea di principio tale informazione dev’essere trasmessa a tutti gli altri broker, ovunque situati, che sono interessati a comprare X o vendere Y. Quindi tale “messaggio” dovrebbe essere trasmesso “a tutti gli interessati” e non solo a un particolare broker. Ma chi produce il messaggio non è detto sappia (o debba necessariamente sapere) chi sono tutti gli altri broker interessati all’operazione, né ha bisogno che tutti rispondano dichiarando la propria posizione. A chi emette la richiesta importa che tutti gli interessati siano avvertiti e che, se qualcuno “prima o poi” risponde, tale informazione venga recapitata al richiedente. Si tratta quindi di uno schema totalmente duale rispetto alla chiamata sincrona di servizio tipica del modello client-server: nel caso dei servizi di brokering, un’informazione viene recapitata in modo *multicast* a un numero a priori sconosciuto di controparti che rimangono in genere *anonime* (non si sa chi riceverà la richiesta); i destinatari potrebbero rispondere o no. È quindi anche uno schema di comunicazione *asincrono*, in quanto non si sa se e quando qualcuno risponderà e quindi chi fa la richiesta non può restare in attesa di una risposta che potrebbe anche non arrivare mai.

Questo schema di funzionamento dà origine a uno stile architettonicale detto *publish-subscribe* o *a eventi*, in quanto ciò che viene inviato da un richiedente è la segnalazione di un evento che è accaduto: nel nostro esempio, la dichiarazione di interesse del broker che vuole comprare X e vendere Y. Lo stile publish-subscribe può essere sinteticamente descritto dalla Figura 7.7.

Ogni nodo o peer che vuole utilizzare i servizi di distribuzione degli eventi ha a disposizione una serie di operazioni offerte da un componente chiamato *dispatcher*.

- Per esprimere l’interesse a ricevere eventi relativi a certi tipi di informazione, un peer invoca il metodo *Subscribe* del dispatcher, specificando le caratteristiche del tipo di eventi al quale è interessato. Per esempio, potrebbe indicare “titoli che riguardano X o Y”. Dal momento in cui il dispatcher ha registrato la dichiarazione di interesse, il peer è abilitato a ricevere la notifica di eventi che avessero le caratteristiche richieste.
- Per segnalare un evento a chiunque fosse interessato, un peer utilizza il servizio *Publish* del dispatcher. Come parametro dell’invocazione, il peer specifica tutte le informazioni che caratterizzano l’evento. Il dispatcher, ricevuto l’evento, ne inoltra una copia a tutti i peer che si erano sottoscritti per quel tipo di eventi.

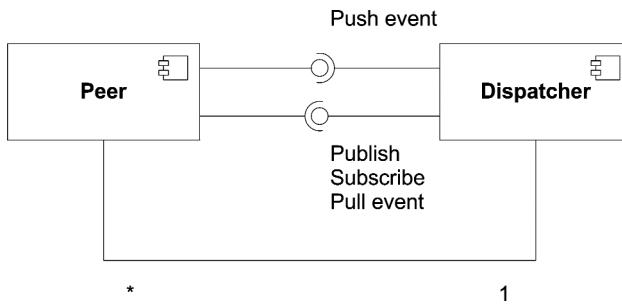


Figura 7.7 Stile publish-subscribe.

- Per ricevere la notifica di un evento, un peer ha a disposizione due meccanismi. Il primo prevede che sia il peer stesso periodicamente a controllare presso il dispatcher l'esistenza di una notifica di proprio interesse (*Pull event*); il secondo prevede che sia il dispatcher a inviare la notifica al peer, invocando un metodo da questi offerto (*Push event*). Normalmente, tale metodo viene reso disponibile dal peer attivando un thread autonomo che resta in attesa delle chiamate da parte del dispatcher. In questo modo, il peer può continuare le proprie elaborazioni senza doversi preoccupare di andare periodicamente a verificare con il dispatcher se esistono notifiche di eventi di proprio interesse.

Lo stile publish-subscribe si è largamente diffuso in diversi ambiti in quanto è un potente strumento di integrazione tra componenti distribuite e lasciamente accoppiate (le parti non entrano in contatto direttamente, non sanno con chi interagiranno, né quanti sono i destinatari di una comunicazione). Ovviamente, viste le sue caratteristiche, non costituisce un'alternativa né al modello client-server né a quello p2p; al contrario, esso è utilizzato per complementarne le funzionalità alla luce dei requisiti e delle caratteristiche dell'applicazione da sviluppare.

7.4 Codice mobile

Una delle maggiori trasformazioni tecnologiche avvenute con l'avvento di Internet è stata l'evoluzione di alcuni principi e approcci alla costruzione ed esecuzione dei programmi. Da sempre il programmatore era abituato a immaginare che un programma venisse eseguito sulla macchina sulla quale era stato installato. Con l'avvento delle moderne tecnologie di programmazione questo assunto viene a cadere. È infatti possibile costruire programmi che, in forme diverse, sono in grado di spostarsi da macchina a macchina. In particolare vi sono tre modalità secondo le quali ciò può avvenire: *code on demand* (COD), *remote evaluation* (RE) e *mobile agent* (MA). Questi tre stili sono in un qualche modo anch'essi estensioni dello stile client-server.

Lo stile COD estende l'approccio client-server in quanto copre una situazione a esso complementare. In un sistema client-server, il client non dispone delle risorse necessarie per effettuare un'elaborazione: codice e dati. Per questo motivo invia una richiesta al server che, al contrario, dispone di tali risorse. Lo stile COD permette di affrontare situazioni nelle quali il client ha risorse di calcolo e dati che permetterebbero lo svolgimento dell'elaborazione richiesta, ma manca del codice applicativo necessario. Se tale codice è disponibile su un server remoto, il client potrebbe inviare i dati al server e richiedere che sia questo a svolgere l'elaborazione e ritornare il risultato richiesto. Spesso quest'approccio non è praticabile, poiché la mole di dati da trasmettere sarebbe troppo alta, il server verrebbe caricato di un'eccessiva quantità di elaborazioni e, nel contempo, non si sfruttarebbe la capacità elaborativa del client. Per questo motivo è possibile copiare il codice dal server sul client, dove a quel punto sono disponibili tutte le risorse per svolgere l'elaborazione richiesta. Si richiede il codice sul client "quando ce n'è bisogno" (on demand).

Lo stile COD è illustrato nella Figura 7.8 con un esempio tra i più noti: il web. Le operazioni compiute da un browser a fronte di una richiesta dell'utente di caricare una



Figura 7.8 Lo stile COD.

140 Capitolo 7 Stili architetturali e design pattern

certa pagina fanno sì che un file HTML originariamente presente sul server sia trasferito sul client. Il browser interpreta il “codice” HTML e ne fornisce un “rendering” sullo schermo. Il file HTML è codice che su richiesta si sposta dal server al client.

Tuttavia, si potrebbe obiettare che un file HTML è una forma molto particolare di “codice” e che quindi mal si adatta all’idea di un “programma che si sposta”. La stessa Figura 7.8 mostra anche il caso in cui l’utente richiede l’esecuzione sul client di un applet Java. Un applet altro non è se non una porzione di codice che viene fatta migrare dal server al client dove è eseguita sotto il controllo del browser (o meglio di un suo plugin). In questo caso, si ha la migrazione di un vero e proprio programma.

Lo stile COD costituisce una potente estensione del concetto di client-server classico. Uno dei principali problemi dei sistemi client-server (specie nel caso di client “ricchi” di funzionalità) risiede nella necessità di gestire l’installazione del codice relativo su tutti i client e il suo continuo aggiornamento a fronte di cambiamenti e correzioni. Con l’avvento del COD, se la rete ha prestazioni adeguate, è possibile installare il codice (l’applet) solo sul server e trasferirlo su tutti i client ogni qual volta essi ne richiedano l’uso. In pratica, si ottiene l’effetto di un client “pesante” senza i problemi dovuti all’aggiornamento del software.

Lo stile RE costituisce un approccio duale rispetto al COD. Se nel COD il codice migra dal server al client, nel RE si ha l’opposto: il codice migra dal client al server. Questo scenario può risultare utile se il server ha a disposizione una grande massa di informazioni, ma non dispone del programma (il codice) necessario per elaborarli. È il caso tipico di molte elaborazioni basate sull’uso del linguaggio SQL, utilizzato per accedere e modificare grandi basi di dati. Tipicamente tali basi di dati risiedono sul server, mentre è il client che sa come comporre la richiesta di elaborazione (il codice SQL necessario).

La Figura 7.9 mostra lo scenario appena descritto. Lo script SQL (il testo dell’interrogazione o del comando da eseguire sulla base di dati) viene inviato sul server affinché possa essere eseguito fornendo al richiedente i risultati attesi.

Sia nel caso dello stile COD che nel RE ciò che si sposta è codice sorgente: il file HTML, il codice Java, il testo dello script SQL. Una volta “giunto a destinazione”, il codice viene lanciato in esecuzione producendo i risultati desiderati. In entrambi i casi si tratta di forme di mobilità di codice “deboli”: in realtà non si sposta un’elaborazione, ma solo parti di codice. Lo stile MA supera questa limitazione dando la reale possibilità di far migrare un programma in esecuzione (mobilità “forte”). Per far ciò in realtà servono tecnologie particolari in grado di identificare lo stato di esecuzione di un programma, spostarlo insieme al codice sulla nuova macchina e rimetterlo in esecuzione ripristinando l’ambiente (program counter, contenuto dei registri, contenuto della memoria centrale, stato dei file aperti ecc.). Proprio per la complessità intrinseca nel processo di spostamento del codice in esecuzione; questo tipo di approccio ha avuto fino a ora una diffusione minore rispetto a COD e RE.



Figura 7.9 Lo stile RE.

Considerazione di carattere generale

Nel corso di questi anni sono stati proposti molti stili architettonici. Alcuni hanno avuto applicazione solo in particolari contesti, come lo stile blackboard, utilizzato principalmente per lo sviluppo di sistemi esperti; altri sono più recenti e costituiscono certamente approcci particolarmente promettenti, come lo stile REST, concepito per lo sviluppo di applicazioni avanzate su Internet. Taluni qualificano come stile architettonico anche l'idea che un programma sia costituito da *layer di servizi* (come nel caso di stack di protocolli). Viene considerato stile architettonico anche la composizione di programmi su Unix attraverso il meccanismo del *pipe-and-filter*. Infine, alcuni concetti che nel seguito saranno qualificati come design pattern in realtà dovrebbero essere classificati come stili architettonici (e viceversa).

Ovviamente non esiste un catalogo ufficiale degli stili architettonici, né regole ferree che ne determinino caratteristiche distintive o necessarie. Né è possibile congelare l'insieme degli stili architettonici a quanto oggi conosciuto: pertanto il lettore deve considerare l'elenco di stili presentati come un campione rappresentativo, ancorché ragionevolmente completo.

7.5 Combinazioni di stili

I sistemi informatici moderni sono sempre più complessi in quanto devono rispondere a requisiti molto articolati in ambiti applicativi che a volte hanno scala mondiale. Un sistema di prenotazione aerea deve offrire servizi a livello planetario alle agenzie viaggio, ai viaggiatori (che sempre più sono in grado via web di svolgere molte operazioni relative al proprio viaggio), agli operatori in aeroporto e ai centri di controllo del traffico della singola compagnia aerea e delle agenzie di controllo nazionali e internazionali. Anche in sistemi più semplici, il numero e l'articolazione delle funzionalità che devono essere offerte e dei requisiti che devono essere soddisfatti fa sì che sia raro il caso in cui la soluzione informatica è basata su un unico stile architettonurale. Ogni stile è in grado di soddisfare al meglio alcuni requisiti e bisogni, ma difficilmente da solo può risolvere in modo praticabile esigenze complesse.

Per questo motivo, nella gran parte dei casi le soluzioni informatiche sono costruite combinando diversi stili architettonurali. Nel caso del sistema di contrattazione titoli di una banca, alcune operazioni relative all'attività di brokering vera e propria potranno essere implementate utilizzando l'approccio a eventi. Tuttavia, vi sono anche altre operazioni di tipo più "convenzionale" che possono (e sono) implementate utilizzando servizi di tipo sincrono più tradizionale. Si consideri ad esempio il caso in cui un broker deve richiedere a un altro specifico broker informazioni su una transazione pgressa. In questo caso, si tratta di un'operazione sincrona per la quale è nota la controparte da contattare. L'architettura complessiva del sistema, quindi, potrebbe essere estesa combinando servizi a eventi con servizi p2p (puri), così come illustrato nelle Figure 7.10 e 7.11 (si tratta di un p2p e non di client-server in quanto ogni peer può fare e ricevere richieste).

Ovviamente, il numero di combinazioni possibili e delle loro varianti è molto elevato. Il progettista informatico deve considerare gli stili architettonurali presentati in questo capitolo come mattoni da combinare in modo flessibile in funzione delle caratteristiche del sistema da costruire.

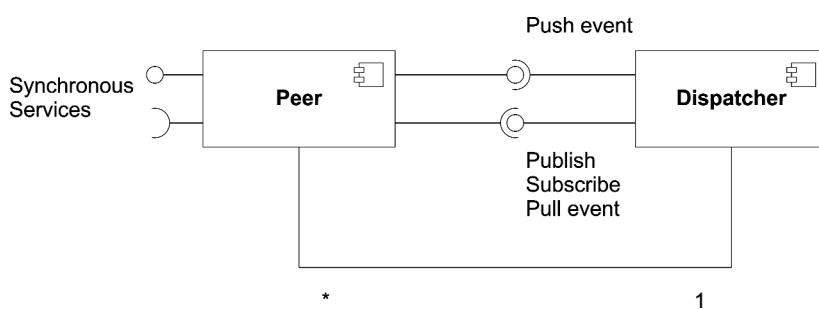


Figura 7.10 Architettura composita publish-subscribe/p2p.

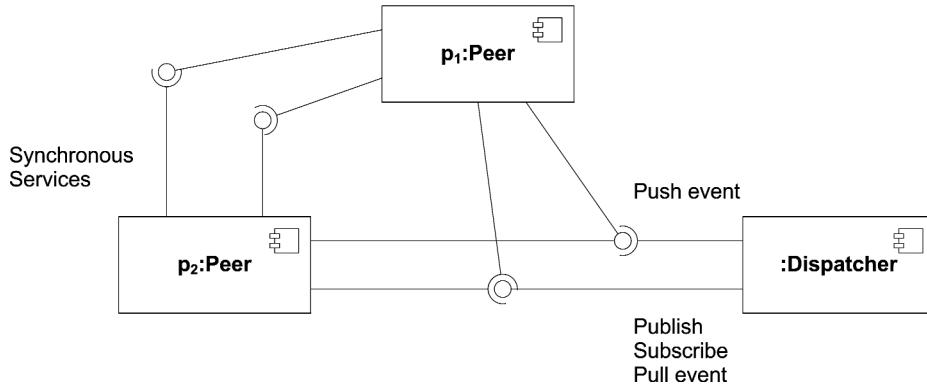


Figura 7.11 Esempio di sistema basato sull'architettura della Figura 7.10.

7.6 Design pattern

Gli stili architetturali non esauriscono i possibili tipi di schemi a disposizione di un ingegnere del software. Quando si passa dal livello di astrazione del sistema nel suo complesso alla descrizione del singolo elemento di un'architettura software (o di una sua parte significativa) diviene molto utile riutilizzare una serie di schemi che vengono in letteratura identificati con il termine *design pattern*. Il numero e la tipologia di pattern oggi disponibile è molto elevato. Vi sono siti e pubblicazioni che offrono cataloghi di pattern di varia natura. In generale, un pattern viene descritto con diagrammi UML e spesso, insieme viene anche fornito uno scheletro del codice Java o C++ che ne implementa le caratteristiche e funzioni. Come esempio, nel seguito viene presentato il design pattern model-view-controller, particolarmente noto e diffuso.

Il pattern *model-view-controller* descrive la tipica struttura di un programma che permette a un utente di modificare in modo interattivo un insieme complesso di informazioni visualizzato secondo diverse modalità. È il caso di tutti gli strumenti interattivi come wordprocessor, spreadsheet, grafica ecc. In tutti questi casi, lo strumento abilita la creazione e/o modifica di una struttura informativa complessa, memorizzata secondo formati non direttamente manipolabili dall'utente. La struttura informativa viene resa visibile e/o manipolabile all'utente secondo diverse view. Nel caso dello spreadsheet, l'utente può manipolare i dati utilizzando la classica visione a matrice (si veda la Figura 7.12); oppure può accedervi utilizzando una form (si veda la Figura 7.13); infine può visualizzare i dati utilizzando un grafico come quello della Figura 7.14. L'insieme dei dati è lo stesso; cambia la modalità di visualizzazione.

La costruzione di programmi di questo tipo richiede una serie di accorgimenti. Non ha senso duplicare i dati in funzione della view che deve essere mostrata. Anzi, è auspicabile vi sia un'unica copia dei dati dalla quale ricavare di volta in volta le diverse view

144 Capitolo 7 Stili architettonici e design pattern

	A	B	C	D	E
1		Colonna A	Colonna B	Colonna C	
2	Dato 1	12		4	22
3	Dato 2		6	35	17
4	Dato 3		28	23	18
5					
6					

Figura 7.12 Un semplice spreadsheet.

in funzione delle richieste dell'utente. Inoltre, la modifica di un dato in una view deve riflettersi in un cambiamento coerente di tutte le altre view costruite a partire da quella stessa informazione.

Il pattern model-view-controller suggerisce la struttura di questo tipo di programmi. In particolare, identifica tre elementi fondamentali (si veda la Figura 7.15).

- *Model:* è il componente che contiene tutti i dati gestiti dall'applicazione (per esempio, i dati che costituiscono lo spreadsheet).

Figura 7.13 Modifica dello spreadsheet tramite form.

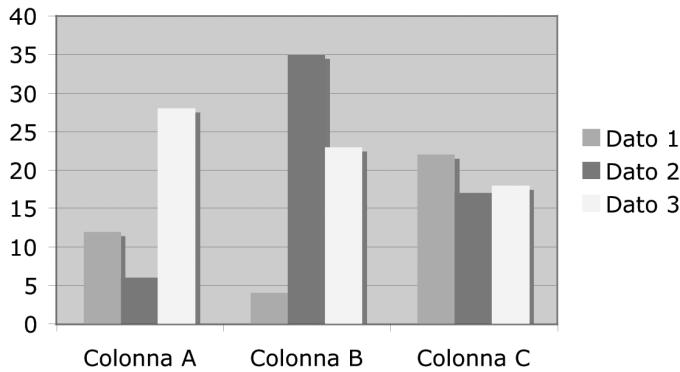


Figura 7.14 Un semplice grafico.

- *View*: è il componente che ha il compito di mostrare i dati all’utente. Poiché ci potrebbero essere diverse view (come nel caso dello spreadsheet), un’applicazione può avere più di un componente di questo tipo (in generale, uno per view).
- *Controller*: è il componente che riceve le richieste da parte dell’utente e le traduce in operazioni di modifica del modello.

Le view sono aggiornate secondo due meccanismi: passivo e attivo. Nel primo caso, è l’utente che in qualche modo richiede di aggiornare la view (*Query*); nel secondo, il modello invia una richiesta *UpdateView* alla view (o alle view) ogniqualvolta vi è un cambiamento nei dati del modello stesso.

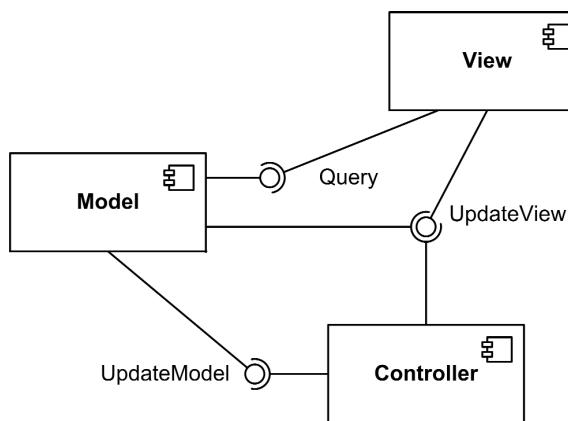


Figura 7.15 Pattern model-view-controller.

Le operazioni di modifica dei dati sono generate dall'utente attraverso il controller che ha il compito da un lato di aggiornare il modello (`UpdateModel`) e dall'altro di provocare l'aggiornamento relativo delle view (`UpdateView`). Si noti che quanto presentato è un esempio piuttosto semplice di pattern model-view-controller. In letteratura sono disponibili proposte più elaborate e ricche di funzioni, in grado quindi di implementare scenari di interazione più complessi e sofisticati.

7.7 Riferimenti bibliografici

Le architetture client-server e p2p sono descritte in miriadi di articoli e pubblicazioni. Un buon punto di partenza per approfondirne lo studio sono le voci relative di Wikipedia. Lo stile publish-subscribe è stato introdotto in Reiss [1990]. Gli stili relativi al codice mobile sono stati descritti e discussi in Fuggetta, Picco, et al. [1998]. Lo stile REST, che descrive il funzionamento del Web, è presentato in Fielding e Taylor [2002].

Per quanto concerne i design pattern, oltre ai moltissimi siti che ne presentano caratteristiche e implementazioni, il testo di riferimento rimane Gamma et al. [1995]. Buschmann et al. [1996] descrive e discute un numero rilevante di stili architetturali, design pattern e programming idiom. Una presentazione sintetica ed efficace del pattern model-view-controller è contenuta in Wikipedia alla voce relativa.

Capitolo 8

Cicli di vita e gestione dei progetti

Nei precedenti capitoli sono stati presentati i metodi e le tecniche che aiutano l'ingegnere del software durante le attività di analisi e descrizione del problema e di sviluppo della soluzione. Queste operazioni sono particolarmente complesse e vengono condotte, in generale, da team di persone su archi di tempo anche molto significativi. Per questi motivi, è essenziale approfondire i problemi e gli strumenti relativi alla gestione del processo di sviluppo nel suo complesso.

Questo capitolo fornisce un'introduzione alle diverse tematiche che sono riconducibili alla gestione del processo di sviluppo del software.

1. Il processo software e i cicli di vita: paradigmi e schemi secondo i quali organizzare le attività di sviluppo e gestione di un sistema informatico complesso.
2. Pianificazione e gestione di progetto: tecniche per organizzare le attività dei progettisti e verificarne lo stato di avanzamento.
3. Configuration management: tecniche e strumenti per la gestione del prodotto software.

Come tutti i prodotti complessi, anche il prodotto software è il frutto di un processo particolare, il processo di sviluppo del software o processo software. In generale, un *processo* è definito come un'insieme di attività concentrate nel tempo finalizzate alla realizzazione di un particolare output; ha un inizio e una fine e produce un qualche risultato tangibile in termini di prodotto o di servizio offerto. Un *processo software* è definito come un insieme strutturato di attività che porta alla costruzione di un prodotto software da parte di un team di sviluppo, utilizzando metodi, tecniche, metodologie e strumenti dell'ingegneria del software. Così come discusso nel Capitolo 3, un *prodotto software* è l'insieme di tutti i semilavorati o artefatti che permettono l'utilizzo di un programma da parte di un utente: codice, documentazione e prodotti intermedi quali casi di test o manuali tecnici.

In un'azienda, lo stesso processo software è di norma seguito da gruppi distinti che hanno il compito di sviluppare diversi prodotti o differenti componenti di uno stesso prodotto. Un *progetto* è una particolare istanza di un processo software.

Così come l'architettura di un sistema informatico è normalmente basata su uno o più stili architettonici, un processo software può essere organizzato utilizzando i *cicli di vita del software*. Un *ciclo di vita* definisce la struttura di massima di un processo software, attraverso l'indicazione delle principali fasi secondo le quali deve articolarsi e i criteri secondo i quali si succedono. Un ciclo di vita non definisce nel dettaglio come devono svolgersi le attività dell'ingegnere del software: è uno schema di lavoro (ancora una volta si parla di schemi!) che viene usato come riferimento nella definizione del processo di sviluppo vero e proprio. Ogni fase prevede il completamento di uno o più deliverable. Un *deliverable* è un prodotto tangibile e verificabile, come ad esempio un documento o un componente informatico.

In sintesi, i concetti di ciclo di vita, processo software e progetto sono tra loro legati in modo analogo a quanto visto per stile architettonico, architettura e installazione (si veda la Tabella 8.1). Il ciclo di vita esprime una logica complessiva secondo la quale condurre le attività di sviluppo, così come uno stile architettonico definisce le caratteristiche di massima di un'architettura. Un processo è una precisa e ordinata organizzazione di attività, così come un'architettura identifica in modo preciso i componenti software di un particolare sistema informatico. Infine, un progetto è un'istanza di un processo, così come un'installazione di un prodotto software complesso costituisce la messa in esercizio di una particolare architettura: ci possono essere molteplici installazioni (con parametri e caratteristiche differenti) della stessa soluzione e quindi di una stessa architettura.

In sintesi, di norma sono definiti diversi processi software per differenti tipologie di prodotti. Ciascun processo è basato su uno specifico ciclo di vita e identifica nel concreto quali attività devono essere condotte, secondo quale temporizzazione e da quale struttura organizzativa di quella specifica azienda. Per ogni diversa tipologia di processo potrebbero essere attivi più progetti, tutti costruiti secondo quanto previsto dal processo di riferimento, ma ovviamente con tempi, deliverable e risorse che cambiano in funzione dello specifico prodotto da rilasciare.

8.1 Il processo software e i cicli di vita

Nel corso degli anni sono stati sviluppati diversi cicli di vita. In questo capitolo ne verranno presentati e discussi alcuni tra i più noti.

- A cascata.
- A spirale.
- A componenti.
- Unified Process (Iterativo-evolutivo).
- Modelli agili ed extreme programming.

8.1 Il processo software e i cicli di vita 149

Processo	Prodotto
Ciclo di vita	Stile architetture
Processo software	Architettura software
Progetto	Installazione

Tabella 8.1 Processo e prodotto.

Il *ciclo di vita a cascata* (*waterfall*) è stato il primo ciclo di vita studiato nell'ingegneria del software (Figura 8.1). Concepito negli anni '70, introduceva una serie di concetti e idee che sono tuttora alla base di molti altri cicli di vita e delle tecniche utilizzate dagli ingegneri del software. Malgrado presenti dei limiti che lo rendono poco utilizzabile in molte situazioni, ha ancora un ruolo importante in quanto permette di introdurre una serie di concetti e termini di validità generale.

Il ciclo di vita a cascata è strutturato secondo una *serie di fasi* organizzate in maniera sequenziale. Inoltre, il ciclo di vita prevede che si possa accedere a una determinata fase solo nel momento in cui quella precedente è terminata e si sono prodotti e validati tutti i deliverable che ne costituiscono il risultato atteso.

Le fasi del ciclo di vita a cascata sono state descritte da diversi autori in vario modo. Nonostante in letteratura si possano trovare descrizioni che differiscono sul numero e la natura delle fasi, è comunque possibile identificare uno schema di massima ragionevolmente condiviso.

- *Studio di fattibilità*: in questa fase si identificano le caratteristiche generali del problema, si analizzano le possibili strategie per la sua soluzione e si effettua una valutazione di massima dei tempi e costi richiesti. Inoltre, per ogni possibile strategia di risoluzione si valutano rischi e opportunità.
- *Descrizione del problema*: è la fase di studio dello spazio del problema. Si analizza il dominio applicativo e si raccolgono i requisiti dell'utente. Sulla base di queste informazioni vengono prodotte le specifiche (o descrizione dell'interfaccia).
- *Progettazione della soluzione*: in questa fase è definita l'architettura e la struttura della soluzione software. In particolare, si definiscono e specificano i componenti, i moduli, le interfacce per la comunicazione e le relazioni tra i diversi componenti.
- *Sviluppo e test di unità*: in questa fase si implementa l'architettura progettata nella fase precedente. Durante lo sviluppo sono eseguiti i test per individuare gli errori presenti nel codice e per verificare che i singoli moduli esibiscano il comportamento preventivato.
- *Integrazione e test di sistema*: in questa fase si integrano i diversi moduli implementati nella fase precedente. La comunicazione tra i moduli ottenuta tramite le interfacce è un aspetto critico della fase di integrazione: la correttezza della comunicazione va attentamente verificata attraverso specifici test.

150 Capitolo 8 Cicli di vita e gestione dei progetti

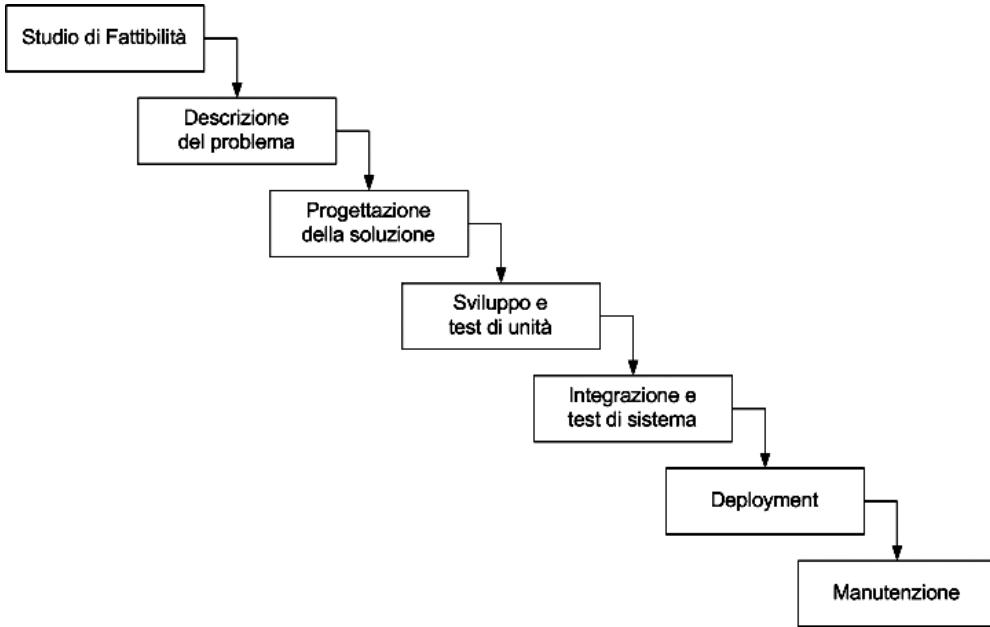


Figura 8.1 Ciclo di vita a cascata.

- *Deployment*: questa fase prevede la distribuzione e installazione del software presso l'utente.
- *Manutenzione*: in questa fase, che segue la distribuzione del prodotto presso il cliente, si garantiscono le correzioni e le operazioni di manutenzione sul prodotto software.

L'introduzione del ciclo di vita a cascata ha i suoi meriti: per la prima volta, infatti, è stata data un'organizzazione al processo di sviluppo di un prodotto software. Sono stati definiti concetti come quello di *deliverable* di un'attività: alla fine di ogni attività è necessario produrre un output dei risultati ottenuti e del lavoro svolto. Inoltre, il ciclo di vita a cascata è stato il primo a evidenziare il peso delle attività che precedono la scrittura del codice: l'implementazione del codice, infatti, è solo una parte del complesso processo di produzione del software.

In generale, il ciclo di vita a cascata nasce come reazione alla mancanza di struttura e organizzazione che aveva caratterizzato l'attività di sviluppo del software negli anni '60. Ciò ha portato alla concezione di un modello molto strutturato e rigido che introduce ordine e metodo in quello che era stato fino ad allora un approccio basato semplicemente sulle capacità delle singole persone. Alcune caratteristiche del ciclo di vita a cascata sono però ormai superate e non si adattano più alla realtà odierna: ad esempio, con le sue atti-

8.1 Il processo software e i cicli di vita 151

vità organizzate in modo strettamente sequenziale, non considera la possibilità di tornare indietro e rifare un'attività (o parte di essa) nel momento in cui ci sia necessità di correggere o modificare il lavoro svolto. Inoltre, non coinvolge abbastanza il cliente finale: nel ciclo di vita a cascata, infatti, egli è coinvolto solo nelle primissime fasi del processo (descrizione del problema) e ricompare solo alla fine quando il prodotto deve essere consegnato e installato. Questo fa sì che a volte non sia possibile soddisfare completamente le esigenze del cliente, perché all'inizio del processo è difficile avere già chiare le funzionalità effettivamente necessarie. Infine, il ciclo di vita a cascata non dà il giusto peso alle attività di manutenzione ed evoluzione del software che invece hanno assunto un ruolo sempre più importante al crescere della diffusione e complessità dei sistemi informatici.

Nel corso degli anni sono stati ideati altri cicli di vita con l'intenzione di affrontare e risolvere gli aspetti problematici di quello a cascata. In particolare, si è passati da una visione senza ricicli (considerati addirittura dannosi) a un'attitudine che riconosce l'importanza di procedere per evoluzioni e incrementi nella costruzione della soluzione finale. Un *ciclo di vita iterativo* è composto da una serie di iterazioni al termine delle quali è sempre prodotto un particolare deliverable. Le *iterazioni* sono mini-progetti formati da una ripetizione di fasi: le attività e i deliverable prodotti cambiano in funzione dello stato di avanzamento del progetto.

Il più famoso ciclo di vita di tipo iterativo è quello a spirale. Il *ciclo di vita a spirale* conserva gli aspetti sistematici di quello a cascata, ma cerca di ridurre la possibilità di errori durante il processo software e, nel caso in cui si verifichino, di riuscire a contenerne i danni. Per far questo, il ciclo di vita a spirale prevede la realizzazione del prodotto software attraverso più rilasci incrementalni. Le attività che portano alla creazione di un rilascio, sono ciclicamente ripetute per ogni versione successiva, secondo una spirale. Al termine delle prime iterazioni, i rilasci prodotti saranno documenti o prototipi; al termine delle ultime iterazioni saranno le versioni sempre più complete del prodotto finale.

Il ciclo di vita a spirale è caratterizzato da una struttura comune di tutte le iterazioni effettuate: tale struttura è composta da *task region*. Ogni task region è costituita da un insieme di attività che possono variare in funzione del grado di avanzamento dell'intero processo. Nella Figura 8.2 è rappresentata la versione originale del ciclo di vita a spirale. Le task region rappresentate sono quattro.

- *Definizione degli obiettivi, delle alternative e dei vincoli.* È la task region che indica l'inizio dell'intero processo e di ogni iterazione: le iterazioni successive devono considerare quelle precedenti e definire i propri obiettivi anche sulla base di quanto già fatto.
- *Valutazione delle alternative, identificazione e risoluzione dei rischi.* Le attività di analisi del rischio sono una novità rispetto al ciclo di vita a cascata. La realizzazione di un prototipo fin dall'inizio dell'iterazione permette di valutare meglio i rischi che la scelta di ogni alternativa comporta.
- *Sviluppo e verifica.* Le attività che caratterizzano questa task region variano a seconda dell'iterazione considerata. Al progredire del progetto, le attività riguardano l'analisi dei requisiti e la definizione delle specifiche, la progettazione dell'architettura,

152 Capitolo 8 Cicli di vita e gestione dei progetti

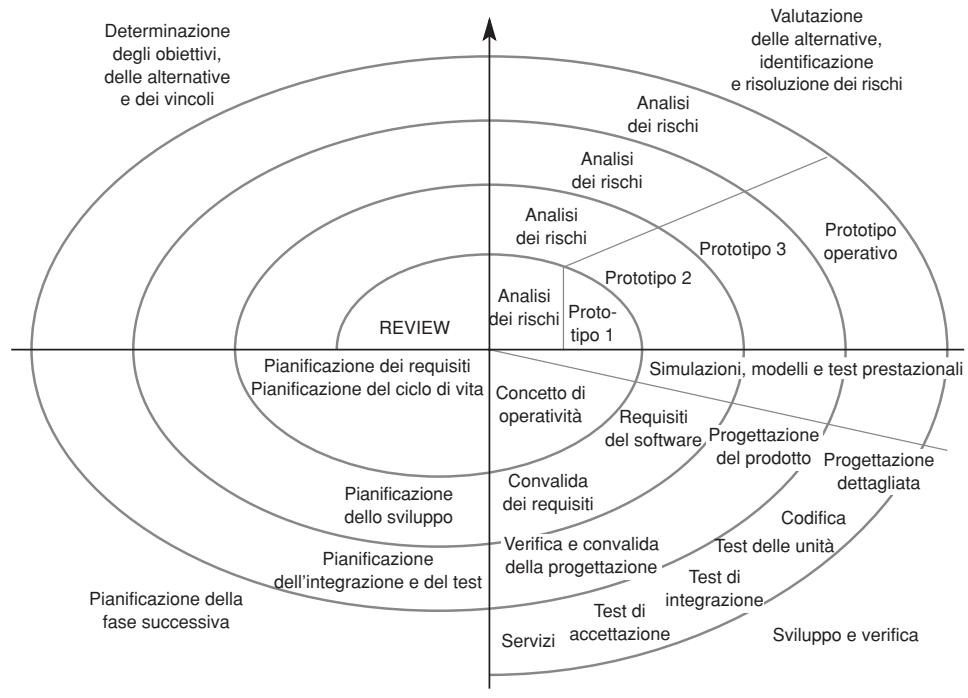


Figura 8.2 Ciclo di vita a spirale di Boehm (© IEEE, 1988).

ra, la progettazione di dettaglio, l'implementazione del prodotto e, infine, i test sul prodotto realizzato. Di volta in volta si produce un risultato che viene sottoposto a verifica.

- *Pianificazione della fase successiva.* Questa task region ha lo scopo di ottimizzare l'uso di risorse e il tempo a disposizione del progetto. Ripetere la pianificazione a ogni iterazione permette di adattare le iterazioni successive alla reale evoluzione del progetto, evitando di cristallizzarsi su attività e output non più aderenti alla realtà.

Il ciclo di vita a spirale qui presentato è quello definito originariamente da Boehm: tuttavia, ne sono derivate molte varianti che differiscono dalla struttura originale in particolare per quanto riguarda le attività che formano le task region. Lo stesso Boehm nel 1998 ha modificato il suo modello per gestire i cambiamenti e le evoluzioni del processo di produzione di un prodotto software, ottenendo la variante mostrata nella Figura 8.3. In essa le task region sono le seguenti.

- *Comunicazione con il cliente:* fase di comunicazione e confronto con il cliente: all'inizio di ogni iterazione, si discutono i requisiti desiderati dal cliente.

8.1 Il processo software e i cicli di vita 153

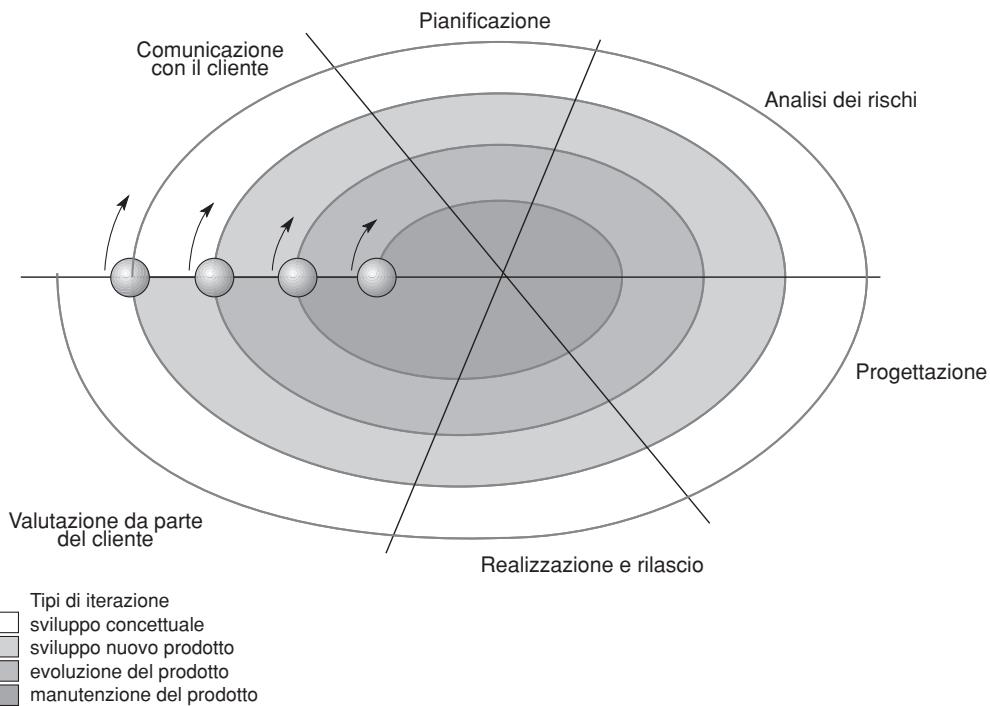


Figura 8.3 Modello a spirale di Boehm del 1998.

- *Pianificazione*: in questa fase sono pianificate le attività da svolgere, le tempistiche e le risorse da usare nell'iterazione considerata. La ripianificazione ripetuta a ogni iterazione è fondamentale per garantire un uso efficace ed efficiente delle risorse al variare delle caratteristiche del progetto. Spesso, infatti, un progetto non evolve secondo i piani prestabiliti perché la realtà nella quale viene a collocarsi cambia in modo significativo nel corso della sua vita.
- *Analisi dei rischi*: stima e prevenzione dei rischi tecnici e di gestione. L'analisi del rischio è stata mantenuta anche in questa evoluzione del modello originario perché ovviamente molto utile.
- *Progettazione*: è la task region in cui si progetta e si struttura il nuovo rilascio, alla luce di quanto emerso durante l'iterazione. Infatti, prendendo in considerazione i risultati del confronto con i clienti e le nuove analisi del rischio, a volte si rende necessario re-ingegnerizzare la soluzione, vale a dire ripensare o ristrutturare il prototipo o la versione da fornire rispetto ai precedenti rilasci.

154 Capitolo 8 Cicli di vita e gestione dei progetti

- *Realizzazione e rilascio:* in questa fase è realizzato quanto definito in fase di progettazione; l'implementazione del rilascio segue le direttive emerse durante la fase di progettazione. Oltre allo sviluppo del rilascio questa task region si occupa anche del test.
- *Valutazione da parte del cliente:* rilevazione delle reazioni da parte del cliente. Nel nuovo modello ideato da Boehm è confermata l'importanza della collaborazione con il cliente, in modo da prevenire e ridurre possibili problemi di fraintendimento o incomprensione degli obiettivi. Le considerazioni e le valutazioni che il cliente fornisce sul lavoro svolto durante l'iterazione sono un punto di partenza da considerare attentamente nell'iterazione successiva.

Il modello di ciclo di vita a spirale descrive al meglio il processo di realizzazione di prodotti software complessi e di notevoli dimensioni. La presenza di task region garantisce la sistematicità propria del ciclo di vita a cascata. Uno dei vantaggi principali di questo modello è l'introduzione dell'analisi del rischio a ogni livello di iterazione. La presenza dell'analisi del rischio però è anche un punto critico perché il successo del progetto dipende in gran parte dalla corretta analisi dei rischi: essa influenza in modo significativo le decisioni prese nel progetto, non solo a livello implementativo.

Il *ciclo di vita basato su componenti* è considerato un'applicazione particolare del ciclo di vita a spirale appena visto. In questo contesto, il componente è inteso con l'accezione di unità di riuso. Il ciclo di vita a componenti è evolutivo e caratterizzato da una serie di iterazioni. Rispetto a quello a spirale, si differenzia nelle task region Progettazione e Realizzazione e rilascio, perché la progettazione e strutturazione della soluzione è focalizzata sui componenti. In particolare, le attività previste in queste due task region sono le seguenti.

- *Identificazione dei componenti necessari per la costruzione della soluzione.* La struttura della soluzione si basa su componenti: il primo passo consiste quindi nel definire quelli che faranno parte del rilascio e della soluzione in generale.
- *Ricerca dei componenti nelle librerie o presso vendor.* Definiti i componenti che costituiscono la soluzione, in questo secondo passo si cerca di capire se ne esistono già alcuni che si possono riusare.
- *Eventuale realizzazione del componente.* Nel caso in cui alcuni componenti non fossero già disponibili né all'interno dell'azienda né sul mercato, è necessario implementarli ex-novo.
- *Integrazione del componente nell'architettura progettata.* Il componente progettato durante l'iterazione o riusato deve essere integrato nell'architettura complessiva progettata.
- *Testing del componente.* Il test deve riguardare non solo il controllo del corretto funzionamento del componente, ma soprattutto la sua corretta integrazione con gli altri componenti della soluzione.

Il ciclo di vita basato su componenti permette un buon riuso del software: in questo modo si ottiene una riduzione del tempo di sviluppo del prodotto software e un'ottimizzazione delle risorse usate.

8.1 Il processo software e i cicli di vita 155

Uno dei modelli iterativi più significativi è lo *Unified Process* (UP) e il suo esempio più famoso, il *Rational Unified Process* (RUP). UP è un modello che definisce una struttura generica di processo: tale struttura è poi specializzata per un'ampia classe di sistemi software, per aree applicative, livelli di competenza e dimensioni del progetto. In questo modello ricopre un'importanza rilevante il linguaggio UML perché è usato per descrivere molti degli artefatti prodotti nel corso del progetto. Il processo RUP è stato definito dalla società Rational sulla base delle esperienze acquisite nella definizione e nell'uso di UML.

UP è iterativo e incrementale: è scomposto in progetti di dimensioni e complessità minori da svolgere in singole iterazioni. Al termine di ogni iterazione è necessario un momento di verifica e, se l'esito è positivo, il progetto passa alla iterazione successiva. Inoltre, è fondamentale avere un feedback di quanto fatto nel corso dell'iterazione, in modo da assecondare eventuali cambiamenti occorsi durante il progetto o modificare obiettivi e requisiti andando incontro alle richieste del cliente. Al termine di ogni iterazione è prodotto un deliverable frutto del lavoro svolto: il deliverable può essere, ad esempio, un documento o un prototipo, in funzione del grado di maturità del progetto stesso.

Il modello UP è strutturato secondo due diverse dimensioni:

1. una dimensione temporale suddivisa in fasi che determina la vita e la maturità del progetto (è la sequenza di iterazioni citate in precedenza);
2. una dimensione di workflow del processo, articolata in attività ripetute in ogni iterazione di ciascuna fase (definisce la struttura di ogni singola iterazione).

La fasi del modello UP sono le seguenti (mostrate nella Figura 8.4).

- *Inception*. In questa prima fase, le diverse iterazioni che si eseguono definiscono l'ambito e l'obiettivo del progetto. In particolare, sono identificate le entità e le interazioni attraverso le quali le entità stesse interagiscono con il sistema: l'identificazione di entità e interazioni è supportata dalla redazione dei diagrammi UML dei casi di uso. Al termine della fase, i risultati prodotti dalle iterazioni saranno: il documento di vision, il documento di business case ed eventuali prototipi. In particolare, il documento di vision contiene le principali caratteristiche, i vincoli e i requisiti del sistema, mentre il documento di business case descrive una prima analisi dei rischi e una prima versione del piano di progetto discusse nel seguito (si veda il Paragrafo 8.2.1).
- *Elaboration*. In questa fase si stabilizzano i requisiti e di conseguenza la descrizione dello spazio del problema. Inoltre, si definisce l'architettura di base del sistema: in questo modo è possibile fare una stima realistica di tempi e costi necessari per la realizzazione del software.
- *Construction*. Questa fase è la più consistente e quella che prevede un maggior numero di iterazioni: comprende, infatti, la realizzazione del prodotto software e l'eventuale evoluzione dell'architettura fino al rilascio finale.

156 Capitolo 8 Cicli di vita e gestione dei progetti

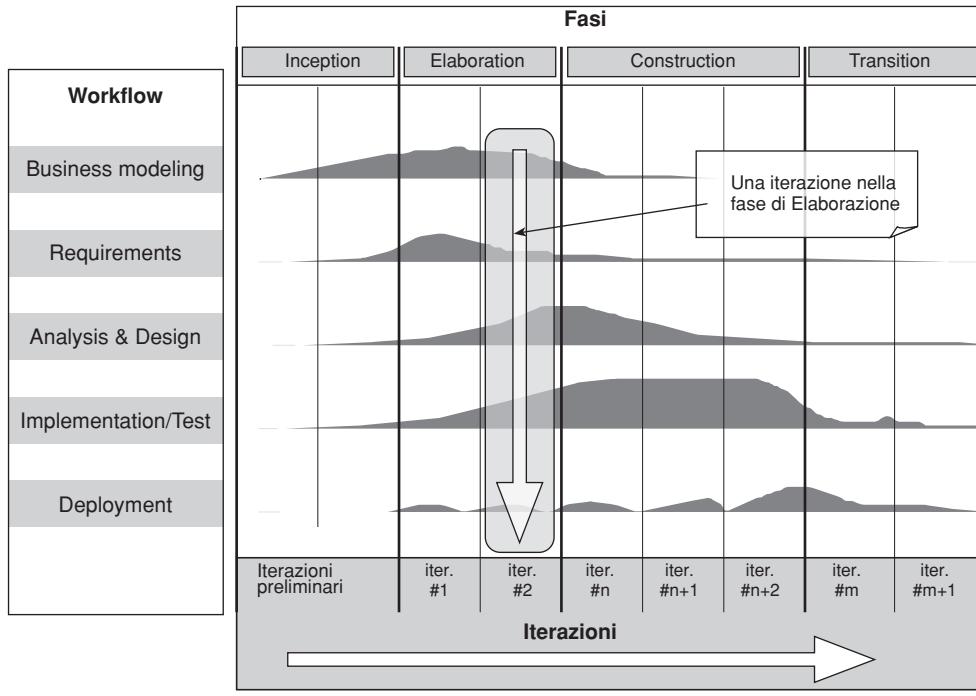


Figura 8.4 Modello Unified Process (UP).

- *Transition.* In questa fase si eseguono le attività successive allo sviluppo vero e proprio. In particolare, le iterazioni eseguite durante la fase di Transition prevedono la correzione dei difetti, la consegna del prodotto e la formazione degli utenti che dovranno interagire con il prodotto.

Le attività individuate a ogni iterazione sono le seguenti.

- *Business modeling:* comprensione dell'organizzazione del progetto e definizione dei requisiti di sistema ad alto livello.
- *Requirements:* analisi, definizione e formalizzazione dei requisiti.
- *Analysis & Design:* analisi e progettazione della soluzione.
- *Implementation:* implementazione della soluzione identificata.
- *Test:* controllo e verifica di quanto realizzato durante l'iterazione.
- *Deployment:* rilascio e installazione presso il cliente della soluzione software sviluppata.

8.1 Il processo software e i cicli di vita 157

In generale, queste attività sono ripetute a ogni iterazione e per ogni fase. Il peso di ogni attività in termini di risorse e tempi dipende dalla fase in cui si trova il progetto. Come mostrato nella Figura 8.4, nella fase iniziale di Inception le attività con un peso preponderante saranno quelle di raccolta e analisi dei requisiti, oltre all'attività di progettazione della soluzione. In questa fase, le attività di sviluppo saranno ridotte e principalmente dedicate allo sviluppo di prototipi o di componenti di tipo generale che sono strumentali alla verifica con l'utente dei requisiti. Al contrario, nella fase di Construction, quando il progetto è a uno stadio avanzato, i requisiti saranno ragionevolmente stabili e l'attività di implementazione della soluzione sarà preponderante. Le attività descritte in precedenza sono affiancate da quelle di gestione e controllo che si svolgono durante tutto il corso del progetto e ne accompagnano l'evoluzione.

I cicli di vita presentati hanno delle caratteristiche comuni: tutti prevedono la redazione di una serie di deliverable intermedi, in particolare documenti, che descrivono i risultati ottenuti durante il processo. Inoltre, sono presenti e acquistano un'importanza di rilievo le attività di pianificazione e gestione del progetto che affiancano quelle più marcatamente orientate alla realizzazione del software. Queste caratteristiche sottolineano come un progetto software non si possa ridurre alla sola implementazione di codice. A volte, tuttavia, queste fasi assumono un peso ritenuto fin troppo rilevante. Per questo motivo, negli ultimi anni sono stati definiti nuovi cicli di vita, complessivamente identificati con l'espressione Agile Software Development.

L'Agile Software Development è un approccio che punta l'attenzione verso la capacità di rispondere in modo veloce ed efficace ai cambiamenti che caratterizzano il processo di sviluppo del software. Nel 2001 è stato pubblicato un manifesto dell'Agile Software Development che ne descrive i principi fondanti e i valori. Il termine *agile* fa riferimento, in generale, alla capacità di reazione al cambiamento, come ad esempio la capacità di adattare la soluzione software in base ai cambiamenti di requisiti richiesti dall'utente, e alla flessibilità del lavoro del team durante l'intero progetto. Nel manifesto sono indicati gli aspetti che differenziano lo sviluppo agile rispetto ai modelli esistenti.

- *Lo sviluppo agile si focalizza sulle persone e sull'interazione piuttosto che sul processo e i tool.* Le metodologie della programmazione agile puntano la loro attenzione sulle singole persone che compongono il team e non tanto sul ruolo che queste hanno all'interno del gruppo. Inoltre, cercano di incoraggiare le interazioni e la comunicazione tra persone per facilitare la formulazione di nuove idee e quindi di nuove soluzioni.
- *L'enfasi è sulla realizzazione del software piuttosto che sulla redazione di una documentazione esaustiva.* La misura del raggiungimento del risultato è data, secondo il manifesto, dall'esecuzione del software realizzato e non dalla documentazione prodotta. È il software che parla da sé.
- *L'aspetto essenziale è la collaborazione con l'utente e non la negoziazione del contratto.* Questo valore pone l'attenzione sul fatto che la cooperazione tra il team che deve pensare e realizzare la soluzione software e il cliente è fondamentale e deve essere costante durante tutto il processo di sviluppo del software. Lo sforzo dello sviluppo agile è di creare un clima di totale collaborazione con il cliente.

- È essenziale rispondere al cambiamento piuttosto che seguire in modo rigido un piano di progetto preordinato. L'importanza di seguire una tabella di marcia, una pianificazione decisa all'inizio del progetto, è ridimensionata per permettere al team di seguire con la dovuta flessibilità i cambiamenti dettati dall'evoluzione del progetto o dalla variazione dei requisiti.

Operativamente, quindi, le metodologie della programmazione agile ridimensionano l'importanza e lo sforzo impiegato nelle attività preliminari del progetto, come ad esempio, la raccolta dei requisiti, la progettazione della soluzione e in particolare dell'architettura di base. Ciò non significa che queste attività non siano eseguite, ma che il tempo e lo sforzo dedicati sono minori rispetto ai modelli di cicli di vita presentati precedentemente. Inoltre, nella programmazione agile il tempo per la stesura della documentazione di progetto è notevolmente ridimensionato.

Un esempio di metodo agile è l'*eXtreme Programming* (XP). Esso è caratterizzato da quattro valori chiave: *semplicità, feedback, comunicazione e coraggio*. Questo approccio, infatti, si focalizza su una progettazione della soluzione il più semplice possibile, inclusiva delle sole funzionalità che di volta in volta il team deve rilasciare all'utente e non alla progettazione completa dell'architettura. L'ideazione e la pianificazione della soluzione avvengono tramite *user story*. Una *user story* è la descrizione, semplice e breve, di una particolare funzionalità richiesta dall'utente: il team di sviluppo analizza tutte le *user story* e decide in che ordine implementarle. Il feedback è ottenuto dal testing continuo eseguito sul codice: spesso i casi di test sono pensati prima dell'implementazione del codice che viene quindi realizzato per rispondere al test definito. Il feedback, inoltre, è garantito dai continui e frequenti rilasci all'utente: in questo modo l'utente può testare il prodotto e verificare eventualmente i propri requisiti. Il team, ricevendo i frequenti feedback dell'utente, può adattare i rilasci successivi in modo flessibile anche in una fase avanzata del progetto.

In un processo agile, la comunicazione assume un ruolo fondamentale: non solo quella continua e diretta con il cliente, ma anche la comunicazione tra i partecipanti al team di sviluppo. L'interazione con l'utente finale è continua tanto da prevedere la presenza dell'utente stesso mentre si procede alla realizzazione del software. L'importanza della comunicazione all'interno del team è alla base della scelta dell'*extreme programming* secondo la quale due programmatore devono lavorare sulla stessa macchina in modo da realizzare una sorta di controllo e testing durante l'implementazione stessa del codice (il cosiddetto *pairwise programming*): due persone che operano sulla stessa porzione di codice possono essere molto più produttive di due che lavorano in modo indipendente.

Dopo aver esaminato le caratteristiche dei principali cicli di vita, va notato che non esiste un ciclo di vita "migliore" in assoluto. L'ingegnere del software dovrà disegnare il proprio processo di sviluppo ispirandosi ai diversi cicli di vita possibili, valutandone di volta in volta i pro e contro. Come nel caso delle architetture software, l'ingegnere non si limita ad applicare in modo acritico questo o quel modello, ma deve raccordarlo alle caratteristiche del problema di cui si sta occupando.

8.2 Pianificazione di progetto e controllo di avanzamento

Nella costruzione di un prodotto software, oltre a scegliere il ciclo di vita di riferimento e a definire nel dettaglio il processo che dovrà essere seguito, è essenziale pianificare lo svolgersi delle attività di sviluppo, al fine di garantirne la riuscita con tempi e costi accettabili. Il ciclo di vita (e il processo definito in base a esso) fornisce un'indicazione di massima sulle fasi da realizzare e sulla loro organizzazione: non dà indicazioni, invece, su quali saranno i tempi e costi che dovranno essere sostenuti e sulle risorse che si renderanno necessarie per la realizzazione del prodotto. La *pianificazione* ha come obiettivo l'organizzazione di un progetto software, e in particolare i vincoli di tempo, costi e risorse assegnati. L'output del processo di pianificazione è il *piano di progetto*: esso descrive il percorso da seguire per raggiungere i risultati finali preventivati.

Il piano di progetto non è solo una guida per la realizzazione efficiente del prodotto software, ma è anche uno strumento di controllo sull'andamento del progetto stesso. Lo scostamento tra il piano e il reale stato di avanzamento del progetto è il punto di partenza per valutare eventuali azioni correttive e per aggiornare il piano secondo nuove esigenze. Inoltre, il piano, esplicitando il processo, e in particolare le risorse dell'azienda usate, può diventare la base per verificare come queste ultime siano legate agli obiettivi strategici aziendali. La redazione precisa e formale di un piano di progetto assume, quindi, un'importanza rilevante per l'intera azienda. Certamente, alcuni cicli di vita (specie quelli basati sugli approcci agili) tendono a limitare, se non addirittura a eliminare, le attività di pianificazione. Sarà compito dell'ingegnere del software identificare di volta in volta un ragionevole equilibrio tra il bisogno di prevedere e controllare le attività di sviluppo e il contenimento di quelle che non sono direttamente collegate alla produzione e al rilascio del software.

Il piano di progetto si articola in una sequenza di attività che realizzano il processo di riferimento (e, di riflesso, il ciclo o i cicli di vita ai quali esso si ispira). Un'*attività* è un insieme definito e preciso di operazioni che dato un particolare input forniscono un output specifico. Una *risorsa* è ciò che viene usato durante un'attività e che concorre alla realizzazione del relativo output. Ad esempio, sono risorse i computer che si usano o i materiali per realizzare eventuale hardware. In un progetto di sviluppo di software, la tipologia di risorsa più importante è in generale quella umana, cioè le persone che lavorano al progetto, partecipano alle riunioni, redigono i documenti, realizzano il codice, effettuano i test, interagiscono con il cliente. Il *costo* di un progetto è dato dalla somma dei costi, calcolati opportunamente, delle diverse risorse utilizzate. Nel caso di prodotti software, il costo maggiore è dato dalle risorse umane, e in particolare da chi lavora sul progetto e da quanto tempo vi dedica. Altre fonti di costo sono i servizi (per esempio i viaggi), le risorse informatiche necessarie e i costi di struttura (spazi, segreteria, affitto, energia, telefono ecc.). I costi sono influenzati da fattori come la dimensione del software da sviluppare, la complessità di programmi e algoritmi, la stabilità dei requisiti.

Un concetto chiave è il *tempo* impiegato per la realizzazione di un'attività (e del progetto nel suo complesso) ovvero la *durata del progetto*. Il tempo, o meglio la sua di-

160 Capitolo 8 Cicli di vita e gestione dei progetti

sponibilità limitata, è uno degli aspetti critici da considerare durante la pianificazione. Esso può essere espresso come durata vera e propria o come tempo elapsed: si possono indicare le sole giornate lavorative o i giorni di calendario. Ad esempio, due settimane di lavoro elapsed corrispondono in realtà a 10 giorni effettivi di lavoro.

L'*effort* è una misura del lavoro che deve essere svolto: normalmente è indicato da espressioni che indicano per quanto tempo un insieme di risorse umane deve operare. Tipicamente, l'unità di misura per l'effort è il mese/uomo, cioè il lavoro che una persona riesce a sviluppare in un mese.

8.2.1 La pianificazione di progetto

Il processo di pianificazione è costituito da una serie di passi che portano alla redazione del piano di progetto.

- Scomposizione del progetto in attività e sottoattività.
- Identificazione delle relazioni fra le attività.
- Stima dei costi e dei tempi.
- Determinazione dello scheduling iniziale.
- Assegnazione delle risorse e adeguamento del piano.
- Definizione del budget.

Scomposizione del progetto in attività e sottoattività

Il primo passo della pianificazione consiste nell'identificazione di tutte le attività necessarie per la realizzazione del progetto. Esse sono individuate considerando le fasi che costituiscono il ciclo di vita scelto come riferimento. In funzione delle fasi presenti, si definiscono delle macroattività che si compongono in sottoattività di dimensioni tali da essere facilmente gestibili. La definizione delle attività può essere svolta sfruttando i seguenti criteri di scomposizione.

- *Per parti*: la scomposizione e la strutturazione delle attività avviene considerando le parti secondo le quali è costituito il prodotto.
- *Per funzioni*: si considerano le funzionalità che il prodotto deve fornire e in base a esse si ottiene l'elenco strutturato delle attività da svolgere.
- *Per obiettivi*: si considerano gli obiettivi posti dal progetto. Il progetto è strutturato in funzione dell'ottenimento degli obiettivi identificati.
- *Per fasi*: si considerano le fasi che devono essere svolte.
- *Per rilasci*: si suddividono e strutturano le attività in funzione dei rilasci che si vogliono fare.
- *Spaziale/geografica*: le attività sono scomposte in funzione delle zone geografiche in cui si trovano, ad esempio, le filiali dell'azienda partecipanti al progetto.

8.2 Pianificazione di progetto e controllo di avanzamento 161

Il criterio adottato non deve essere obbligatoriamente unico per tutto il progetto: è essenziale, tuttavia, che sia coerente in funzione del livello di scomposizione. Ad esempio, al primo livello si può scomporre per rilasci, identificando tante attività quanti sono i rilasci previsti. Ogni attività così definita può essere poi scomposta in ulteriori attività seguendo il criterio delle fasi o delle funzioni.

L'aspetto critico di questo primo passo consiste nell'individuare e definire attività che siano gestibili e realizzabili: spesso, infatti, le attività identificate sono troppo grandi e complesse per essere gestite in maniera chiara ed efficace o, al contrario, sono così piccole da rendere inefficiente il progetto perché richiede un uso di risorse ingiustificato. Alcune regole che si possono considerare per scomporre le attività in sottoattività sono le seguenti.

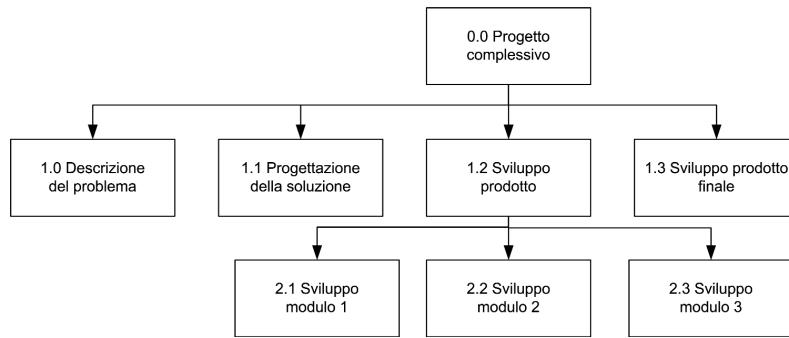
- *Regola dell'8/80:* la durata delle attività deve essere compresa tra le 8 ore, pari a una giornata lavorativa, e le 80 ore, pari a due settimane lavorative. Un'attività che dura meno di un giorno rappresenta una parcellizzazione eccessiva del lavoro; una che dura troppo, invece, corre il rischio di diventare ingestibile.
- *Regola del periodo di reporting:* un'attività deve durare al massimo quanto l'intervallo di tempo che intercorre tra due meeting sullo stato di avanzamento del progetto. A ogni meeting devono esserci attività la cui esecuzione non è cominciata, è terminata oppure è a metà.
- *Regola dell'utilità:* un'attività va scomposta se le attività risultanti dalla scomposizione sono più facili da gestire, da stimare o da monitorare, o anche se diventa più facile assegnare la responsabilità dell'attività in maniera chiara.

Al termine di questo passo si ottiene un elenco strutturato delle attività da eseguire e una prima indicazione dei ruoli professionali richiesti per la loro esecuzione. Lo strumento che aiuta a scomporre il progetto in attività e a darne una raffigurazione chiara e comprensibile è la *Work Breakdown Structure (WBS)*. Nella WBS le attività sono rappresentate in forma testuale, con un elenco indentato, o in forma grafica, con una struttura ad albero le cui foglie sono le attività vere e proprie mentre i padri sono delle attività fittizie che servono solo per raggruppare logicamente, oltre che graficamente, attività che hanno caratteristiche comuni. Le attività foglie sono chiamate *workpackage*; le attività che ne raggruppano altre, permettendo così di avere un'organizzazione strutturata, sono dette *summary task*. La Figura 8.5 mostra un esempio di WBS organizzata per fasi.

Identificazione delle relazioni fra le attività

Nel passo precedente, la pianificazione ha portato all'identificazione delle attività da svolgere. Il passo successivo consiste nell'identificare i vincoli temporali che legano tra loro le varie attività. Le attività di un progetto software, infatti, non sono indipendenti le une dalle altre: alcune, ad esempio, hanno bisogno, come dati di input, del risultato prodotto da altre attività. È importante definire, quindi, quali attività sono vincolate da altre, e in che modo, per definire la loro sequenza di esecuzione. Questo serve soprattutto per ottimizzare il tempo e l'uso delle risorse: si corre il rischio, infatti, di bloccare alcune risorse e quindi sostenere costi per attività che in realtà non possono partire.

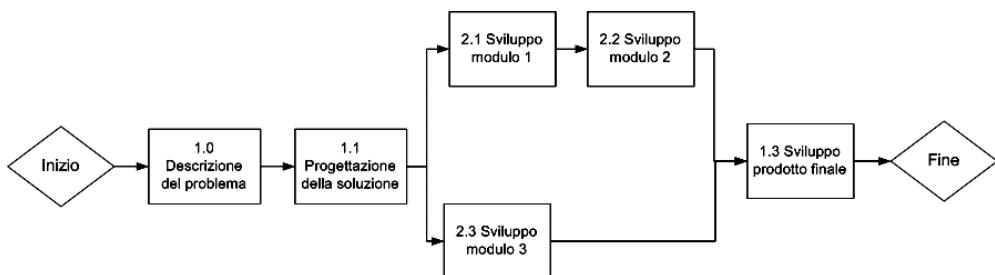
162 Capitolo 8 Cicli di vita e gestione dei progetti

**Figura 8.5** Esempio di WBS organizzata per fasi.

Due attività (spesso si usa anche il termine *task*), chiamate A e B possono essere legate da uno dei seguenti vincoli.

- *Finish-to-start*: "A f-to-s B" indica che il task B può iniziare solo dopo la fine del task A.
- *Start-to-start*: "A s-to-s B" indica che il task B è attivato contemporaneamente ad A o in un istante successivo all'inizio di A, ma non prima.
- *Finish-to-finish*: "A f-to-f B" indica che il task B termina contemporaneamente ad A o in un istante successivo alla fine di A, ma non può terminare prima.
- *Lag*: indica il tempo di attesa dalla fine del task A prima che il task B possa iniziare. Ad esempio, un lag pari a 5 giorni indica che l'attività B può partire solo 5 giorni dopo la conclusione dell'attività A.

I vincoli temporali identificati sono descritti in una tabella o in un *diagramma PERT* come quello della Figura 8.6. Per esempio, l'attività 1.0 è *finish-to-start* con l'attività 1.1, mentre l'attività 2.1 è *start-to-start* con l'attività 2.3. I diagrammi PERT possono essere disegnati utilizzando strumenti di supporto alle attività di pianificazione di progetto come Microsoft Project.

**Figura 8.6** PERT.

8.2 Pianificazione di progetto e controllo di avanzamento 163

Stima dei costi e dei tempi

Per ogni attività devono essere stimati i costi e i tempi. Nella stima dei costi sono considerati il costo del lavoro, i costi fissi dell’azienda, il costo del materiale usato, i servizi ecc. Nei progetti software è necessario considerare anche il costo relativo all’uso o acquisto di hardware e licenze software. Per ciò che concerne la stima dei tempi, essa deve considerare anche la durata solare di un’attività. Ad esempio, se un’attività prevede l’ordine di materiale per proseguire nel progetto, e questo richiede 5 giorni per la consegna, vanno contati anche i 5 giorni di attesa.

Stimare tempi e costi non è un compito semplice. Nel corso degli anni sono stati identificati una serie di metodi di stima molto sofisticati. Uno dei più noti è COCOMO: esso permette di calcolare tempi ed effort a partire da una stima delle dimensioni complessive del software da sviluppare. In questa sede non sono discussi nel dettaglio questi metodi: il lettore è invitato a considerare i riferimenti suggeriti al termine del capitolo. Per quanto visto, avendo identificato in modo dettagliato la WBS del progetto, l’ingegnere del software può direttamente stimare i costi e i tempi delle singole attività sulla base delle proprie esperienze. È la forma più semplice di *stima per analogia*: riuso dell’esperienza accumulata per fornire una previsione di quello che accadrà. L’uso di una WBS dettagliata semplifica questo compito, in quanto l’ingegnere del software non deve stimare costi e tempi del progetto nel suo complesso, ma solo delle singole attività. La stima complessiva si ottiene per aggregazione dei dati lungo la WBS.

Calcolo dello scheduling iniziale

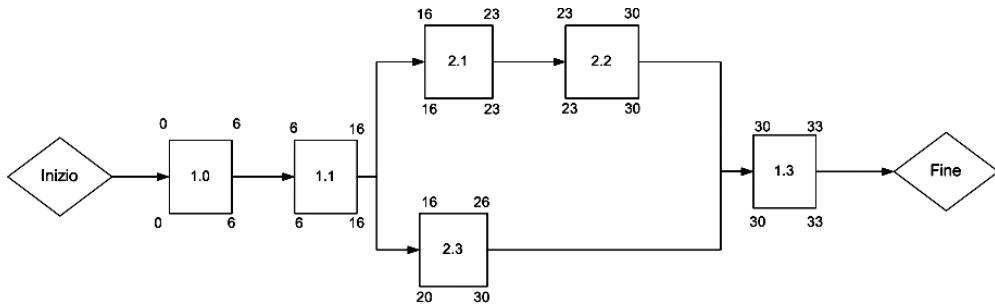
In questa fase della pianificazione è necessario collocare le varie attività in un calendario dei tempi. Nella fase di identificazione delle relazioni, infatti, sono stati solamente indicati i vincoli temporali che legavano le attività. È ora necessario collocare le attività in precisi intervalli di tempo.

Per collocare le attività su un asse temporale occorre avere la data di inizio del progetto, la stima dei tempi necessari per ogni attività e il diagramma PERT con le relazioni tra attività definito nel secondo passo della pianificazione. PERT supporta la stima della durata di un’attività considerando tre valori principali: la durata ottimistica, la durata media e quella pessimistica.

Partendo dalla data iniziale del progetto, che si può indicare come giorno zero, e considerando la durata delle varie attività proposte nel diagramma nell’ordine con cui si trovano, è possibile identificare le *quattro date di riferimento* di ogni attività (si veda la Figura 8.7).

- *Early start ed early finish*: la prima data utile in cui l’attività può iniziare e finire, nel rispetto dei vincoli temporali con le altre attività (collocate nella figura sul lato superiore dell’attività).
- *Late start e late finish*: la data ultima in cui l’attività considerata può iniziare e terminare senza causare ritardi all’intero progetto (collocate nella figura sul lato inferiore dell’attività).

164 Capitolo 8 Cicli di vita e gestione dei progetti

**Figura 8.7** PERT tempificato.

Sfruttando il calcolo di queste date è possibile calcolare lo slack time. Lo *slack time* indica l'intervallo di tempo entro il quale l'attività considerata può partire senza creare ritardi e conseguenti problemi allo svolgimento del progetto. In particolare, lo slack time è dato dalla differenza tra late finish e early finish (o anche tra late start e early start). Nel caso in cui early start e late start siano uguali, lo slack time è nullo: l'attività così caratterizzata è definita critica. Un'attività critica è un'attività che non ha alcun margine di ritardo: un suo ritardo causa lo slittamento della fine dell'intero progetto. L'insieme delle attività critiche forma il *cammino critico*.

Individuato l'intervallo di tempo in cui ogni attività può partire, è necessario eseguire il posizionamento delle attività lungo l'asse temporale, indicando in maniera specifica il loro inizio e la loro fine.

I metodi per collocare l'inizio preciso di un'attività all'interno del suo slack time sono i seguenti.

- *As Soon As Possible (ASAP):* l'attività deve partire il prima possibile, per esempio, il primo giorno utile che coincide con l'early start.
- *As Late As Possible (ALAP):* la partenza dell'attività è schedulata il più tardi possibile, nel limite dello slack time.
- *Fixed:* l'attività è schedulata alla data indicata dal progettista. È il caso, ad esempio, di attività in cui l'interazione con l'utente è fondamentale ed è possibile solo in una specifica data.

Il posizionamento sull'asse temporale delle attività del progetto va eseguito con attenzione perché indicare che una particolare attività inizia in uno specifico giorno significa garantire, con un sufficiente grado di sicurezza, che in quel dato giorno ci siano le risorse disponibili per iniziare l'attività e, al tempo stesso, che non rimangano inutilizzate.

Un modo sintetico per rappresentare tutte le informazioni relative alla pianificazione del progetto è costituito dal *diagramma di Gantt*: la Figura 8.8 ne illustra un esempio relativo al progetto descritto dal diagramma PERT della Figura 8.7.

8.2 Pianificazione di progetto e controllo di avanzamento 165

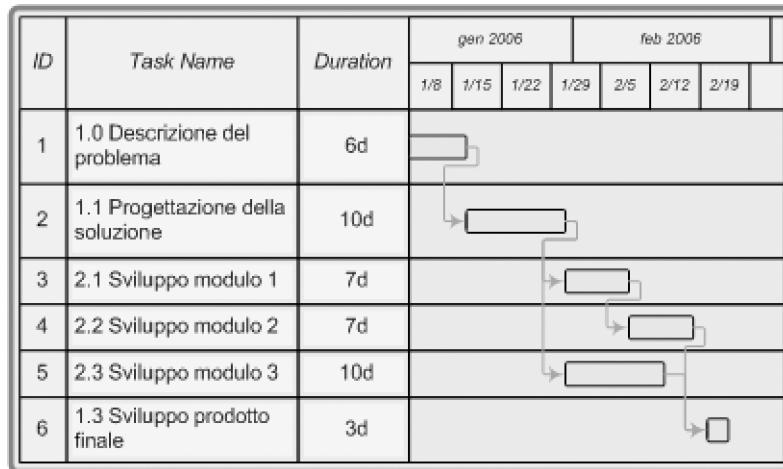


Figura 8.8 Esempio di diagramma di Gantt.

Un diagramma di Gantt mostra lo scheduling delle attività su un asse temporale, indicandone inizio, fine, durata e precedenze. Ogni attività è rappresentata con un rettangolo di lunghezza proporzionale alla sua durata, collocato sull'asse temporale a partire dal giorno di inizio dell'attività stessa. Le relazioni di precedenza tra attività sono descritte con frecce che collegano i rettangoli che sono in relazione tra loro. Il diagramma permette di visualizzare efficacemente il calendario delle attività, la loro sequenza, durata e gli eventuali punti critici. In particolare, sono messi in evidenza il cammino critico, le attività che lo compongono e le attività che sono in relazione con queste ultime. Ad esempio, nella Figura 8.8 il cammino critico è formato dai task 1, 2, 3, 4 e 6.

Assegnazione delle risorse e adeguamento del piano

Lo scopo di questo passo consiste nell'ottimizzare l'uso delle risorse in modo da garantirne l'allocazione ottimale. A questo punto della pianificazione si conoscono le attività, i ruoli coinvolti in ognuna di esse, i costi e i tempi oltre alle dipendenze temporali tra attività. È possibile, quindi, assegnare le risorse a ogni attività nel rispetto dei vincoli di tempo.

Come già detto, la risorsa principale nei progetti software è la persona, il professionista. L'obiettivo primario è assegnare in modo ottimale la risorsa, evitando, ad esempio, che venga allocata su un'attività per un tempo superiore alla sua disponibilità effettiva o per un tempo nettamente inferiore. Nel primo caso, le attività subiranno un ritardo per la mancanza della risorsa, nel secondo caso si avrà un costo non proporzionato all'effort, con il rischio che la risorsa non possa essere allocata per altri progetti.

L'allocazione è definita in termini temporali. Una risorsa può essere allocata indicando la percentuale di tempo che deve dedicare al progetto, ad esempio, il 50%. È pos-

sibile indicare anche direttamente il numero di ore che devono essere impiegate dalla risorsa per lavorare sull'attività considerata, come per esempio 30 ore per l'attività di definizione dell'architettura di base. Infine, è possibile indicare sia il numero totale di ore che la risorsa deve dedicare all'attività considerata che il numero di ore giornaliero.

A causa della disponibilità generalmente limitata delle risorse, dopo un'allocazione iniziale, è spesso necessario dover "livellare" le allocazioni, vale a dire "aggiustare" l'assegnazione per coprire tutte le attività con le risorse a disposizione, tenendo conto dei tempi di lavoro di una risorsa (tipicamente 8 ore al giorno). Sono presenti diversi metodi di leveling.

- *Cambiamento delle allocazioni.* Questo metodo prevede due possibilità: la prima consiste in un aumento della percentuale di tempo in cui la risorsa è allocata, la seconda nell'inserimento di una nuova risorsa che lavori sul progetto. Entrambe queste alternative presentano alcuni aspetti problematici. La prima alternativa risulta spesso non realizzabile perché la risorsa considerata può essere già allocata al 100%, e questo rende impossibile aumentare ulteriormente la sua allocazione sul progetto; la seconda alternativa può essere realizzata solo nel caso in cui ci siano risorse da poter aggiungere per alleggerire il carico di quelle sovra-allocate, e questo non sempre accade. Nell'esempio della Figura 8.9, un'attività richiede 24 giorni/uomo di lavoro. Le persone inizialmente allocate sono 4: questo fa sì che impieghino 6 giorni di calendario per completare l'attività. Si supponga che con tale allocazione non si rispettino i tempi. Una possibile soluzione è cambiare le allocazioni, dirottando sull'attività altre due persone per far diminuire il numero di giorni totali e rientrare così nei tempi del progetto. Si noti che, come ampiamente discusso in molti testi di ingegneria del software, non è sempre possibile applicare questa tecnica in quanto in generale non si può scambiare tempo con persone: non è sempre vero che più persone sul progetto fanno lo stesso lavoro in meno tempo, in quanto aumenta l'attività di comunicazione e coordinamento.
- *Ritardo di un task.* Se una risorsa è allocata a due attività diverse per un tempo superiore alle sue disponibilità, è possibile cercare di ritardare l'inizio di un'attività

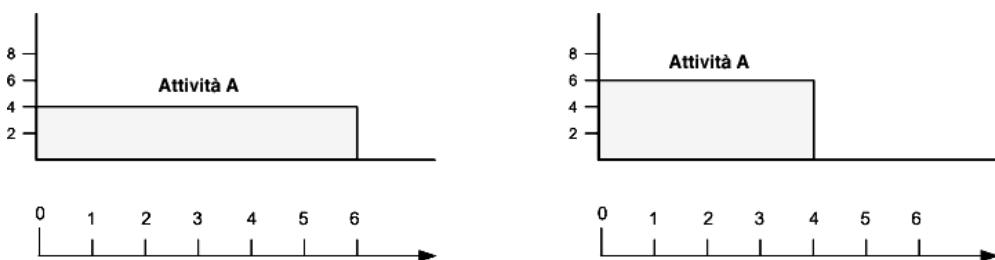


Figura 8.9 Cambiamento delle allocazioni.

8.2 Pianificazione di progetto e controllo di avanzamento 167

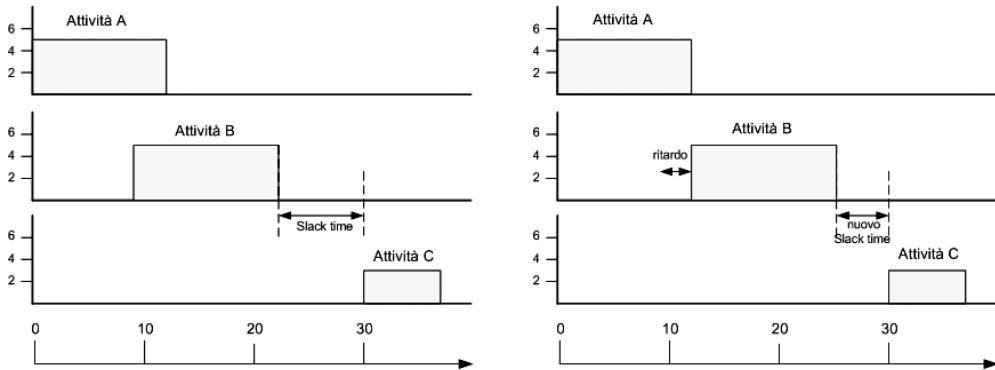


Figura 8.10 Ritardo di un task.

per dare il tempo alla risorsa sovra-allocata di terminare la precedente (Figura 8.10). Tuttavia, occorre considerare che non tutte le attività possono essere ritardate e non è possibile assegnare un ritardo qualsiasi. Possono essere ritardate quelle che presentano uno slack time superiore a zero, vale a dire hanno dei giorni di ritardo di tolleranza. In particolare, si cerca sempre di ritardare prima le attività che hanno uno slack time maggiore. Ovviamente, le attività critiche, i cui ritardi influenzano il ritardo dell'intero progetto, non devono essere ritardate se non è strettamente necessario.

- *Divisione delle assegnazioni e dei task.* Può capitare che per un certo periodo due attività siano schedurate in parallelo e richiedano un numero di risorse superiore a quello previsto per il progetto. In questo caso si può decidere di spezzare l'attività che ha una durata maggiore per liberare le risorse che vanno a eseguire l'attività parallela con durata minore (Figura 8.11). Al termine di quest'ultima, le risorse ritornano sull'esecuzione della precedente. Questa soluzione è adottata solo se il ritardo che si introduce è accettabile.
- *Allocazione non uniforme.* Anche questa soluzione prevede l'interruzione di un'attività per permettere alle risorse contese di eseguire un'attività parallela che non può attendere. In questo caso però, non ci si limita a spezzare l'attività ma si aumentano il numero di persone assegnate durante la prima parte (Figura 8.12). Lo scopo di questa riallocazione è di annullare o almeno ridurre il ritardo subito dall'attività interrotta.

Definizione del budget

Dopo aver individuato le attività che concorrono alla realizzazione del progetto, e dopo aver indicato per ognuna le relazioni con le altre, la durata e l'effort necessario per realizzarle, è possibile calcolare il costo complessivo (*budget*) del progetto per aggregazione tra tutte le attività presenti nella WBS.

168 Capitolo 8 Cicli di vita e gestione dei progetti

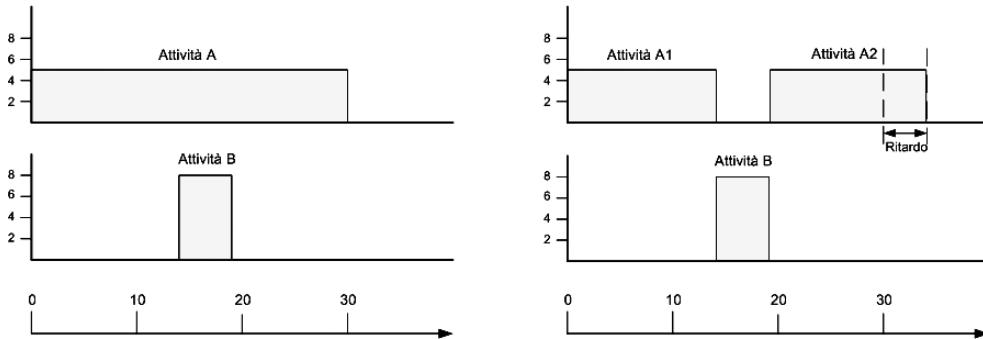


Figura 8.11 Divisione delle assegnazioni e dei task.

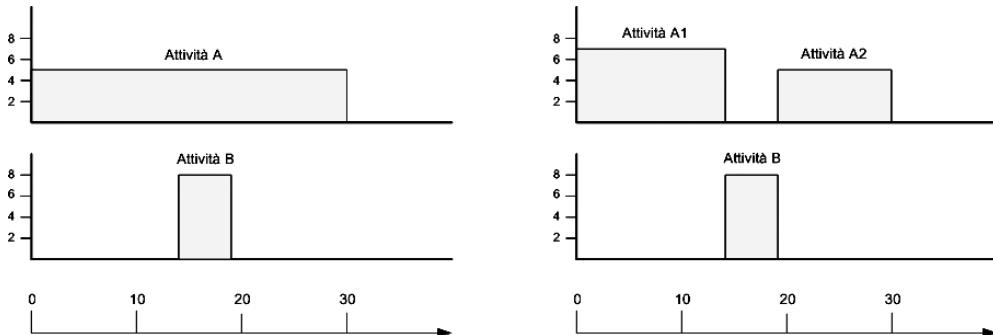


Figura 8.12 Allocazione non uniforme.

8.2.2 Il controllo di avanzamento

La pianificazione è il primo passo per una buona gestione di progetto, ma da sola non basta. Occorre, infatti, controllare costantemente lo stato di avanzamento di un progetto e confrontarlo con quanto preventivato. Spesso anche il piano di progetto frutto di una pianificazione effettuata correttamente e con precisione non è seguito con esattezza: diventa così importante sapere quanto l'esecuzione reale si scosta dal piano definito. Un metodo per misurare lo stato di avanzamento del progetto e garantire la possibilità di confronto con il piano è l'*Earned Value Management* (EVM). L'EVM è uno strumento di misurazione delle prestazioni e un meccanismo di raccolta e analisi dei feedback.

Il diagramma dell'EVM della Figura 8.13 riporta sull'asse delle ascisse il tempo e su quello delle ordinate il costo cumulativo (cioè quanto si è speso complessivamente a una

8.2 Pianificazione di progetto e controllo di avanzamento 169

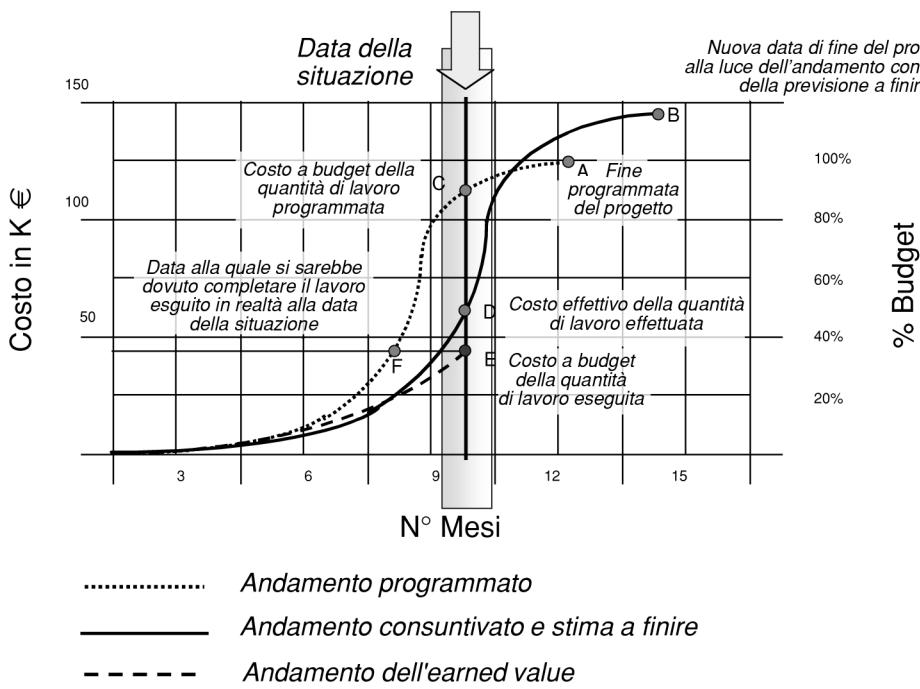


Figura 8.13 Il diagramma dell'earned value.

certa data) e in percentuale sul budget (i valori relativi a costi e tempi hanno valore di esempio). In realtà sul diagramma dell'EVM sono disegnate tre diverse curve.

La prima curva è quella del *Planned Value* (PV): essa coincide con la pianificazione iniziale o *andamento programmato*. Descrive l'evoluzione del progetto se tutto si svolgesse secondo quanto pianificato. L'indice di riferimento è il *Budgeted Cost for Work Scheduled* (BCWS, o *costo a budget della quantità di lavoro programmata*) che fornisce un'indicazione sul costo a budget del lavoro pianificato. La curva è tracciata all'inizio del progetto utilizzando direttamente i dati prodotti dalla pianificazione (il piano di progetto iniziale).

Le altre due curve sono tracciate periodicamente per verificare in momenti specifici (la cosiddetta *data della situazione*) lo stato di avanzamento del progetto. In particolare, la seconda curva, *Actual Cost* (AC), indica il costo effettivo che si sta affrontando (*andamento consuntivato*). L'indice di riferimento è l'*Actual Cost for Work Performed* (ACWP, o *costo effettivo della quantità di lavoro effettuata*) che fornisce l'indicazione sul costo affrontato per il lavoro realmente svolto. La curva è tracciata utilizzando le informazioni raccolte in azienda (in particolare dall'amministrazione) su quelli che sono stati i costi realmente sostenuti fino a quel momento nel progetto. La terza curva è detta dell'*Earned Value* (EV) e indica lo stato di avanzamento del progetto. L'indice di riferimento è il *Budgeted Cost for Work Performed* (BCWP) che fornisce l'indicazione sul costo preventi-

170 Capitolo 8 Cicli di vita e gestione dei progetti

vato per il lavoro effettivamente svolto. In pratica, l'earned value indica quale sarebbe stato secondo la pianificazione originaria il costo corrispondente al lavoro effettivamente svolto.

La curva dell'earned value è tracciata dal progettista o dal capo progetto, che deve valutare "quanto vale" il lavoro svolto. La criticità della misurazione dell'EV consiste nel calcolare il valore assorbito da un'attività non ancora completata e per la quale non è possibile rilasciare un qualche risultato che ne testimoni il completamento: quanto manca "per finire" e quindi quanto vale quanto fatto sinora? Ovviamente, si tratta di un compito particolarmente delicato e critico in quanto non è possibile misurare in modo diretto questa quantità. Per facilitare il compito del progettista sono state definite alcune tecniche di misurazione.

- *Fixed formula*: si assegna una percentuale predefinita del valore pianificato all'inizio dell'attività. Al termine dell'attività si assegna anche il valore pianificato rimanente.
- *Weighted milestone*: l'assegnamento del valore del lavoro è subordinato alla produzione di risultati osservabili. Il valore è assegnato al conseguimento dei milestone definiti in un'attività.
- *Percent complete*: si assegna un valore di EV proporzionale alla percentuale di attività svolta. Questa tecnica presenta un punto critico nella definizione della percentuale di attività svolta, perché tale definizione dipende dalla soggettività del manager.
- *Apportioned effort*: si stima l'EV di un'attività considerando il valore già ottenuto di un'attività correlata a quella considerata.
- *Level of effort*: è riferito alle attività che non producono output tangibili per il progetto, come ad esempio quelle di gestione e controllo. In questo caso si distribuisce il costo dell'intera attività in modo uniforme lungo tutta la sua durata: il valore assorbito a un dato istante è proporzionale al tempo trascorso dall'inizio dell'attività. Al termine, si assegna automaticamente il valore PV stimato alla fine del periodo di misurazione.

Le tre curve che caratterizzano il diagramma dell'EV rappresentano in modo sintetico lo stato di avanzamento di un progetto. Nel caso ideale, le tre curve dovrebbero sovrapporsi: si è speso quanto preventivato per fare il lavoro originariamente previsto. In generale, le tre curve non sono sovrapposte e il loro posizionamento relativo sta a indicare lo stato del progetto. Per esempio, nella Figura 8.13, alla data della situazione il costo effettivamente sostenuto (punto D) è inferiore a quanto pianificato (punto C): si è speso meno. La curva dell'EV è posizionata sotto quella dei costi sostenuti: in pratica, quanto prodotto (punto E) vale meno di quanto si è speso. Ciò corrisponde a una situazione pessima: il progetto è in ritardo e si è speso più del previsto.

In generale, le grandezze misurate saranno il punto di partenza per effettuare l'analisi delle prestazioni ed eseguire eventuali previsioni sul proseguimento del progetto. L'analisi delle prestazioni richiede il confronto delle grandezze e la definizione di indicatori opportuni. Tali indicatori servono per verificare l'uso adeguato del tempo e delle risorse

8.2 Pianificazione di progetto e controllo di avanzamento 171

e sono qui di seguito illustrati (si noti che nelle formule che seguono, EV corrisponde al punto E del grafico della Figura 8.13, PV al punto C e AC al punto D).

- *Schedule variance (SV)*: indica se, in una certa data, il progetto è in anticipo o in ritardo rispetto a quanto preventivato. La formula che calcola SV è: $SV = EV - PV$. Se SV è maggiore di zero vuol dire che il valore generato è maggiore di quello preventivato.
- *Schedule Performance Index (SPI)*: indica l'efficacia con cui si sta consumando il tempo. Si calcola attraverso la formula $SPI = EV / PV$, vale a dire il rapporto tra il lavoro generato e quello preventivato. Un rapporto superiore all'unità indica che nel progetto c'è efficienza nell'uso del tempo.

Gli indicatori relativi all'uso delle risorse sono i seguenti.

- *Cost Variance (CV)*: indica se il progetto ha richiesto, fino al momento considerato, un consumo di risorse maggiore o minore rispetto al budget preventivato. La formula di calcolo è $CV = EV - AC$, vale a dire la differenza tra il valore generato dal progetto e il costo effettivo (*Actual Cost*) affrontato. Un valore maggiore di zero è favorevole perché indica che il valore creato è maggiore rispetto al costo sostenuto, vale a dire che si è sotto il budget preventivato. CV può essere dato anche in forma percentuale con la seguente formula $CV\% = CV / EV$ che indica il posizionamento percentuale rispetto al budget.
- *Cost Performance Index (CPI)*: indica l'efficacia con cui si stanno sfruttando le risorse del progetto. La formula del CPI è $CPI = EV / AC$, vale a dire il rapporto tra il valore creato e i costi che sono stati effettivamente affrontati. Un valore maggiore di uno è favorevole perché indica che a parità di lavoro le risorse sono state sfruttate meglio di quanto pianificato.

La tecnica dell'EV richiede come requisito essenziale che sia eseguita una pianificazione accurata e precisa. Inoltre, fornisce i risultati migliori quando i task definiti nel progetto presentano alcune caratteristiche precise: in particolare, i task sono in numero sufficientemente elevato e di durata breve rispetto al periodo di reporting scelto. Al momento dell'analisi, i task completati devono essere in numero sufficiente, in modo da effettuare un'analisi degli scostamenti significativa, così come ci devono essere un certo numero di task in lavorazione, in modo che l'avanzamento di progetto non coincida con l'avanzamento del singolo task.

La tecnica dell'EV è una metodologia molto diffusa e permette di avere un'unica unità di misura e di confronto per tutta la durata del progetto. È integrata con la maggior parte degli strumenti di pianificazione di progetto (come il già citato Project).

8.2.3 Il ciclo di pianificazione e controllo

Da quanto discusso in precedenza si evince che le attività di pianificazione e controllo non sono eseguite solo all'inizio del progetto o in particolari circostanze. In realtà, accanto alle attività di sviluppo vero e proprio, è necessario attivare un processo sistematico

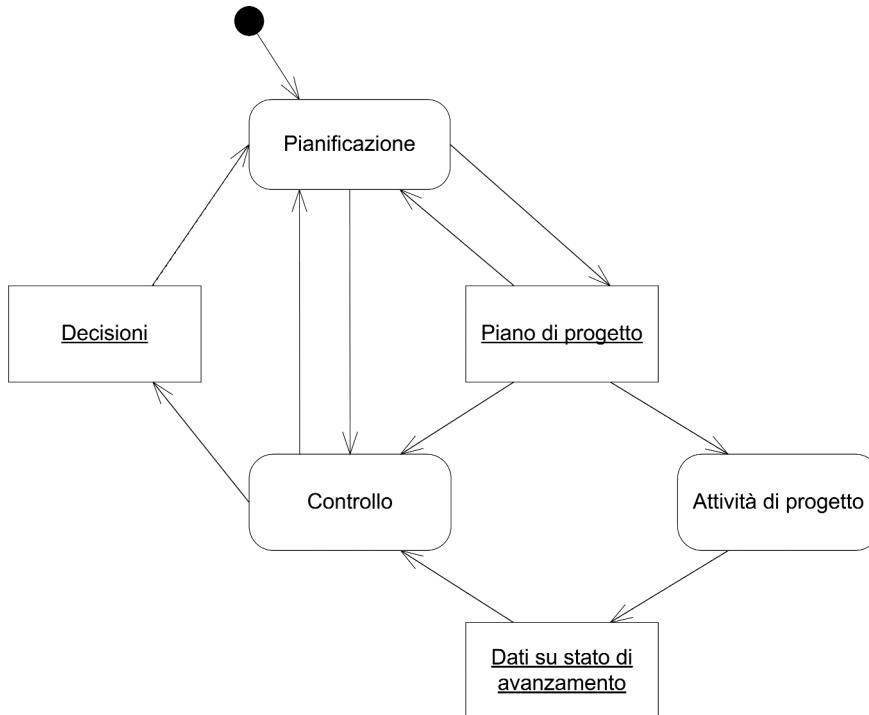


Figura 8.14 Il ciclo di pianificazione e controllo.

co di monitoraggio dell'andamento del progetto e di sua ripianificazione. Si tratta, quindi, di un ciclo continuo, così come illustrato dall'activity diagram della Figura 8.14.

La pianificazione produce un piano di progetto che viene utilizzato per guidare le attività di sviluppo vere e proprie. Periodicamente vengono prodotti dei dati sullo stato di avanzamento del progetto che sono passati all'attività di controllo. Il confronto tra piano e stato di avanzamento genera delle decisioni che sono utilizzate per un nuovo ciclo di pianificazione che può aggiornare (anche significativamente) il piano di progetto. Come si può notare dal diagramma, il ciclo ha inizio, ma non ha mai fine (o meglio termina con la chiusura del progetto).

8.3 Configuration management

Durante il processo di sviluppo del software sono realizzati e implementati semilavorati di varia natura: documenti, codice sorgente, casi di test ecc. La complessità dei semilavorati prodotti e la durata nel tempo delle attività necessarie al loro sviluppo rende fondamentale riuscire a gestire queste informazioni in modo organico e strutturato all'interno

8.3 Configuration management 173

del team di progetto. Spesso, infatti, il progetto prevede che più persone lavorino in tempi diversi e talvolta a più riprese sullo stesso deliverable. Le tecniche e gli strumenti per il configuration management sono state introdotte per rispondere alle sfide e ai problemi che nascono in questo scenario.

Il *Configuration Management* (CM, o *gestione delle configurazioni*) è un insieme di metodologie e tecniche per la gestione delle versioni dei documenti e delle configurazioni dei prodotti e dei semilavorati manipolati nel corso di un progetto (o di gruppi di progetti correlati). L'adozione del configuration management permette di ridurre gli errori e contribuisce a mantenere la coerenza e l'integrità tra i semilavorati e prodotti di un progetto software. Inoltre, concorre al miglioramento del processo di sviluppo nel suo complesso perché, facilitando la gestione dei semilavorati, incrementa la capacità di riusarli e sfruttarli in altre fasi del processo.

Alla base del configuration management si trovano una serie di concetti e principi (Figura 8.15).

- *Versione*. Il concetto di versione rappresenta lo stato di un semilavorato in un preciso istante di tempo. In un processo software sono *versionati* tutti i documenti e in particolare i moduli e le parti di codice realizzate durante tutto il ciclo di vita del progetto. Un documento versionato è nominato ponendo accanto al nome anche la versione che lo caratterizza. Normalmente, ogni versione viene identificata in modo univoco attraverso la cosiddetta *dot notation*: una versione è indicata con un numero composto da due (o più) cifre separate da punti. Nel caso più semplice, lo schema utilizzato ha la seguente struttura: #major.#minor. La cifra minor è incrementata quando il semilavorato subisce delle variazioni oppure è ampliato ma rimane comunque un semilavorato interno. Quando le modifiche sono rilevanti oppure quando il semilavorato è rilasciato e non è più in lavorazione, si cambia la cifra major. Si consideri, ad esempio, il documento dei requisiti in un progetto che segue un ciclo di vita iterativo: al momento della creazione il documento sarà identificato come versione 0.1 e a ogni incremento o modifica verrà incrementata la seconda cifra (0.2, 0.3, 0.4,...). Nel momento in cui il documento viene consolidato al termine della prima iterazione, è aggiornata anche la cifra major: essa viene incrementata e la versione finale della prima iterazione sarà la 1.0. Alla seconda iterazione si partirà modificando il documento dei requisiti esistente e quindi la versione iniziale sarà la 1.1 e la finale, una volta consolidata, sarà la 2.0.

Identificare e tenere traccia delle diverse versioni di ogni documento o semilavorato è estremamente importante in quanto in questo modo è possibile ripercorrerne lo sviluppo nel corso del tempo. Di conseguenza, le versioni "vecchie" non vengono eliminate quando ne viene creata una più recente. Le tecniche di memorizzazione delle versioni sono sostanzialmente due. Nell'approccio *completo*, sono memorizzate tutte le versioni complete del semilavorato. Nell'approccio *a delta*, un semilavorato è memorizzato in modo completo solo nella sua versione iniziale: nuove versioni sono memorizzate solo come variazioni incrementalì rispetto alla copia iniziale (e alle sue successive versioni).

174 Capitolo 8 Cicli di vita e gestione dei progetti

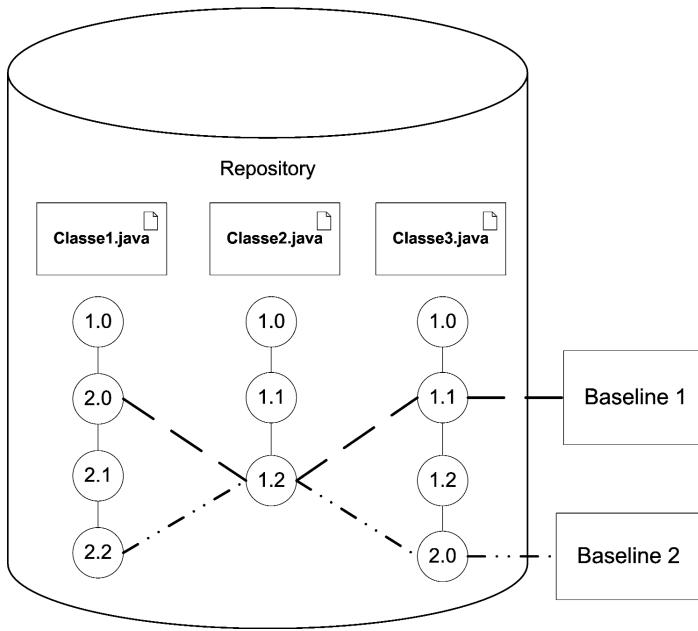


Figura 8.15 I concetti della gestione delle configurazioni.

- **Configurazione.** È l'insieme di semilavorati usati per formare il prodotto. Nella scelta della configurazione la versione del semilavorato è fondamentale: cambiando la versione si cambia necessariamente anche configurazione. Due configurazioni si differenziano perché formate da semilavorati diversi oppure perché le versioni dei semilavorati della prima configurazione differiscono da quelle della seconda.
- **Baseline.** È una configurazione che è stata formalmente validata e accettata: cambiamenti sui semilavorati che la compongono possono essere effettuati solo attraverso un formale processo di revisione. Una baseline fa da punto di riferimento durante il ciclo di vita del progetto del software. In alcuni cicli di vita può coincidere con l'output di un'iterazione, ad esempio con una versione del prodotto o un prototipo che siano stati rilasciati al cliente.
- **Repository.** Contiene tutte le versioni dei semilavorati realizzati durante il progetto. È dal repository che gli sviluppatori prelevano i semilavorati su cui lavorare. Le versioni di un semilavorato conservato nel repository sono collegate tra loro e formano una linea di sviluppo chiamata branch. Un *branch* è un insieme di versioni dello stesso semilavorato, poste in ordine sequenziale di sviluppo. Ogni volta che si crea una nuova versione il branch è incrementato. Dal branch principale è possibi-

le creare branch secondari che seguono una diversa linea di sviluppo, creando così una struttura complessiva a grafo. Nel caso in cui è necessario fondere due rami, vale a dire ottenere un unico semilavorato con le modifiche fatte nelle versioni dei due rami, viene eseguita l'operazione di merge. Il *merge* consiste nel "mettere insieme" le modifiche delle versioni più recenti di due rami, in modo da ottenere un'unica versione integrata. Quest'operazione è particolarmente delicata, perché non può essere effettuata in modo automatico: è solo il progettista che sa come comporre in modo corretto e appropriato diversi frammenti di codice.

- *Workspace*. È lo spazio personale di lavoro che contiene gli elementi su cui opera il singolo progettista. Egli ha completo controllo sui semilavorati memorizzati nel proprio workspace. Uno degli aspetti maggiormente critici del configuration management è mantenere la coerenza e l'integrità dei semilavorati che si trovano nei workspace dei partecipanti al progetto rispetto ai semilavorati che si trovano nel repository centrale. In particolare, è necessario coordinare la sincronizzazione delle diverse versioni dei partecipanti al team di progetto con la versione del repository, senza perdere le modifiche effettuate da ciascuno. A questo scopo esistono vari tool che permettono di gestire gli accessi al repository e il suo aggiornamento. Tutti i tool disponibili si basano su due concetti elementari, il check-out e il check-in. Con il termine *check-out* si intende l'operazione grazie alla quale un progettista copia dal repository centrale nel proprio workspace una copia del semilavorato che intende modificare: in questo modo il progettista può lavorare senza modificare direttamente la versione del repository e senza impedire che altri possano accedervi. Con il termine *check-in* si intende l'operazione tramite la quale il progettista deposita la propria versione del semilavorato nel repository, se gli è consentito.

Le operazioni di check-in e check-out devono gestire la sincronizzazione di accessi multipli a uno stesso semilavorato. Il caso più critico si ha quando due progettisti accedono allo stesso semilavorato per modificarlo in parallelo. In mancanza di meccanismi adeguati, il primo dei due che eseguirà il check-in creerà una nuova versione collegata a quella originariamente presente nel repository. Quando il secondo progettista effettua il proprio check-in, la versione che inserisce apparirà come la più recente, ma non ingloberà in alcun modo i cambiamenti effettuati dal primo progettista: il primo check-in è stato "sovrascritto" (e perso).

Per gestire questo problema, sono state definite diverse politiche di gestione degli accessi, chiamate politiche di locking. Esse si basano sostanzialmente su due diversi paradigmi.

1. *Lock-modify-unlock*: consiste nel bloccare nel repository un elemento al momento del check-out in modo che nessun altro possa prelevarlo fino a quando non viene effettuato il corrispondente check-in.
2. *Copy-modify-merge*: in questo paradigma tutti possono fare il check-out di un particolare semilavorato. Al momento del check-in, il tool di configuration management controlla che non ci siano stati check-in precedenti: se così fosse l'ingegnere è obbligato e supportato a fare un merge del semilavorato, vale a dire a importare le modifiche fatte sulla versione modificata presente nel repository.

176 Capitolo 8 Cicli di vita e gestione dei progetti

- *Release.* La release, o rilascio, è una baseline approvata e validata, che viene data all'esterno del gruppo di sviluppo. Una release può essere interna se rimane comunque all'interno dell'organizzazione, esterna se viene affidata al cliente. I controlli sulla release sono ovviamente maggiori se il rilascio è esterno cioè verso il cliente.
- *Building.* È il processo di costruzione di una copia funzionante del prodotto a partire da una configurazione conservata nel repository. Il team di progetto scrive un build script con le dipendenze tra i componenti della soluzione software e tutte le informazioni necessarie per installare ed eseguire il programma partendo dai singoli componenti posti sotto configuration management. Durante il processo di building è necessario garantire che tutti i componenti che concorrono a realizzare il sistema siano considerati nel building con le loro versioni corrette.

I concetti sopra definiti sono alla base del processo di CM. Tale processo, in realtà, è molto complesso e può essere ulteriormente strutturato e discusso analizzando le diverse attività necessarie che devono essere attuate per garantire una corretta gestione dei semilavorati.

- *Configuration identification.* Lo scopo di quest'attività è decidere quali sono gli elementi da porre sotto configuration management e secondo quale meccanismo di identificazione. I documenti scelti saranno versionati e memorizzati nel repository.
- *Configuration control.* Quest'attività ha lo scopo di definire e verificare l'applicazione delle politiche e delle linee guida secondo le quali gestire cambiamenti significativi dei semilavorati posti sotto configuration management. In particolare, il configuration control pone l'attenzione su tre aspetti: cosa controllare, a chi affidare la responsabilità di validare i cambiamenti, come effettuare operativamente il controllo. In generale, sono soggetti all'attività di controllo formale solo le baseline: la scelta è dovuta al fatto che il controllo dei cambiamenti e la loro conferma e validazione sono attività che richiedono l'intervento di più persone e quindi, per migliorare l'efficacia e l'efficienza dell'intero progetto, vanno eseguite non per ogni minimo cambiamento, ma solamente per le modifiche sostanziali. Di norma, esiste un *Configuration Control Board* che ha la responsabilità di accettare i cambiamenti relativi alla particolare baseline di sua competenza: esso si riunisce periodicamente per analizzare lo stato delle richieste di modifica e per coordinare e autorizzare eventuali cambiamenti. Il Configuration Control Board è formato, in generale, da persone che risultano responsabili delle parti che subiscono modifiche: ad esempio, quello che controlla le modifiche sui componenti del codice sarà formato dal capo progetto e dai responsabili dei singoli componenti, mentre il Configuration Control Board responsabile della baseline funzionale vedrà la presenza tra i partecipanti anche del cliente.
- *Configuration status accounting.* Quest'attività prevede la raccolta delle informazioni sullo stato dei diversi semilavorati di cui si gestiscono le versioni. In particolare, le informazioni raccolte riguardano le baseline e lo stato delle richieste di modifica proposte o in corso. Se necessario, durante quest'attività sono prodotti alcuni report, che contengono le informazioni raccolte: sono di vari tipi in funzione del tipo di informazione o dei loro destinatari.

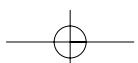
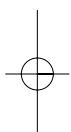
- *Configuration audit and review.* Quest'attività consiste nella verifica di conformità dei vari semilavorati rispetto alle specifiche, agli standard applicabili e agli impegni contrattuali. Nel configuration management ci sono tre tipi di audit: il primo, il *functional configuration audit*, considera gli elementi gestiti e verifica che siano rispettate le caratteristiche funzionali e di prestazioni in accordo con quanto definito nella baseline funzionale. Il secondo tipo, il *physical configuration audit*, considera la documentazione di progettazione e verifica la sua corrispondenza con quanto effettivamente implementato. Infine, il terzo tipo di audit, *in-process audit*, considera l'intero processo di configuration management e verifica che sia effettivamente adottato.

8.4 Riferimenti bibliografici

I cicli di vita del software sono discussi e presentati in moltissime pubblicazioni. Oltre al lavoro originario di Royce sul ciclo a cascata (citato nel Capitolo 1), altri utili riferimenti su questi temi sono Beck [2000], che discute in dettaglio le tecniche di eXtreme Programming, e Cockburn [2002], che presenta una serie di considerazioni tecniche e organizzative sugli approcci agili. Lo Spiral Model, nella sua versione più recente, è presentato in Boehm [2000]. Lo Unified Process è descritto, tra gli altri, in Arlow e Neustadt [2005]. Per gli altri cicli di vita, il lettore può far riferimento, almeno in prima battuta, ai testi generali sull'ingegneria del software citati nel Capitolo 1.

Per la pianificazione e controllo di progetto, un testo di riferimento è Royce [1998]. Altre informazioni utili sono contenute sul sito del Project Management Institute (PMI), che costituisce un riferimento generale sulle tematiche del project management. COCOMO (I e II) e in generale il tema della stima dei costi è trattato nei libri di Boehm e in particolare Boehm et al. [2000]. Un primo approfondimento della tecnica dell'earned value è disponibile su Wikipedia alla voce corrispondente.

Per quanto riguarda le tecniche e i metodi di configuration management, due lavori certamente storici sono Feldman [1979] e Tichy [1985]. Il primo presenta make, lo strumento che ha introdotto il concetto di program building. Il secondo descrive RCS, il primo ambiente di configuration management e change control. Una descrizione organica e dettagliata delle moderne tecnologie di configuration management è proposta da Hass [2002].



Capitolo 9

Qualità del prodotto e del processo

I prodotti software sono sempre più complessi e svolgono funzioni fondamentali per l'utenza. I guasti derivati dal malfunzionamento di un sistema basato su componenti informatiche possono portare a pericolose conseguenze. In molti casi hanno provocato seri danni non solo a cose ma, purtroppo, anche a persone: dai problemi di controllo del traffico aereo alle apparecchiature mediche, ai sistemi informativi di aziende private e di organizzazioni pubbliche. Per questo motivo è essenziale che in un processo di sviluppo software siano previste adeguate fasi di *verifica e validazione della qualità del prodotto*. Queste fasi non possono però ridursi semplicemente a un controllo o "collaudo" da effettuarsi al momento del rilascio del software: la qualità di un prodotto va "costruita e verificata" passo dopo passo, su ogni semilavorato del processo. Spesso i problemi più gravi e perniciosi derivano da errori commessi in fase di studio del problema. In generale, più tardi ci si accorge di un problema, maggiore sarà il costo per porvi rimedio.

Analogamente, *anche il processo di sviluppo ha una sua qualità (maturità)* che può e deve essere valutata e incrementata. Maggiore è la maturità del processo, maggiore è la confidenza che il prodotto possa essere sviluppato con i requisiti di qualità richiesti, nel rispetto dei tempi e costi previsti.

Per questi motivi, l'ingegneria del software ha sviluppato una serie di tecniche e metodi per verificare e validare la qualità di prodotti e processi. In questa sede non è possibile fornire un'analisi dettagliata e completa di quanto oggi disponibile. Ci si limiterà a introdurre i principali concetti e metodi che devono essere conosciuti da tutti coloro che in una qualche misura sono coinvolti in un processo di sviluppo del software.

9.1 Verifica e validazione

Nel Capitolo 8 sono stati introdotti i cicli di vita e le fasi che li caratterizzano. È stato evidenziato come il processo di realizzazione di un prodotto software non si riduce alla sola implementazione del codice, ma è il frutto di una serie di attività che devono essere accuratamente pianificate e gestite e che includono sia le fasi preliminari, come l'analisi dei requisiti e la progettazione, sia le fasi che seguono la codifica, come il deployment e la gestione dell'applicazione. Tutte queste attività devono prevedere un'attenta e puntuale verifica e validazione di quanto viene prodotto o gestito, in particolare, i documenti che descrivono il problema, il documento di progetto, il codice. Se è vero che la qualità non è ottenuta a valle, ma è costruita insieme al prodotto, allora è evidente che *ogni passo del processo di sviluppo deve dedicare una particolare attenzione al problema della qualità*. Ovviamente, i termini verifica e validazione assumono una particolare visibilità e importanza quando l'oggetto per il quale si vuole valutare la qualità è il codice sorgente vero e proprio. Nel seguito, verranno brevemente discusse alcune tecniche che si applicano principalmente al codice sorgente (in particolare il testing) con alcuni cenni anche ad altri strumenti di uso più generale.

Va innanzitutto notato che, seppur spesso utilizzati in modo indifferenziato, i concetti di verifica e validazione non sono sinonimi e hanno significati e connotazioni diverse. Riprendendo alcune considerazioni introdotte nel Capitolo 3, l'attività di *verifica* ha l'obiettivo di controllare che il prodotto sia stato realizzato coerentemente con le specifiche formulate, quindi che sia privo di bachi o problemi intrinseci: "il programma è giusto". La *validazione* ha l'obiettivo di controllare che si sia realizzato "il giusto prodotto": il software risponde alle reali esigenze dell'utente.

Le tecniche e i metodi di verifica e validazione discussi nel resto del capitolo sono due.

- *Testing*: è l'osservazione del comportamento del programma in esecuzione a fronte di un insieme di dati in ingresso e la verifica della coerenza dei risultati ottenuti rispetto alle specifiche.
- *Tecniche di analisi manuali*: è l'esame "manuale" del codice (o di altri semilavorati) allo scopo di trovare anomalie e difetti.

In realtà esistono anche altre tecniche e metodi di verifica e validazione; nel seguito ne sono enumerate tre che, per motivi di spazio, non saranno discusse.

1. *Analisi formale di modelli statici*: il codice (o una qualche descrizione espressa in un linguaggio formale) viene analizzato (senza che vi sia una reale esecuzione) con tecniche formali per valutarne o provarne proprietà.
2. *Esecuzione simbolica*: il codice di un programma, o una descrizione espressa con un linguaggio formale, viene eseguito associando espressioni simboliche alle variabili. Ciò permette di studiare il comportamento generale del programma senza che sia necessario passare a una sua reale esecuzione.

3. *Model checking*: il sistema viene descritto mediante un automa a stati finiti. Le proprietà di interesse sono espresse attraverso formule. La verifica avviene osservando se il comportamento del sistema è coerente con le formule (e quindi le proprietà).

9.1.1 Testing

Lo scopo dell'attività di testing è *scoprire errori che sono causa di malfunzionamenti*. Un *malfunzionamento* è definito come un comportamento non previsto e non desiderato che denota una mancata aderenza alle specifiche. Un *errore*, invece, è la causa che provoca il malfunzionamento. Una volta scoperto il malfunzionamento è necessario localizzare l'errore ed eliminarlo (*debugging*).

L'attività di testing viene svolta sollecitando il programma attraverso i *casi di test*: essi sono insiemi di dati forniti in ingresso al software da testare. Hanno lo scopo di rendere possibile il controllo di corrispondenza dei risultati ottenuti con quanto descritto nelle specifiche. La corretta identificazione dei casi di test è fondamentale per la buona riuscita di un test: infatti, i casi di test dovrebbero essere definiti in modo da valutare tutte le funzionalità che il software deve fornire.

È essenziale osservare che il testing non ha come scopo quello di dimostrare l'assenza di errori. Il *testing dimostra la presenza di errori*. Ciò significa che per avere una ragionevole certezza che il programma contenga "pochi errori" è necessario condurre un'attività di testing estesa e sistematica: maggiore è il numero di test eseguiti, maggiore è la probabilità che il programma funzioni "bene". È peraltro evidente che condurre attività di testing costa: è necessario definire i casi di test e, soprattutto, utilizzarli in modo sistematico per eseguire il programma e verificare la corrispondenza dei risultati con le specifiche. Esiste quindi un *trade-off* tra la sicurezza che il programma funzioni bene e il costo delle attività di testing: il progettista deve di volta in volta capire "quanto testare" in funzione della criticità del programma, dei rischi ai quali sono esposti gli utenti e del costo complessivo dell'attività di testing. Non ha senso, infatti, "testare" nello stesso modo un videogioco e il software di controllo di un aeroplano. Nel corso degli anni sono stati anche sviluppati modelli matematici che correlano affidabilità del software, intensità delle attività di testing e loro costo (per esempio, quello sviluppato da John Musa all'AT&T negli anni '80). Tali modelli hanno come scopo proprio quello di fornire un supporto metodologico per capire "dove fermarsi" in una campagna di testing.

Ma qual è il significato dell'espressione "quanto testare"? Cosa significa "andare avanti nel testing" o "fermarsi"? Per rispondere a queste domande è necessario introdurre le due tecniche di testing disponibili:

- test white box
- test black box

182 Capitolo 9 Qualità del prodotto e del processo

Il *test white box* è un *test strutturale*, eseguito sulla base di un'analisi diretta dell'implementazione. In particolare, i casi di test sono definiti in modo da sollecitare le diverse parti del codice sorgente: maggiore è la capacità di sollecitarle in modo esaustivo, più efficace (e più costosa) è l'attività di testing.

Per definire in modo più puntuale il significato dell'espressione "sollecitare in modo esaustivo", nel caso del test strutturale si utilizza il concetto di copertura. Con *copertura del codice* si intende l'intensità secondo la quale il codice è stato sollecitato attraverso l'utilizzo dei casi di test selezionati. Maggiore è la copertura, più accurata è l'attività di testing. Il concetto di copertura viene espresso attraverso una *serie di criteri* che ne esplicitano l'intensità e, conseguentemente, l'accuratezza. Essi indicano come selezionare i casi di test in modo da ottenere un certo *grado di copertura*.

- *Statement coverage*. Secondo questo criterio, i casi di test sono selezionati in modo da sollecitare almeno una volta tutte le istruzioni (*statement*) del codice considerato.

```
1 if (a>0) {
2   a = a; }
```

L'esempio riportato sopra mostra un semplice costrutto if-then-else scritto in linguaggio C. Secondo il criterio dello statement coverage, bisogna garantire l'esecuzione delle due istruzioni presenti. Nell'esempio mostrato, basta un caso di test con valore $a > 0$: per esempio $a = 3$. In questo modo viene sollecitata sia l'istruzione *if* che l'assegnamento presente nel ramo *then*. Si noti che questo test risulta in molte circostanze largamente insufficiente. Supponendo che il frammento dell'esempio rappresenti il codice che deve calcolare il valore assoluto di un numero, il programma manca ovviamente di un ramo *else* nel quale, se il numero è negativo, viene effettuato l'assegnamento $a = -a$. Tale errore, molto banale, non viene in alcun modo identificato dal caso di test prescelto. Il motivo è molto semplice: il caso di test sollecita solo il ramo *then* dell'*if* e non valuta ciò che accade se la condizione dell'*if* non è verificata. Se si scegliesse anche il caso $a = -5$, allora diventerebbe evidente che il programma non è in grado di calcolare il valore assoluto di un numero negativo. Per rilevare questo problema occorre introdurre un nuovo caso di test che non sarebbe necessario secondo il criterio dello statement coverage. Quanto visto introduce un criterio più stringente: *edge coverage*.

- *Edge coverage*. Secondo questo criterio i casi di test sono definiti in modo da sollecitare almeno una volta tutte le diramazioni (*flusso di controllo*) presenti nel codice considerato, come ad esempio i due rami del costrutto if-then-else. In questo caso, la criticità è data dal numero di diramazioni che possono essere presenti in un programma. Per garantire che tutti i rami siano percorsi almeno una volta, è utile tracciare il *grafo di controllo* che rappresenta, attraverso nodi (le istruzioni) e archi (il flusso di controllo), ogni possibile diramazione del codice: la copertura è ottenuta facendo sì che ogni diramazione del grafo sia eseguita almeno una volta. La Figura 9.1 mostra un esempio di programma, mentre la Figura 9.2 mostra il grafo di controllo corrispondente. Il caso di test $a=-2, b=5$ permette di sollecitare le

```
1   if (a<0)
2   {
3       if (b>0)
4       {
5           a = a+2*b;
6       };
7       printf(a);
8   }
9   else
10  {
11      a = a*2;
12  }
```

Figura 9.1 Esempio per criterio di copertura degli archi (edge coverage).

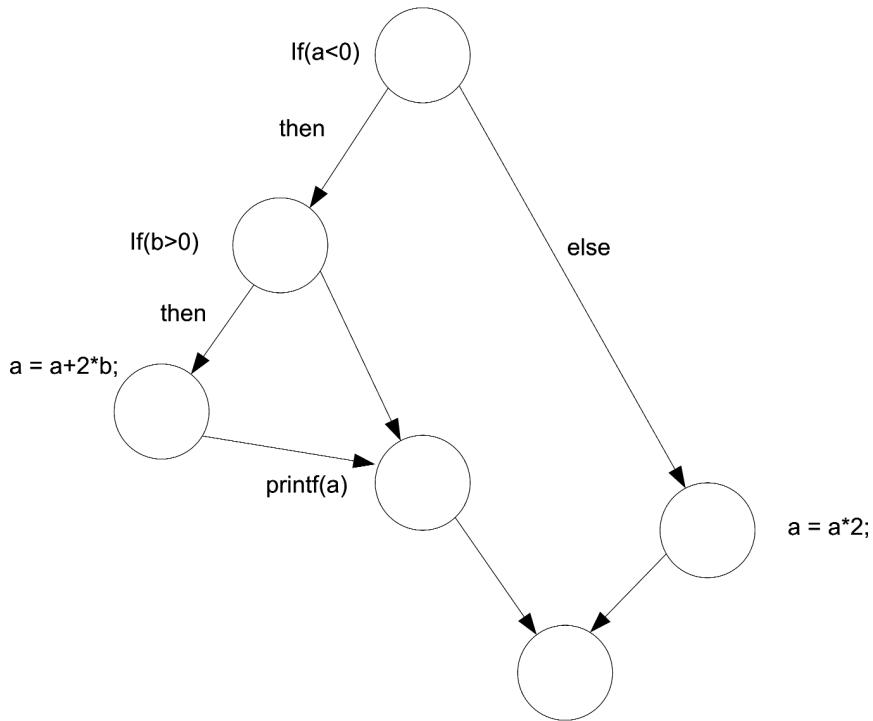


Figura 9.2 Grafo di controllo del programma della Figura 9.1.

184 Capitolo 9 Qualità del prodotto e del processo

istruzioni del ramo 1, 2, 3, 4, 5, 6, 7. Il caso di test $a=-2, b=-3$ non esegue il ramo `then` dell'`if` della riga 3. Il caso di test $a=6, b=qualunque$ sollecita il ramo `else` alla riga 9 (e seguenti). È facile verificare che i tre casi di test permettono di sollecitare tutti i rami del grafo di controllo della Figura 9.2.

- *Condition coverage.* Negli esempi visti si è sempre verificato il caso in cui le condizioni che determinano l'esecuzione di un particolare insieme di istruzioni fosse di tipo molto semplice. In generale, accade che le condizioni utilizzate nel corpo di un programma siano di tipo composito. Per esempio, si consideri la condizione seguente:

```
if(a<0 || b>0)
```

Il ramo `then` è eseguito se una o entrambe le condizioni elementari sono vere. Coerentemente, se entrambe le condizioni sono false, viene eseguito il ramo `else`. In questa situazione possono accadere errori dovuti al fatto che la combinazione dei valori delle condizioni elementari presenti nell'istruzione `if` produce situazioni non desiderate e quindi malfunzionamenti. Per avere una più attenta copertura del codice (e quindi un test più efficace) i casi di test devono sollecitare tutte le combinazioni possibili delle condizioni che permettono di seguire un particolare ramo del codice. Nell'esempio considerato i casi di test necessari sono quattro: a maggiore di zero e b minore di zero, a minore di zero e b maggiore di zero, a e b maggiori di zero, a e b minori di zero. Di questi, il primo attiva il ramo `else`, mentre gli altri tre attivano il ramo `then`.

- *Path coverage.* Questo criterio è il più stringente. Secondo tale criterio è necessario coprire tutti i possibili cammini del flusso di controllo del programma. Nell'esempio della Figura 9.3, che include due costrutti `if-then-else`, non basta attraversare le diverse diramazioni almeno una volta, ma è necessario farlo secondo tutte le combinazioni possibili. Anche in questo caso può essere d'aiuto il grafo di controllo: la Figura 9.4 mostra tutte le combinazioni presenti nell'esempio. Come si vede, il numero di casi di test necessari per soddisfare il criterio aumenta molto velocemente al crescere della complessità del grafo di controllo.

Nella definizione dei casi di test è necessario tenere conto non solo del criterio di copertura prescelto, ma anche delle *boundary condition*. Spesso, le condizioni che determinano il flusso di esecuzione del programma presentano situazioni "di confine" che sono molto critiche. Si pensi al caso di un test che deve verificare che il contatore utilizzato per scorrere un array abbia un valore minore o uguale al numero massimo di elementi presenti. Se ciò fosse espresso da una condizione del tipo $i < n$ (nell'ipotesi che l'array abbia n elementi memorizzati nelle posizioni da 0 a $n-1$) risulta conveniente definire anche un caso di test che verifichi $i == n$, il caso di confine, che potrebbe segnalare errori di solito difficili da scoprire.

Un'ultima osservazione concerne la gestione dei cicli. Per definire i casi di test che sollecitano opportunamente i cicli presenti in un programma è necessario considerare sia

9.1 Verifica e validazione 185

```

1  if(a>0) {
2      printf("caso a>0");
3  }
4  else {
5      printf("caso a<0");
6  };
7  if(b<0) {
8      printf("caso b<0");
9  }
10 else {
11     printf("caso b>0");
12 }

```

Figura 9.3 Esempio per criterio di copertura dei cammini (path coverage).

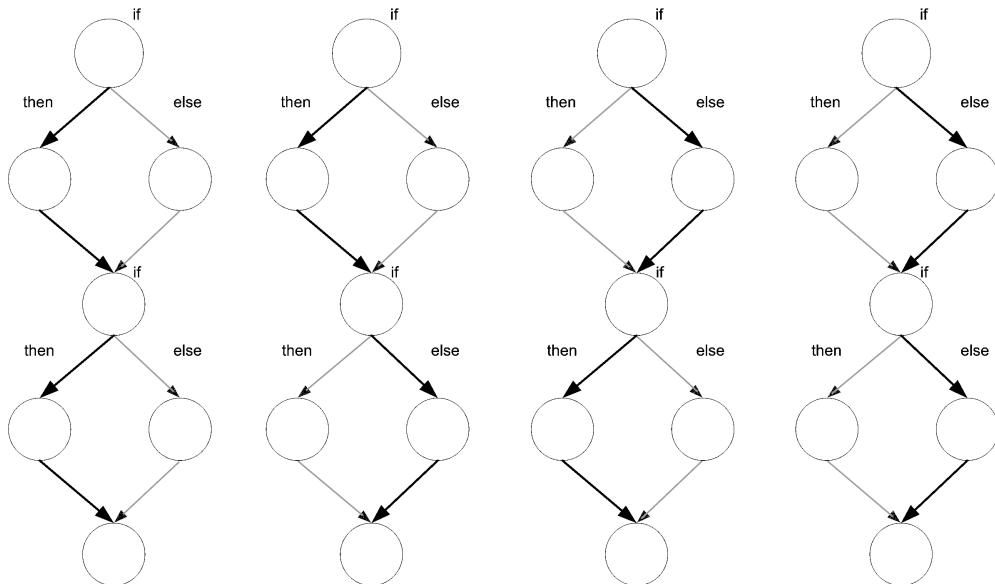


Figura 9.4 Grafo di controllo per l'esempio della Figura 9.3.

i cicli con numero di iterazioni predeterminato (`for`) che quelli a condizione iniziale e finale (`while`, `repeat-until`). In generale, il numero di casi di test necessario per provare in modo esaustivo questo tipo di istruzioni può crescere fino a rendere il testing im-

186 Capitolo 9 Qualità del prodotto e del processo

praticabile. Alcuni criteri utilizzabili per determinare un numero praticabile di casi di test sono i seguenti:

1. si esegue il ciclo zero volte o un numero di volte minimo
2. si esegue il ciclo il numero massimo di volte
3. si esegue il ciclo un numero medio di volte.

Il test white box è impegnativo e costoso. Per questo motivo si applica solo alle parti di codice ritenute particolarmente critiche. Purtroppo, il test white box, oltre a non essere adatto per programmi di dimensioni elevate, ha un ulteriore limite: non riesce a individuare malfunzionamenti dovuti all'assenza di parti di codice. È possibile individuare la mancanza di una funzionalità del sistema solo con test che si basano sulle specifiche, come il test black box.

Il *test black box* è un *test funzionale*: i casi di test sono definiti considerando le specifiche e non il codice implementato. Il test black box consente di controllare tutte le funzionalità che il software deve fornire, individuando anche le eventuali mancanze o carenze. L'efficacia di questo tipo di test dipende dalla precisione delle specifiche: le specifiche descritte in linguaggio formale sono più efficaci e utili, in quanto sono meno ambigue di quelle descritte in linguaggio naturale, e quindi da esse è più facile derivare casi di test affidabili e completi.

Per effettuare il test funzionale di un programma occorre definire due tipologie di casi di test: input validi e input non validi o volutamente inesatti. Ad esempio, si consideri un programma che dato un numero fornito in una certa base numerica lo traduce in un'altra base: le basi ammesse vanno dalla binaria all'esadecimale. Il primo caso di test per verificare la correttezza del programma sarà formato da un numero di ingresso corretto e da una base prevista dal programma, come ad esempio il numero binario 110110 da trasformare in decimale. Il secondo caso di test può prevedere un numero di partenza non corretto, ad esempio che contenga dei caratteri non numerici come il numero 1*0r&40, o una base non prevista dal programma. Il terzo caso di test può prevedere che si scelga come base lo zero.

Nel caso del test black box, le boundary condition sono valori limite che il progettista ricava dalle specifiche. Se tra le specifiche è richiesta la divisione tra due numeri, il valore limite da considerare è lo zero, valore per cui la divisione è notoriamente impossibile. Occorre verificare su questa boundary condition che il programma reagisca gestendo correttamente la situazione.

L'aspetto critico del test black-box è trovare il maggior numero di casi di test con valori particolari o non validi che si possano estrapolare dalle specifiche, in modo da controllare che il programma si comporti in tutti questi casi in modo accettabile e controllato. La definizione dei casi di test può essere svolta anche prima della completa realizzazione del codice, proprio perché i casi sono definiti sulla base delle specifiche che il sistema dovrà rispettare.

In generale, il progettista dovrà di volta in volta decidere la strategia di testing più adatta in funzione delle caratteristiche del prodotto in corso di sviluppo. Ciò richiederà non solo la determinazione della combinazione più appropriata di test funzionale e strutturale per le diverse parti del codice, ma anche il livello di intensità con il quale effettuare il testing.

9.1.2 Tipologie di test

Esistono diverse tipologie di attività di test, svolte con scale, tecniche e interlocutori differenti, in funzione della loro collocazione all'interno del processo di sviluppo del software. In particolare, le principali sono:

- test di modulo
- test di integrazione
- test di sistema: alfa e beta
- test di regressione.

Il *test di modulo* ha l'obiettivo di verificare il corretto funzionamento di un componente elementare del programma come una classe o un tipo di dato astratto. È questa la fase dove si utilizzano in modo particolarmente intenso le tecniche di testing funzionale e strutturale. Poiché spesso un programma è composto da più moduli, per poterlo sottoporre a test è necessario "simulare" le altre parti del programma con cui interagisce. In caso contrario, sarebbe necessario sottoporre a test l'intero programma, con un'evidente crescita di complessità. La simulazione del contesto nel quale un modulo è utilizzato si ottiene tramite opportune componenti software dette stub e driver (Figura 9.5). Un *driver* è un programma che simula il modulo chiamante, vale a dire quella parte di architettura che nella realtà fornisce gli input al modulo in fase di test. Uno *stub* è un programma che simula un modulo chiamato, cioè la parte di architettura che il modulo in fase di test utilizza per svolgere le proprie funzioni. Stub e driver nei casi più semplici sono delle interfacce che servono, da una parte a immettere i dati dei casi di test definiti, dall'altra a visualizzare i risultati e controllare che corrispondano agli output attesi dalle specifiche.

Il *test di integrazione* è eseguito su un sottoinsieme di moduli per verificare che cooperino correttamente. I moduli possono essere aggregati in base alle macrofunzionalità offerte.

Il *test di sistema* ha l'obiettivo di controllare che il prodotto software risponda ai requisiti espressi dall'utente. In particolare, sono due i test di sistema più conosciuti:

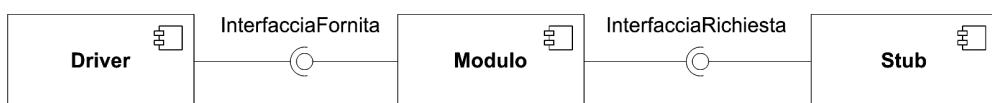


Figura 9.5 Driver e stub per il test di un modulo.

alfa test e *beta test*. Entrambi consistono nell'uso del prodotto prima di rilasciarlo ufficialmente al cliente, in modo da identificare malfunzionamenti non evidenziati dai test precedenti. Si tratta di attività svolte secondo un approccio black box: si utilizza il programma e se ne osserva il comportamento. Solitamente, i test alfa e beta non sono affidati ai programmatore che hanno realizzato il prodotto, ma a persone estranee al team di sviluppo. In particolare, l'alfa test è rivolto a persone interne all'impresa, mentre il beta test prevede come utilizzatori piloti alcuni clienti selezionati a cui viene rilasciato il prodotto "in prova". Sempre più spesso, in realtà, i programmi vengono rilasciati in beta test al pubblico, rendendoli liberamente scaricabili dai siti web dei produttori.

Il *test di regressione* è effettuato a fronte del rilascio di una nuova versione del software. Spesso accade che in tali frangenti il software sia stato modificato con l'introduzione (ovviamente involontaria) di errori che rendono inutilizzabili funzionalità precedentemente disponibili. Il test di regressione ha lo scopo di verificare che tutti i test che erano stati "passati" dalla precedente versione del programma lo siano anche per la nuova. Si tratta, in sintesi, di verificare che la nuova versione, oltre a offrire nuove funzionalità, non precluda o pregiudichi quanto già in precedenza offerto al cliente.

9.1.3 Tecniche di analisi manuali

Queste tecniche fanno parte dei metodi di analisi statica: non richiedono l'esecuzione del codice. Sono sostanzialmente riconducibili a un esame "manuale" del codice o di altri semilavorati prodotti durante il progetto, con l'obiettivo di identificarne anomalie e difetti. Normalmente, queste attività sono condotte da persone differenti da coloro che hanno sviluppato il codice o creato i documenti oggetto di analisi. Ciò perché è spesso più facile per una persona esterna al team di sviluppo identificarne limiti, carenze ed errori.

Un primo insieme di tecniche di questo tipo sono le *software inspection*, metodi formali di revisione del prodotto software. Una software inspection si articola su una serie di incontri cui partecipa un team di ispettori esterni al team di progetto. Lo svolgimento dell'ispezione si svolge in tre fasi: una prima di preparazione all'incontro, poi la riunione vera e propria di tutto il team di ispettori e, infine, una fase di follow-up. Il coordinamento del team durante le tre fasi e la gestione dell'ispezione è affidata all'*inspection leader*, che è un membro dello staff tecnico o del gruppo di controllo della qualità aziendale. Per garantire una maggior obiettività, in alcuni casi può essere scelta anche una persona esterna al progetto.

La *fase preparatoria* dell'ispezione consiste nell'analisi, da parte di ogni ispettore, del documento oggetto di ispezione (sia esso codice o un documento di progetto) al fine di trovare errori, anomalie o problemi. In questa fase ogni persona è guidata da una checklist. La *checklist* è un elenco di punti di attenzione che devono essere considerati durante l'ispezione. Vi sono diverse tipologie di checklist in funzione del tipo di documento analizzato e dell'obiettivo dell'ispezione stessa. La checklist è fornita agli ispettori dall'*inspection leader*, insieme a tutto il materiale necessario per l'ispezione, come ad esempio le politiche di ispezione, la descrizione delle procedure e l'agenda degli incontri. L'ispezione eseguita individualmente da ogni membro del team identifica una prima serie di errori, ambiguità e anomalie nel documento. La *riunione di ispezione* prevede la let-

Considerazione di carattere generale

I due approcci discussi, testing e tecniche di analisi manuali, sono complementari: hanno caratteristiche diverse e rispondono a esigenze e problemi differenti. In particolare, le tecniche di analisi manuali sono mediamente più efficienti perché il numero di errori trovati a parità di effort è maggiore. Le software inspection sono in grado di trovare degli errori che possono sfuggire alle procedure di testing, ad esempio per problemi di copertura. I walkthrough possono mettere in luce problemi progettuali o carenze nel processo di studio del problema. Rispetto ai test di tipo black box, però le revisioni non possono essere eseguite dagli utenti finali e di conseguenza non colgono il loro punto di vista. Inoltre, a differenza dei test, le ispezioni non possono considerare e controllare i requisiti non funzionali.

In sintesi, la verifica e validazione di un prodotto software complesso richiede un mix di interventi e di attività che deve essere definito caso per caso, in funzione del tipo di prodotto che si sta sviluppando, della sua criticità e dei vincoli e requisiti posti dall'utente finale.

tura attenta del documento considerato e la discussione all'interno del team degli errori e problemi trovati nella fase precedente. In particolare, un membro del team nominato dall'inspection leader ha il compito di registrare tutto quanto discusso durante la riunione. Al termine dell'incontro viene prodotto un documento in cui si registrano tutti i problemi riscontrati. La *fase di post ispezione (follow-up)* prevede la risoluzione, da parte del team di sviluppo, dei problemi evidenziati nella riunione di ispezione. È compito dell'inspection leader verificare che tutti i problemi segnalati siano stati considerati e risolti: solo allora il processo di ispezione può dirsi concluso.

Il *walkthrough* è un metodo di revisione tecnica meno formale dell'ispezione. Consiste nell'analisi, da parte di un team di esperti opportunamente selezionato, del documento di progetto o del codice: questi due tipi di documenti, infatti, sono quelli che meglio si prestano a essere revisionati con questo metodo. All'inizio dell'incontro sono forniti dei dati di input per il codice da analizzare: partendo da quei dati i membri del team analizzano riga per riga il codice giocando il ruolo del computer, vale a dire come se eseguissero manualmente il codice. Il responsabile della revisione è colui che ha realizzato il documento da analizzare: molti ritengono che l'efficienza della tecnica del walkthrough sia legata proprio al fatto che il responsabile dell'attività di revisione conosca bene il documento da analizzare.

9.2 Miglioramento del processo

Per quanto visto nei precedenti capitoli, il processo che porta alla realizzazione di una soluzione informatica complessa è esso stesso un'entità molto articolata. Descrivere il problema, progettare e sviluppare la soluzione, verificarne le caratteristiche di qualità sono attività che richiedono grandi capacità personali, un'organizzazione adeguata e tecnologie appropriate ed efficaci. Anche se non esiste una correlazione diretta e immediata, è

190 Capitolo 9 Qualità del prodotto e del processo

indubbio che *la qualità del processo influenza la qualità del prodotto realizzato*: se il processo è di scarsa qualità, maggiori saranno i rischi che la soluzione prodotta possa essere inadeguata rispetto agli obiettivi e ai bisogni per i quali è stata concepita.

L'affermazione suddetta utilizza volutamente le parole "influenza" e "rischi". Non esiste infatti una relazione di causalità diretta e immediata tra qualità del processo e qualità del prodotto. Esistono esempi di aziende che, pur non avendo processi di altissima qualità, riescono a essere "di successo", fornendo al cliente un prodotto che viene ritenuto accettabile e pienamente rispondente alle proprie esigenze. In realtà, il problema di fondo è proprio quello del *rischio che si intende accettare*: con un processo di bassa qualità aumentano fortemente i rischi che il prodotto non abbia le caratteristiche che lo rendono accettabile (o meglio desiderabile) dal cliente finale. Studiare e migliorare la qualità dei processi non costituisce automaticamente una garanzia di successo, ma senza dubbio offre una serie di indicazioni estremamente importanti per coloro che devono valutare la capacità di un'azienda di fornire prodotti e servizi di elevata qualità.

Chi è interessato a valutare la qualità di un processo? Fondamentalmente tre tipi di soggetti.

- *Gli acquirenti di applicazioni informatiche.* L'esempio più noto è quello del Dipartimento della Difesa statunitense (DOD), uno dei più grandi acquirenti di software al mondo. Il DOD ha definito un metodo per valutare la *maturità* dei processi di sviluppo utilizzati dai propri fornitori.
- *Gli utenti finali.* Spesso chi compra non è la stessa persona che usa il bene acquistato. Una grande azienda (per esempio una banca) acquista soluzioni informatiche attraverso i propri uffici tecnici. Tali soluzioni sono poi utilizzate da una miriade di utenti (gli impiegati della banca) che non sono stati direttamente coinvolti nelle attività di sviluppo del software. Più in generale, nel caso di software venduto a singoli utenti (si pensi a programmi come Office acquistati anche da singoli cittadini), essi non hanno la possibilità (e il peso contrattuale) di interagire direttamente con gli sviluppatori. In questo caso, esiste un problema di garanzia del mercato: chi compra vorrebbe essere certo che ciò che viene venduto rispetta requisiti minimi di qualità (così come avviene in altri settori merceologici).
- *Le aziende stesse che producono software.* Un fornitore di software attento ai bisogni della propria clientela sarà per primo interessato a far sì che il proprio processo di sviluppo sia conforme a elevati standard qualitativi.

In generale, la qualità o maturità di un processo viene valutata per due diverse esigenze: l'affidabilità di un fornitore (i primi due casi discussi in precedenza) o la propria capacità di sviluppare software (il terzo). Qualunque sia il motivo per il quale si vuole valutare la bontà di un processo, è possibile identificare tre fasi di questa attività.

1. *Valutazione di maturità (capability determination).* In questa fase si studia il processo al fine di determinarne il livello di qualità attuale, la sua maturità. Si tratta di definire in quale stato il processo si trovi. Questo passaggio è necessario e propedeutico a qualunque altra iniziativa.

9.2 Miglioramento del processo 191

2. *Definizione di un programma di miglioramento.* Normalmente, alla valutazione di maturità segue la definizione di un programma di miglioramento: è necessario porre rimedio alle carenze eventualmente identificate. Un programma di miglioramento deve essere definito alla luce delle carenze riscontrate in fase di valutazione di maturità, degli obiettivi aziendali e dei vincoli di mercato. Ci sono tanti modi per “migliorare” e ciascuno deve individuare i passaggi più adatti al proprio contesto. È essenziale il pieno, convinto e “visibile” coinvolgimento e impegno del management aziendale.
3. *Attuazione del programma di miglioramento.* Ovviamente, la parte più complessa è l'esecuzione e la “messa in pratica” di quanto previsto dal programma di miglioramento. Quand'anche ci fossero pieno consenso e chiarezza sui passaggi da seguire, è indubbio che cambiare e innovare le modalità secondo le quali un'organizzazione complessa conduce le proprie attività rappresenta da sempre uno dei problemi più critici studiati dai ricercatori di management e teorie organizzative.

Il miglioramento di un processo è, per quanto visto, un'attività critica che richiede risorse altamente qualificate e il pieno coinvolgimento di tutte le persone di un'impresa. Per raccogliere con successo questa sfida così complessa nel corso degli anni sono stati sviluppati una serie di metodi, approcci e linee guida che indicano percorsi e suggeriscono passaggi intermedi sulla strada dell'eccellenza. Nel seguito sono presentati tre metodi molto diffusi: CMMI, ISO 9000 e QIP. Essi si fondano su due approcci teorici diversi. CMMI e ISO 9000 definiscono un profilo ideale e misurano la maturità di un'azienda in base alla sua “distanza” da tale profilo. QIP è un metodo basato sull'approccio giapponese del miglioramento continuo: non esiste un modello di riferimento e di volta in volta si definiscono gli obiettivi di miglioramento considerati più adatti alla situazione della singola azienda. A valle della presentazione di questi tre approcci, sono introdotti il concetto di metrica del software e un metodo per la definizione di un programma di raccolta dati denominato GQM. Tale metodo può essere utilizzato in molte attività di miglioramento di processo, ogni qual volta è necessario identificare e raccogliere dati quantitativi che illustrino in modo obiettivo la situazione dell'azienda.

9.2.1 Miglioramento per stadi: il CMMI

Il *Capability Maturity Model Integration* (CMMI) definisce un modello di riferimento per analizzare lo stato del processo di un'azienda e in particolare la sua maturità. Inoltre, fornisce un supporto per la selezione delle aree critiche e dei punti in cui intervenire all'interno del processo considerato al fine di migliorarne il livello di maturità.

Il CMMI è stato sviluppato dal Software Engineering Institute (SEI) della Carnegie Mellon University di Pittsburgh (USA). A sua volta, il SEI è stato costituito dal DOD per sostenere la crescita e lo sviluppo delle aziende di software fornitrice della Difesa. In particolare, il DOD ha richiesto al SEI la definizione di un metodo di valutazione della maturità dei propri fornitori.

Il CMMI permette di definire il livello di maturità di un processo software esaminandone diverse caratteristiche: processo di sviluppo del software in senso stretto e processi di ingegneria di prodotto in generale (si pensi al caso in cui il software è parte di un prodotto più complesso). Il livello di maturità di un processo è valutato osservando il grado di reale applicazione di una serie di pratiche dell'ingegneria del software. Le pratiche sono raggruppate in alcune macro aree.

1. *Gestione del processo*: comprende le pratiche relative all'organizzazione e gestione del processo nel suo complesso. In particolare, tutto ciò che concerne lo studio del processo e il suo miglioramento. Un processo di qualità prevede esso stesso gli strumenti per crescere e "migliorare".
2. *Gestione del progetto*: comprende le pratiche relative alle attività di pianificazione e controllo di un progetto. Inoltre, appartengono a quest'area anche le tematiche relative alla gestione dei rapporti con i fornitori e alla gestione del rischio.
3. *Ingegnerizzazione*: comprende le pratiche relative alle attività di sviluppo del prodotto software vere e proprie, come ad esempio, la definizione dei requisiti, la progettazione della soluzione, la verifica e la validazione.
4. *Supporto*: comprende le pratiche relative alle attività svolte durante il progetto, ma che non rientrano nella realizzazione diretta del prodotto, come ad esempio il configuration management.

La maturità di un processo è espressa attraverso una serie di livelli che misurano la vicinanza dell'azienda a un modello ideale di processo.

- Livello 1 (*Initial*): a questo livello il processo non è organizzato, è definito implicitamente e in modo non chiaro. Il flusso delle attività non segue un percorso preciso. Quand'anche si raggiungono gli obiettivi aziendali, ciò è dovuto in massima parte alla volontà e capacità dei singoli. Un'azienda il cui processo si trovi al livello 1 è particolarmente fragile ed esposta a ogni genere di rischio, come ad esempio, l'incapacità di operare in modo efficace a fronte della perdita di alcune delle proprie persone chiave.
- Livello 2 (*Repeatable*): a questo livello le performance del processo sono ripetibili in progetti simili. Questo perché sono diffusamente impiegate tecniche per la pianificazione, monitoraggio e controllo di progetto, condotte a fronte di una definizione chiara degli obiettivi dell'azienda.
- Livello 3 (*Defined*): alla base del processo c'è una definizione comune, a livello aziendale, delle metodologie, delle tecniche e delle tecnologie che in esso devono essere utilizzate.
- Livello 4 (*Quantitatively managed*): il processo è monitorato e controllato ricorrendo all'uso di metriche di prodotto e di processo.
- Livello 5 (*Optimizing*): l'uso di metodi quantitativi è consolidato e istituzionalizzato all'interno dell'azienda. I dati sono utilizzati in modo sistematico e continuo per migliorare le prestazioni del processo.

9.2 Miglioramento del processo 193

Per qualificarsi a un certo livello di maturità, un processo deve soddisfare i requisiti di quel livello e di quelli “precedenti”. Per esempio, un processo di livello 4 soddisfa tutti i requisiti dei livelli 2, 3 e, ovviamente, 4.

La definizione del livello di maturità avviene attraverso interviste e analisi in loco che valutano il grado di esecuzione delle diverse pratiche. Con il termine *grado di esecuzione* si intende la reale e sistematica applicazione della pratica considerata nelle diverse attività dell’azienda. Un’azienda potrebbe anche aver definito nel dettaglio che fare, ma se ciò non si traducesse in una reale innovazione di processo, il livello di maturità deve essere necessariamente considerato basso.

Le interviste effettuate hanno lo scopo di far conoscere meglio l’azienda non solo a chi utilizza il CMMI a fini della valutazione, ma soprattutto all’azienda stessa, vale a dire al management e alle persone che operano nel processo. In funzione delle informazioni ricavate dalle interviste, si definisce con il management come e dove intervenire per migliorare le aree ritenute più deboli e per far sì che tutte le pratiche considerate conseguano il grado di maturità necessario per permettere all’azienda di passare al livello successivo.

9.2.2 Standard ISO 9000

Uno degli standard più conosciuti, non solo nel settore dell’ingegneria del software, è ISO 9000, o meglio la famiglia di standard ISO 9000. Questi standard sono definiti dall’*International Organization for Standardization* (ISO). L’organizzazione produce una serie di standard rivolti alle imprese e alle pubbliche amministrazioni.

Prima di procedere a una valutazione puntuale dello standard ISO 9000, è utile ricordare brevemente alcuni concetti di base.

- **Standard:** insieme di requisiti che definiscono proprietà di un prodotto o di un processo. Per esempio, lo standard ANSI C definisce le caratteristiche del linguaggio C secondo l’American National Standards Institute. Un prodotto o processo si dice *conforme* a uno standard se rispetta i requisiti in esso definiti.
- **Ente di standardizzazione:** organismo che emette lo standard. Per esempio, sono enti di standardizzazione il già citato ISO e l’UNI (Ente Nazionale Italiano di Unificazione). L’UNI partecipa, in rappresentanza dell’Italia, all’attività normativa degli organismi sovranazionali di normazione: ISO (International Organization for Standardization) e CEN (Comité Européen de Normalisation).
- **Ente di certificazione:** organismo che verifica se un prodotto o processo è conforme a uno standard. Per esempio, in Italia un ente di certificazione è l’Istituto Marchio Qualità (IMQ).
- **Ente di accreditamento:** organismo che riconosce e autorizza a operare gli enti di certificazione. In Italia, tale ruolo è svolto da SINCERT (Sistema Nazionale per l’Accreditamento degli Organismi di Certificazione e Ispezione).

Lo standard ISO 9000 definisce le *caratteristiche di un sistema di qualità aziendale*. Si applica a un vasto insieme di settori industriali e di servizi. È ampiamente applicato alle imprese che progettano, sviluppano e gestiscono software. Viene utilizzato per verificare se

194 Capitolo 9 Qualità del prodotto e del processo

il processo di sviluppo dell'azienda considerata è conforme ai requisiti di qualità contenuti nello standard. È importante sottolineare che lo standard ISO 9000 è distinto dall'ISO 9126 (discusso nel Capitolo 3). Anche se entrambi mantengono una numerazione simile, hanno scopi e finalità diverse. Il primo è uno *standard di processo*, mentre il secondo è uno *standard di prodotto*. I requisiti che sono contenuti nei due standard sono quindi sostanzialmente diversi, anche se è indubbio che vi sia una forte correlazione logica: un processo di qualità deve avere come obiettivo lo sviluppo di un prodotto di qualità.

Al contrario del CMMI, nel caso dell'ISO 9000 non esiste una gradualità di giudizio: *un processo può solo essere o meno conforme allo standard* (e quindi certificato). Vi sono più enti di certificazione che nei vari paesi verificano la conformità di un processo allo standard ISO 9000. Normalmente, la certificazione avviene attraverso una visita ispettiva che verifica la presenza di un sistema di qualità conforme allo standard e la sua piena applicazione all'interno del processo di sviluppo. Eventuali scostamenti o carenze sono definite *non conformità*. Nel caso di non conformità gravi, la certificazione è rifiutata. La certificazione può essere riconosciuta anche in presenza di non conformità lievi. In ogni caso, la conformità viene rilasciata per un periodo di tempo definito (tipicamente un anno). Al termine del periodo, la certificazione scade e deve essere rinnovata attraverso una nuova visita ispettiva.

Gli standard che fanno parte della famiglia ISO 9000 si sono evoluti e modificati nel tempo: negli anni che vanno dal 1996 al 2000 è stata effettuata una revisione della struttura e dei contenuti degli standard in base al feedback dell'applicazione delle norme contenute nella versione del 1994. Questa nuova versione è identificata come ISO 9000:2000, per indicare la versione più recente con le modifiche apportate nel 2000. Essa include tre documenti principali.

- ISO 9000 "Sistemi di gestione per la qualità - Fondamenti e terminologia": contiene informazioni simili a quanto qui proposto, con lo scopo di fornire un'introduzione allo standard.
- ISO 9001 "Sistemi di gestione per la qualità - Requisiti": questo standard definisce i requisiti veri e propri dei sistemi di gestione per la qualità che devono essere soddisfatti da un'organizzazione perché possa essere certificata.
- ISO 9004 "Sistemi di gestione per la qualità - Linee guida per il miglioramento delle prestazioni": questo standard fornisce una guida per il miglioramento continuo del processo.

Gli standard della famiglia ISO 9000 sono stati definiti sulla base di otto principi.

- *Focalizzazione sul cliente*: l'organizzazione dipende dai propri clienti e deve essere in grado di capire i loro bisogni e possibilmente anticiparli.
- *Leadership*: la leadership del management è fondamentale per garantire l'unità di intenti e di direzione dell'organizzazione. Inoltre, il management deve essere in grado di coinvolgere tutti i partecipanti al processo in modo da ottenere il raggiungimento dei risultati attesi.

9.2 Miglioramento del processo 195

- *Coinvolgimento delle persone*: è importante che l'intero team che è coinvolto in un processo si senta partecipe nelle decisioni dell'organizzazione. In questo modo migliora la collaborazione all'interno del team e tra il team e il management.
- *Orientamento al processo*: questo principio si riferisce al fatto che è più semplice ottenere un risultato se le attività sono strutturate secondo un processo ben definito e organizzato.
- *Gestione integrata ("sistematica") dei processi*: l'efficienza e l'efficacia di un'organizzazione migliorano se diversi processi sono gestiti come facenti parte di un unico sistema integrato focalizzato sull'ottenimento dei risultati aziendali.
- *Miglioramento continuo*: il miglioramento continuo del processo deve essere un obiettivo costante dell'azienda.
- *Processi decisionali basati su evidenze e dati di mercato*: è necessario che le decisioni da prendere siano fondate sullo studio e sull'analisi di dati oggettivi.
- *Relazioni con i fornitori mutuamente convenienti*: questo principio evidenzia il fatto che un'organizzazione e i suoi fornitori sono interdipendenti. È necessario instaurare rapporti con i fornitori che puntino all'ottenimento di vantaggi da ambo le parti, ad esempio in termini di crescita e condivisione del know-how.

9.2.3 Miglioramento continuo: QIP (Quality Improvement Paradigm)

Il metodo *Quality Improvement Paradigm* (QIP) suggerisce che il miglioramento di un processo si debba articolare su tre fasi ripetute ciclicamente.

1. *Understanding*: è la fase conoscitiva del processo utilizzato dall'azienda. In questa fase l'obiettivo è conoscere e analizzare i problemi e i punti di forza del processo, le tipologie di software sviluppato e le caratteristiche distintive dei singoli prodotti realizzati.
2. *Assessing*: in questa fase si misura l'impatto sul processo derivante dall'adozione di specifici metodi e tecnologie o anche di cambiamenti strutturali e organizzativi del processo stesso.
3. *Packaging*: i risultati della fase precedente devono essere valutati al fine di consolidare e arricchire la base di esperienze che costituiscono il patrimonio conoscitivo dell'azienda. È in base a queste conoscenze che l'organizzazione cresce e matura.

Il QIP è in realtà uno schema di lavoro più che un metodo prescrittivo. Se nel CMMI la valutazione del livello di maturità indicava implicitamente anche le priorità di intervento, il QIP suggerisce l'approccio complessivo e la logica secondo le quali procedere, ma lascia del tutto indeterminate le azioni correttive. Esse devono essere identificate e valutate di volta in volta da chi conduce le attività di studio e miglioramento del processo.

L'importanza del QIP risiede nel fatto che costituisce un'alternativa concettuale ai modelli progressivi ("staged") come il CMMI. È una filosofia di fondo diversa che rifiuta l'idea di un modello di riferimento "ottimo" e favorisce il miglioramento graduale guidato da una forte analisi del contesto.

9.2.4 Le metriche e il metodo GQM

Tutte le metodologie presentate si basano sulla comprensione della situazione iniziale del processo e sulla valutazione dell'efficacia degli sforzi fatti per raggiungere gli obiettivi di miglioramento identificati. Queste due operazioni, valutazione dello stato iniziale e dello stato finale, sono eseguibili in modo convincente solo nel caso in cui sia effettivamente possibile una valutazione oggettiva dello stato del processo. In sintesi, per valutare il grado di miglioramento di un processo è necessario ricorrere al concetto di metrica. IEEE definisce *metrica* la valutazione quantitativa del grado di possesso di un attributo da parte di un'entità: un esempio di metrica è il numero di linee di codice di una classe oppure il numero medio di errori trovati in un componente software. Le metriche sono importanti perché forniscono una valutazione oggettiva delle caratteristiche del prodotto software e del processo di sviluppo. Va notato che con il termine metrica s'intende non solo la misura in sé, ma anche le procedure e le modalità di misurazione.

La definizione delle metriche da considerare in un particolare contesto applicativo è un'operazione fondamentale e delicata: il numero di potenziali metriche è a priori illimitato e il costo delle attività di raccolta dati può essere molto elevato. Per questo sono stati definiti alcuni metodi per la definizione e gestione di programmi per la raccolta di dati che minimizzino il numero di informazioni da raccogliere in funzione dell'obiettivo del che si pone il programma stesso: il più noto e diffuso è denominato *Goal/Question/Metric* (GQM) ed è parte integrante del QIP descritto in precedenza. Peraltro, può essere utilizzato ogni volta è necessario definire un programma di raccolta di dati particolarmente complesso o critico, indipendentemente dal fatto che si adotti l'approccio QIP.

Il metodo GQM è basato su tre macro fasi, come suggerisce il nome stesso. In primo luogo devono essere definiti gli obiettivi (*Goal*) che l'azienda si pone nello sviluppare il programma di raccolta dati. A partire dagli obiettivi vengono derivate le domande (*Question*) le cui risposte siano in grado di dare indicazioni sullo stato di raggiungimento degli obiettivi (si veda la Figura 9.6). Infine, il metodo suggerisce come identificare le metriche (*Metric*) che forniscono elementi quantitativi per rispondere alle domande e quindi valutare il raggiungimento degli obiettivi originariamente formulati.

Nel dettaglio, il metodo si articola su una serie di attività (si veda la Figura 9.7).

- *Studio preliminare*: si raccolgono le informazioni che permettono lo studio del contesto nel quale opera l'azienda. In particolare, le informazioni ricercate riguardano le caratteristiche del processo considerato e del prodotto, gli obiettivi strategici e il mercato in cui si colloca l'impresa.
- *Identificazione degli obiettivi GQM*: in funzione del contesto analizzato nell'attività precedente, si definiscono gli obiettivi per il miglioramento del processo. Gli obiet-

9.2 Miglioramento del processo 197

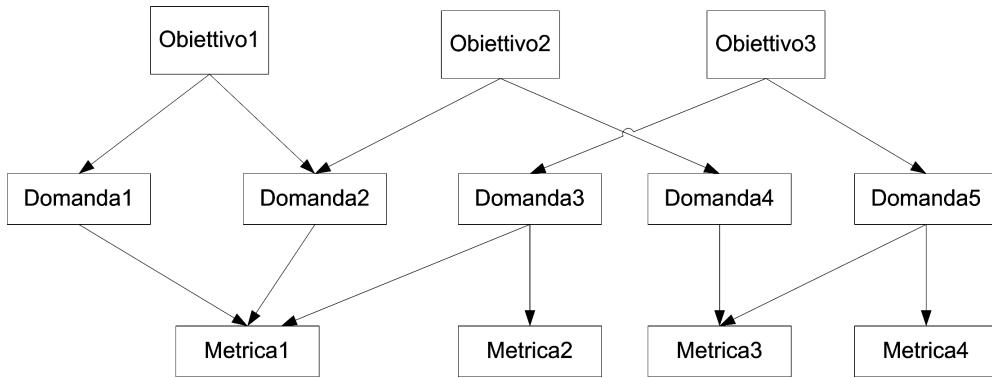


Figura 9.6 Goal (obiettivi), Question (domande), Metric (metriche).

tivi definiti sono poi classificati e organizzati considerando la loro rilevanza ai fini della strategia aziendale.

- *Stesura del piano GQM*: durante quest'attività si redige il documento che descrive il piano GQM. Nel documento, vengono associate le metriche che permettono di valutare il grado di raggiungimento di ogni obiettivo identificato.
- *Stesura del piano di misurazione*: una volta identificate le metriche da considerare, è necessario indicare con precisione le modalità secondo le quali i dati relativi devono essere raccolti.
- *Raccolta e validazione dei dati*: si raccolgono e validano i dati necessari. La validazione consiste nel valutare la correttezza dei dati raccolti, in modo da avere dati completi, coerenti e corretti. In alcuni casi reperire i dati può essere un compito complesso: a volte, infatti, l'azienda non dispone dei dati richiesti o non li gestisce in maniera efficiente, ad esempio non li raccoglie in formato digitale e li lascia solo su carta.
- *Analisi dei dati*: i dati raccolti sono analizzati e interpretati in funzione degli obiettivi cui sono legati, in modo da capire se l'obiettivo è stato raggiunto o meno.
- *Consolidamento dell'esperienza*: l'esperienza accumulata, in termini di conoscenza dell'azienda e di gestione delle metriche relative ai processi aziendali, viene raccolta e razionalizzata al fine di poterla sfruttare in progetti futuri.

Il processo GQM può richiedere che alcune attività siano ripetute in modo ciclico. Ad esempio, può rendersi necessario ripetere l'esecuzione dell'attività di definizione delle metriche associate agli obiettivi perché ci si rende conto che l'azienda non è in grado di fornire i dati richiesti: in questo caso è, quindi, indispensabile definire una nuova metrica che descriva l'obiettivo considerato.

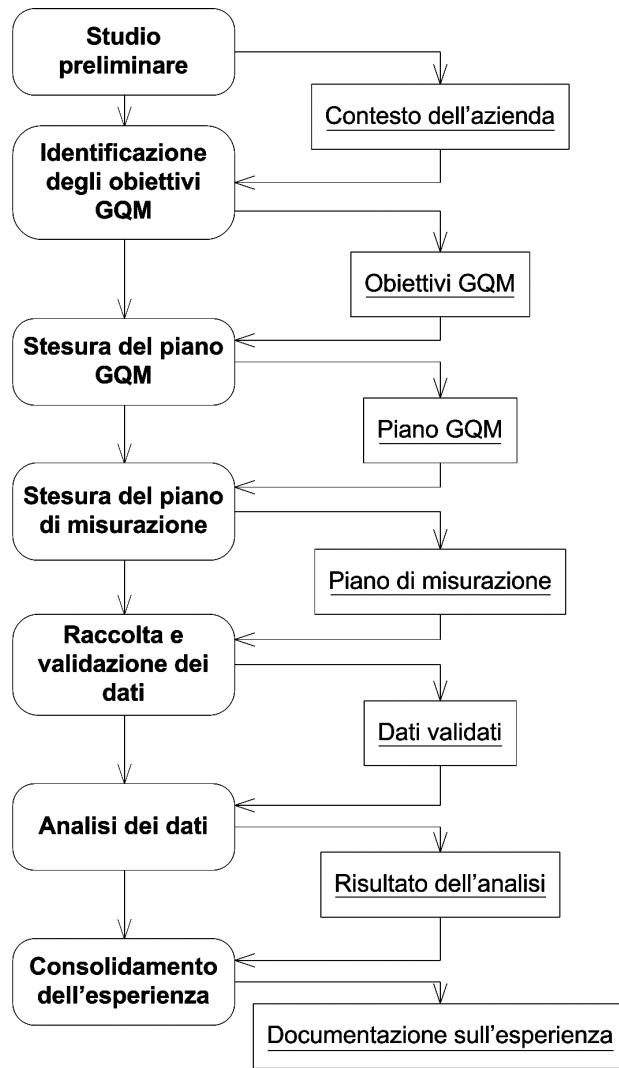


Figura 9.7 Attività del processo GQM.

Un obiettivo è definito dalle seguenti informazioni.

- *Oggetto dello studio:* è la parte effettivamente osservata e studiata relativa all'obiettivo posto. Ad esempio, l'oggetto, o ambito dello studio, può essere il processo di produzione software dell'azienda o una sua fase specifica oppure ancora il prodotto.
- *Scopo:* è definito dalle motivazioni che portano a studiare l'oggetto in funzione dell'obiettivo considerato.

9.2 Miglioramento del processo 199

- *Qualità principali*: sono le qualità fondamentali che caratterizzano l'ambito di studio e sulle quali si concentra l'analisi. Le qualità possono fare riferimento al processo, come ad esempio, la produttività, o al prodotto, come la sua robustezza.
- *Punto di vista*: indica la persona, o il gruppo di persone, alla quale è rivolto lo studio dell'oggetto. Ad esempio, lo studio del processo può essere realizzato per l'utente, per il management o per il team di sviluppo. È infatti indubbio che il punto di vista influenza sensibilmente il modo con cui viene valutata una certa proprietà. Per esempio, la qualità del software secondo l'utente non corrisponde necessariamente a quanto percepito da un membro del team di sviluppo.
- *Ambiente*: è il contesto applicativo in cui si studia l'oggetto. Può essere l'azienda nel suo complesso o una sua particolare area.

Un esempio di obiettivo è il seguente (si riconoscono abbastanza facilmente gli elementi ora introdotti).

Analizzare le attività di progettazione e verifica del processo di sviluppo (oggetto e ambiente), al fine di valutare l'efficacia del processo di rilevazione dei difetti presenti nel software (scopo e qualità principali) dal punto di vista del management e del team di sviluppo (punto di vista). Si vuole verificare se il field test ha un rapporto costo/benefici soddisfacente.

La Figura 9.8 rappresenta un frammento del diagramma che illustra alcune domande e metriche per l'obiettivo ora introdotto.

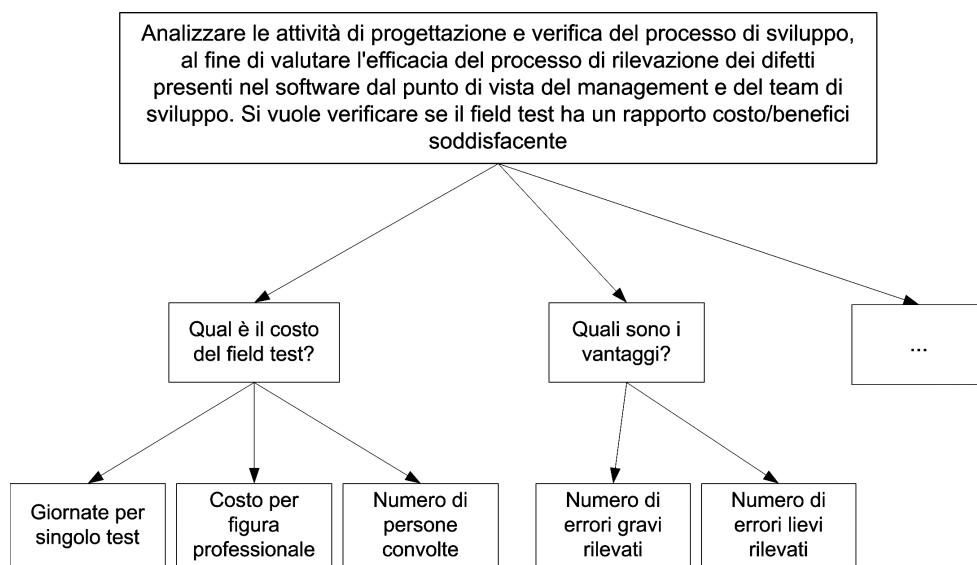


Figura 9.8 Un esempio di applicazione del metodo GQM.

9.3 Riferimenti bibliografici

Il tema della qualità del software è discusso in dettaglio in Ghezzi et al. [2003]. Oltre che delle principali tecniche di testing, il testo discute anche delle altre forme di verifica e validazione. Un altro riferimento importante è Myers et al. [2004]. Sulle software inspection il lettore può consultare direttamente l'articolo originale che introdusse questa tecnica (Fagan [1986]).

Da un punto di vista generale, il tema del processo di sviluppo del software è discusso in Fuggetta [2000]. Bandinelli et al. [1995] presenta un caso di studio nel quale un processo di sviluppo del software viene descritto in modo preciso e analizzato al fine di individuarne carenze e opportunità di miglioramento. Un'introduzione e discussione generale sul tema del miglioramento del processo è Conradi e Fuggetta. [2002]. Fuggetta, Lavazza, et al. [1998] presenta un caso di studio relativo all'utilizzo di QIP e GQM nella definizione di un programma di raccolta dati. CMMI è discusso in Ahern et al. [2003]; molti documenti sono anche disponibili sul sito del SEI. Infine, Cattaneo et al. [2001] propone alcune considerazioni sulla relazione tra aspetti ingegneristici e organizzativi nelle attività di miglioramento di processo.

Capitolo 10

Il middleware e le tecnologie per lo sviluppo software

Lo sviluppo di una soluzione informatica complessa deve basarsi su un processo rigoroso di progettazione. Nei precedenti capitoli si è studiato come basare tale processo su un metodo strutturato (Capitolo 6) e su schemi riusabili (le architetture software presentate nel Capitolo 7). Ma oltre a questi aspetti di carattere metodologico, l'ingegnere del software ha anche bisogno di specifiche tecnologie di supporto. Nel corso degli ultimi decenni, esse si sono evolute: oltre ai linguaggi di programmazione tradizionali e ai servizi offerti dai sistemi operativi, il progettista di software ha oggi a disposizione un insieme di tecnologie sofisticate che facilitano grandemente lo sviluppo di applicazioni distribuite ed eterogenee (cioè basate su piattaforme hardware e software differenti). Tali tecnologie estendono le funzioni offerte dai sistemi operativi, rendendo possibile la creazione di componenti software distribuiti e riusabili. In realtà, il catalogo di tecnologie disponibili è particolarmente ampio e ricco. Quest'insieme di tecnologie è normalmente identificato con il termine *middleware*.

Il middleware è forse una delle innovazioni più significative degli ultimi anni. Già a partire dai primi anni '90, innovazioni importanti come RPC e CORBA hanno radicalmente mutato l'approccio nella costruzione di sistemi distribuiti. RPC (*Remote Procedure Call*) è una tecnologia che permette di trasferire in ambito distribuito il principale concetto della programmazione tradizionale: la chiamata di procedura. RPC consente a un programma di invocare una procedura offerta da un programma remoto come se fosse una procedura locale tradizionale. In questo modo, si trasferisce nel contesto distribuito la potenza del paradigma di programmazione utilizzato fin dagli albori dell'informatica. CORBA si spinge ancora più avanti introducendo due concetti importanti. In primo luogo definisce un mediatore (il *broker*) che è in grado di localizzare il fornitore di un servizio (una procedura). In questo modo si disaccoppia chi offre un servizio da chi lo richiede. Inoltre, CORBA rende possibile l'interazione e l'integrazione di applicazioni costruite anche con linguaggi e tecnologie differenti: per esempio, è possibile che un programma scritto in COBOL (uno dei principali e più "antichi" linguaggi usati nel mondo dei sistemi informativi aziendali) sia invocato da una procedura scritta in C++ o C.

202 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

RPC e CORBA sono stati senza dubbio i precursori delle moderne tecnologie di middleware. Oggi tali tecnologie si sono ulteriormente evolute fornendo servizi di comunicazione, di gestione di transazione e di accesso a basi di dati distribuite, di gestione di componenti software riusabili, di sicurezza e gestione delle configurazioni. È un mondo particolarmente complesso e ancora in piena evoluzione, anche se sono emersi almeno due fieri competitori: l'ambiente Java e il mondo delle tecnologie Microsoft incentrate sul linguaggio C#.

Scopo di questo capitolo non è illustrare nel dettaglio il middleware, quanto fornire una panoramica che mostri come queste tecnologie possano essere utili nello sviluppo delle architetture descritte in precedenza: in particolare, saranno brevemente introdotte quelle principali legate al mondo Java in quanto rappresentative delle funzionalità tipiche offerte oggi dai prodotti di middleware. Per una panoramica più ricca e puntuale di quanto oggi realmente disponibile, il lettore è invitato a consultare la bibliografia consigliata.

10.1 RMI

Remote Method Invocation (RMI) è una tecnologia che permette di realizzare comunicazioni tra programmi Java distribuiti. In sintesi, è l'implementazione nel mondo Java di quanto offerto da RPC. RMI, infatti, permette a un oggetto Java in esecuzione su un computer di accedere a oggetti che si trovano su un altro computer attraverso chiamate di metodo remoto. La sintassi di tali chiamate è identica a quella delle chiamate eseguite in locale, vale a dire sulla stessa Java Virtual Machine¹. L'uso di RMI per realizzare applicazioni distribuite in Java permette di conservare la semantica e le caratteristiche di Java anche in un contesto distribuito.

L'architettura di RMI è articolata su tre elementi principali.

- I *server*, che mettono a disposizione di altre applicazioni oggetti e relativi metodi.
- I *client*, che utilizzano i metodi offerti dai server.
- Un *registry* centrale (denominato *RMI Registry*), che rende possibile il meccanismo di chiamata distribuita, attraverso funzionalità che mettono in contatto tra loro client e server.

In generale, vi possono essere più server e client, mentre il registry è normalmente unico e indipendente dai server e dai client. Ogni nodo può essere client e server contemporaneamente perché questi ruoli non sono rigidamente fissati, ma valgono per ogni singola interazione. Ciò significa che in un'interazione un programma può operare da client e invocare metodi remoti, mentre in un'altra potrà ricoprire il ruolo di server fornendo propri metodi invocabili da remoto.

¹ *Java Virtual Machine* (JVM): è la macchina virtuale che esegue i programmi in linguaggio bytecode, ovvero i prodotti della compilazione di sorgenti Java. La JVM è l'ambiente di esecuzione di programmi Java.

La struttura di un programma che utilizza RMI, sia esso client o server, è organizzata su più livelli.

- *Livello applicativo.* È costituito dal codice sorgente che contiene nel client le istruzioni che invocano il metodo remoto e nel server l'implementazione dei metodi invocati. A questo livello, il programma client invoca il metodo come se fosse offerto da un oggetto in locale.
- *Stub/skeleton.* Lo *stub* è lo strato che fornisce al client il riferimento locale dell'oggetto remoto: rende così trasparente il fatto che il metodo invocato è eseguito su un'altra macchina e da un'altra JVM. Lo *skeleton* è lo strato lato server che riceve la chiamata remota e la inoltra al livello applicativo che implementa il metodo richiesto. In pratica, il livello applicativo del client chiama lo stub immaginando che si tratti del metodo vero e proprio. In realtà, lo stub invoca lo skeleton che si trova sulla macchina remota. A sua volta lo skeleton invoca il metodo vero e proprio (il livello applicativo del server), originariamente richiesto dall'applicazione.

Nella Figura 10.1 sono mostrati i componenti presenti in un'applicazione distribuita basata su RMI. Il server applicativo pubblica nel registry i metodi che mette a disposizione

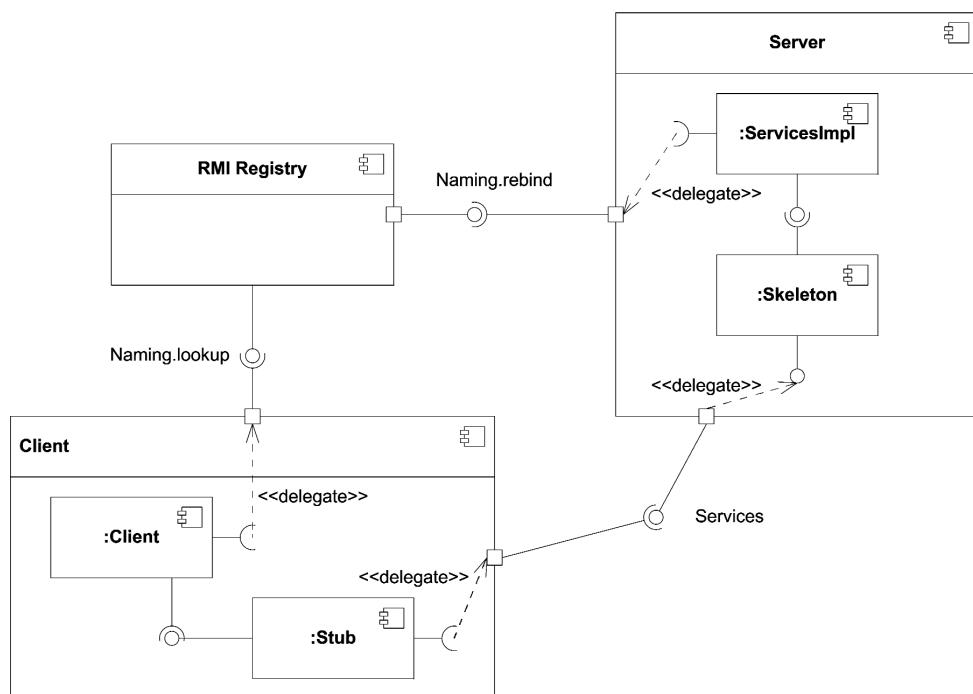


Figura 10.1 Elementi di una chiamata RMI.

204 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

dei client attraverso `Naming.rebind()`. Prima di poter invocare un metodo di un oggetto remoto, il client deve ottenere dal registry il riferimento a un oggetto che realizza i metodi esportati dal server. Per fare ciò, il client utilizza il metodo `Naming.lookup()`. Quando il client invoca un metodo remoto tramite il riferimento ottenuto dal registry, in realtà viene invocato lo stub (presente sulla macchina del client). Questo attiva una connessione con il server e in particolare con lo skeleton (la sua controparte) che invoca il metodo remoto richiesto. Stub e skeleton sono generati automaticamente dall'ambiente Java nel momento in cui il codice che include le diverse chiamate viene tradotto nel codice intermedio della JVM.

Un semplice esempio di chiamata RMI è costituito dai seguenti frammenti di codice Java.

Un server che voglia esportare un metodo chiamato `remoteService` deve innanzitutto definire un'interfaccia Java con la definizione del metodo:

```
public interface Services extends Remote {
    public void remoteService () throws RemoteException; }
```

L'interfaccia estende la classe `Remote` e ogni metodo esposto nell'interfaccia deve lanciare l'eccezione `RemoteException`. `Remote` e `RemoteException` sono fornite come elementi standard dell'ambiente di sviluppo Java.

Il server deve poi includere una classe che implementa l'interfaccia `Services`:

```
public class ServicesImpl extends UnicastRemoteObject
    implements Services {

    ...
    public void remoteService throws RemoteException()
    {
        // codice che implementa il metodo remote
    }
} // fine della dichiarazione della classe
```

Il server dichiara al registry gli oggetti che potranno essere invocati dal client attraverso il seguente frammento di codice:

```
...
ServicesImpl temp = new ServicesImpl();
String serverName = "URL dell'oggetto remoto";
Naming.rebind(serverName,temp);
...
```

La prima istruzione crea un oggetto in grado di eseguire le chiamate remote richieste. La seconda istruzione memorizza in una variabile l'URL (*Universal Resource Locator*) che permette di identificare il server che offrirà il servizio remoto. Queste due informazioni vengono comunicate al registry.

Per poter chiamare il servizio `remoteService`, un client deve compiere poche semplici operazioni:

```
...
String serverName = "URL dell'oggetto remoto";
Services s = (Services) Naming.lookup(serverName);
```

Da questo momento, il client può invocare il metodo `remoteService` tramite l'oggetto `s`:

```
... s.remoteService() ...
```

Si noti che `UnicastRemoteObject` è una classe predefinita in Java e che le uniche informazioni che devono essere condivise tra client e server sono l'interfaccia Java visibile al client e l'indirizzo (URL) dell'oggetto remoto che offre i servizi. Inoltre, il programmatore non ha coscienza del fatto che il sistema crea stub e driver in modo automatico al momento della compilazione.

In conclusione, grazie a RMI è possibile costruire programmi Java distribuiti che si richiamano attraverso invocazione di metodi remoti. Questo meccanismo di base permette di realizzare le diverse architetture discusse nei precedenti capitoli.

In realtà RMI è un meccanismo molto flessibile, ma tutto sommato di basso livello: permette "solo" di remotizzare chiamate di metodi. Per esempio, il programmatore di un'applicazione Java distribuita, usando RMI, ha completamente a suo carico la gestione di aspetti quali la sicurezza o la transazionalità delle comunicazioni fra gli oggetti remoti. Per affrontare questi problemi e rendere ancora più semplice lo sviluppo di sistemi informatici distribuiti, è stata introdotta la piattaforma J2EE che estende in modo estremamente significativo quanto offerto da RMI.

10.2 JNDI e JMS

La comunicazione tra applicazioni è fondamentale nel contesto delle architetture distribuite; di conseguenza le tecnologie e i servizi che gestiscono la comunicazione, migliorandola e rendendola più facilmente realizzabile acquistano un'importanza rilevante. Nel seguito sono proposte due tecnologie particolarmente significative: JNDI che permette una gestione semplificata dei servizi di naming e directory e JMS che offre delle funzionalità per lo scambio di messaggi tra applicazioni.

Java Naming and Directory Interface (JNDI) è una tecnologia che fornisce alle applicazioni i servizi di naming e directory, vale a dire le funzionalità necessarie per eseguire le operazioni standard di un sistema di directory:

- associazione di attributi a oggetti
- ricerca di oggetti in base al valore dei loro attributi.

206 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

Il servizio di naming è indispensabile per ottenere i riferimenti all'applicazione distribuita che si desidera usare. Un esempio di questo servizio è stato descritto nella presentazione del middleware RMI. In esso, l'applicazione client doveva ricevere prima il riferimento al server in grado di fornire il metodo richiesto. Il servizio di directory consiste nel reperire gli attributi degli oggetti memorizzati in una particolare directory, una volta trovato il riferimento per raggiungerla.

JNDI risulta indipendente da una specifica implementazione. JNDI fornisce un'interfaccia che consente alle applicazioni Java di accedere sempre nello stesso modo a differenti servizi di naming e directory. In particolare, le applicazioni possono utilizzare JNDI per accedere a servizi di naming e directory come il DNS (*Domain Name Service*). L'uso di JNDI quindi, rende trasparente all'applicazione l'effettiva implementazione del servizio di directory e naming. La Figura 10.2 descrive le due tipologie di interazioni di JNDI: la prima verso l'applicazione e la seconda verso i diversi servizi di directory possibili.

Java Messaging Service (JMS) è una tecnologia usata per la gestione della comunicazione tra applicazioni Java. Essa fornisce un insieme di interfacce e di procedure che permette a programmi Java di comunicare tramite funzionalità di messaging. In particolare, le comunicazioni realizzate da JMS sono asincrone, affidabili, lasciamente accoppiate. Una comunicazione è "lasciamente accoppiata" se il mittente e il destinatario non devono essere disponibili nello stesso istante per comunicare e non devono conoscere alcunché l'uno dell'altro.

Un'applicazione JMS è costituita dai seguenti elementi.

- *JMS provider*: elemento che implementa le interfacce di JMS. È il fornitore dei servizi di messaging.
- *JMS client*: entità che partecipa alla connessione inviando e consumando i messaggi.
- *Messaggio*: è l'informazione scambiata.
- *Oggetti amministrati*: oggetti preconfigurati che sono messi a disposizione del client per realizzare l'operazione di spedizione o ricezione di un messaggio.

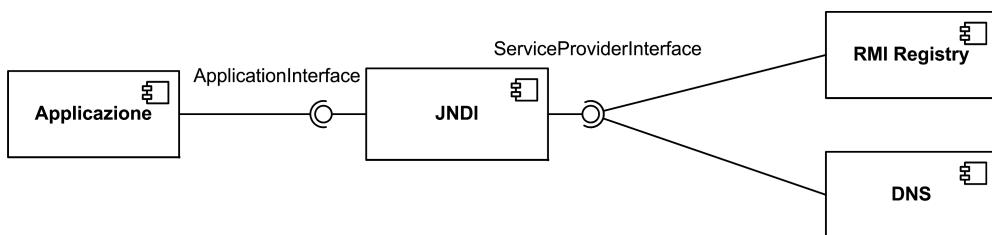


Figura 10.2 Architettura di JNDI.

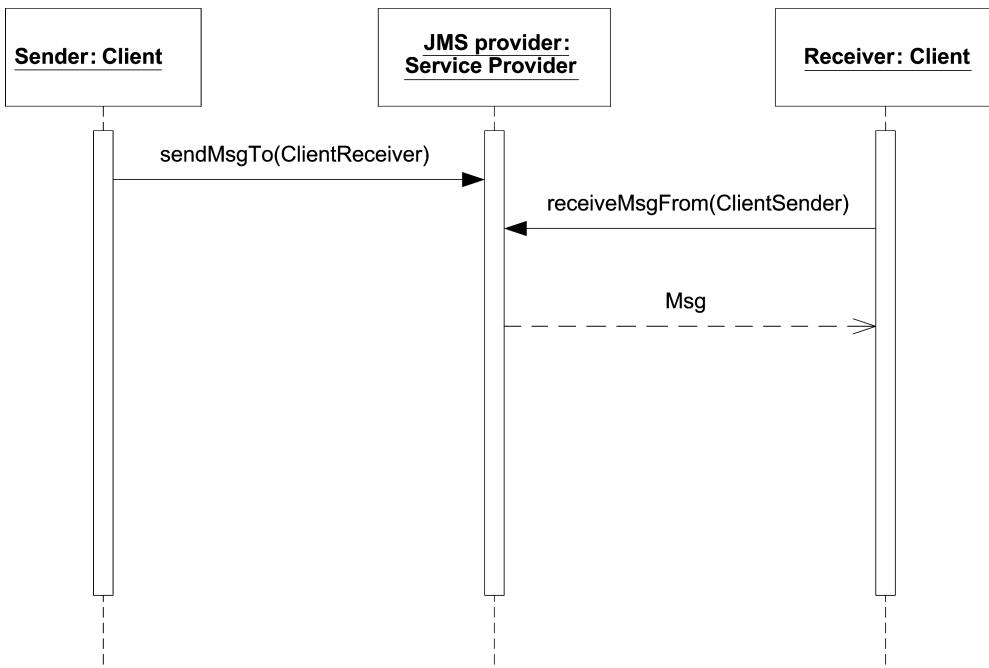


Figura 10.3 Comunicazione punto-punto.

Gli oggetti amministrati sono di due generi.

- *Connection factory*: oggetto che il client usa per creare una connessione con il provider.
- *Destination*: oggetto che il client usa per specificare il destinatario del messaggio che produce o la sorgente del messaggio che consuma. Vi sono due tipi di destination: le code e i topic. Essi sono utilizzati per le due tipologie di comunicazione offerte da JMS: punto-punto e publish-subscribe.

La *connessione punto-punto* avviene usando una coda. Come illustrato nella Figura 10.3, il mittente manda il messaggio a una coda che conserva i messaggi finché non sono ritirati dal destinatario o finché non scadono.

Le caratteristiche principali della connessione punto-punto sono le seguenti.

- Presenza di un solo destinatario.
- Asincronicità delle operazioni di scrittura e lettura: mittente e destinatario non devono essere connessi nello stesso momento per comunicare.
- Possibilità, da parte del client destinatario, di decidere se prelevare o meno il messaggio.
- Conoscenza, da parte del client mittente, del successo o meno dell'invio.

208 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

La connessione *publish-subscribe* prevede che il mittente pubblichi il messaggio associan-dolo a un *topic* (cioè una sorta di “argomento del messaggio”). Tutti i client che si sono sottoscritti a quel topic, e che quindi sono interessati al messaggio, ne ricevono una copia. La Figura 10.4 mostra con un esempio questo tipo di comunicazione. Tale operazione è curata dal JMS provider che si occupa della distribuzione dei messaggi ai client sottoscrittori.

Le caratteristiche principali di questa connessione sono.

- Presenza di più destinatari.
- Dipendenza dal tempo: un client può prelevare i messaggi da un topic solo dopo che si è sottoscritto al topic considerato. Per continuare a ricevere deve rimanere sottoscritto al topic. Il client può visualizzare solo i messaggi arrivati dopo la sua sottoscrizione.

Ogni JMS client che intende interagire tramite messaggi deve connettersi a un JMS provider che gli fornisce servizi per creare, spedire, ricevere, leggere messaggi. Il client JMS deve eseguire il lookup dell’oggetto amministrativo desiderato e, una volta trovato il suo riferimento, creare una connessione con quest’ultimo con il supporto del JMS provider.

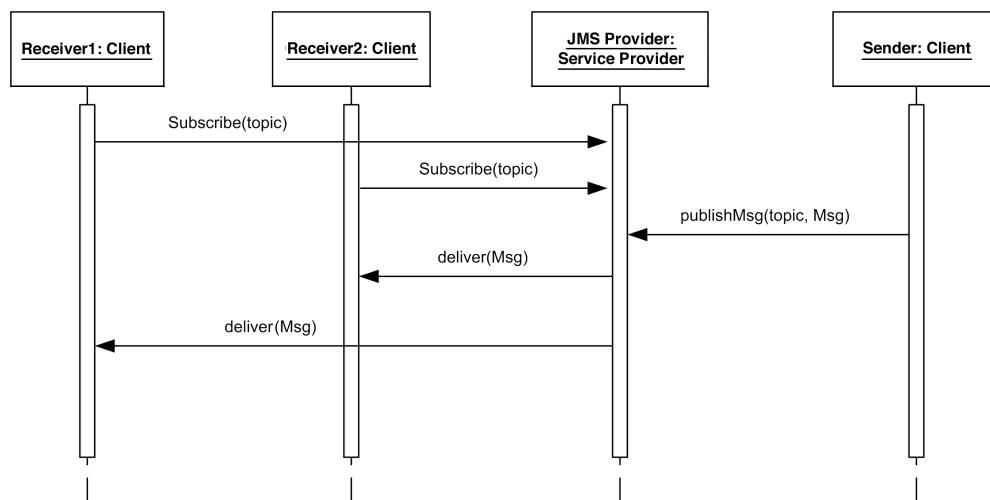


Figura 10.4 Comunicazione publish/subscribe.

10.3 J2EE

*Java 2 Enterprise Edition*² (J2EE) è una piattaforma per la realizzazione di applicazioni distribuite basate sul linguaggio Java. È fondata su un approccio a componenti e ha un modello architettonico multilivello (*multi-tier*). Tale modello ben si concilia con le architetture software distribuite (e in particolare, le architetture client-server a più livelli).

La Figura 10.5 mostra il modello architettonico di J2EE. I livelli logici principali secondo cui l'architettura è strutturata sono tre.

- *Client tier*. È il livello di *presentazione*, che permette all'utente di inviare richieste al sistema informatico e di prendere visione del risultato di tali richieste. Risiede sulla macchina client utilizzata dall'utente.
- *Middle tier*. È il livello *applicativo*. A questo livello si trovano le componenti informatiche che implementano i servizi offerti all'utente. Risiede su una o più macchine server sulle quali è stato installato l'ambiente J2EE.
Il middle tier può suddividersi a sua volta in due sottolivelli.
 - *Web tier*: è il livello che si occupa di strutturare i risultati e le informazioni da inviare al client, laddove invece il tipo di elaborazioni richieste è generalmente più semplice.
 - *Business tier*: è il livello nel quale sono collocate le componenti più complesse di elaborazione dei dati.
- *Enterprise Information System (EIS) tier*. È il livello che si occupa della gestione dei dati e di incapsulare i sistemi informativi preesistenti (denominati sistemi legacy).

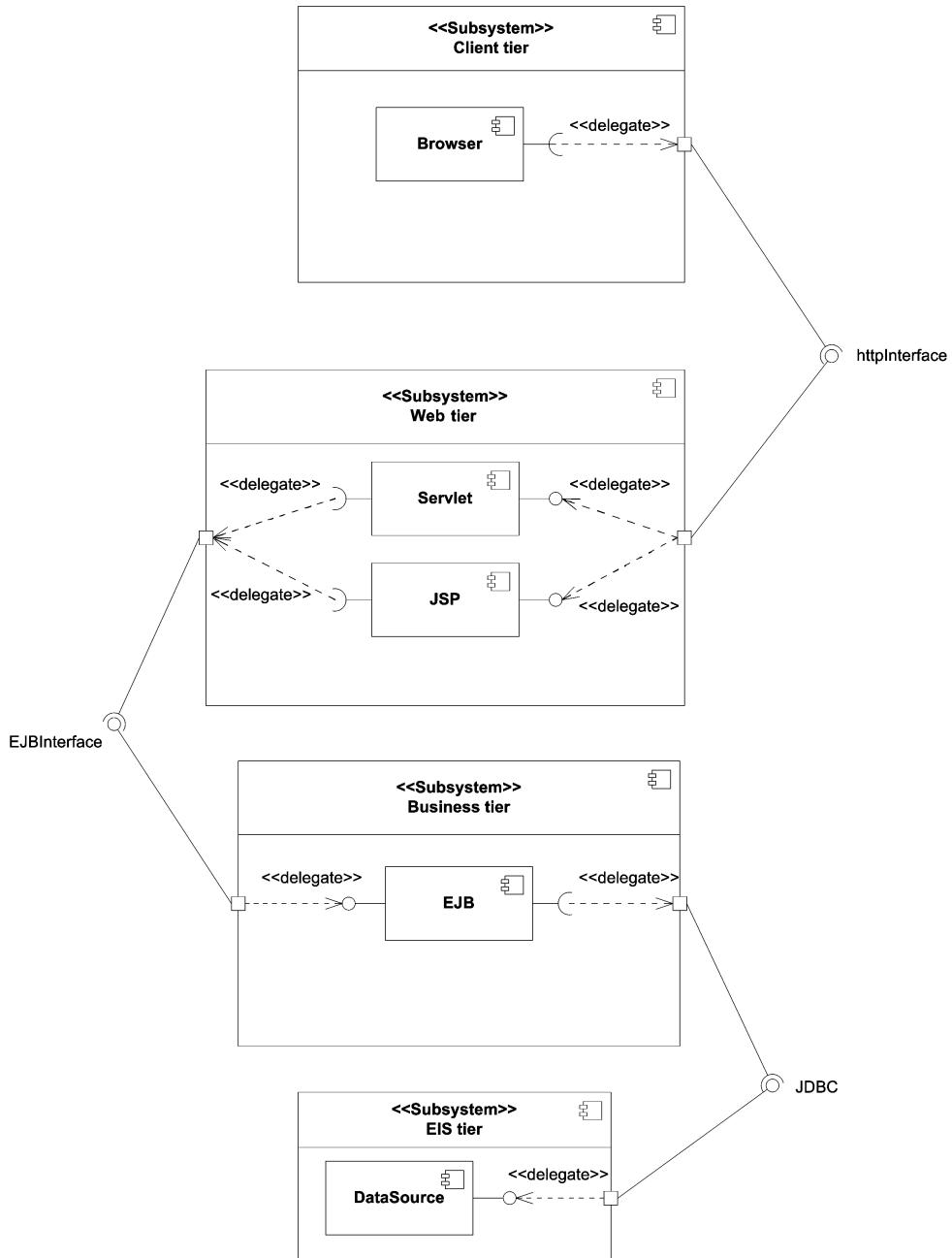
In realtà, la figura illustra solo una tra le diverse configurazioni possibili di un sistema J2EE, anche se certamente particolarmente significativa. In generale, lo schema della Figura 10.5 può essere adattato in funzione delle specifiche necessità e requisiti del progettista.

La piattaforma J2EE è basata sul concetto di componente. In J2EE, un *componente* è una *unità di riuso*, cioè un insieme di moduli software sorgenti. I componenti sono suddivisi in tre tipologie.

- *Componenti client tier*: sono quelli eseguiti lato client. Esempi di questi componenti sono le applicazioni client in generale e in particolare le applet.
- *Componenti web tier*: sono implementati tramite servlet e *Java Server Page (JSP)*. In particolare le servlet sono indicate per ricevere ed espletare richieste di servizio, mentre le JSP sono più indicate per la presentazione di risultati.

² Sito di J2EE: <http://java.sun.com/j2ee/>

210 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

**Figura 10.5** Modello architettonico di J2EE.

- *Componenti business tier*: sono i componenti, quali gli *Enterprise Java Beans* (EJB), che implementano le logiche applicative e le informazioni proprie dello specifico dominio considerato quali, ad esempio, il settore finanziario o il commercio elettronico.

In funzione della complessità e dimensioni delle applicazioni, il progettista dovrà selezionare le tipologie di componenti da utilizzare per realizzare le diverse parti del sistema. Ovviamente, ogni specifica tecnologia ha proprie caratteristiche che la rendono più o meno adatta a svolgere in modo efficiente una specifica funzione. La possibilità di combinare in modo flessibile le diverse tipologie di componenti viste in precedenza permette di sfruttare le caratteristiche e i punti di forza delle tre tecnologie di J2EE citate: servlet, JSP e EJB.

I componenti J2EE hanno delle caratteristiche particolari rispetto a programmi Java generici. In particolare, essi devono implementare specifiche interfacce o ereditare da classi predefinite (così come accade su scala più limitata nel caso di RMI). In questo modo, i componenti costruiti dal progettista riusano porzioni esistenti dell'ambiente J2EE e implementano interfacce standard richiamabili da altri componenti o dal supporto run-time dell'ambiente stesso.

I componenti J2EE sono gestiti attraverso i container. Un *container* è un ambiente di esecuzione che gestisce una particolare tipologia di componente J2EE, fornendo una serie di servizi quali la gestione del ciclo di vita del componente (per esempio la sua inizializzazione), servizi di sicurezza e funzionalità per il suo deployment. Grazie ai servizi forniti dal container, il progettista può concentrarsi sulla realizzazione della logica applicativa peculiare al contesto considerato. Il programmatore può personalizzare il comportamento del proprio componente non solo agendo direttamente sul codice sorgente del componente sviluppato, ma anche a livello di *deployment descriptor*³. Il livello di sicurezza e di transazionalità che il componente deve garantire può essere dichiarato in tale descrizione, senza essere gestito a livello di codice.

I container forniti da J2EE sono i seguenti.

- *Application client container*: gestisce l'esecuzione di programmi sul client tier.
- *Applet container*: gestisce l'esecuzione di applet sul client tier.
- *Web container*: gestisce l'esecuzione di servlet e JSP sul middle tier.
- *EJB container*: gestisce l'esecuzione sul middle tier dei componenti EJB.
- *J2EE server*: è il supporto run-time di un'installazione J2EE.

³ Il deployment descriptor è un documento XML che permette di descrivere le proprietà delle tecnologie J2EE, come ad esempio servlet, JSP ed EJB.

212 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

I componenti J2EE possono interagire tra di loro sia all'interno di uno stesso livello architettonico che tra due diversi. Per gestire tali interazioni il modello J2EE mette a disposizione differenti servizi quali ad esempio l'accesso a database (come in JDBC, che verrà discusso nel seguito del capitolo) e servizi di naming e directory (JNDI) o messaging (JMS). Tali servizi sono implementati utilizzando un ampio spettro di tecnologie e protocolli di comunicazione, come http e RMI.

I vantaggi dell'uso di J2EE sono molteplici.

- *Semplificazione nello sviluppo dell'architettura e nella sua implementazione.* Il modello su cui si basa J2EE facilita la realizzazione delle applicazioni distribuite permettendo allo sviluppatore di concentrarsi sulle caratteristiche peculiari dell'applicazione.
- *Scalabilità.* J2EE fornisce un meccanismo per ottenere un alto livello di scalabilità (per esempio, nella gestione di transazioni o di connessioni con i client) senza che lo sviluppatore debba implementare esplicitamente tali funzionalità.
- *Integrazione di sistemi informativi esistenti.* J2EE include una serie di interfacce standard per accedere a sistemi informativi esistenti di tipo *legacy* (per esempio applicazioni COBOL o C preesistenti all'adozione della piattaforma J2EE).
- *Sicurezza.* J2EE offre un servizio di *single sign-on* per l'accesso a un'applicazione. I requisiti di sicurezza possono essere specificati per ogni metodo.
- *Riconfigurabilità e dinamicità (naming).* È possibile riconfigurare a run-time la collocazione dei diversi componenti, sfruttando anche la possibilità di accedere a un servizio di naming e di directory che permette di localizzare in modo semplice ed efficiente i diversi componenti.

Nei successivi paragrafi verranno discusse le caratteristiche più importanti delle diverse tecnologie presenti in J2EE.

10.3.1 Componenti web tier

Le servlet e le JSP sono componenti la cui logica applicativa viene eseguita lato server a fronte di un'esplicita richiesta effettuata da un browser sul client tier.

Una *servlet* è un programma Java che estende le funzionalità del web server, fornendo specifici servizi come la generazione dinamica di HTML o XML. È invocata attraverso un opportuno comando http. Tipicamente, una servlet fa da intermediario tra la macchina che esegue una richiesta e i sistemi legacy e le basi di dati che contengono le informazioni necessarie alla costruzione della risposta alla richiesta dell'utente.

La Figura 10.6 mostra il funzionamento di una servlet nel contesto di un'applicazione distribuita. Il client invia una richiesta al web server che gestisce, attraverso un apposito container, la servlet oggetto della richiesta. La richiesta può essere, ad esempio, derivante da una form HTML compilata lato client per ottenere determinate informazioni. Nel codice HTML della form, è indicato l'URL della servlet da invocare: quando l'utente conferma l'invio della form, il client avvia una connessione http e invia la ri-

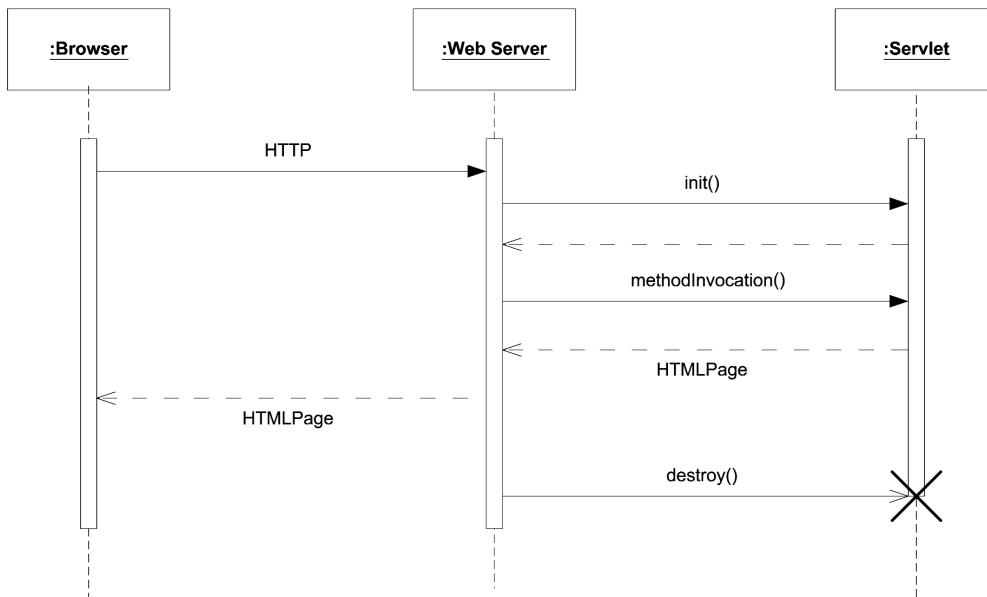


Figura 10.6 Funzionamento di una servlet.

chiesta all'URL specificata. I metodi http più usati per inviare richieste alle servlet sono il metodo `GET` e il metodo `POST`. La servlet soddisfa una richiesta attraverso i suoi metodi di servizio. Quelli usati sono principalmente `doGet()` e `doPost()`: essi rispondono rispettivamente alla richieste http `GET` e `POST`.

Una caratteristica della servlet è il suo caricamento dinamico: di norma, infatti, una servlet è caricata alla sua prima chiamata e non quando è attivato il web server. Il caricamento di una servlet nel web server prevede che essa sia inizializzata, attraverso l'apposito metodo `init()` implementato in ogni servlet.

Una servlet può estrarre dalla richiesta le informazioni necessarie per elaborare la risposta. Le informazioni utili sono sia quelle strettamente inerenti la richiesta, come ad esempio i parametri di quest'ultima, sia informazioni inerenti la comunicazione con il client, come le caratteristiche del browser, il nome dell'host o la presenza e le eventuali caratteristiche dei *cookies* (dati depositati dal server sulla macchina client attraverso il browser per memorizzare informazioni applicative). Una volta estratte le informazioni necessarie, la servlet elabora la risposta: questo può richiedere una connessione a database o un'invocazione di altre applicazioni. Elaborata la risposta, la servlet la formatta per la presentazione all'utente sul client (nella maggior parte dei casi i risultati della servlet sono inseriti all'interno di una pagina HTML). Infine, interagisce con il container per definire il risultato che verrà inviato al client richiedente.

214 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

La creazione e la manutenzione dello strato di presentazione (HTML) introdotto dalle servlet sono attività complesse e soggette a errori. Per questo, nel caso in cui la presentazione delle informazioni abbia un peso rilevante, è preferibile utilizzare le *Java Server Pages* (JSP). Una JSP è un file di testo che prevede una chiara separazione tra presentazione e logica dell'applicazione. La struttura di una JSP richiede due tipologie di contenuto ben definite.

- *Template data*: sono gli elementi di presentazione della pagina. Tra questi, ad esempio, vi è il codice HTML che descrive la pagina web, i relativi stylesheet ed eventuali elementi Javascript.
- *Elementi JSP*: costituiscono il codice Java che elabora i dati. Gli elementi JSP forniscono informazioni relative al codice che il container interpreta al momento della chiamata di un client; essi sono racchiusi dal tag `<% %>`. I diversi tipi di elementi JSP sono i seguenti.
 - *Direttive*: rappresentano informazioni e messaggi sulla JSP che questa manda al JSP container. Non producono output per il client ma hanno effetto sul comportamento complessivo della JSP. Per esempio, la direttiva `<%@ include file="relativeURLspec" %>` indica che il JSP container dovrà includere nella JSP considerata anche il contenuto della sorgente indicata nell'URL descritto.
 - *Scripting*: contengono il codice Java vero e proprio, tra cui la dichiarazione delle variabili ed eventualmente le espressioni da calcolare e il cui risultato va inviato al client.
 - *Azioni*: sono indicazioni al container che hanno effetto a run-time. Ad esempio, l'azione di forward mostrata di seguito consente di inoltrare la request a un servlet o a un'altra risorsa.
`<jsp:forward page="url" />`
 - *Oggetti impliciti*: oggetti accessibili da qualunque JSP, che non devono essere dichiarati o istanziati, ma sono forniti dal JSP Container all'interno della classe servlet d'implementazione. Un esempio di oggetto隐式的 sono gli oggetti `request` e `response`.
 - *Expression language*: sono espressioni i cui valori vengono valutati e inseriti nell'output di risposta al client.
 - *Custom tag*: lo sviluppatore ha la possibilità di definire e usare propri tag XML nella pagina realizzata. I tag definiti sono contenuti in una libreria che lo sviluppatore deve indicare usando l'apposita direttiva `taglib`.

La JSP è molto simile a una servlet: infatti, anche la JSP risiede sul web server e implementa il paradigma richiesta/risposta. L'invio della richiesta da parte del client è pressoché identico: una JSP, infatti viene chiamata attraverso una URL nella quale si specifica il web server e la JSP in grado di rispondere alla richiesta. Il web server che riceve la ri-

chiesta traduce la JSP in una classe Java e poi la compila come una servlet: nel tradurre e nel compilare la JSP il container considera le informazioni racchiuse negli elementi JSP della pagina stessa. L'elaborazione della risposta e l'invio della stessa procedono come nel caso della servlet. Il vantaggio di usare una JSP deriva dalla sua caratteristica principale, che è quella di separare la presentazione dal contenuto: usando una JSP, infatti, diventa molto più gestibile scrivere il contenuto statico di presentazione in HTML, o eventualmente modificarlo, senza toccare il codice e la parte dinamica.

10.3.2 Componenti business tier

Gli *Enterprise Java Bean* (EJB) sono anch'essi componenti riusabili scritti in Java. Nella costruzione di un EJB devono essere rispettate una serie di regole. Queste impongono, in particolare, che una classe contenga l'implementazione di specifiche interfacce. In questo modo, la classe così realizzata può interagire con il resto del sistema secondo meccanismi standardizzati. Allo stesso tempo, così è possibile accedere e riusare una serie di servizi predefiniti.

I componenti EJB risiedono sul middle tier e sono gestiti dall'EJB container che fornisce una serie di servizi a livello di sistema. Il container gestisce il ciclo di vita di un EJB e i servizi inerenti la sicurezza e le transazioni. La sua presenza è un vantaggio per lo sviluppatore, che può riutilizzare i servizi di sistema dell'EJB e può concentrarsi sulla soluzione dei problemi veri e propri legati all'applicazione in corso di sviluppo.

Un EJB è composto da una serie di elementi.

- *EJB Class*: è il bean vero e proprio. È la classe che contiene l'implementazione dei metodi che questo mette a disposizione degli altri componenti della piattaforma.
- *EJBHomeInterface* ed *EJBObjectInterface*: sono interfacce che definiscono rispettivamente i metodi legati al ciclo di vita del componente (creazione, distruzione, lookup) e i metodi legati propriamente alle sue funzionalità.
- *Deployer Descriptor*: è un file XML che fornisce al container i dati fondamentali per utilizzare il bean.

Ogni EJB è caratterizzato da un proprio stato. Lo *stato di un EJB* è l'insieme dei valori assegnati alle variabili dichiarate nel bean stesso.

Gli EJB si suddividono in tre tipi.

- *Session EJB*.
- *Entity EJB*.
- *Message-Driven Bean*.

Il *Session EJB* è un bean che fa da tramite tra il client e l'applicazione che si trova sul server: quando un client chiama un'applicazione distribuita che risiede sul server, sfrutta i metodi del session EJB.

216 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

Il session EJB si divide a sua volta in due diversi tipologie, in funzione del modo secondo il quale è gestito il suo stato.

- *Stateful*: lo stato del session bean è mantenuto per tutta la durata della connessione con il client. Questo significa che se il client esegue più invocazioni di metodi all'interno della stessa sessione, il session EJB stateful mantiene comunque lo stato anche tra un'invocazione e la successiva. Lo stato viene perso nel momento in cui il client termina la sua esecuzione o elimina il bean. Di conseguenza, un session EJB è associato sempre allo stesso client durante tutta la sessione.
- *Stateless*: date due interazioni consecutive con lo stesso EJB, la seconda interazione non è in alcun modo condizionata dalla precedente, in quanto le due sono completamente slegate e indipendenti l'una dall'altra. Quando l'interazione richiesta dal client termina, lo stato viene perso anche se la sessione del client è ancora attiva: in questo modo il session EJB stateless può servire un altro client, ed essere così associato a un client diverso per ogni invocazione di metodo.

Il session EJB stateless supporta meglio la scalabilità dell'applicazione perché può servire molti client. Di conseguenza, a parità di client che interagiscono con l'applicazione, sono richiesti un numero minore di session EJB stateless piuttosto che stateful.

Un *Entity EJB* rappresenta un oggetto associabile a una struttura di memoria persistente. La maggior parte delle applicazioni J2EE prevedono l'elaborazione di dati contenuti in un database relazionale: l'uso dell'entity EJB rende la gestione e l'elaborazione delle informazioni più semplice e sicura. Un entity EJB, quindi, rappresenta una particolare tabella di un database e una sua istanza rappresenta una singola tupla di quella tabella. Le tabelle di un database che descrivono clienti, ordini o prodotti sono descrivibili tramite gli entity EJB: in questo caso si avranno gli entity EJB *ClienteEJB*, *OrdineEJB* e *ProdottoEJB*. Un entity EJB è usato per rendere più semplice ed efficiente l'interazione dell'applicazione con il database, soprattutto nel caso in cui più client accedano alle stesse informazioni o le modifichino. L'accesso molteplice agli stessi dati richiede di dover affrontare una problematica estremamente delicata: il mantenimento dell'integrità del dato a cui si accede. L'uso dell'entity EJB permette di gestire al meglio questa problematica: infatti, il container e il deployer descriptor offrono servizi per gestire anche l'integrità dei dati, sollevando lo sviluppatore da questo compito oneroso.

Un entity EJB presenta delle caratteristiche peculiari che lo differenziano da un session EJB.

- *Persistenza*: i dati relativi a un entity EJB sono persistenti perché sono la descrizione di tuple di un database che esistono indipendentemente dall'entity EJB stesso. La persistenza del dato è di due tipi, in funzione di chi la gestisce.
 - *Persistenza gestita dall'entity EJB*: l'accesso al database è gestito dall'entity EJB, che deve contenere i metodi per la chiamata al database stesso.
 - *Persistenza gestita dal container*: l'accesso al database è gestito dal container.

La gestione della persistenza da parte del container rende l'entity EJB maggiormente portabile: infatti, nel momento in cui l'applicazione interagisce con un altro database, non occorre reinstallare il bean, che non avendo metodi di accesso al database risulta essere trasparente al tipo di database usato.

Oltre ai dati anche le elaborazioni sono persistenti: la modifica di un'informazione fatta accedendo a uno specifico bean non si perde quando questo viene terminato ma persiste nel database contenente l'informazione descritta proprio dall'entity EJB considerato.

- *Relazione*: un entity EJB può essere associato ad altri entity EJB, per esempio è possibile associare un `ClienteEJB` con un `OrdineEJB`.
- *Accesso condiviso*: un entity EJB non è associato a un unico client ma, al contrario più client vi possono accedere anche contemporaneamente. Diventa necessario per l'applicazione, quindi, lavorare all'interno di transazioni.

Le diverse caratteristiche dei session EJB e degli entity EJB rendono preferibile usare questi due tipi di bean in contesti e situazioni diverse. I session EJB sono destinati all'implementazione della logica applicativa e hanno un ruolo funzionale. Gli entity EJB, invece, sono destinati alla gestione delle informazioni persistenti e contengono la sola parte di logica per accedere e gestire la sincronizzazione con la sorgente dati.

Un terzo tipo di bean è il *Message-Driven Bean* che gestisce l'invio e la ricezione dei messaggi asincroni. Esso agisce come un message listener: i messaggi gestiti possono essere inviati da qualsiasi componente J2EE, ma anche da sistemi che non usano tale tecnologia. Una caratteristica di questo bean è che il client non vi accede attraverso le interfacce perché in realtà il message-driven bean è costituito dalla sola bean class. Il Message-Driven Bean sfrutta la tecnologia JMS per gestire la fruizione dei messaggi.

10.3.3 Java DataBase Connectivity

Molto spesso, un'applicazione Java richiede l'esecuzione di operazioni su una base di dati. La tecnologia che permette di eseguire queste operazioni è chiamata *Java DataBase Connectivity* (JDBC).

JDBC è un package contenente una serie di classi e interfacce che permette a un'applicazione Java di accedere a database eterogenei. JDBC, infatti, rende trasparente all'applicazione il tipo di database utilizzato: l'applicazione chiamerà gli stessi metodi e istruzioni indipendentemente dal tipo di database col quale deve interagire.

Le funzionalità di JDBC coprono una serie di operazioni.

- Connessione al database. Questo tipo di operazioni rende disponibile una specifica base di dati all'applicazione.
- Esecuzione di operazioni vere e proprie di accesso alla base di dati.
- Elaborazione dei risultati delle operazioni di accesso al database. Questo permette di elaborare i dati ricavati e inviarli come risposta al client.

218 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

Il linguaggio usato per interrogare e modificare una base di dati è lo *Structured Query Language* (SQL). SQL è un linguaggio standard di interrogazione di database, adottato dalla gran parte dei DBMS (*Data Base Management System*) oggi sul mercato, come ad esempio Access e PostgreSQL.

La Figura 10.7 illustra i componenti principali di JDBC.

- *Application*: è l'applicazione che si connette al database per ottenere le informazioni desiderate. L'applicazione comunica direttamente solo con il Driver Manager.
- *Driver Manager*: è una libreria che gestisce la comunicazione tra l'applicazione e il driver necessario per la connessione e la comunicazione con il database considerato.
- *Driver*: è una libreria specifica per ogni tipologia di database, permette di eseguire gli operazioni nel linguaggio proprio del database. Il driver maschera l'eterogeneità dei sistemi DBMS sottostanti.
- *Data Source*: è la base di dati a cui si deve accedere dall'applicazione.

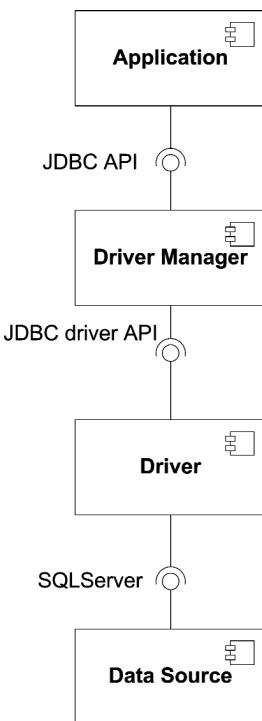


Figura 10.7 Architettura di JDBC.

Le interfacce di JDBC sono fondamentali per realizzare la connessione e la comunicazione tra applicazione e database.

- *Driver Manager e Connection*: sono le due interfacce che realizzano la connessione al database. Il driver manager dipende dal tipo di database che si vuole usare ed è l'oggetto che apre e gestisce la connessione, creando così l'oggetto connection.
- *Statement*: è l'oggetto usato per realizzare i comandi SQL inviandoli al database. È legato all'oggetto connection. L'oggetto statement definisce, infatti, metodi come `executeQuery(String)`, che esegue l'interrogazione specificata come parametro sul database a cui si è connessa l'applicazione, oppure come `executeUpdate(String)`, che esegue la modifica specificata come parametro di ingresso.
- *ResultSet*: rappresenta l'oggetto che contiene il risultato dell'interrogazione. Tale risultato è organizzato come una tabella. Attraverso l'oggetto ResultSet è possibile scorrere le tuple che costituiscono il risultato ed estrarne i dati: questi possono essere elaborati anche dall'applicazione e inviati al client. I metodi principali offerti dal ResultSet sono il metodo `next()`, che permette di scorrere la tabella dei risultati, e la serie di metodi `get`, che estraggono un particolare dato della tupla considerata, creando un oggetto di un tipo specifico. Ad esempio, il metodo `getString(numeroColonna)` estraie dalla tupla considerata il dato, sottoforma di stringa, della colonna indicata dal parametro di ingresso.

10.4 SOAP e web service

In questo capitolo sono state introdotte le principali tecnologie della piattaforma J2EE che permettono la realizzazione di applicazioni distribuite. Questo tipo di applicazioni è realizzato all'interno di una singola organizzazione ed è legato al linguaggio Java. La tecnologia dei web service permette di realizzare, attraverso opportuni standard, applicazioni distribuite service-based fondate su tecnologie eterogenee. In questo modo, il linguaggio di programmazione non è più un vincolo. Inoltre, è possibile sviluppare e integrare applicazioni tra organizzazioni diverse (*cross-enterprise*).

Un *web service* è un servizio messo a disposizione da un server e accessibile attraverso messaggi XML inviati utilizzando il protocollo http. I vantaggi di un web service sono principalmente due.

- Uso di protocolli aperti e standard. In particolare, il protocollo http utilizza una porta che è normalmente lasciata aperta dai firewall: in questo modo è possibile realizzare architetture distribuite anche tra macchine situate dietro un firewall.
- Uso del linguaggio XML. XML è un linguaggio fortemente portabile perché è facilmente modificabile con la maggior parte dei linguaggi di programmazione e sulla maggior parte delle piattaforme esistenti: tutti i principali linguaggi di programmazione offrono opportune interfacce per leggere, modificare, creare, spedire e ricevere documenti XML.

220 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

Queste due caratteristiche affrancano l'applicazione dal limite esercitato dal linguaggio di programmazione. Infatti, l'uso del web service rende trasparente la piattaforma hardware e software usata per implementare il servizio e soprattutto il linguaggio usato per realizzarlo.

Un'applicazione che fa uso di web service è basata sul concetto di *agente*.

- *Service Provider*: sono gli agenti che forniscono i servizi mettendoli a disposizione di altri agenti.
- *Service Requestor*: sono gli agenti che richiedono i servizi messi a disposizione dai provider.
- *Service Registry, Service Brokers, Meta Information Provider*: sono gli agenti che forniscono le descrizioni delle informazioni necessarie per permettere l'uso dei servizi pubblicati da parte dei vari richiedenti.

Le interazioni che legano queste tre tipologie di agenti ricordano molto le interazioni realizzate con RMI per usufruire di metodi remoti. La Figura 10.8 mostra uno scenario di interazione tra agenti in un'applicazione basata su web service. Il service provider che vuole rendere disponibili i propri servizi pubblica sul service registry una loro descrizio-

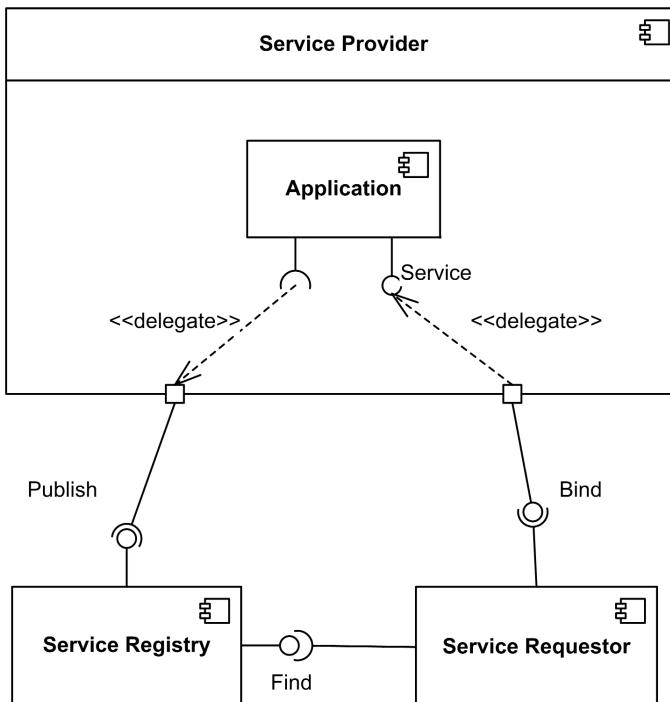


Figura 10.8 Il modello dei web service.

ne (*publish*). Un service requestor che ha bisogno di un servizio lo cerca nel service registry (*find*): una volta trovata l'interfaccia che descrive il servizio concreto desiderato, riceve i riferimenti al service provider ed esegue l'operazione di *bind* and *execution* per poter usufruire di una o più funzionalità offerte dal servizio.

I web service si fondano su una serie di tecnologie.

- *Network* (livello di trasporto): è il protocollo utilizzato per la comunicazione tra i diversi elementi dell'architettura (generalmente, si tratta del protocollo http).
- *XML-messaging* (livello di protocollo applicativo): i messaggi trasmessi via http sono strutturati come documenti XML che rispettano la specifica del protocollo SOAP.
- *Service description*: i servizi sono descritti tramite il linguaggio WSDL.
- *Service Publication & Discovery*: la pubblicazione, ricerca e localizzazione dei servizi viene effettuata utilizzando la tecnologia UDDI.

Simple Object Access Protocol (SOAP) è un protocollo per la comunicazione tra applicazioni basato sull'invio di messaggi XML fra requestor e service provider. SOAP è basato sul linguaggio XML e questo lo rende semplice ed estendibile; è indipendente sia dalla piattaforma che dal linguaggio usati per realizzare e installare requestor e service provider.

SOAP è uno standard e quindi la scelta più naturale è quella di abbinarlo con un protocollo di trasporto come http, che è supportato da tutti i browser e i server di Internet. Sfruttando la comunicazione http e il linguaggio XML, SOAP fornisce un modo per far comunicare applicazioni eseguite su sistemi operativi diversi e realizzate con diverse tecnologie e linguaggi.

Web Service Description Language (WSDL) è usato per specificare le regole attraverso cui le applicazioni possono ricevere e spedire correttamente i messaggi SOAP. Inoltre, WSDL è il linguaggio usato per descrivere i servizi pubblicati. Un servizio, come già accennato, è descritto come un insieme di operazioni: WSDL permette di specificare il nome dell'operazione, i parametri di ingresso e uscita, l'URL per raggiungere il servizio che implementa la descrizione.

Universal Description Discovery and Integration (UDDI) è una tecnologia per la pubblicazione e scoperta dei servizi sul web. È usata per implementare il Service Registry che contiene le descrizioni di tutti i servizi pubblicati. È basato sul linguaggio XML ed è indipendente dalla piattaforma hardware/software utilizzata.

10.5 Uno sguardo allargato

Il middleware costituisce solo una parte degli strumenti e delle tecnologie a disposizione dell'ingegnere del software. In realtà, nel corso degli anni, la disciplina dell'ingegneria del software si è evoluta significativamente, portando allo sviluppo di un ricco insieme di concetti, principi, metodi e tecnologie per progettare, realizzare e manutenere un prodotto software. Anche se lo scopo principale di questo capitolo è quello di introdurre le principali tecnologie di middleware, è utile quanto meno allargare lo sguardo e delineare i princi-

222 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

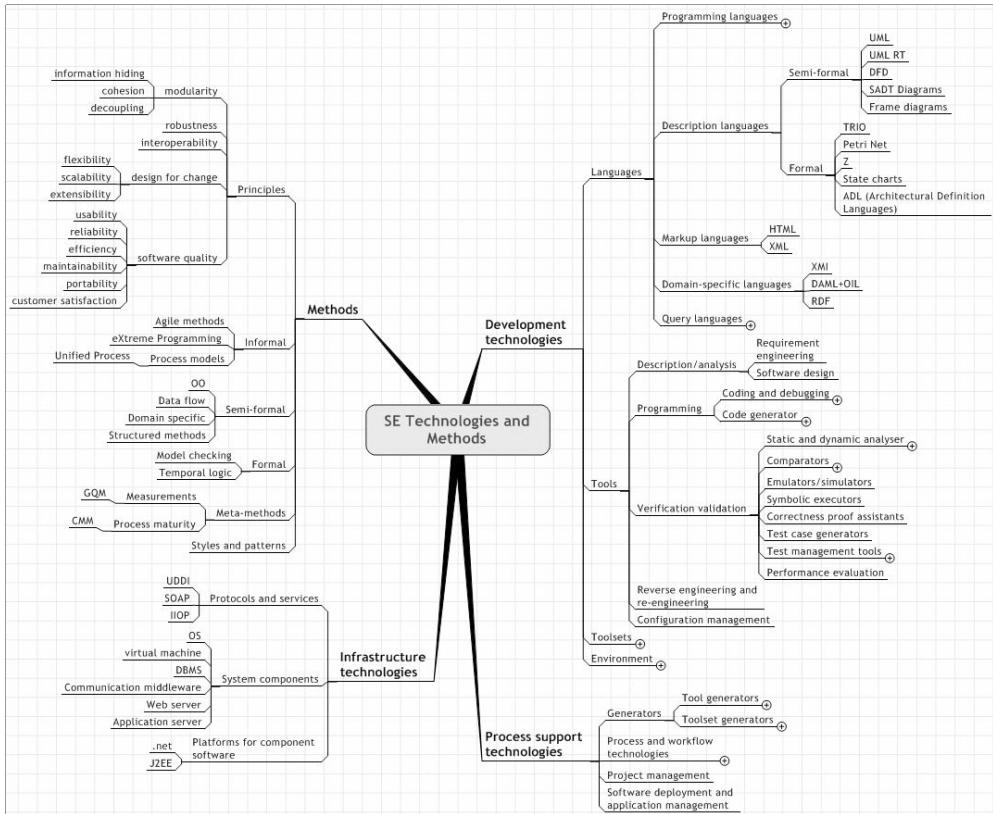


Figura 10.9 Metodi e tecnologie per il processo di sviluppo del software.

pali tratti caratteristici dell'insieme delle tecnologie oggi disponibili. Per ulteriori approfondimenti sul tema, il lettore è invitato a consultare i riferimenti consigliati.

Quanto oggi a disposizione di chi deve sviluppare un'applicazione informatica può essere organizzato in quattro categorie principali (Figura 10.9): metodi, tecnologie di sviluppo, tecnologie a supporto del processo di sviluppo e infine tecnologie infrastrutturali.

10.5.1 Metodi

Con il termine "metodi" si identificano i concetti, le linee guida e le tecniche che guidano da un punto di vista metodologico lo sviluppo efficace ed efficiente di un prodotto software. I metodi possono essere a loro volta suddivisi in quattro sottocategorie.

- *Principi*. Sono i concetti base su cui si fonda la realizzazione di un prodotto software: per esempio, i concetti di modularità, interoperabilità e robustezza e i principi di qualità del software discussi nel Capitolo 3.

- *Tecniche di sviluppo.* Sono le diverse metodiche sviluppate per guidare il processo di sviluppo di un sistema informatico. Si suddividono in tre tipi di approcci a seconda del grado di formalismo tenuto:
 - *informali*, come l'eXtreme Programming e i metodi agili (discussi nel Capitolo 8);
 - *semiformali*, come l'approccio object-oriented;
 - *formali*, come le tecniche di model checking o le logiche temporali.

Le tecniche di sviluppo forniscono gli strumenti concettuali e metodologici al progettista di software. Molte di queste tecniche sono poi supportate attraverso specifici strumenti.

- *Meta-metodi.* Sono metodi che non hanno come scopo diretto quello di aiutare il progettista del software a sviluppare il codice (o i documenti a esso correlati), quanto quello di migliorare l'organizzazione e il funzionamento del processo di sviluppo nel suo complesso. Un esempio di meta-metodo è GQM (*Goal Question Metric*), un approccio strutturato alla creazione di un sistema di raccolta dati e metriche sul processo e sul prodotto software.
- *Stili e pattern.* Come discusso in dettaglio nel Capitolo 1, l'ingegnere del software consolida la conoscenza attraverso schemi (stili e pattern) che descrivono la struttura di diverse tipologie di problemi o di soluzioni.

10.5.2 Tecnologie di sviluppo

Le tecnologie di sviluppo costituiscono la categoria forse più conosciuta di strumenti a servizio dell'ingegnere del software. Esse includono tutti gli strumenti utilizzati per l'effettiva implementazione di un sistema informatico complesso. Le tecnologie di sviluppo possono essere classificate in ulteriori sottocategorie.

- *Linguaggi.* Gli esempi più noti sono i *linguaggi di programmazione* come ad esempio Java e C++, vale a dire i linguaggi utilizzati per la creazione vera e propria dei programmi sorgenti. Vi sono poi i *linguaggi di descrizione*, come UML, che hanno acquisito sempre maggior importanza negli ultimi anni e che sono usati per tradurre dal linguaggio corrente a un linguaggio più formale la descrizione dello spazio del problema concordata con il cliente o l'architettura complessiva della soluzione. Con l'avvento di Internet e del web è emerso il ruolo dei *linguaggi di mark-up* che sono usati per descrivere tipologie di informazioni e concetti. I linguaggi di mark-up più noti sono HTML e XML. Infine, è importante ricordare i *linguaggi domain-specific*, usati per uno specifico dominio o applicazione. Un esempio è il linguaggio SQL per l'interrogazione delle basi di dati.
- *Tool.* Sono gli strumenti che supportano il programmatore in una particolare fase della produzione del software. Spesso essi si basano su uno o più linguaggi tra quelli descritti nel precedente paragrafo. Gli strumenti si possono suddividere in funzione della fase del processo di sviluppo nella quale sono usati.
 - *Descrizione e analisi:* gli strumenti di questo tipo permettono di descrivere il problema e la soluzione tramite uno o più linguaggi di descrizione. Ad esem-

224 Capitolo 10 Il middleware e le tecnologie per lo sviluppo software

pio, per il linguaggio UML sono disponibili tool quali Rational Rose, Poseidon, o l'editor presente in Microsoft Visio. Spesso, questo tipo di strumenti offre funzioni di vario tipo per analizzare le descrizioni prodotte e fornire indicazioni sulla loro qualità e correttezza.

- *Programmazione*: gli strumenti di programmazione sono usati per la scrittura e la correzione del codice sorgente. Si tratta tipicamente dei compilatori, dei generatori di codice e dei debugger.
- *Verifica e validazione*: questo tipo di strumenti fornisce un supporto (semi)automatico a molte attività legate alla verifica e alla validazione del codice. Un tipico esempio sono gli strumenti di supporto alla generazione dei casi di test.
- *Reverse engineering*: gli strumenti di questo tipo permettono di estrarre informazioni dal codice esistente e di ristrutturarlo in modo da facilitare le attività di manutenzione ed evoluzione del prodotto.
- *Gestione delle configurazioni*: gli strumenti per la gestione delle configurazioni forniscono un supporto alle attività legate alla gestione del prodotto e dei processi di cambiamento e generazione delle sue componenti.
- *Toolset e ambienti di sviluppo*. Sempre più spesso, i tool di sviluppo non sono venduti o resi disponibili separatamente. In pratica, è sempre più comune che diversi tool di sviluppo siano integrati all'interno di un unico prodotto. Quest'integrazione permette di condividere dati in modo semplice e diretto, avviare in modo automatico operazioni che coinvolgono diversi tool e fornire un'interfaccia omogenea e di più facile utilizzo. Per esempio, prodotti come JBuilder integrano un editor per Java, un compilatore, un editor grafico per la creazione di interfacce utente, un generatore di codice e un debugger.

La distinzione tra toolset e ambienti di sviluppo è legata al grado di copertura offerto da uno specifico prodotto al processo di sviluppo del software. Il termine toolset viene normalmente utilizzato per identificare un insieme integrato di tool orientato a una specifica fase del processo (per esempio, un toolset che include strumenti di programmazione e debugging per la fase di codifica). Un ambiente di sviluppo include strumenti che coprono l'intero spettro di attività, dalla codifica vera e propria alla produzione di descrizioni in UML, alle attività di pianificazione e controllo del progetto e di gestione delle configurazioni.

10.5.3 Tecnologie a supporto del processo

Gli strumenti e ambienti di supporto al processo di sviluppo del software sono sistemi software particolarmente complessi e sofisticati. Il processo stesso di sviluppo è un insieme complesso di attività condotte in generale da team di progettisti. Per questi motivi, nel corso degli ultimi anni sono state sviluppate una serie di tecnologie che sono di supporto alla creazione degli ambienti di sviluppo e all'esecuzione efficiente delle attività presenti nel processo di sviluppo del software. In particolare, è possibile distinguere le seguenti tipologie di prodotti.

- *Generatori.* Sono strumenti per la creazione di tool e ambienti. Un esempio è il componente per lo sviluppo di plug-in fornito da Eclipse (un ambiente per lo sviluppo di applicazioni Java). Un plug-in è un modulo che estende le funzioni di base di Eclipse. Per esempio, esistono plug-in per la costruzione di diagrammi UML e la generazione (semi)automatica di scheletri di programmi Java.
- *Tecnologie di processo e workflow.* Sono usate per descrivere il processo di sviluppo del software, per automatizzarne alcune parti o per coordinare le operazioni di diversi progettisti. Si tratta di strumenti che forniscono una sorta di *orchestrazione complessiva* delle attività che vanno a costituire il processo di sviluppo nel suo complesso.
- *Project management.* Gli strumenti di project management supportano il manager nella pianificazione e controllo delle tempistiche, dell'effort speso e degli obiettivi da raggiungere nel corso del progetto.
- *Tecnologie di sviluppo del software e di gestione dell'applicazione.* Sono usate per installare il software nell'ambiente dove dovrà essere eseguito. Inoltre, queste tecnologie sono usate per controllare il funzionamento del software, in modo da poter rilevare eventuali errori che si manifestano a run-time.

10.5.4 Tecnologie infrastrutturali

Le tecnologie infrastrutturali sono quelle identificate con il termine middleware e che sono state trattate all'inizio di questo capitolo. Rispetto a quanto già discusso in precedenza, è opportuno identificare una classificazione delle diverse tecnologie disponibili, così da avere un quadro complessivo dello stato dell'arte nel settore.

- *Protocolli e servizi.* Includono le tecnologie che offrono le funzionalità per scambiare informazioni e controllare le interazioni tra componenti software distribuiti. Fanno parte di questa classe le tecnologie UDDI e SOAP.

Considerazione di carattere generale

Quanto discusso in questo capitolo non ha assolutamente l'ambizione di fornire una descrizione completa ed esaustiva delle tecnologie a supporto del processo di sviluppo del software. Ciò richiederebbe ben altro spazio e livello di approfondimento. Inoltre, questo tipo di studi è soggetto a rapida obsolescenza, in quanto lo sviluppo delle tecnologie è molto più rapido della velocità con cui i libri di testo vengono prodotti, aggiornati e immessi sul mercato.

Lo scopo di questo capitolo può essere riassunto nei seguenti punti.

1. L'analisi offerta vuole indicare le principali categorie di tecnologie e strumenti a supporto del progettista. Lo sviluppo delle tecnologie è stato in questi anni impressionante e a volte risulta particolarmente difficile e ostico avere un quadro d'insieme organico e strutturato.
2. Lo studio delle tecnologie, specie quelle di middleware, è importante per capire come concepire e strutturare una soluzione informatica. Il Capitolo 11, infatti, discuterà come passare dallo studio del problema a quello della soluzione e come il middleware influisce nella scelta e nella costruzione di una specifica architettura software.

- *Componenti di sistema.* Fanno parte di questa classe i sistemi operativi, le macchine virtuali come la JVM, i database management system (DBMS), le tecnologie di middleware come RMI, i web server come Tomcat e gli http server come Apache.
- *Piattaforme per componenti software.* Fa parte di questa classe la piattaforma J2EE. Essa in realtà include sia un modello architettonico, che componenti di sistema basati, tra l'altro, su una serie di protocolli e servizi di base.

10.6 Riferimenti bibliografici

Le pubblicazioni che discutono le tecnologie presentate in questo capitolo sono numerosissime. Poichè la materia è in rapida evoluzione, il lettore è invitato a considerare innanzi tutto il materiale e i documenti disponibili sul web, presso i siti delle società e degli organismi che sviluppano o definiscono le tecnologie di interesse.

Tutte le tecnologie legate direttamente a Java sono ampiamente descritte e approfondite sul sito di Sun Microsystems. In particolare, tale sito contiene documenti e tutorial su Java, RMI, JMS e l'architettura J2EE nel suo complesso.

Alcuni testi recenti su Java sono Arnold et al. [2006] e Carrano e Pritchard [2006]. Il primo è un testo sulla programmazione in Java, mentre il secondo, dopo una parte iniziale sul linguaggio, discute nel dettaglio come sviluppare e valutare algoritmi e strutture dati.

Un testo di carattere generale sui protocolli e le tecnologie di Internet è Sebesta [2005], che presenta tecnologie quali XML, applet e servlet, Perl, Javascript, PHP e JDBC. JNDI è presentato in dettaglio in Lee [2000]. Un sito che contiene informazioni dettagliate e aggiornate sulle tecnologie del web (in particolare, WSDL, SOAP) è quello del World Wide Web Consortium (W3C). Informazioni su UDDI sono disponibili sul relativo sito.

Le tecnologie e i metodi di supporto al processo di sviluppo del software sono state classificate in Fuggetta [1993] e, più recentemente, in Fuggetta e Sfardini [2005].

Capitolo 11

Dal problema alla soluzione

I precedenti capitoli hanno offerto una descrizione dei principali concetti, tecniche e metodi che caratterizzano il processo di sviluppo di soluzioni software complesse. Per poter apprezzare compiutamente quanto discusso, è importante ripercorrere idealmente il ciclo di vita di un'applicazione informatica per analizzare e commentare alcuni passaggi critici che segnano il processo di sviluppo del software. In particolare, questo capitolo contiene una serie di considerazioni e riflessioni che evidenziano i problemi e le scelte che un ingegnere del software si trova ad affrontare nel corso di un progetto nel passare dallo studio di un problema alla realizzazione e messa in esercizio (“in produzione”) della corrispondente soluzione informatica.

11.1 Alcune linee guida

Nell'ambito di un progetto complesso, l'ingegnere del software deve valutare una serie di aspetti metodologici e di impostazione generale che determinano lo stile e l'approccio complessivo secondo i quali sono condotte le attività di sviluppo.

11.1.1 Completezza, formalità, rigore

Nel corso degli ultimi decenni si sono spesso contrapposte due scuole di pensiero.

- I sostenitori degli approcci e metodi formali sostengono che le attività di sviluppo del software devono essere centrate sull'uso di tecniche formali che garantiscano rigore nel processo di sviluppo e coerenza, correttezza e completezza di tutte le informazioni che vengono prodotte o raccolte dall'ingegnere del software.
- Quelli che privilegiano la facilità di espressione, l'immediatezza della comunicazione e l'efficacia dei risultati finali ritengono che il successo nei progetti di sviluppo del software è garantito da un'organica combinazione di fattori umani e organizzativi. Anzi, poiché lo sviluppo di software è un'attività creativa nella quale la qualità

228 Capitolo 11 Dal problema alla soluzione

delle persone è decisiva, l'uso di tecniche formali che vincolino e limitino le capacità del singolo sono addirittura viste come una possibile fonte di complessità e criticità.

La discussione è per molti versi ancora aperta in quanto l'ingegneria del software è lungi dall'essere una disciplina consolidata. D'altro canto, è altrettanto vero che entrambe le posizioni hanno meriti che, peraltro, non sono mutuamente esclusivi o alternativi. L'ingegnere del software ha il compito di sviluppare e gestire prodotti estremamente complessi. Ipotizzare che ciò possa avvenire senza l'utilizzo di approcci rigorosi e precisi è illusorio. Nella descrizione del problema o nella progettazione della soluzione è necessario essere, almeno in alcuni passaggi chiave, precisi e completi. Le attività di verifica e validazione devono garantire che il software sia coerente con i requisiti di affidabilità e sicurezza richiesti dall'utente. Tuttavia, durante le attività di sviluppo può accadere che le informazioni siano incomplete o incoerenti: ciò è tipico di un complesso processo di ingegneria (si veda il Capitolo 1). Si tratta quindi di coniugare opportunamente rigore e creatività, formalità e immediatezza di espressione.

L'esigenza di avere un equilibrio tra queste spinte in apparenza contrastanti è particolarmente evidente nel caso della descrizione del problema. Si consideri a questo proposito il seguente esempio, proposto da Jackson (Jackson [1995]) e ispirato a un caso reale.

Si vuole costruire un sistema per automatizzare le procedure di atterraggio di un aeroplano. Il dominio applicativo è caratterizzato dalle seguenti informazioni.

- L'aeroplano è dotato di motori per i quali al momento dell'atterraggio è possibile comandare l'inversione della spinta (e facilitare quindi la riduzione della velocità). L'inversione della spinta può essere comandata tramite attuatori controllati digitalmente.
- Le ruote dell'aeroplano sono dotate di sensori di rotazione che inviano un segnale quando la ruota gira. Le ruote girano quando l'aeroplano si trova sulla pista.

I requisiti del sistema di atterraggio possono essere sintetizzati nel modo seguente.

- L'inversione della spinta deve essere comandata quando l'aeroplano si trova sulla pista.

Dalla descrizione del dominio applicativo e dei requisiti si può dedurre la specifica del sistema:

il calcolatore che controlla il funzionamento dell'aereo deve invertire la spinta quando riceve il segnale di rotazione delle ruote.

Il segnale del sensore è il fenomeno condiviso che, alla luce di quanto indicato dalle informazioni di dominio, corrisponde al fatto che l'aereo si trova sulla pista. Quest'ultimo fenomeno non è direttamente "visibile" al computer: quando l'aereo è sulla pista, le ruote girano e il sensore emette un segnale che è visibile al computer di bordo (è "condiviso") e può essere quindi utilizzato dall'algoritmo di controllo come indicatore del fatto che sia necessario comandare l'inversione della spinta.

Questo livello di descrizione può essere certamente più che accettabile in una fase preliminare, quando l'obiettivo primario è definire contorni e tratti complessivi del progetto. Tuttavia, non appena si approfondisce lo studio del problema, ci si rende conto che tale descrizione deve essere resa molto più accurata, al fine di capire nel dettaglio proprietà e vincoli di quanto si sta analizzando. Ciò può essere ottenuto tramite l'utilizzo di metodi formali. Per esempio, utilizzando semplici espressioni logiche, è possibile descrivere il problema in modo molto più preciso e puntuale.

Il dominio applicativo è descritto come segue:

- RUOTE_GIRANO: variabile logica che indica se le ruote stanno girando (vero) o meno (falso);
- INVERSIONE_SPINTA: indica se è in corso l'inversione della spinta (vero) o no (falso);
- AEREO_A_TERRA: indica se l'aereo è sulla pista (vero) o no (falso);
- SENSORE_ACCESO: indica se il sensore è acceso (vero) o no (falso).

Queste variabili sono utilizzate per descrivere le altre caratteristiche del dominio applicativo:

- RUOTE_GIRANO iff AEREO_A_TERRA: “è vero che le ruote girano se e soltanto se è vero che l'aereo è a terra”, dove iff significa “if and only if” (se e solo se);
- SENSORE_ACCESO iff RUOTE_GIRANO: “è vero che il sensore delle ruote è acceso se e soltanto se è vero che le ruote girano”.

Il requisito può essere di conseguenza sintetizzato come segue (si noti l'uso del verbo “deve” per indicare “ciò che si vuole ottenere”, cioè il requisito):

INVERSIONE_SPINTA iff AEREO_A_TERRA

“deve essere vero che vi è inversione della spinta
se e soltanto se è vero che l'aereo è a terra”

La specifica del sistema può essere ottenuta utilizzando la seguente espressione ricavata congiungendo le espressioni logiche che descrivono il dominio applicativo e il requisito:

INVERSIONE_SPINTA iff SENSORE_ACCESO

“deve essere vero che vi è inversione della spinta
se e soltanto se è vero che il sensore è acceso”

L'uso di una (semplice) notazione formale permette di risolvere alcuni problemi della descrizione testuale utilizzata in origine. Nella descrizione testuale era stata utilizzata la parola “quando” per indicare l'accadere o meno di un fatto. Per esempio, si è detto che “le ruote girano quando l'aeroplano si trova sulla pista.”. Nella descrizione formale, la parola “quando” è sostituita dall'operatore “iff” che fornisce l'informazione in modo preciso

230 Capitolo 11 Dal problema alla soluzione

e non ambiguo: condizione necessaria e sufficiente perché le ruote girino è che l'aereo si trovi sulla pista. La descrizione testuale non è così precisa: si poteva intendere sia l'esistenza di una condizione necessaria e sufficiente sia una condizione solo necessaria (o solo sufficiente).

Tuttavia, anche la descrizione formale è viziata da una grave carenza che ha fatto sì che il codice sviluppato a partire da quella specifica causasse un grave incidente aereo. Il problema è che non è vero che le ruote girano “se e solo se” l'aereo è a terra. In effetti, è accaduto che un aeroplano in atterraggio sia stato soggetto a venti laterali e ad aquaplaning. Per questi motivi, le ruote non hanno girato immediatamente e l'inversione della spinta è stata ritardata e ciò ha causato l'uscita di pista dell'aereo.

Chi ha descritto il dominio applicativo ha fornito (o raccolto) un'informazione sbagliata: *l'aereo potrebbe trovarsi sulla pista senza che le ruote girino*. Si tratta di un errore dovuto a scarsa conoscenza del dominio applicativo ed è quindi riconducibile ad aspetti organizzativi, di capitale umano e di processo. A chi è stato dato l'incarico di studiare il dominio applicativo? Aveva le competenze e le conoscenze adeguate? Chi doveva controllare la descrizione di dominio? Questo controllo è stato effettuato? *Non esiste metodo formale che possa prevenire questo tipo di errore*, in quanto la formalità agisce come elemento di razionalizzazione sulle informazioni che sono messe a disposizione del progettista. Tramite descrizioni formali è possibile ragionare sulla completezza e coerenza delle informazioni fornite (coerenza interna, si veda il Capitolo 1), ma non provarne la veridicità. *A qualunque livello di approfondimento ci si spinga nel formalizzare le informazioni ricevute, prima o poi ci si dovrà basare su una qualche affermazione che deve necessariamente essere assunta come vera e coerente con la realtà osservata*: qualcuno deve assumersi la responsabilità di dire se è vero o meno che “le ruote girano se e solo se l'aereo è sulla pista” (coerenza esterna).

Nella sua semplicità, questo esempio mostra come *aspetti formali e capacità personali/organizzative sono complementari*.

11.1.2 La conoscenza di dominio

Un aspetto particolarmente critico di ogni progetto di sviluppo di software è la conoscenza del dominio applicativo. Nel caso precedente, l'errore nella specifica deriva da una conoscenza e/o descrizione errata di un fenomeno legato al funzionamento in atterraggio di un carrello.

Certamente, l'ingegnere del software non può conoscere in modo dettagliato tutti i domini applicativi nei quali potrebbe trovarsi a operare. Si pone quindi il problema di *come coniugare competenze specifiche di dominio con conoscenze informatiche in senso stretto*. A questo proposito vi sono almeno due approcci possibili.

1. Il team di sviluppo è composto da ingegneri del software e da altre persone che sono *esperte del dominio applicativo*: nel caso in questione, uno o più ingegneri aerospaziali.
2. Il team include ingegneri del software che hanno sviluppato, dal punto di vista del percorso formativo e/o per esperienze acquisite in progetti passati, le competenze di dominio necessarie.

Questi due diversi approcci si riflettono anche sui percorsi formativi che sono oggi riscontrabili nel mondo dell'ingegneria del software. Taluni sostengono che l'ingegnere del software non necessita di competenze applicative, in quanto dovrà di volta in volta integrare con esperti di dominio; altri affermano che i corsi di ingegneria informatica devono essere costruiti in funzione dei settori applicativi più rilevanti, in modo da preparare professionisti capaci di comprendere e analizzare le caratteristiche del dominio applicativo di interesse.

Indipendentemente dal punto di vista prescelto, è indubbio che *un team di sviluppo deve includere professionisti che conoscono in modo approfondito il dominio applicativo*: il capo progetto dovrà prestare la massima attenzione affinché non vengano a mancare queste competenze necessarie allo svolgimento del progetto. Certamente, se l'ingegnere del software possiede già conoscenze di dominio, la sua capacità di incidere in modo positivo sul processo di acquisizione e validazione delle informazioni aumenta in modo significativo.

11.2 Le scelte di progetto

All'inizio di un progetto (ma anche durante il suo svolgimento), l'ingegnere del software si trova a dover fare delle scelte relative all'impostazione e gestione complessiva delle attività di sviluppo: quale ciclo di vita, quali strumenti, quali tecnologie?

Non esistono soluzioni ottime o universalmente valide. È necessario di volta in volta *prendere o cambiare decisioni*, sulla base delle proprie conoscenze e del proprio bagaglio di esperienze. Esistono peraltro alcuni criteri e linee guida che è utile ricordare.

11.2.1 Il ciclo di vita e la pianificazione di progetto

Innanzitutto l'ingegnere del software deve decidere come organizzare e gestire il progetto. Quale ciclo di vita preferire? Come organizzare il piano di progetto? Come passare dalla scelta del generico ciclo di vita a un diagramma di Gantt dettagliato delle "cose da fare"?

La scelta dipende dalle caratteristiche del progetto e del committente. Laddove possibile, potrebbe risultare conveniente optare per il ciclo di vita a cascata, che garantisce il modo di lavoro più semplice e lineare. Come discusso nel Capitolo 8, questo modo di procedere è spesso impraticabile. Generalmente non ci sono le condizioni per congelare la descrizione del problema, né per attendere la fine del progetto prima di consegnare un qualunque prototipo o componente al cliente. Occorre scegliere un ciclo di vita che preveda iterazioni e ricicli, magari con fasi prototipali volte a raccogliere informazioni attraverso sperimentazioni sul campo. In sintesi, il ciclo di vita stesso *deve essere progettato e aggiornato dall'ingegnere del software, combinando e adattando gli schemi di riferimento discussi nel Capitolo 8*. Spesso, in aziende di dimensioni medio-grandi esistono cicli di vita standard, con i relativi processi di sviluppo. Anche in questo caso però è indubbio che l'intelligenza e le competenze dell'ingegnere del software sono essenziali per adattare ed

estendere quanto eventualmente previsto dallo standard aziendale alle esigenze e specificità rilevate “sul campo”.

Alla luce del ciclo di vita e del relativo processo di sviluppo, l’ingegnere del software deve creare il piano di progetto. Spesso ciclo di vita, processo e piano di progetto sono considerati in modo indipendente e separato. Si comincia a pianificare come se quanto definito a livello di ciclo di vita e di processo non fosse rilevante. In realtà il piano di progetto è *il raffinamento e l’attuazione di quanto previsto dal ciclo di vita e dal relativo processo*. Se, per esempio, nel ciclo di vita si indica che ci deve essere un fase iniziale di definizione architettonica con il cliente, il piano di progetto dovrà riportare una serie di task che nel loro complesso permettono di raggiungere l’obiettivo di definire tale architettura. In generale, *il piano di progetto deve essere creato in modo logicamente collegato con quanto previsto dal ciclo di vita e dal processo software da quest’ultimo derivato*. Deve essere sempre possibile collegare un task del piano di progetto a una macro fase del ciclo di vita, spiegandone ruolo e funzione.

Troppò spesso, in realtà, manca questa visione d’insieme: non ci si rende conto che il piano di progetto è l’attuazione operativa di una strategia che è espressa dalla natura e struttura del ciclo di vita, o si inizia a lavorare senza sapere bene secondo quale piano si giungerà alla realizzazione e messa in esercizio del prodotto.

Per questi motivi, *è essenziale che l’ingegnere del software colleghi in modo organico i tre concetti chiave di ciclo di vita, processo e progetto, al fine di definire una strategia di sviluppo del prodotto che trovi coerente realizzazione nei diversi passaggi che caratterizzano il piano di progetto*.

11.2.2 La stima dei costi

Uno degli aspetti più critici e delicati di un progetto di sviluppo software è la stima dei costi. Tutti vogliono sapere “quanto costerà” il software che deve essere sviluppato: il cliente, il capo progetto, i responsabili delle aziende interessate. Normalmente, la stima del costo è richiesta fin dall’inizio, quando le informazioni sono scarse e i contorni stessi del prodotto da sviluppare non sono ancora ben definiti.

In realtà, *essendo appunto una “stima”, è soggetta a errori*. Non esistono soluzioni magiche che permettano di eliminare tali errori. La certezza assoluta nella valutazione dei costi è ottenibile solo quando il progetto è finito (o vicino alla conclusione), “a consumo”. Tipicamente, se si traccia l’andamento dell’errore di stima in funzione del tempo, si ottiene una curva del tipo indicato nella Figura 11.1.

Da cosa dipende l’errore nella stima? Dalla mancanza di informazioni: maggiore è l’incertezza sull’evoluzione del progetto, maggiore è il rischio di commettere errori anche grossolani. *La stima dei costi di un progetto di sviluppo del software è un’attività che non può essere compiuta “una tantum”*. Il Capitolo 8 ha discusso nel dettaglio le tecniche di pianificazione di progetto e ha sottolineato il fatto che ci deve essere un controllo periodico dello stato di avanzamento del progetto. In quest’ambito, deve essere condotta anche una sistematica attività di stima e revisione dei costi. All’inizio, tale stima sarà basata principalmente sull’esperienza e conoscenza del capo progetto. Con l’avanzare del progetto, avendo acquisito informazioni sul prodotto da sviluppare, si potranno utilizzare

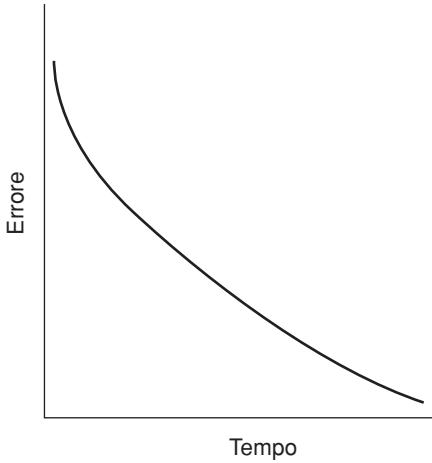


Figura 11.1 Andamento nel tempo dell'errore di stima.

strumenti come COCOMO o WBS per effettuare stime sempre più dettagliate. In generale, con il passare del tempo, le informazioni acquisite permetteranno di aggiustare o anche radicalmente ridurre l'errore nella stima dei costi.

Ancora una volta, è importante sottolineare che *la stima dei costi non è un'attività che possa essere condotta indipendentemente dalle attività di progetto*. Molti problemi e fallimenti che si osservano nei progetti reali sono causati da questa sorta di schizofrenia secondo cui i costi sono una variabile indipendente, valutati in funzione di rapporti di natura commerciale con il cliente o alle "convinzioni" del manager. In realtà, essi derivano direttamente dalla natura e struttura del progetto: maggiore sarà la capacità di pianificazione e gestione di progetto, più bassa sarà la curva della Figura 11.1 e migliore sarà la capacità del capo progetto di prevedere e controllare i reali costi di sviluppo.

11.2.3 Quali e quanti linguaggi di descrizione?

Molte organizzazioni si imbarcano in discussioni e dibattiti sul tipo di linguaggio di descrizione da utilizzare nelle attività di sviluppo. Talvolta questa diviene una sorta di scelta di campo sulla quale si confrontano persone e organizzazioni. Oppure, una particolare opzione rende l'organizzazione ingessata e acritica di fronte a problemi che tale scelta può comportare.

I linguaggi di descrizione devono essere scelti in funzione delle esigenze del progetto. Non è detto che esista un linguaggio unico per tutte le soluzioni. Il fatto stesso che UML includa un insieme piuttosto ricco di notazioni sta a indicare che esigenze diverse danno origine a scelte differenti. In alcuni casi, UML potrebbe essere la soluzione più conveniente per una certa tipologia di progetto; in altri, dove magari è necessario provare e studiare proprietà di sicurezza o safety, potrebbe risultare più opportuno utilizzare altri linguaggi di tipo formale che rendano possibile questo tipo di controlli.

234 Capitolo 11 Dal problema alla soluzione

Un fattore non marginale nella scelta del linguaggio è anche la tipologia di utenza con la quale i progettisti devono interagire. La descrizione del dominio, in particolare, deve essere discussa e analizzata con l'utente per verificarne coerenza, correttezza e completezza. Pertanto, è auspicabile che il linguaggio di descrizione sia conosciuto dall'utente, così che l'interazione possa essere efficace e proficua.

In sintesi, *la scelta del linguaggio di descrizione è essa stessa parte del progetto: va fatta in funzione delle sue caratteristiche e criticità*.

11.2.4 Quali strumenti e tecnologie?

Analogamente a quanto accade per i linguaggi di descrizione, anche nella scelta delle tecnologie possono emergere criticità e conflitti. Molto spesso si assiste a riunioni di pianificazione di un progetto nelle quali si dibatte sui vari tipi di tecnologie: ambienti di sviluppo, middleware, database, strumenti di gestione delle configurazioni ecc. Ovviamente, la scelta di una tecnologia, o di uno strumento di sviluppo, è particolarmente difficile e, poiché incide in modo significativo sullo svolgimento del progetto, è essenziale che sia compiuta sulla base di considerazioni tecnico-economiche molto dettagliate quali, ad esempio, le seguenti.

- *Qualità intrinseca della tecnologia:* quali sono le prestazioni e funzioni garantite?
- *Allineamento con le caratteristiche del progetto:* è inutile scegliere una tecnologia eccellente, ma inadatta al progetto che si deve svolgere.
- *Compatibilità e vincoli tecnologici:* questa tecnologia deve integrarsi con altri componenti preesistenti? Se sì, è in grado di farlo?
- *Transizione e facilità di apprendimento:* è semplice adottare questa tecnologia e quanto è già conosciuta dal team di progetto?
- *Total cost of ownership:* quanto costa acquisire e soprattutto mantenere nel tempo una tecnologia? Spesso ci si concentra sul suo costo iniziale e non si presta attenzione ai costi di manutenzione e/o aggiornamento.
- *Lock-in:* quanto questa tecnologia vincolerà scelte future? L'azienda sarà in grado di sostituirla in modo ragionevolmente facile, nel momento in cui ci fossero alternative più convenienti?
- *Affidabilità del fornitore:* quanto affidabile è il fornitore della tecnologia? L'azienda può contare sul suo supporto stabile e competente?

Queste osservazioni sono solo alcune tra quelle che il responsabile di un progetto deve considerare al momento di una scelta tecnologica importante. Purtroppo, molto spesso queste decisioni sono governate da criteri di carattere commerciale o addirittura ideologico. *È invece essenziale che tutti gli aspetti chiave di una scelta siano opportunamente considerati e pesati.*

Considerazione di carattere generale

Quanto visto mette in luce un fatto essenziale: il responsabile di un progetto di sviluppo di software deve fare una serie di scelte particolarmente critiche e complesse di tipo sia tecnologico che organizzativo-gestionale che si ripresentano spesso nel corso del progetto in funzione dell'evoluzione delle attività. È quindi indubbio che *l'ingegnere del software deve saper coniugare le competenze tecnologiche con adeguate conoscenze di carattere gestionale/organizzativo, al fine di dominare l'intero spettro di problemi che nel corso di un progetto complesso si troverà ad affrontare.*

11.2.5 Gli aspetti organizzativi

Uno dei problemi più delicati che il responsabile di un progetto si trova ad affrontare è l'organizzazione del team di progetto. Alcune tra le domande e questioni più critiche sono le seguenti.

1. Quali persone coinvolgere nel progetto?
2. Con quali responsabilità?
3. Come gestire i problemi di natura organizzativa che nel corso del progetto certamente nasceranno?

Non è scopo di questo testo discutere nel dettaglio le problematiche di carattere organizzativo. Ciononostante, è essenziale e opportuno ricordare che *il responsabile di un progetto deve conoscerle e allineare le sue decisioni alla struttura complessiva del progetto.*

A titolo di esempio, è particolarmente significativo considerare il rapporto tra organizzazione e gestione delle configurazioni. Come discusso nel Capitolo 8, la gestione delle configurazioni definisce alcune regole di gestione del prodotto che sono collegate alle responsabilità dei singoli e alla struttura dei team di progetto. Non ha senso ed è anzi dannoso e controproducente definire procedure di configuration management e change control in modo incoerente con quanto previsto dalla struttura organizzativa presente in azienda e nel team di progetto.

11.3 Le scelte architetturali

I due passaggi chiave che caratterizzano il processo di sviluppo di una soluzione informatica complessa sono lo studio del problema e la progettazione della soluzione. Queste due fasi sono state ampiamente discusse. È importante comprendere come sono collegate. Come si passa dal problema alla soluzione? Come si sceglie l'architettura della soluzione? Come si può essere certi che la scelta fatta sia la più adatta per affrontare in modo convincente ed economicamente conveniente il problema?

Il tema è stato per lungo tempo sottovalutato, in particolar modo fintanto che la complessità delle architetture software è rimasta tutto sommato limitata, ma nel corso

degli ultimi anni è emerso in modo evidente con il crescere delle dimensioni dei sistemi software sviluppati e con l'aumentare della complessità delle tecnologie a disposizione dell'ingegnere del software. Concepire un'architettura informatica complessa richiede una serie di scelte e alternative che non possono essere sottovalutate o ignorate. In particolare, per caratterizzare e affrontare la questione in modo efficace, è necessario considerare tre aspetti chiave: come si passa dal problema alla soluzione; come si sceglie un'architettura e qual è il ruolo giocato in questi passaggi dalle tecnologie di middleware; come si procede nella progettazione e sviluppo di un sistema complesso. Nel seguito del capitolo questi aspetti sono discussi da un punto di vista concettuale. L'Appendice propone un esempio completo che ne mostra un'applicazione concreta.

11.3.1 Come si passa dal problema alla soluzione?

Lo sviluppo dell'informatica negli anni '70 ha portato alla formulazione di alcuni concetti chiave come *information hiding* e *modularità del codice*. Essi affermano che un programma di qualità deve essere *strutturato*, così come anche gli elementi che lo compongono. In particolare, a qualunque livello di granularità, un componente informatico deve avere un'*interfaccia*, che definisce le funzioni/operazioni offerte al "resto del mondo", e un'*implementazione*, che indica come il programma realizza al suo interno quelle funzioni/operazioni. Un programma "ben formato" deve avere interfacce che nascondono l'implementazione e offrono funzioni chiare e coerenti all'utilizzatore.

Un altro dei capisaldi dell'informatica è il concetto di *stepwise refinement* o *raffinamento per passi successivi*. Esso afferma che lo sviluppo del codice di un programma avviene per raffinamenti continui: si parte con l'identificazione dell'operazione che il programma deve svolgere e la si scomponete progressivamente in sequenze di operazioni più semplici. In questo processo, si passa in modo incrementale e progressivo dall'enunciazione di un'operazione di alto livello all'articolazione delle singole operazioni che la implementano, cioè il codice. Per questo motivo si utilizza a volte anche l'espressione *functional decomposition*.

Si noti che mentre il ciclo di vita del software definisce lo schema secondo il quale condurre e organizzare l'intero processo di sviluppo del software, le tecniche tipiche della programmazione strutturata presentate in precedenza hanno lo scopo di guidare il programmatore/progettista nello sviluppo di un singolo semilavorato del processo, come un programma o un documento. Per esempio, nella concezione di un programma per effettuare l'ordinamento di un array, si parte dall'enunciazione generale dell'obiettivo che si vuole ottenere: "ordinare un array di interi in ordine crescente". Tale operazione può essere decomposta in operazioni più semplici:

- acquisisci i dati che costituiscono l'array
- applica uno degli algoritmi di ordinamento noti
- stampa l'array ordinato.

La seconda operazione è ovviamente tutt'altro che semplice. Può essere a sua volta scomposta in operazioni più elementari, che dipendono dal tipo di algoritmo di ordinamento utilizzato: bubblesort, mergesort, swapsort ecc. Secondo il principio dello stepwise

refinement, si procede iterativamente per raffinamenti successivi fino ad arrivare a dettagliare la struttura del codice sorgente che realizza l'operazione originaria. Il codice sorgente è ottenuto incrementalmente a partire dalla formulazione di alto livello dell'operazione che deve essere realizzata.

Taluni contrappongono il principio dell'information hiding e quello dello stepwise refinement. Il secondo è considerato un metodo top-down che mal si presta alla soluzione di problemi complessi: se a livello "alto" si è fatta una scelta progettuale sbagliata, risulta difficile cambiarla. Al contrario, il principio dell'information hiding suggerisce di nascondere le scelte progettuali all'interno di un modulo o di una singola procedura, in modo che eventuali cambiamenti siano limitati e non impattino sul resto del programma. In realtà, i due principi non sono alternativi. Nel progettare una soluzione informatica complessa si procede in parte top-down per raffinamenti successivi, in parte bottom-up. Ai diversi livelli si cerca di applicare il principio dell'information hiding per rendere il codice modulare e "resistente" ai cambiamenti. Come spesso accade, non si tratta di sposare in modo acritico e unilateralmente un principio o l'altro, quanto di applicarli con "creatività e metodo" al fine di ottenere una soluzione che risolva in modo convincente il problema da cui si è partiti.

Quando si studiano sistemi informatici di dimensioni non banali emerge in modo chiaro un problema critico. *Il principio dello stepwise refinement tende a confondere le due fasi di studio del problema e progettazione della soluzione:* la struttura della soluzione nascerebbe incrementalmente attraverso la scomposizione del problema in sottoproblemi. La struttura del problema e quella della soluzione sarebbero in un qualche modo *omologhe*. Tale commistione è ragionevole e indolore nel caso di programmi di complessità limitata. Tuttavia, nel caso di sistemi complessi, essa non ha ragione di esistere: *non è vero che la struttura del problema determina necessariamente la struttura della soluzione.* Per esempio, la Figura 6.6 illustra la module view del client di un sistema client-server. Tale view contiene classi che rappresentano, tra gli altri, il gestore della finestra dell'applicazione e il gestore delle comunicazioni con il server. Questi elementi non hanno nulla a che fare con il problema di partenza (il negozio elettronico), né si può considerare il diagramma delle classi della figura come un "raffinamento" della descrizione del problema. Quest'ultima conterrebbe informazioni quali libri, scaffali e acquisti, ma nulla direbbe sul fatto che queste entità sono le vere e proprie classi che costituiscono la soluzione, né che in questo caso è opportuno avere, per esempio, un'architettura client-server con le caratteristiche discusse nel Capitolo 6.

In generale, *non è sempre vero che la struttura della soluzione è derivata per raffinamenti successivi a partire dalla descrizione del problema.* Questo approccio è largamente suggerito (a volte implicitamente) nella letteratura che presenta e discute il metodo object-oriented. Si parla a questo proposito di "analisi e progettazione object-oriented" come di una *sequenza logicamente continua di attività che portano progressivamente dall'analisi del problema fino alla realizzazione del codice* (la "famosa" *functional decomposition*). Esiste un *unico "modello"* che a diversi livelli di precisione rappresenta "l'analisi" o "il progetto": in particolare, il diagramma delle classi che descrive il problema può essere incrementalmente trasformato nel progetto e quindi nel codice della soluzione.

238 Capitolo 11 Dal problema alla soluzione

Tale approccio avrebbe dei vantaggi non banali: il progettista sarebbe aiutato e indirizzato nel concepire la soluzione dalla struttura e natura stessa del problema. Purtroppo, questo modo di procedere rappresenta l'eccezione e non certo la regola. In progetti complessi, normalmente *la struttura del problema, o meglio, la struttura della descrizione del problema, non è per nulla indicativa della struttura della soluzione*. Guardando il diagramma delle classi (o qualunque altro diagramma UML) che descrive libri, acquirenti e cestini della spesa nell'esempio relativo al sistema di acquisto online, non si nota nulla che possa suggerire una soluzione client-server (a quanti livelli?) o p2p. *Le scelte progettuali sono complesse e non sono implicitamente determinate dal modo secondo il quale è stato descritto il problema.* Inoltre, la descrizione del problema costruita con tecniche e linguaggi come UML normalmente considera solo i requisiti funzionali. In particolare, un diagramma delle classi può certamente illustrare “gli oggetti” presenti nel dominio applicativo, indicandone caratteristiche e proprietà funzionali, ma nulla può dire sui requisiti di sicurezza, fault-tolerance o prestazione.

L'approccio suggerito da Michael Jackson e adottato in questo testo è che, essendo problema e soluzione due entità diverse che hanno struttura e natura differenti, *devono in generale esistere due “modelli” (o meglio descrizioni) diversi: quello che racconta la natura del problema e quello che indica la struttura della soluzione.* Tra i due esiste un legame logico determinante: *le scelte progettuali dell'ingegnere del software.* È tramite queste scelte, non ricavabili direttamente dalla descrizione funzionale del problema, che si determina la struttura della soluzione. *È l'ingegnere del software che deve decidere quale sia il miglior tipo di architettura, quali siano i componenti da sviluppare e la natura e organizzazione delle classi che li realizzano.* Ciò è ancora più vero quando si tratta di estendere un sistema informatico esistente, che ha già una sua struttura, oppure quando si sviluppano sistemi informatici riusando componenti preesistenti.

Per questi motivi, *nel condurre un progetto di sviluppo di software è essenziale distinguere sempre e comunque la fase di studio del problema da quella di progettazione della soluzione.* Ciò non significa in alcun modo procedere sempre secondo un ciclo di vita a cascata: le due fasi possono accavallarsi e ripetersi in funzione del ciclo di vita prescelto. L'aspetto essenziale che l'ingegnere del software deve sempre tenere presente è lo scopo ultimo delle attività che in quel momento sta svolgendo: “sto studiando le caratteristiche del problema, e quindi ‘ascoltando l’utente’, oppure sto facendo scelte progettuali sfruttando le mie competenze e le tecnologie che ho a disposizione.” *Nello studio di ciascun ambito è possibile procedere per raffinamenti successivi, ma non si può immaginare che tale criterio si possa utilizzare per gestire il passaggio complessivo dalla descrizione del problema all’identificazione della soluzione.*

11.3.2 Come si sceglie un'architettura?

Il Capitolo 7 ha presentato un certo numero di stili architettonici. Per alcuni di essi, come client-server e p2p, sono possibili varianti significativamente diverse che ne mutano in modo non marginale caratteristiche e proprietà. L'ingegnere del software ha quindi a di-

11.3 Le scelte architetturali 239

sposizione un insieme ragionevolmente ampio di scelte e alternative. Inoltre, i diversi stili architetturali possono essere combinati per ottenere strutture e architetture complesse.

Come si giunge a scegliere uno stile architettonico (o una combinazione di stili) tra i tanti possibili? Se è vero che l'architettura di un sistema non è derivabile in modo diretto dalla natura e struttura del problema, quali criteri e linee guida occorre seguire nel progettare la soluzione? In realtà questo passaggio costituisce *uno degli elementi più critici e complessi del lavoro dell'ingegnere del software*. In questa fase entrano in gioco aspetti quali l'esperienza del progettista, la sua capacità di analisi e sintesi, l'intuito, la creatività. Il problema da risolvere è “osservato e studiato”; la soluzione (la sua architettura) deve essere “ideata e pensata” dal progettista. In generale, è difficile ricondurre la scelta di una soluzione architettonica a un mero processo o metodo sistematico che determina in modo meccanico quale alternativa preferire. Ci sono peraltro una serie di criteri e di linee guida che possono essere seguiti.

In primo luogo, esistono soluzioni architettoniche per specifici settori applicativi. Per esempio, nel caso dei sistemi di controllo, vi sono architetture di riferimento che possono essere raffinate e adattate al caso specifico. Nel mondo delle applicazioni per Internet, la stessa architettura J2EE identifica una serie di elementi e componenti per la costruzione di applicazioni distribuite client-server multilivello. L'ingegnere del software, a fronte di un problema, può quindi innanzitutto *valutare le scelte architettoniche già perseguiti in casi simili e che si sono rivelate efficaci per quel tipo di problemi*. Negli anni scorsi, è stata creata l'espressione *Domain Specific Software Architecture* proprio per indicare soluzioni architettoniche ottimali per “specifici domini applicativi”.

Più in generale, la definizione dell'architettura di un sistema complesso viene effettuata studiando i requisiti e le caratteristiche del problema sia da un punto di vista funzionale che non funzionale. *I risultati di queste analisi vengono confrontati con le caratteristiche e proprietà tipiche dei diversi stili e da questo nascono le ipotesi architettoniche*. Se il problema, per esempio, è l'integrazione di archivi dati gestiti da organizzazioni autonome, è evidente che nascono (almeno) due possibilità: o si consolidano tutti i dati in un archivio centrale, con un sistema client-server (magari a più livelli per ottimizzare le prestazioni complessive) oppure si adotta un'architettura p2p nelle quale ciascuno continua a essere “padrone e gestore unico” dei propri dati. In questo secondo caso, si pone il problema di come propagare query e operazioni di aggiornamento delle informazioni. Quindi si potrebbe pensare a un sistema p2p ibrido oppure a un'architettura composita p2p + publish-subscribe. La scelta tra queste diverse alternative deve considerare i trade-off (vantaggi e svantaggi) che ciascuna soluzione comporta, così come discusso nel Capitolo 7. Se per esempio le funzioni e operazioni svolte da ciascun nodo autonomo sono le stesse e avvengono su dati assolutamente identici, non ha senso avere un replicazione dei sistemi: un sistema centralizzato (magari ridondato per garantire fault-tolerance e affidabilità) è la soluzione più conveniente dal punto di vista economico e gestionale. Se invece i diversi nodi hanno, almeno in parte, dati diversi con funzioni che possono mutare o evolvere in modo indipendente, è preferibile (se non obbligatorio) attuare una strategia distribuita p2p.

In ogni caso, è evidente che *il progettista si trova di fronte a un insieme non limitato di alternative*. Spesso, esse non sono presenti solo o principalmente a livello di stile architettonico, ma anche e soprattutto a livello di variante o specifica implementazione dello stile o di combinazione di stili. *Sono l'esperienza, le competenze, l'intuito e la professionalità dell'ingegnere del software a costituire la chiave di volta per affrontare questo passaggio in modo efficace e convincente.*

La scelta di un'architettura deve essere guidata anche da *considerazioni di carattere prestazionale*. Potrebbe essere necessario, per garantire un certo tipo di prestazioni, dover valutare quale architettura risulti più conveniente, oppure come una certa architettura debba essere rivista o adattata per conformarsi al problema specifico. Per esempio, dovendo gestire un alto carico di richieste in un sistema client-server multi-tier, potrebbe essere utile avviare studi che valutano, dal punto di vista della capacità complessiva di risposta, architetture a tre o quattro livelli, con diversi meccanismi di distribuzione delle funzioni applicative o dei criteri di replica e gestione dei tier intermedi. Da tempo, questo tipo di analisi è condotto simulando e studiando il comportamento delle diverse architetture tramite modelli matematici basati sulla teoria delle code (o tecniche simili). Tali approcci sono piuttosto diffusi in molti settori dell'ingegneria e sono certamente di ausilio anche all'ingegnere del software.

11.3.3 Il ruolo del middleware

Un aspetto importante che influisce in modo significativo sull'attività di progettazione è la scelta del middleware. Infatti, è indubbio, e alcuni studi lo hanno dimostrato, che *l'utilizzo di una particolare tecnologia di middleware influenza sulle scelte architettoniche*. Può persino accadere che l'adozione di un tipo di middleware renda più o meno facile (e persino possibile) la realizzazione di una soluzione basata su una specifica architettura.

Un caso esemplare è costituito dalle tecnologie di middleware a eventi (tipo JMS). Esse forniscono operazioni e servizi per gestire la pubblicazione e ricezione di eventi: offrono il dispatcher e i servizi di publish, subscribe, push e pull. Dovendo realizzare un'architettura publish-subscribe, è indubbio che la disponibilità di questo tipo di tecnologia facilita grandemente tutto il processo di sviluppo. Il progettista può concentrarsi nell'identificazione dei diversi componenti dell'architettura, dei tipi di eventi da scambiare e della gestione della chiamata dei servizi offerti dal middleware, ignorando gli aspetti implementativi della gestione degli eventi. Al contrario, se si dovesse realizzare un sistema client-server utilizzando un middleware a eventi, il progettista si troverebbe di fronte a problemi non marginali. Certamente è possibile realizzare una chiamata sincrona attraverso pubblicazioni di eventi e sottoscrizioni, ma il compito del progettista sarebbe alquanto ostico.

- Chi richiede un servizio deve pubblicare un evento con le informazioni sul servizio richiesto.
- Il ricevente deve essersi già sottoscritto a quel tipo di evento. In caso contrario, la richiesta va persa e il richiedente rimane inascoltato (senza avere meccanismi efficienti per venirlo a sapere in tempi rapidi).

- Quando chi offre il servizio riceve l'evento di richiesta, deve generare un nuovo evento con la risposta.
- Il richiedente del servizio deve essersi registrato per tempo, per poter essere sicuro di ricevere l'evento di risposta.

Come si può facilmente desumere, pensare di gestire interazioni client-server con questo meccanismo è semplicemente improponibile: appare quindi evidente che *la scelta del middleware può avere un'influenza più o meno significativa sull'architettura (e viceversa)*. In generale, ogni tecnologia di middleware, in modo più o meno invasivo, *induce uno o più stili architetturali*. Esistono tecnologie di middleware molto semplici e flessibili che possono essere utilizzate per tutti gli stili architetturali. Altre sono particolarmente adatte a realizzare solo uno o pochi stili. Per esempio, con RMI è possibile realizzare sistemi client-server a più livelli e, con un po' di programmazione aggiuntiva, anche sistemi p2p. J2EE è particolarmente adatta a sistemi client-server multi-tier con codice mobile di tipo COD (tramite gli applet), mentre può risultare (nella sua configurazione completa) particolarmente pesante e onerosa per gestire sistemi p2p con funzioni RE (*Remote Evaluation*).

Avendo scelto uno stile architetturale, è *quindi necessario assicurarsi che il middleware prescelto (o da scegliere) sia "compatibile" con la scelta fatta. Il middleware gioca inoltre un ruolo importante anche nella scelta di componenti da integrare in un sistema informatico esistente o già parzialmente implementato*. Infatti, se il componente utilizza meccanismi di middleware e logiche di funzionamento (interfacce) non compatibili con il "sistema ricevente", si ha una situazione di integrazione impossibile ("rigetto") o estremamente costosa (è necessario costruire adattatori che facciano da interprete tra le diverse parti).

Ancora una volta, *emerge un problema di scelta progettuale che vede l'ingegnere del software giocare un ruolo essenziale*. Nel determinare l'architettura del sistema e il middleware di supporto, l'ingegnere deve assicurarsi che non vi siano *architectural mismatch*, cioè incompatibilità architetturali tra i diversi elementi dell'architettura e in particolare tra essi e il middleware scelto per implementarli.

11.3.4 Bottom-up o top-down

In diverse circostanze, è emerso un tema particolarmente rilevante: nella progettazione e realizzazione di un sistema informatico si procede in modo top-down o bottom-up?

L'approccio top-down deriva direttamente dalla logica dello stepwise refinement: si parte dall'astrazione massima e progressivamente la si raffina in funzioni elementari e quindi in codice. Più in generale, si definisce l'architettura del sistema e si raffinano i diversi componenti ed elementi che ne emergono. "Scendendo" a livello di componente si giunge a progettare il modulo e quindi il codice dei singoli metodi o procedure.

Nell'approccio bottom-up, si parte da componenti già pronti o facilmente realizzabili e li si assembla progressivamente fino a costituire il sistema completo. È l'approccio che sfrutta al meglio i cicli di vita basati sul riuso di componenti (*component-based development*).

242 Capitolo 11 Dal problema alla soluzione

Nei progetti reali, non si segue quasi mai in modo “puro” né l’uno né l’altro approccio. Spesso si procede con una progettazione architettonale di massima e poi la si “popola” con componenti preesistenti o sviluppati ad hoc. Talvolta, esistono settori dell’azienda che sono focalizzati sulla produzione bottom-up di componenti di uso generale riusabili dai gruppi di progetto. *Molto spesso ci si trova quindi di fronte a una sorta di approccio jo-jo, nel quale in funzione del ciclo di vita prescelto si procede alternativamente e iterativamente in parte top-down e in parte bottom-up.*

11.4 Dal progetto al codice

Obiettivo di un progetto di sviluppo del software è mettere in esercizio un’applicazione informatica. È ovvio, quindi, che lo sviluppo del codice è una fase essenziale. Laver progettato correttamente la soluzione informatica è il prerequisito affinché si possa procedere alla realizzazione del codice che costituirà il prodotto vero e proprio consegnato all’utente.

Non è certo scopo di questo testo discutere come si procede nella scrittura di codice ben formato. Il lettore dovrebbe aver acquisito queste competenze attraverso corsi base di informatica, casi di studio e/o la propria attività professionale. Vi sono peraltro alcune osservazioni che meritano specifici approfondimenti e che possono essere utili per collegare tra loro idee, tecniche e indicazioni emerse nel corso della trattazione.

11.4.1 Il ruolo di UML

UML è il linguaggio di descrizione utilizzato nei precedenti capitoli per descrivere il problema e la soluzione. Non è certamente l’unico linguaggio utilizzabile e secondo alcuni non è neanche il migliore. Soffre di una serie di limitazioni, non ultima l’essere in realtà un insieme per certi versi disorganico di notazioni grafiche informali e non logicamente correlate. Che legame c’è tra un diagramma use case e un sequence? Solo il progettista sa se e come sono legati. Ciononostante, UML è molto diffuso ed è quindi indubbio che costituisca nei fatti quasi uno standard. Pochi saranno coloro che non si troveranno a usarlo, presto o tardi.

Una questione che è importante sottolineare è la relazione tra UML e linguaggi non object-oriented. UML può essere usato per descrivere sia il problema che la soluzione. Specie nel secondo caso, emerge in maniera evidente l’approccio e l’idea di fondo che ne hanno guidato la concezione: *UML nasce principalmente come strumento per facilitare la descrizione di programmi scritti con linguaggi object-oriented.* Una classe UML può facilmente essere trasformata in una classe Java o C++. Viceversa, da un programma object-oriented è abbastanza facile derivare un diagramma delle classi che ne rappresenta struttura e organizzazione in forma sintetica e grafica.

In generale, *UML può essere utilizzato anche per progettare e descrivere sistemi che non sono realizzati con linguaggi object-oriented.* Alcune delle notazioni di UML non hanno

nulla che le vincoli al modello object-oriented: activity, state, use-case, deployment e component diagram sono quasi totalmente slegati dal mondo degli oggetti e delle classi. Altri diagrammi, come i class e i sequence diagram, sono invece sostanzialmente e strutturalmente connessi al paradigma object-oriented. Questi diagrammi sono quelli utilizzati, in particolare, nella module view per mostrare la struttura del codice sorgente. Quindi occorre chiedersi se e come utilizzarli per descrivere codice scritto con linguaggi meno sofisticati come C o lo stesso assembler.

In realtà ciò è non solo possibile, ma anzi auspicabile. Il concetto di classe in UML ha due aspetti che sono difficilmente replicabili o adattabili alla programmazione tradizionale: l'ereditarietà e l'istanziabilità. È difficile, se non impossibile, scrivere un modulo C che eredita proprietà da un "supermodulo" (si possono usare operazioni di inclusione, ma ovviamente non si tratta dello stesso tipo di meccanismo) ed è altrettanto difficile creare più istanze di uno stesso modulo C. Ciononostante, si possono costruire tipi di dati astratti con qualunque linguaggio di programmazione. Anche in assembler è possibile costruire moduli che hanno una struttura interna nascosta e che dispongono di procedure invocabili dall'utente del modulo. Ovviamente, non è così facile creare più istanze di uno stesso tipo di dato astratto, né controllare che l'utilizzo di tale struttura sia conforme a quanto specificato. In assembler (ma anche in C) si potrebbero utilizzare meccanismi di vario tipo per accedere all'implementazione di un modulo senza usare i metodi dell'interfaccia. Nonostante queste avvertenze e potenziali problemi, è *peraltro indubbio che anche con linguaggio di livello più basso è possibile e fortemente auspicabile applicare i principi della programmazione strutturata e dell'information hiding*.

Di conseguenza, *UML può essere utilizzato per descrivere codice strutturato che verrà (o è stato) sviluppato con linguaggi non object-oriented*. Tipicamente, il concetto di classe può essere utilizzato per rappresentare un tipo di dato astratto. Gli attributi rappresentano le variabili del modulo e i metodi della classe costituiscono le procedure offerte dal modulo. È un modo semplificato di utilizzare UML e che non ne sfrutta appieno potenzialità e caratteristiche: tuttavia, risulta molto efficace in quanto permette di rappresentare in forma grafica (e particolarmente leggibile) programmi anche molto complessi.

11.4.2 Come si scrive il codice

Avendo a disposizione la descrizione in UML (o in un qualche linguaggio di descrizione) di un modulo o di una classe, come si scrive il codice? Se la descrizione in UML è accurata e completa, il programmatore è veramente a buon punto: può utilizzare la descrizione delle interfacce e tramite i sequence diagram (o altri diagrammi UML) ha indicazioni su come costruire il corpo dei metodi (le istruzioni).

Da un punto di vista pratico, esistono una serie di semplici criteri e suggerimenti che possono essere seguiti per costruire un codice ben formato, di facile lettura e che può agevolmente essere modificato ed esteso.

Naming standard

È opportuno che tutti i nomi di classi, procedure, metodi e variabili siano selezionati usando criteri condivisi. Non esistono standard ufficiali, anche se ci sono convenzioni sia a livello di alcune comunità di programmatore che all'interno delle singole aziende. Conviene comunque che in ogni progetto vi siano alcune regole standard alle quali tutti i componenti del team di sviluppo si adeguano.

Per esempio, è possibile avere regole come le seguenti.

- Tutte le classi hanno nomi composti da una o più parole che iniziano tutte con una lettera maiuscola: AutoInServizio, PazienteRicoverato ...
- Tutte le variabili hanno nomi composti da una o più parole che iniziano tutte con una lettera maiuscola tranne la prima: codiceAuto, nomePaziente ...
- Tutti i nomi di metodo sono strutturati come i nomi di variabili, ma hanno come prima parola un verbo che spiega il senso del metodo: modificaCodice, cancellaPaziente ...
- Le variabili usate come contatori o come dati di supporto hanno nomi semplici e facilmente intellegibili scritti in minuscolo: i, j, cont, trovato, ultimo ...

Quanto visto costituisce solo un'ipotesi di massima che ciascun team/azienda può estendere e/o modificare secondo i propri gusti e preferenze.

Coding standard

I linguaggi di programmazione possono essere utilizzati in vario modo ed è quindi opportuno che, per garantire uniformità di scrittura del codice e una sua alta leggibilità, esistano delle convenzioni adottate da tutti i programmatore. Tali convenzioni possono essere utili anche per evitare errori.

Un semplice esempio è il seguente. In C (e in C++ e Java) è possibile scrivere un'istruzione `if-then-else` nel seguente modo:

```
if (a > 0) b = a; else b = -a;
```

Uno stile di scrittura (o anche di *indentazione*) che rende maggiormente leggibile il codice è il seguente:

```
if (a > 0)
    b = a;
else
    b = -a;
```

In questo modo, risulta molto più visibile e leggibile il fatto che ci si trova di fronte a una alternativa.

È tuttavia possibile avere regole e standard di codifica che affrontano problemi più significativi e che vanno oltre la semplicità e leggibilità del codice. Per esempio, se si volesse aggiungere l'istruzione `c=-a` al ramo `else` dell'`if`, sarebbe un errore scrivere quanto segue:

11.4 Dal progetto al codice 245

```
if (a > 0)
    b = a;
else
    b = -a;
c = -a;
```

Infatti, solo un'istruzione viene associata all'`else`: il programmatore si è dimenticato di inserire le parentesi graffe che definiscono l'istruzione composita associata all'`else`. Per prevenire ed evitare questi errori è buona regola scrivere da subito l'istruzione nel seguente modo:

```
if (a > 0)
{
    b = a;
}
else
{
    b = -a;
}
```

Così, quand'anche si dovesse aggiungere un'istruzione, appare subito evidente dove essa deve collocarsi.

Questo approccio alla programmazione può spingersi ancora oltre. Esistono tecniche dette di *defensive programming* che permettono di evitare di scrivere codice errato o che generi malfunzionamenti in situazioni particolari. Un caso tipico è la gestione dei casi non previsti in un'istruzione `switch`. Anche se si è ragionevolmente certi di aver considerato tutte le possibili alternative tramite opportune clausole `case`, è buona norma prevedere una clausola `default` che gestisca situazioni impreviste. Un'altra importante tipologia di controlli riguarda la validazione dei dati ricevuti in ingresso dal programma. In generale, in Java l'uso delle eccezioni è un altro potente meccanismo che permette di scrivere codice "resistente" a errori o malfunzionamenti.

Il tema dell'affidabilità e robustezza del codice sarà ulteriormente discusso nel seguito del presente capitolo.

Documentation standard

Anche il codice deve essere documentato. Addirittura, approcci come l'eXtreme Programming teorizzano che il codice "è" la documentazione. A maggior ragione è necessario che esso contenga informazioni che lo rendano comprensibile a chiunque lo legga.

Per questo motivo è opportuno definire regole di documentazione del codice che specificano una serie di informazioni chiave.

- Informazioni relative all'autore del codice, alle modifiche effettuate e alle date nelle quali sono state effettuate. Questo tipo di informazioni possono essere ridondanti in quanto già raccolte dal sistema di configuration management. Il capo progetto dovrà decidere quali informazioni includere nel codice sorgente, in funzione degli strumenti e delle tecnologie in uso nel progetto.

- Funzione e scopo di classi, metodi e variabili.
- Logiche di utilizzo dei metodi presenti nella classe.
- Aspetti particolarmente critici o delicati nella struttura di un metodo (per esempio, vincoli sulla dimensione di buffer o array interni al metodo).

"Esegui il codice subito"

Una tecnica molto efficace sia per motivare il programmatore sia per verificare incrementalmente il funzionamento di un programma è quella di procedere nelle attività di sviluppo in modo da poter eseguire da subito e regolarmente il codice. In pratica, si tratta di scrivere il programma in modo da avere sempre codice funzionante. In principio, il programma sarà costituito semplicemente da quelle istruzioni che segnalano il corretto avvio del codice (per esempio, la stampa di una qualunque scritta tipo "sono nel main!"). Incrementalmente, il programmatore aggiunge le diverse funzioni (per esempio una voce di menu e il codice relativo), verificandone subito il funzionamento. In questo modo, il programma "cresce sotto gli occhi del programmatore" che ha la sensazione immediata che "c'è qualcosa che va". In generale, ogni incremento è deciso ed effettuato cercando di renderlo da subito eseguibile e verificabile.

Questo semplicissimo approccio incrementale permette anche di verificare da subito problemi di integrazione o di "mismatch" tra classi e metodi.

Allineamento tra UML e codice

Un problema particolarmente importante è l'allineamento tra il codice sviluppato e la sua descrizione (tipicamente i diagrammi UML). Molto spesso accade che il progettista modifichi il codice perché si rende conto di errori o carenze che non erano emerse in fase di progettazione. Tipicamente, egli tende a cambiare il codice, senza riportare le modifiche sui diagrammi di progetto che diventano, quindi, rapidamente obsoleti.

Vi sono due tipi di soluzioni a questo problema. La prima è di tipo organizzativo-metodologico: il team e il capo progetto si organizzano in modo da verificare periodicamente l'allineamento tra codice e documentazione di progetto. Ovviamente, in questo modo è alto il rischio che l'urgenza delle scadenze metta in secondo piano l'aggiornamento della documentazione, in quanto essa non contribuisce direttamente alla "consegna del codice al cliente". Per questo motivo, alcuni strumenti per il disegno di diagrammi UML iniziano a disporre di funzionalità che permettono di generare il codice (o scheletri di codice) a partire dai diagrammi UML. Inoltre, questi strumenti prevedono sempre più spesso funzioni di *reverse engineering* in grado di ricostruire automaticamente almeno parte della descrizione UML di codice esistente.

È comunque indubbio che è responsabilità primaria del capo progetto e del team di sviluppo assicurarsi che, periodicamente e specialmente in vista di milestone o release importanti di progetto, sia fatto un allineamento completo tra documentazione e codice.

11.5 Affidabilità e robustezza del codice

Le tecniche di defensive programming suggeriscono criteri per la scrittura di codice che sia resistente a errori e malfunzionamenti. In generale, il tema dell'affidabilità e robustezza del codice è sentito come uno dei problemi più critici e importanti per garantire il successo di un progetto di sviluppo di software. Inoltre, è importante avere criteri e metodi che guidino il progettista nel passaggio dal progetto di dettaglio al codice vero e proprio.

Per questo motivo, nel corso degli anni questi temi sono stati approfonditi e studiati sia dal punto di vista metodologico che dal punto di vista delle tecnologie di supporto. Le soluzioni sviluppate nel corso di questi anni si applicano a diverse tipologie di linguaggi. Tre contributi risultano particolarmente significativi.

- La tecnica del design by contract, che ha una valenza e un'applicabilità di tipo generale.
- Il meccanismo delle asserzioni, sviluppate per vari tipi di linguaggi e, più recentemente, per Java.
- JML (*Java Modeling Language*), linguaggio sviluppato per facilitare l'applicazione delle tecniche del design by contract a Java.

11.5.1 Design by contract

Il passaggio da problema a soluzione vede come elemento di congiunzione il concetto di descrizione dell'interfaccia, cioè la specifica. Essa rappresenta il contratto tra chi ha un problema da risolvere e chi che ne progetta e sviluppa una soluzione. Questo rapporto "contrattuale" in realtà non si manifesta solo al confine tra problema e soluzione. Ogni qual volta nello scrivere un frammento di codice si invoca un metodo non si fa altro che richiedere un servizio offerto "da qualcun altro" secondo una specifica condivisa: si invoca il metodo sapendo che fornirà un risultato conforme a "quanto ci si aspetta". I concetti di modularità e di information hiding, normalmente applicati per sviluppare codice strutturato e di elevata qualità, altro non sono se non un'applicazione di semplici concetti.

1. Ciascuna porzione di codice offre dei servizi tramite la sua interfaccia.
2. I servizi sono conformi a una specifica che costituisce il "contratto" stabilito con l'utilizzatore dell'interfaccia stessa.
3. Il modo secondo il quale il contratto viene rispettato è nascosto all'interno dell'implementazione.

Nel corso degli anni, queste idee si sono arricchite e formalizzate fino a giungere alla formulazione del principio del *Design By Contract* (DBC): *la progettazione del software avviene definendo in modo organico e coerente i "contratti" che ciascuna porzione di codice si impegna a rispettare nei confronti dei propri utilizzatori*.

248 Capitolo 11 Dal problema alla soluzione

Nell'interpretazione comune, un contratto si articola in clausole e impegni. Nel caso dell'ingegneria del software, le "clausole e gli impegni" sono sostanzialmente di tre tipi.

- *Precondizioni*. Una precondizione è un'espressione logica che deve essere verificata perché una certa porzione di codice possa essere eseguita. È quanto l'utilizzatore deve accertarsi sia garantito al momento dell'invocazione del codice.
- *Postcondizioni*. Una postcondizione indica la condizione che deve essere verificata al termine di un'elaborazione. È quanto chi ha scritto il codice chiamato si impegna a garantire al chiamante a valle della sua esecuzione.
- *Invarianti*. Un invariante è un predicato che esprime una proprietà o caratteristica che non varia nel corso dell'esecuzione.

Precondizioni, postcondizioni e invarianti costituiscono informazioni importanti per caratterizzare il funzionamento di una porzione di codice. Per esempio, si consideri il caso di una semplice operazione di divisione tra due interi positivi implementata dal seguente metodo:

```
public int dividi(int a, int b)

// Precondizione: b != 0
// Postcondizione: a = risultato*b + resto

{return a/b;}
```

La precondizione che deve essere verificata perché il metodo sia invocato correttamente è che `b` non sia uguale a zero, altrimenti la divisione genererebbe un errore a run-time. La postcondizione illustra il risultato che si ottiene invocando il metodo. In questo caso non era necessario prevedere invarianti, in quanto il programma scelto come esempio è particolarmente semplice. In generale, gli invarianti definiscono vincoli di integrità e coerenza su strutture dati o all'interno di elaborazioni complesse. Esempi di invarianti saranno presentati nel seguito del capitolo.

I concetti di precondizioni e postcondizioni permettono di definire un altro importante aspetto. Per come è stato concepito, il metodo `dividi` è una *funzione parziale* poiché fornisce un risultato corretto solo se i parametri di ingresso soddisfano la precondizione: se il valore passato per `b` fosse zero, quando si arriva all'esecuzione della divisione, si verifica un errore run-time e il metodo terminerebbe in modo anomalo. Se un metodo non avesse precondizioni, vorrebbe dire che è in grado di offrire un risultato per qualunque valore in ingresso. In questo caso si dice che realizza una *funzione totale*. Per esempio, il metodo `dividi` può essere trasformato in funzione totale estendendola come segue:

```
public int dividi(int a, int b)

// Postcondizione
// Se b==0, risultato = -1
// Se b!=0, a = risultato*b + resto
```

11.5 Affidabilità e robustezza del codice 249

```
{
    if (b==0)
    {
        return -1;
    }
    else
    {
        return a/b;
    }
}
```

Alternativamente, in Java è possibile utilizzare il meccanismo delle eccezioni:

```
public int dividi(int a, int b) throws Exception

// Postcondizione
// Se b==0, throws Exception
// Se b!=0, a = risultato*b + resto

{
    if (b==0)
    {
        throw new Exception();
    }
    else
    {
        return a/b;
    }
}
```

Ovviamente, sarebbe auspicabile avere sempre funzioni totali. In presenza di funzioni parziali, è necessario che il programmatore si assicuri che i parametri passati alla funzione siano coerenti con la precondizione, altrimenti potrebbero verificarsi errori run-time o risultati scorretti. Sfortunatamente, è impossibile avere solo funzioni totali in quanto potrebbero appesantire in modo inaccettabile le prestazioni del sistema. Si pensi, per esempio, al caso di una ricerca binaria su un array di numeri interi. Essa si basa sul principio che l'array sia ordinato. Se non lo fosse, l'algoritmo darebbe risultati imprevedibili. Un ipotetico metodo `binarySearch` è quindi solitamente una funzione parziale che ha come precondizione il fatto che l'array sul quale si richiede la ricerca sia già ordinato. Per rendere la funzione totale, essa dovrebbe preliminarmente scandire tutto l'array per verificarne l'ordinamento: tanto varrebbe effettuare direttamente una ricerca per scansione sequenziale completa!

Dagli esempi fatti si evince che è effettivamente utile e importante specificare in modo preciso il comportamento di un metodo secondo i principi del design by contract. In generale, *lo standard di documentazione del codice dovrà prevedere che ogni metodo sia opportunamente descritto dal contratto che realizza*. Tuttavia, se ci si limitasse a inserire alcuni semplici commenti testuali, nulla vietterebbe al programmatore di scrivere un codi-

250 Capitolo 11 Dal problema alla soluzione

ce errato che viola il contratto: come controllare che il codice sia coerente con un commento testuale? Per questi motivi, sono state realizzate specifiche tecnologie che aiutano a formalizzare la definizione dei contratti e ad automatizzarne la verifica. In particolare, il linguaggio Java, come altri, prevede il costrutto linguistico delle asserzioni tramite le quali è possibile dichiarare i predicati che devono essere verificati nel corso del programma. Un ulteriore sviluppo delle tecniche di design by contract è costituito dall'avvento di strumenti come JML. Mentre le asserzioni sono istruzioni del linguaggio Java, JML è un linguaggio a sé stante, utilizzato per specificare il contratto a cui deve sottostare un metodo e per verificarne il suo effettivo rispetto.

11.5.2 Asserzioni

Un'asserzione è un meccanismo che consente, in uno specifico punto dell'esecuzione del programma, di verificare se una determinata condizione è vera. Qualora fosse falsa, l'esecuzione viene interrotta o comunque viene segnalata una situazione di errore.

Molti linguaggi di programmazione supportano le asserzioni in modo nativo, attraverso un apposito costrutto sintattico. In Java, un'asserzione è definita tramite l'istruzione `assert`, che ha due formati possibili:

```
assert booleanExpression;
assert booleanExpression : errorMessage;
```

L'istruzione `assert` può essere inserita in un qualunque punto di un programma Java per controllare se, in quel particolare momento dell'esecuzione, l'espressione logica indicata è vera. In caso affermativo, l'esecuzione procede normalmente. Altrimenti, viene generata un'eccezione di tipo `AssertionError` e viene stampato, se presente, il relativo messaggio di errore.

Il meccanismo delle asserzioni può essere utilizzato per definire precondizioni, postcondizioni e invarianti. Per esempio, per effettuare una divisione, deve essere vero che il denominatore è maggiore di zero.

```
assert b != 0 : "Errore! Divisione per zero";
c = a/b;
```

Come esempio di postcondizione, a valle dell'inserimento di un numero intero in un array il contatore che indica quanti elementi sono presenti nell'array deve risultare incrementato di uno (`num` è il numero di elementi presenti nell'array, `element` il numero da inserire e `size` la dimensione dell'array).

```
// precondizione: l'array non deve essere pieno

assert num < size : "L'array è pieno!";

int oldNum = num;
array[num++] = element;
```

11.5 Affidabilità e robustezza del codice 251

```
// postcondizione

assert num == oldNum+1 && array[num-1] == element :
    "Errore nella gestione del contatore";
```

L'istruzione assert può essere utilizzata anche per definire un invarianto che caratterizza specifiche proprietà degli oggetti di una classe. Nel caso dell'array utilizzato in precedenza, il contatore di elementi presenti non deve essere “fuori dai limiti”. Può quindi essere utile che in punti specifici del programma sia inserita la seguente asserzione:

```
assert num >= 0 && num < size:
    "Errore nella gestione dell'array!" ;
```

Il controllo delle asserzioni può appesantire l'esecuzione del programma. Per questo motivo, è possibile disabilitare il controllo delle asserzioni. In genere, le asserzioni vengono *abilitate* quando il programma è in fase di test, mentre sono *disabilitate* quando si è raggiunta una ragionevole stabilità e affidabilità del codice e il programma viene messo in esercizio o, con un'espressione di uso comune, “passa in produzione” (inizia a essere usato regolarmente dagli utenti).

11.5.3 JML

JML (*Java Modeling Language*) è stato concepito per facilitare lo sviluppo di codice affidabile e robusto secondo i principi del design by contract. È un linguaggio separato da Java ed è utilizzato per specificare il comportamento di metodi e classi. Rispetto alle asserzioni Java, JML introduce una serie di funzionalità che arricchiscono in modo significativo la capacità di intervento e controllo del progettista.

Il programmatore inserisce le istruzioni JML all'interno del codice Java. Il programma risultante viene utilizzato in vari modi. I più frequenti sono i seguenti.

- Tramite un'apposita estensione del compilatore Java (denominata `jmlc`) è possibile tradurre in byte code il programma Java arricchito con istruzioni JML. Il codice risultante include i controlli run-time che verificano quanto specificato in JML.
- Lo strumento di supporto al test `JUnit` è stato esteso dando origine a `jmlunit`. Questo nuovo strumento facilita grandemente lo svolgimento delle procedure di test, in quanto è in grado di utilizzare le istruzioni JML per verificare se il codice si comporta secondo le specifiche.
- Grazie a `jmldoc`, è possibile generare documentazione di programma per codice Java che include istruzioni JML.

Le funzionalità e l'utilizzo di JML sono illustrati dal seguente esempio. Si consideri il caso del metodo di ricerca binaria `binarySearch` citato in precedenza. Si è osservato che tale metodo è di norma realizzato tramite una funzione parziale. Se si volesse applicare la tecnica del defensive programming e si volesse trasformare il metodo in una funzione totale, sarebbe necessario inserire all'inizio del metodo stesso una serie di istruzioni che

252 Capitolo 11 Dal problema alla soluzione

controllano se l'array è ordinato prima di procedere alla ricerca. Ciò renderebbe il metodo inutilizzabile.

Utilizzando JML si può ottenere un triplice vantaggio: specificare correttamente la natura del metodo; effettuare i controlli run-time durante la fase di test, disabilitandoli quando il programma va in produzione; documentare in modo efficace il codice del metodo. In particolare, il codice Java del metodo può essere così arricchito:

```
/*@ requires vet != null
@           && (\forall int i;
@           0 < i && i < vet.length;
@           vet[i-1] <= vet[i]);
@*/
int binarySearch(int[] vet, int x) { ... }
```

Il codice JML è racchiuso dai simboli speciali `/*@`, `@` e `@*/`. Oppure è indicato su una sola riga che inizia con il simbolo `//@`. Tali simboli sono utilizzati per distinguere il codice JML da quello Java vero e proprio. Come si nota nell'esempio, le istruzioni JML sono molto simili a comuni istruzioni Java, con alcune differenze. Nell'esempio, la clausola `requires` è utilizzata per introdurre una precondizione: perché il metodo possa essere eseguito correttamente, “è richiesto” che quanto segue sia vero. In particolare, tramite il quantificatore `\forall` si richiede che tutti gli elementi siano ordinati.

Il lettore potrebbe obiettare che quanto specificato in JML altro non è se non almeno parte del codice che si sarebbe inserito volendo perseguire l'approccio “difensivo” scartato in precedenza perché troppo oneroso. In realtà, è possibile far sì che le istruzioni JML siano compilate ed eseguite solo quando il programma è in fase di test, mentre possono essere considerate una specifica formale e precisa del funzionamento del metodo, una volta che si sia conclusa la fase di verifica del corretto funzionamento del programma.

Oltre a `requires`, JML introduce una serie di clausole particolarmente interessanti.

- `invariant`: permette di definire invarianti.
- `ensures`: permette di definire postcondizioni.
- `signals`: permette di definire una postcondizione per situazioni particolari (come quelle che si verificano quando il metodo termina generando un'eccezione).
- `assignable`: se presente, indica che all'interno di un metodo è possibile assegnare un valore solo alle variabili specificate dalla clausola.

Oltre a `\forall`, sono presenti altri operatori particolarmente utili come `\old`, il valore “vecchio” di una variabile o attributo, e `\result`, il risultato dell'esecuzione di una certa operazione.

Un esempio più completo dell'uso di JML è quello presentato nella Figura 11.2. Esso descrive una classe che implementa un semplice borsellino elettronico.

11.5 Affidabilità e robustezza del codice 253

```
public class BorsellinoElettronico {  
    final int VALORE_MAXIMO;  
    int saldo;  
    // L'invariante specifica che il borsellino contiene  
    // IN UN QUALUNQUE MOMENTO una quantità di denaro  
    // maggiore di zero e minore del massimo  
    //@ invariant 0 <= saldo && saldo <= VALORE_MAXIMO;  
    byte[] pin;  
    // Il pin deve essere sempre di 4 caratteri e  
    // composto da cifre  
    /*@ invariant pin != null && pin.length == 4  
     * && (\forall int i; 0 <= i && i < 4;  
     * @ 0 <= pin[i] && pin[i] <= 9);  
     */  
    // Si può prelevare solo se ci sono fondi sufficienti  
    // Controlla l'aggiornamento finale del saldo  
    /*@ requires valore >= 0;  
     * @ assignable saldo;  
     * @ ensures saldo == \old(saldo) - valore  
     * @         && \result == saldo;  
     * @ signals (BorsellinoException) saldo == \old(saldo);  
     */  
  
    int pagamento(int valore) throws BorsellinoException  
    {  
        if (valore <= saldo)  
            { saldo -= valore; return saldo; }  
        else  
            { throw new BorsellinoException("scoperto di " + valore); }  
    }  
    ... altri metodi della classe ...  
}
```

Figura 11.2 Un esempio di uso di JML.

11.6 Riferimenti bibliografici

Molte considerazioni presenti in questo capitolo sono discusse in modo diffuso in libri e articoli di carattere generale.

L'esempio sul sistema di atterraggio dell'aeroplano discusso all'inizio del capitolo è tratto da Jackson [1995].

Il tema delle Domain Specific Software Architecture è discusso in Tracz [1995].

Il rapporto tra middleware e architetture software è stato studiato e discusso da Di Nitto e Rosemblum [1999]. I problemi legati al mismatch architetturale sono discussi in Garlan e Allen [1995].

JML dispone di un sito con materiale, articoli e tutorial (Leavens).

Appendice A

Questa appendice propone un caso di studio che ha lo scopo di illustrare ed esemplificare quanto trattato nei precedenti capitoli. Per motivi di spazio, l'esposizione sarà focalizzata solo su alcuni aspetti e in particolare sulla descrizione del problema e su quella della soluzione. Al termine, sono presentati anche porzioni di codice corrispondenti ad alcune classi identificate in fase di progetto e alcuni casi di test black box derivati dalla descrizione del problema.

Il tema del caso di studio riguarda lo sviluppo di un sistema di interrogazione di archivi dispersi sul territorio e contenenti dati relativi a beni culturali. Tale caso di studio è fondato sull'analisi delle reali problematiche del sistema museale italiano, costituito da una miriade di siti e sistemi informativi gestiti in modo indipendente da una varietà di soggetti istituzionali e privati. Obiettivo del progetto illustrato è quello di realizzare un'infrastruttura (la soluzione software) che permette di accedere alle diverse sorgenti informative in modo integrato come se fossero un unico archivio, senza necessariamente crearne uno centralizzato. In ultima analisi, si vuole permettere agli studiosi e a qualunque persona interessata al tema di fruire delle informazioni conservate dai diversi organismi ed enti operanti sul territorio come se fossero un'unica sorgente informativa.

Il caso di studio riprende parte del materiale e del codice che è stato realmente sviluppato per costruire un prototipo di sistema p2p in grado di rispondere ai requisiti descritti: si è cercato di mostrare sia il collegamento logico tra le diverse parti del progetto che la relazione con quanto illustrato nei diversi capitoli del testo.

Nella realizzazione del progetto sono state utilizzate alcune tecnologie e prodotti.

1. Tomcat, un web server basato su tecnologia Java per la gestione di servlet e pagine JSP.
2. RMI, per la comunicazione tra processi e thread.
3. Microsoft Access XP, per l'implementazione e la gestione dei singoli data base contenenti gli archivi con i dati relativi ai beni culturali.
4. JDBC, per gestire l'accesso da parte di metodi Java agli archivi Access.
5. SQL, per la formulazione delle query.
6. JBuilder 4.0, per lo sviluppo applicativo vero e proprio. JBuilder è un ambiente di Borland per lo sviluppo di applicazioni Java.

256 Appendice A

7. Microsoft Visio Professional, per la creazione dei diagrammi UML. Visio è un'applicazione che consente di realizzare vari tipi di disegni. Per facilitare la realizzazione di diagrammi complessi, è personalizzabile tramite il meccanismo degli *stencil*, librerie di oggetti grafici predefiniti e relative regole di composizione e uso. In particolare sono disponibili stencil per realizzare diagrammi secondo la sintassi di UML 2.0.

L'appendice è strutturata in 5 sezioni.

- La Sezione 1 introduce il caso di studio tramite una descrizione informale del problema.
- La Sezione 2 contiene la descrizione del problema in termini formali: domini, requisiti e interfaccia.
- La Sezione 3 contiene la descrizione della soluzione pensata in base al risultato della fase di descrizione del problema.
- La Sezione 4 mostra il codice relativo ad alcune delle classi realizzate.
- La Sezione 5 mostra alcuni aspetti relativi al test del sistema.

A.1 Il caso di studio

Il caso di studio riguarda la realizzazione di un *ambiente collaborativo* per la gestione e la fruizione delle informazioni sui beni culturali presenti sul territorio italiano (ad esempio, musei, collezioni private ecc.). Esso deve realizzare una rete collaborativa che integra le diverse sorgenti informative senza che sia necessario copiarle in un unico archivio. Ciò per rendere possibile a professionisti e utenti del settore una facile condivisione delle rispettive conoscenze.

I dati consultabili sono riferiti a beni culturali. Con *bene culturale* si intende un'opera o un oggetto di interesse artistico (ad esempio, un quadro, una statua, un palazzo ecc.). I dati che caratterizzano un bene culturale sono i seguenti.

- Dati identificativi dell'opera, quali titolo, autore, tipologia, periodo di riferimento, corrente di riferimento ecc.
- Dati concernenti la collocazione attuale quali città, museo, collezione ecc.

Gli utenti del sistema sono gli operatori del settore e in particolar modo gli studiosi. Fanno parte di questa categoria: professori universitari, tesisti, ricercatori nel campo dei beni culturali, archeologi, restauratori, storici. Tali utenti posseggono le conoscenze e la terminologia caratteristiche del settore artistico, mentre non è detto che siano capaci di eseguire consultazioni efficienti usando la rete.

Uno studioso deve poter fruire delle informazioni gestite dal sistema in accordo con le sue necessità. In particolare, può cercare informazioni di vario tipo: un preciso be-

ne (un quadro, un edificio o una scultura) oppure un insieme di beni aventi una qualche caratteristica in comune (quadri che abbiano lo stesso soggetto, sculture realizzate con la stessa tecnica o edifici realizzati nello stesso stile architettonico).

Nel caso in esame si è deciso di fornire allo studioso diverse possibilità di esplorazione dei contenuti culturali presenti sul sistema.

1. Deve essere possibile recuperare direttamente i dati di uno specifico bene (ricerca diretta). In tal caso, il sistema dovrà fornire il modo di localizzare e consultare i dati del bene. Di ogni bene si possono visualizzare i dati identificativi: il nome dell'autore, il titolo, il sito dove è conservato ecc.
2. Deve essere possibile ottenere informazioni su più beni aventi caratteristiche uguali, come ad esempio tutti i quadri del Caravaggio presenti nei siti del sistema (ricerca guidata).
3. Per facilitare ulteriormente la consultazione delle informazioni da parte degli studiosi, dev'essere possibile accedere a insiemi predefiniti di beni (ricerca per insiemi predefiniti). Tali insiemi sono costituiti da tutti i beni che sono caratterizzati da uno stesso insieme di parole chiave.

Si noti come in precedenza i termini e concetti relativi al dominio applicativo sono stati presentati utilizzando semplici verbi in modo indicativo: "vi sono diversi tipi di utenti", "un bene è ..." ecc. Per i requisiti, invece, si sono utilizzate frasi introdotte dal verbo "dev'è" proprio per indicare l'effetto che si vuole ottenere: "dev'essere possibile accedere a ..." ecc.

A.2 Descrizione del problema

Volendo procedere a una descrizione più precisa e formale del problema, il primo passo che deve essere affrontato è relativo all'analisi del dominio applicativo. Per poter definire chiaramente gli aspetti del problema di cui tener conto e/o sui quali intervenire in fase di design si è scelto di procedere per raffinamenti successivi, tracciando dapprima un context diagram, che viene in seguito arricchito di informazioni tramite l'uso di un problem diagram e di una serie di diagrammi UML.

A.2.1 Context diagram

Il context diagram realizzato è riportato nella Figura A.1. Il diagramma include tre domini.

- **Gestore beni culturali:** è il machine domain costituito dal sistema da sviluppare e che gestirà le informazioni sui beni culturali.
- **Informazioni relative ai beni:** sono le informazioni già esistenti e presenti sui siti dei possessori dei beni (ad esempio musei, gallerie d'arte ecc.). Esse costituiscono un designed domain in quanto la loro struttura viene determinata dal progettista in funzione delle informazioni raccolte sulle diverse tipologie di beni presenti.

258 Appendice A



a: visualizzazione delle informazioni
 b: gestione delle informazioni

Figura A.1 Context diagram.

- **Utenti:** è un dominio di tipo given, in quanto si suppone costituiscano una categoria di persone che preesistono alla nascita del sistema.
- **Fenomeni condivisi.**
 - **Utenti - Gestore beni culturali:** i fenomeni condivisi riguardano le informazioni che sono rese disponibili all'utente da parte del gestore.
 - **Gestore beni culturali - Informazioni:** i fenomeni condivisi riguardano la ricerca delle informazioni e la loro gestione.

A.2.2 Problem diagram

Il problem diagram raffina quanto visto nel context diagram, introducendo una descrizione più precisa dei fenomeni di interfaccia tra i vari domini. In questo diagramma vengono anche aggiunte le informazioni relative ai requisiti.

Nella Figura A.2 viene mostrato il problem diagram per la situazione considerata. Esso rappresenta una particolare istanza di uno dei problem frame presentati nel Capito-

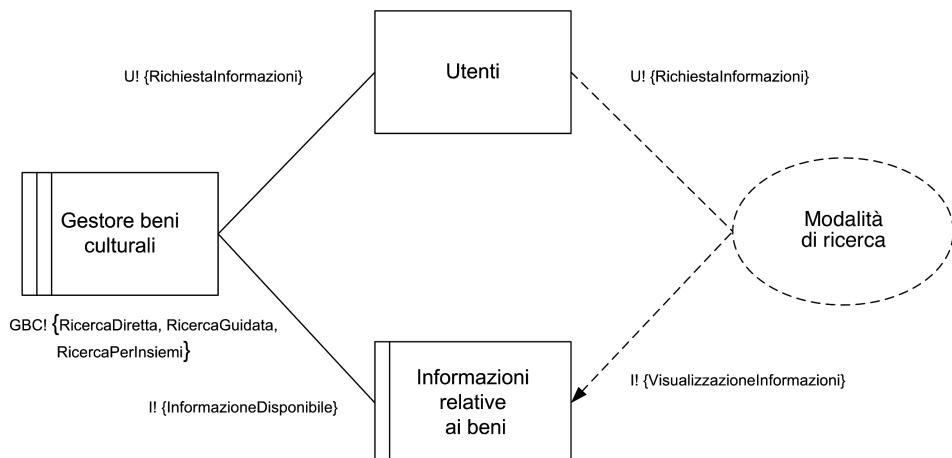


Figura A.2 Problem diagram.

lo 5. In particolare, si tratta di un problem frame di tipo simple workpiece: è possibile per un operatore accedere a una serie di informazioni strutturate. In realtà, nel caso del simple workpiece l'operatore può anche modificare e creare informazioni. Nel caso di studio, le operazioni si limitano a quelle di consultazione (peraltro necessarie anche a una qualunque operazione di modifica).

L'utente vuole poter emettere delle richieste di informazioni che rendono possibile la visualizzazione dei dati sui beni ricercati. Tale visualizzazione vincola il dominio Informazioni relative ai beni in quanto richiede che tali informazioni siano presentate così come se le aspetta l'utente. Il dominio Informazioni relative ai beni offre dei dati al gestore che deve essere sviluppato. A sua volta questo, in base ai comandi emessi dall'utente, invoca tre tipi di operazioni sul dominio.

A.2.3 La descrizione del problema in UML

Per fornire una descrizione ancora più dettagliata del dominio applicativo che contraddistingue il problema in esame, a partire dalla descrizione informale e dai diagrammi già visti è possibile derivare un class diagram che illustra in modo dettagliato l'insieme delle entità significative (si veda la Figura A.3). Le classi di cui si compone il diagramma sono le seguenti.

- La classe Utente descrive gli utenti del sistema che sono costituiti dagli operatori del settore che si occupano dei beni culturali o vi accedono. Tale classe viene specializzata in ulteriori sottoclassi, per indicare le diverse tipologie di utenti presenti: professori universitari, tesisti, ricercatori nel campo dei beni culturali, archeologi, restauratori, storici.
- La classe Bene descrive i beni culturali che costituiscono il cuore del sistema. Ogni bene ha i propri dati identificativi: titolo e autore dell'opera, descrizione testuale e, infine, una serie di parole chiave che lo caratterizzano. Le parole chiave sono usate nella creazione di insiemi predefiniti di beni e, in particolare, nelle operazioni di ricerca di più beni aventi la stessa caratteristica. Ogni bene deve essere caratterizzato da almeno una parola chiave.
- La classe Descrizione identifica un testo di commento al bene considerato. Ogni bene può avere più descrizioni che lo caratterizzano.
- La classe InsiemePredefinito definisce un insieme di chiavi e, di conseguenza, tutti i beni che sono caratterizzati da *tutte* le chiavi specificate. Un insieme predefinito ha un nome e l'insieme delle chiavi che lo identificano (ad esempio: "pittura del '700" è il nome di un insieme di quadri realizzati nel 1700 che ha come chiavi "pittura" e "1700").
- La classe Sito descrive i siti che fanno parte del sistema di condivisione delle informazioni sui beni culturali. Un sito viene descritto tramite un codice identificativo e l'indirizzo del sito stesso. Ad esempio, un sito potrebbe essere quello degli Uffizi

260 Appendice A

il cui indirizzo è www.uffizi.it. Le associazioni appartieneA e contiene descrivono la relazione tra i beni e i siti. Ogni sito deve contenere informazioni su almeno un bene.

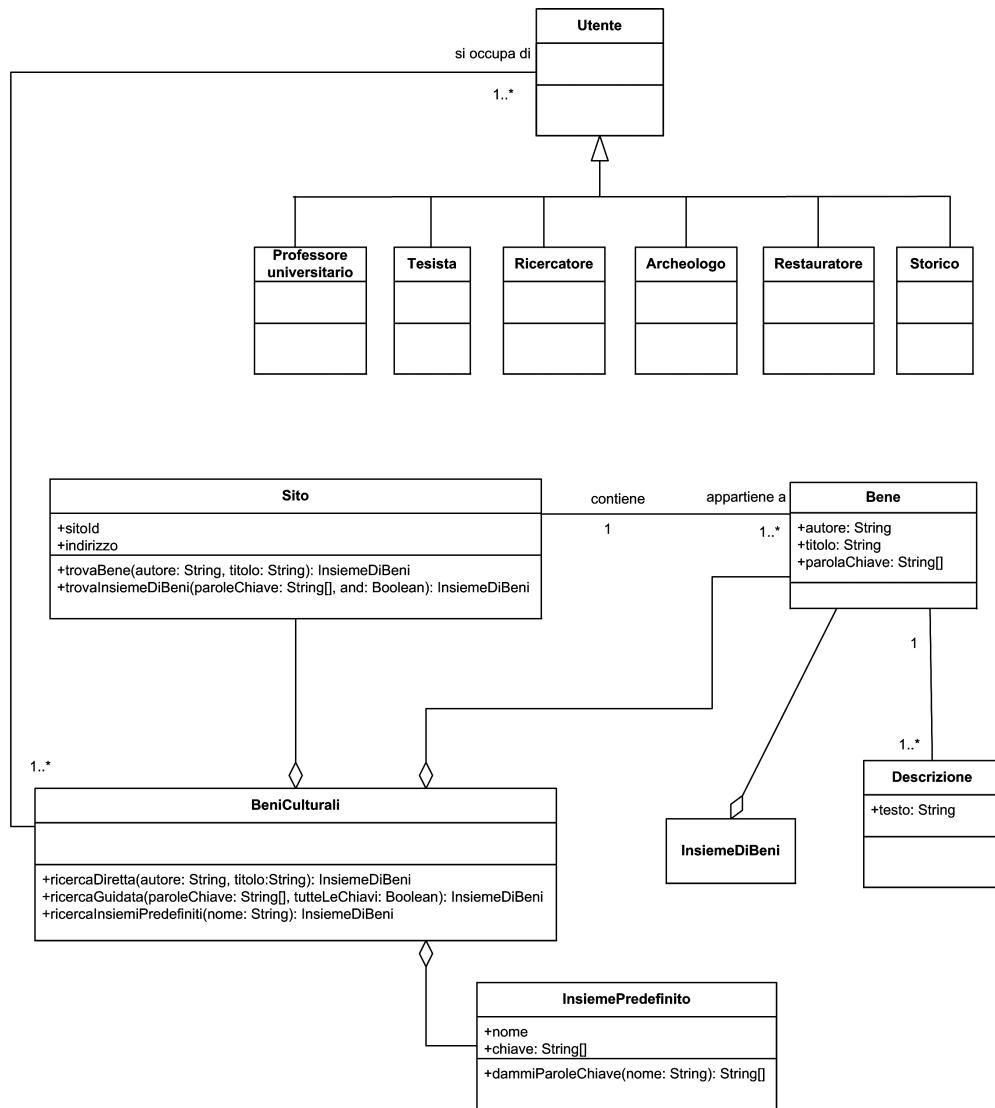


Figura A.3 Class diagram di descrizione del dominio applicativo.

- La classe `BeniCulturali` rappresenta l'insieme delle informazioni che sono messe a disposizione degli utenti. I beni culturali sono definiti da un insieme di beni, un insieme di siti e una serie di insiemi predefiniti di beni, predisposti per facilitare operazioni di consultazione rapida.
- La classe `InsiemeDiBeni` rappresenta il gruppo di beni che di volta in volta costituirà il risultato della ricerca.

La descrizione del dominio applicativo deve essere arricchita tramite altri diagrammi. In particolare, il class diagram di per sé rappresenta una visione statica del dominio ed è necessario descrivere anche le informazioni che lo caratterizzano da un punto di vista dinamico.

A.2.4 La descrizione dei requisiti

Dalla descrizione informale del problema si evince che i requisiti funzionali del sistema sono costituiti dalle operazioni di ricerca che devono essere messe a disposizione degli utenti. Accanto a questi requisiti se ne possono considerare anche altri di tipo non funzionale come, per esempio, quelli relativi all'affidabilità. Per fornirne una rappresentazione generale è opportuno utilizzare una matrice dei requisiti. Una descrizione più dettagliata (limitatamente ai requisiti funzionali) può essere ottenuta tramite diagrammi UML, così come illustrato nel seguito.

A.2.4.1 La matrice dei requisiti

I requisiti funzionali e di affidabilità che caratterizzano il problema possono essere illustrati a livello macro attraverso la matrice rappresentata nella Tabella A.1. Tale descrizione di alto livello risulta molto utile poiché da un lato permette di fornire una panoramica completa e facilmente leggibile dei requisiti del sistema e dall'altro può essere interpretata come una check list delle funzionalità che devono essere implementate nel sistema.

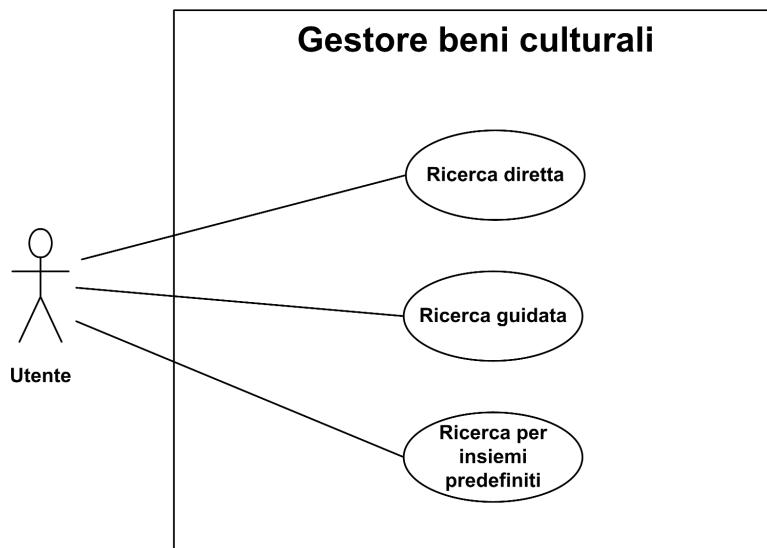
Affinché sia possibile in fase di progettazione comprendere in modo esauriente come i requisiti dovranno esser implementati, è necessario descriverli in modo più approfondito rispetto a quanto fatto nella matrice. A tale scopo, facendo ad esempio riferimento solo ai requisiti di tipo funzionale, si possono utilizzare una serie di diagrammi UML.

A.2.4.2 I requisiti funzionali in dettaglio

Un primo passo per raffinare i requisiti funzionali in modo più dettagliato rispetto a quanto fatto nella matrice è costituito dall'identificazione delle tipologie di utenti cui si devono rivolgere le funzionalità del sistema. Nel caso di studio, le funzionalità di ricerca diretta, ricerca guidata e ricerca per insiemi predefiniti devono essere messe a disposizione di ogni utente. In questo caso, quindi, la descrizione di questa relazione non è particolarmente complessa. In generale, per descrivere quest'informazione può essere impiegato uno use case diagram così come illustrato nella Figura A.4.

262 Appendice A

nome requisito	id requisito	tipo	priorità	criticità	rischio	data	descrizione	fonte
ricerca diretta	1	funzionale	2	alta	basso	n.d.	deve consentire di ottenere informazioni su un preciso bene di cui si conosce sia il titolo che l'autore	descrizione informale del problema
ricerca guidata	2	funzionale	3	media	basso	n.d.	deve consentire di ottenere informazioni su più beni aventi caratteristiche uguali	descrizione informale del problema
ricerca per insiemi	3	funzionale	4	bassa	basso	n.d.	deve consentire di visualizzare un insieme di beni predefinito	descrizione informale del problema
affidabilità della ricerca	4	tolleranza ai guasti	1	alta	alto	n.d.	per evitare ripercussioni sui risultati, durante la ricerca devono essere segnalate eventuali indisponibilità dei componenti del sistema	intervista presso committente

Tabella A.1 La matrice dei requisiti.**Figura A.4** Use case diagram relativo ai requisiti funzionali.

A.2.4.2.1 Ricerca diretta

L'utente utilizza quest'operazione nel caso in cui desideri avere informazioni relative a un bene di cui conosce il titolo e/o l'autore.

Precondizioni

L'utente ha indicato almeno autore e/o titolo, non ha selezionato un insieme predefinito e non ha indicato parole chiave.

Descrizione dell'operazione

L'operazione ricercaDiretta riceve in ingresso dall'utente il titolo e l'autore dell'opera da cercare e restituisce sia le informazioni sul bene sia l'indirizzo del sito che le contiene. L'utente può anche riservarsi di inserire solo il titolo dell'opera, oppure solo il nome dell'autore o nessuno dei due valori. Nel primo caso ottiene le informazioni su tutte le opere che hanno quel nome, nel secondo su tutte le opere dell'autore indicato, nel terzo su tutte le opere presenti nel sistema.

Postcondizioni

Sono visualizzate all'utente le informazioni relative ai beni che soddisfano alla richiesta, organizzati in base al sito ove sono conservati.

La Figura A.5 descrive l'operazione di ricercaDiretta tramite sequence diagram.

A.2.4.2.2 Ricerca guidata

L'utente utilizza quest'operazione nel caso in cui desideri ottenere informazioni su più beni aventi uno stesso insieme di caratteristiche presenti nei diversi siti del sistema, come ad esempio tutti i dipinti classificati con le parole chiave "affresco" e "tema religioso".

Precondizioni

L'utente non ha selezionato un insieme predefinito e ha indicato almeno una parola chiave.

Descrizione dell'operazione

L'operazione ricercaGuidata riceve in ingresso una o più parole chiave e restituisce un insieme di dati identificativi e un sito su cui si trovano le informazioni dei beni aventi le caratteristiche desiderate. Vi sono due possibili interpretazioni di quest'operazione relative ai vincoli sulle chiavi. La prima pone come vincolo che tutte le chiavi inserite dall'utente siano associate a ciascuna delle opere restituite come risultato; la seconda interpretazione pone il vincolo meno stringente che almeno una delle parole chiavi inserite dall'utente sia associata a ciascuna delle opere restituite come risultato. L'utente dovrà scegliere attraverso l'interfaccia quale interpretazione deve essere utilizzata.

Postcondizioni

Sono visualizzate all'utente le informazioni relative ai beni che soddisfano alla richiesta, organizzati in base al sito ove sono conservati.

Il sequence diagram relativo all'operazione ricercaGuidata è descritto nella Figura A.6.

264 Appendice A

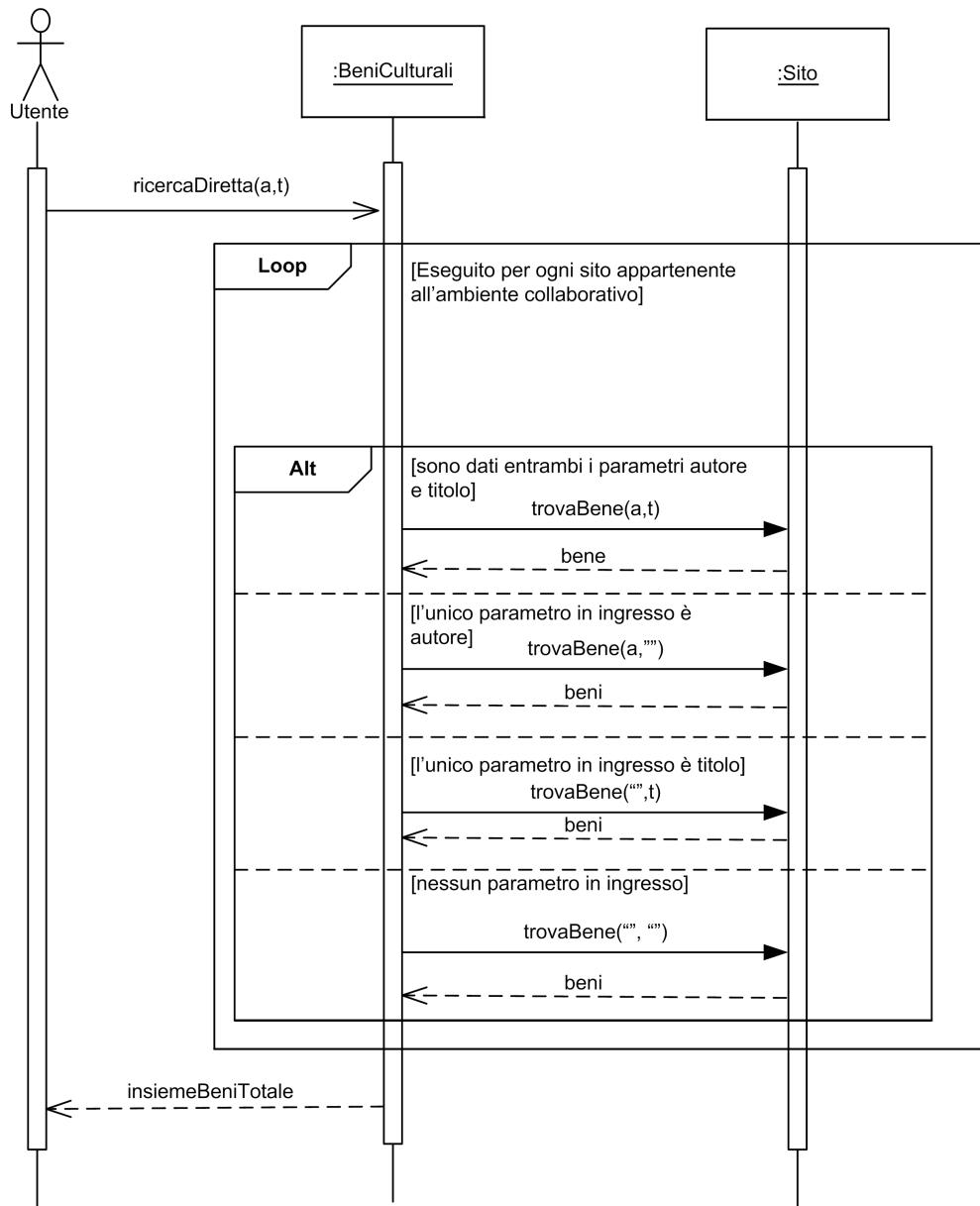


Figura A.5 Sequence diagram della ricerca diretta.

A.2.4.2.3 Ricerca per insiemi predefiniti

È l'operazione tramite la quale l'utente può visualizzare un insieme di beni predefinito come, ad esempio, tutte le opere di pittura rinascimentale.

Precondizioni

L'utente ha selezionato un insieme predefinito.

Descrizione dell'operazione

Ricevuto in ingresso il nome che identifica l'insieme desiderato, restituisce tutte le informazioni sulle opere che fanno parte dell'insieme e la loro collocazione sui siti.

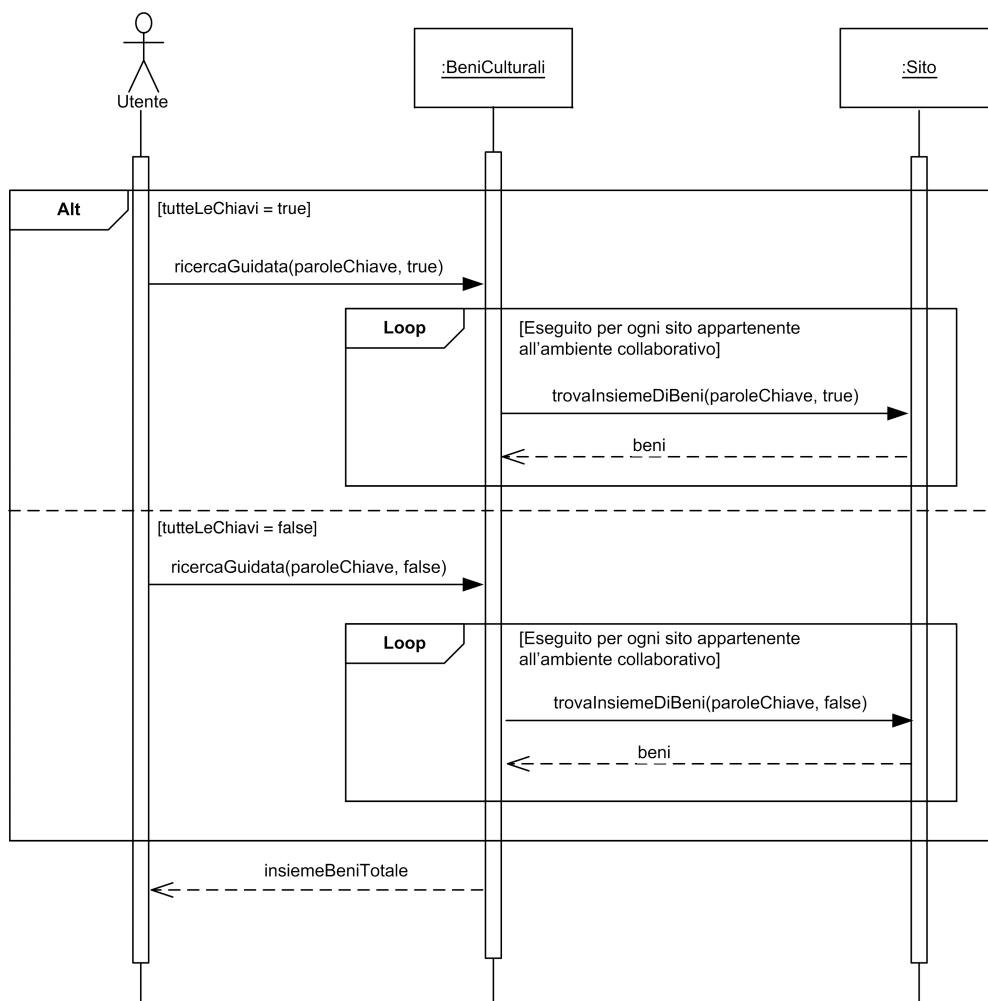


Figura A.6 Sequence diagram della ricerca guidata.

266 Appendice A

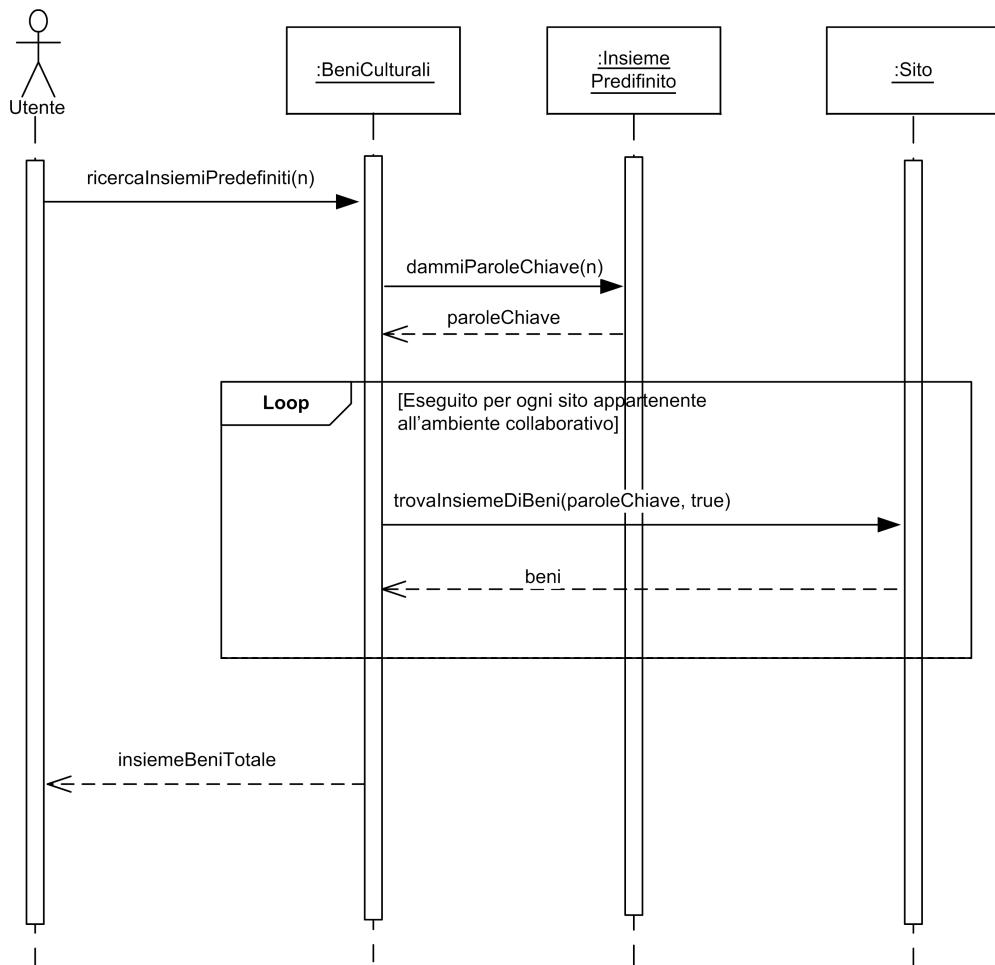


Figura A.7 Sequence diagram della ricerca per insiemi predefiniti.

Postcondizioni

Sono visualizzate all’utente le informazioni relative ai beni che soddisfano alla richiesta, organizzate in base al sito ove sono conservati.

Nella Figura A.7 viene mostrato il sequence diagram relativo all’operazione ricercaInsiemiPredefiniti.



Figura A.8 Esempio di query su insieme predefinito.

A.2.5 La specifica dell'interfaccia utente

L'analisi del dominio applicativo e dei requisiti permette di definire l'interfaccia applicativa che è illustrata nella Figura A.8. Si noti che gli insiemi predefiniti sono già visualizzati come radio button: la loro selezione fa sì che venga ignorata qualsiasi altra informazione inserita negli altri tre campi.

Un esempio di risultato della ricerca è quello mostrato nella Figura A.9. Si noti che l'utente riceve l'insieme delle informazioni richieste che includono la descrizione dei beni e anche la loro localizzazione (il sito ove sono conservati).

268 Appendice A

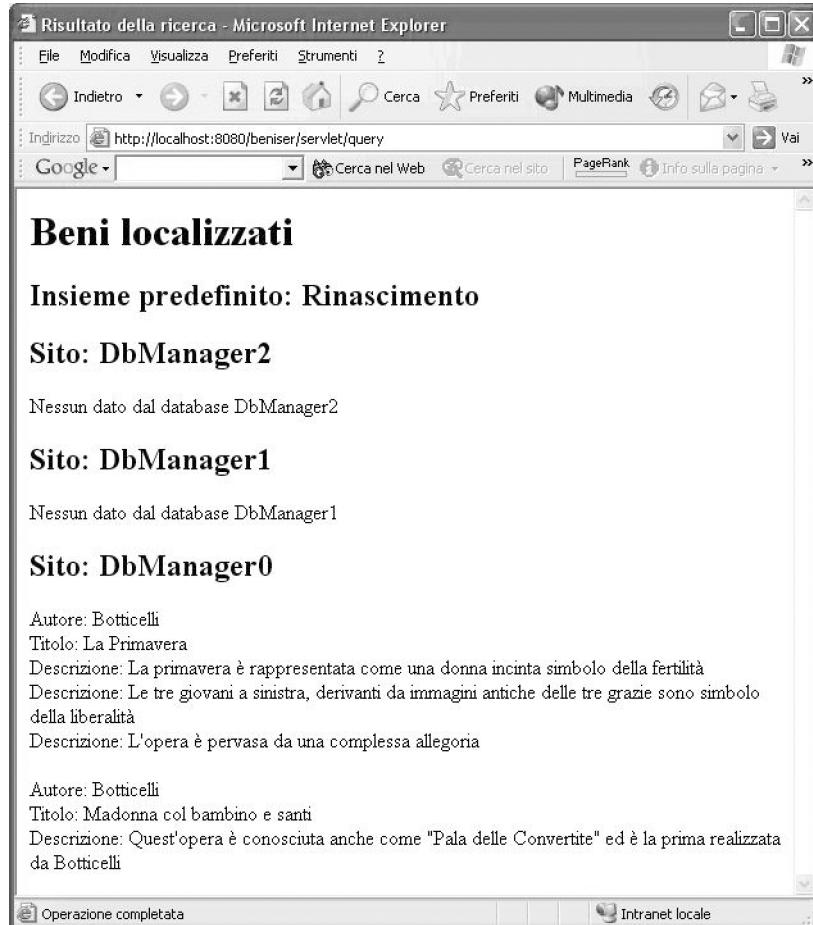


Figura A.9 Finestra con i risultati della query della Figura A.8.

A.3 Progetto della soluzione

Il lavoro svolto fin qui si è concentrato sulla descrizione e sull'analisi del problema. Partendo dalle informazioni ottenute durante lo studio del problema, l'ingegnere deve concepire una soluzione architettonale adeguata per rispondere alle esigenze che sono state evidenziate. Nel caso considerato, in base ai requisiti e alla descrizione di dominio offerta, la soluzione architettonale ideata si basa su un'architettura ibrida client-server e peer-to-peer (p2p). Il modello p2p viene utilizzato per federare i diversi siti presenti nel sistema. Ogni utente interagisce in modalità client-server con uno dei siti presenti. Sarà

quest'ultimo che si rivolgerà direttamente agli altri siti per reperire le informazioni che non sono in suo possesso.

Nel seguito sono presentati una serie di diagrammi UML che illustrano le diverse view della soluzione identificata. Il sistema distribuito proposto utilizza http e RMI per le proprie comunicazioni. Nei diagrammi, le chiamate relative all'utilizzo di RMI non sono dettagliate (per esempio, non viene rappresentato RMI Registry e le operazioni che lo riguardano). Questa scelta è stata fatta per semplificare la presentazione delle diverse viste e per focalizzare l'attenzione del lettore sugli aspetti più significativi dell'architettura proposta.

A.3.1 Functional view

Il primo aspetto che deve essere considerato dall'ingegnere del software è la descrizione logico-funzionale della soluzione ideata. Inizialmente, quindi, sono identificati gli elementi che compongono la soluzione, descritti nella Figura A.10. Il diagramma mostra anche la molteplicità dei vari elementi. In particolare, la soluzione sarà costituita da alcuni Client che si indirizzeranno ai siti presenti per ottenere le informazioni. Ogni sito prevede due elementi diversi: il Sito Web vero e proprio, che gestisce l'interazione con il client, e la parte, chiamata DataManager, di gestione dei dati contenuti su quello speci-

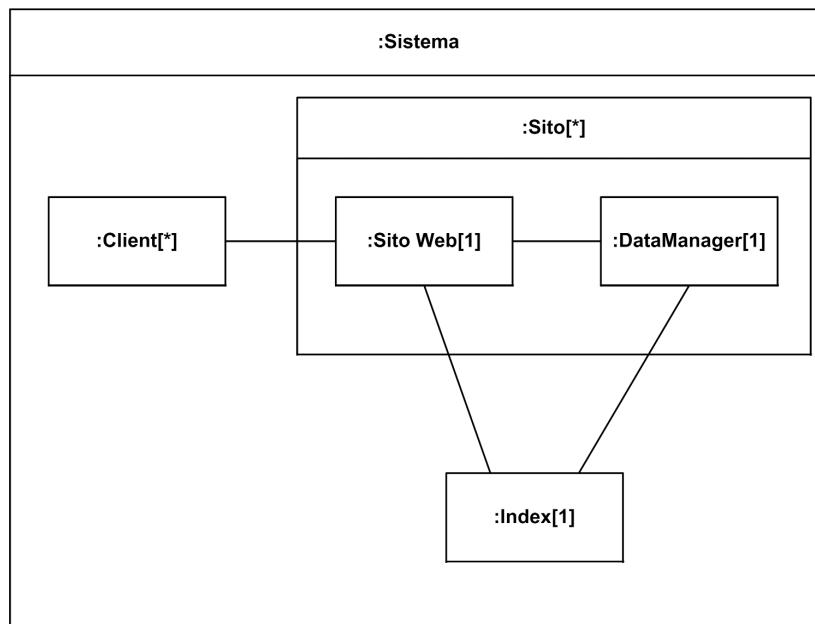


Figura A.10 Composite structure diagram.

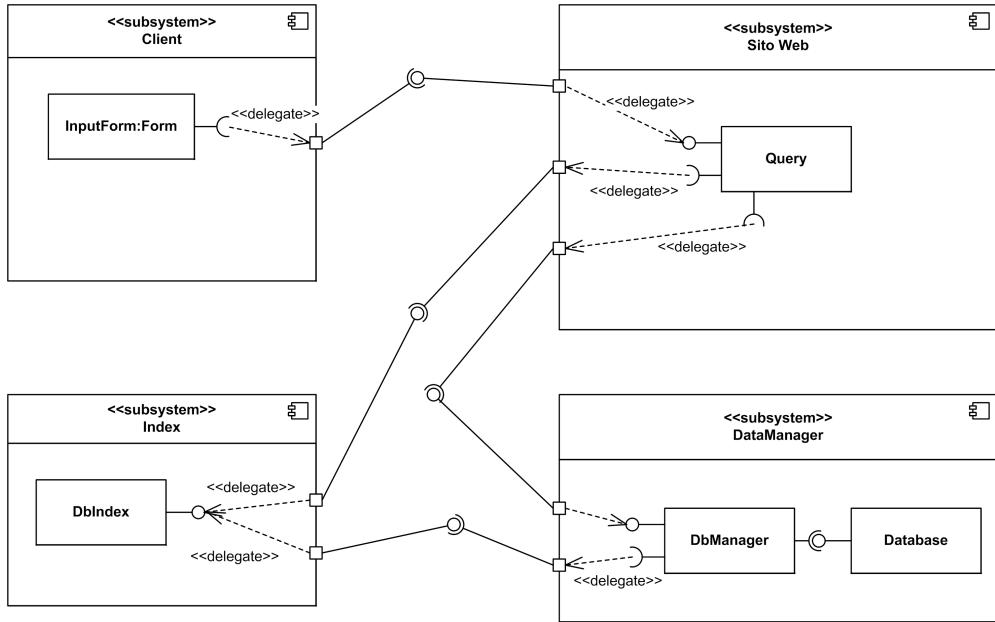


Figura A.11 Vista logico-funzionale.

fico sito. I due componenti del Sito sono in relazione con un componente Index, che è presente in un'unica istanza per tutto il sistema. Tale componente gestisce l'interazione p2p tra i diversi siti presenti nel sistema. La scelta fatta è quindi di avere un sistema p2p ibrido che consenta una localizzazione immediata e certa di tutti i beni dell'ambiente collaborativo.

Per poter comprendere il dettaglio della struttura e del funzionamento delle diverse parti identificate nel composite structure diagram della Figura A.10, è necessario descriverle meglio, specificando anche il tipo di interazione che le lega. A questo scopo è stato utilizzato il component diagram della Figura A.11.

Ogni utente è dotato di un'interfaccia che permette di formulare una query e di inviarla a uno dei siti appartenenti al sistema. Ogni sito è dotato di un server che ha due parti: il gestore dell'interazione con l'utente (che come vedremo è stato realizzato tramite un sito web e una servlet) e il gestore vero e proprio dei dati. Quindi l'interazione tra utente e sistema si articola su tre livelli: client, sito web e gestore dei dati.

I diversi siti che gestiscono i dati relativi ai beni culturali possono cooperare tra loro. In un'architettura p2p ibrida, vi è un indice che ha lo scopo di identificare gli altri server presenti (secondo uno schema alla Napster). Nel caso considerato, l'indice è realizzato dal componente `Index` presso il quale si registrano, al momento della loro attivazione, tutti i gestori dei dati (`DataManager`). In questo modo, `Index` può costruire l'in-

A.3 Progetto della soluzione 271

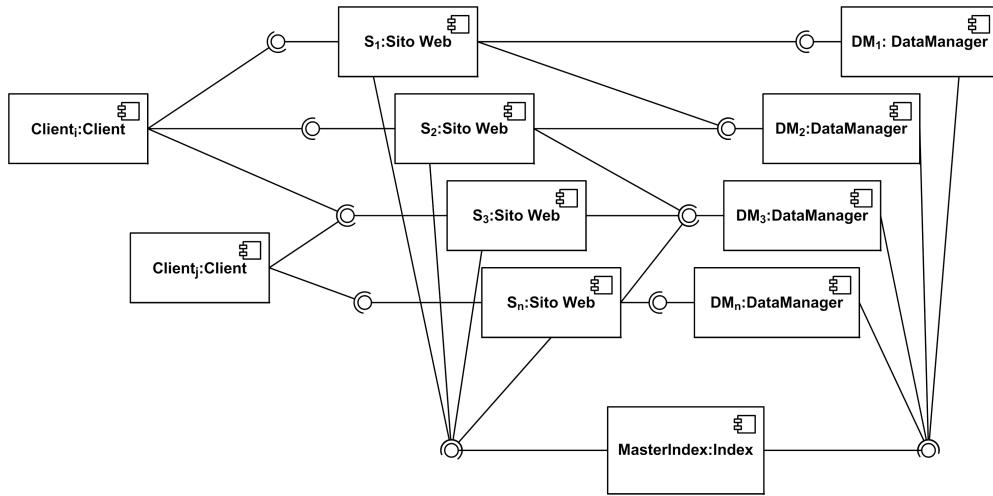


Figura A.12 Diagramma di istanze della vista logico-funzionale.

dice di tutti i **DataManager** in grado di rispondere a interrogazioni relative ai beni culturali. L'indice viene interrogato dal **Sito Web** di uno specifico sito per localizzare gli altri siti presenti nel sistema e, in particolare, i rispettivi **DataManager**.

La Figura A.12 mostra un diagramma delle istanze relativo alla vista funzionale: nel sistema è prevista la presenza di più client e siti che collaborano tra loro. In particolare, nella figura non sono state introdotte tutte le relazioni possibili tra client, siti web e gestori di dati: ciò è unicamente dovuto all'esigenza di garantire una sufficiente leggibilità del diagramma.

In realtà, dallo studio del diagramma potrebbe sorgere il dubbio che un client per effettuare una singola interrogazione deve necessariamente interagire direttamente con più siti. Il diagramma delle istanze vuole solo rappresentare il fatto che un client, di volta in volta, può interagire con siti diversi. Ma ogni singola interazione avviene con un unico sito: è questo che si preoccupa di interrogare gli altri siti. Questa modalità di funzionamento verrà esplicitata andando a esaminare il comportamento dettagliato dei diversi componenti, così come presentato nella module view.

A.3.2 Module view

La functional view visualizza in modo semplice ed efficace l'architettura complessiva della soluzione prescelta e i principali componenti necessari alla sua realizzazione. La module view offre una descrizione di maggior dettaglio dell'architettura scelta e, in particolare, illustra quella che sarà la struttura vera e propria del codice sorgente e il suo comportamento dinamico.

272 Appendice A

Nel realizzare i diagrammi che costituiscono la module view, è necessario considerare costantemente tutte le informazioni fin qui disponibili.

- Descrizione funzionale della soluzione (*functional view*): fornisce la linea guida della struttura di dettaglio che avrà la soluzione. Infatti, nella realizzazione dei diagrammi della module view si partirà dall'analisi dei componenti individuati nella functional view.
- Descrizione formale del problema: fornisce le informazioni e i dati che rendono la soluzione specifica per il problema considerato. Ad esempio, in questo caso sono i dati relativi ai beni e la natura e struttura delle operazioni che dovranno essere implementate.

A titolo di esempio, nel seguito sono riportati i class diagram dei due componenti ritenuti più interessanti. In particolare, la Figura A.13 descrive il componente DataManager, mentre la Figura A.14 descrive la classe Query del componente Sito Web.

Classe Bene

La classe `Bene` definisce le caratteristiche che deve avere un bene culturale e le operazioni che possono essere eseguite su di esso. Un bene è descritto oltre che dagli attributi `Autore` e `Titolo`, anche dall'attributo `Descrizioni`, un vector Java che ha come elementi le descrizioni relative al bene considerato. Le operazioni eseguibili sul bene sono per la maggior parte di tipo `get`, restituiscono cioè i dati che descrivono il bene.

Classe SetBeni

Nella classe `SetBeni` viene definita la struttura `InsiemeBeni`: si tratta di un vector i cui elementi sono istanze della classe `Bene`. Le operazioni definite servono per gestire gli elementi di `InsiemeBeni`, in particolare per inserire un bene e per leggere le informazioni che descrivono il bene presente in una specifica posizione del vector.

Classe DbManager

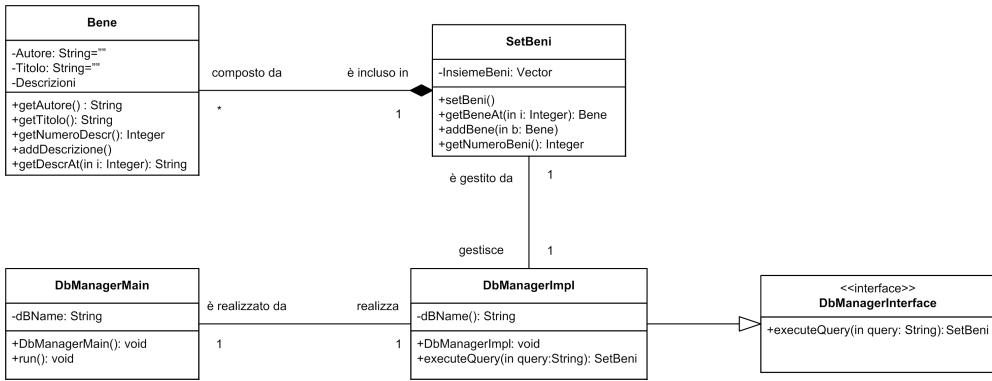
La classe `DbManagerImpl` descrive i gestori dei singoli database. Un `DbManagerImpl` è legato a uno specifico database e ne gestisce le interrogazioni dopo essersi registrato presso l'`Index`.

Classe Query

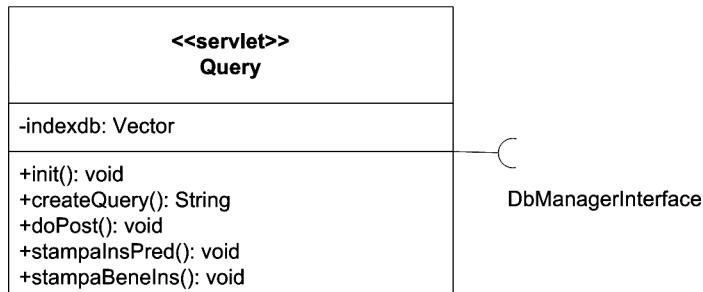
La classe `Query` realizza una servlet che viene invocata da Tomcat a fronte di un opportuno comando inserito nella form gestita dal browser sulla macchina client. Il metodo principale della servlet è `doPost` che analizza la richiesta dell'utente e la invia ai `DataManager`. Inoltre, dopo aver ricevuto come risultato della ricerca un oggetto `SetBeni`, provvede a comporre la pagina HTML che illustra i risultati della ricerca.

Nel descrivere le classi è stata fatta una serie di commenti circa il loro comportamento dinamico. Tali osservazioni sono strumentali a sintetizzare il ruolo di ciascuna classe, ma

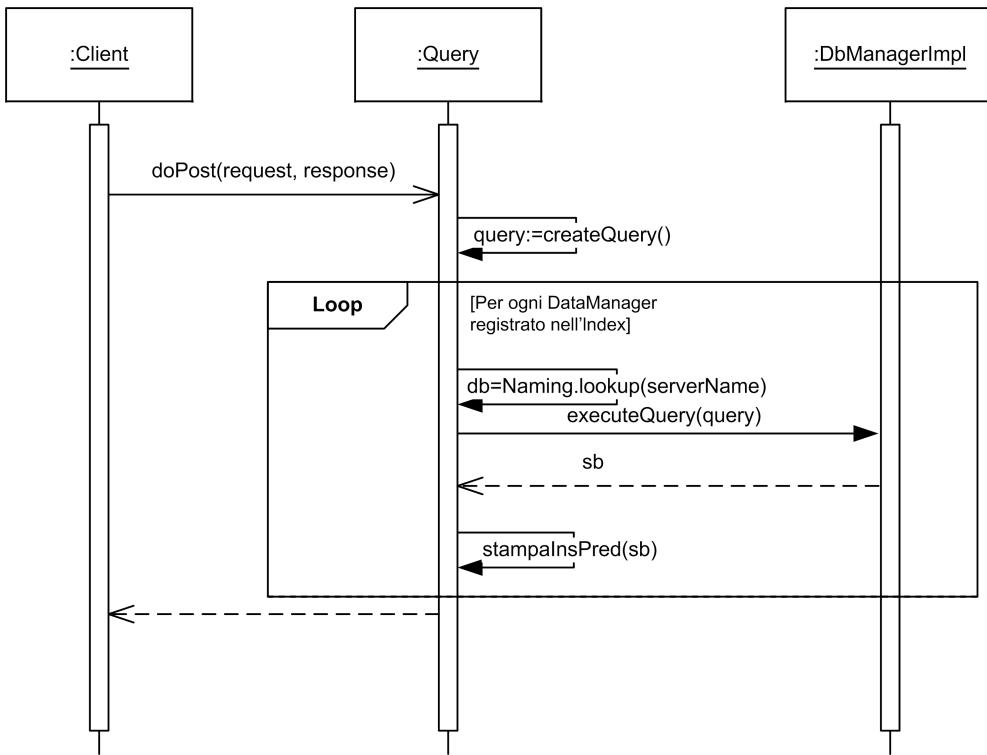
A.3 Progetto della soluzione 273

**Figura A.13** Module view del componente DbManager.

non sono certamente sufficienti per caratterizzarne appieno il comportamento. Per rappresentare in modo compiuto le caratteristiche dinamiche della soluzione è conveniente utilizzare una serie di sequence diagram. Ogni sequence diagram descrive un'operazione o una funzionalità che la soluzione deve fornire. I diagrammi sono realizzati partendo dai class diagram. In particolare, ogni oggetto che compare nel sequence è un'istanza di una classe del class diagram. Inoltre, ogni chiamata di metodo presente nel sequence corrisponde a un metodo definito nella classe corrispondente presente nel class diagram. La Figura A.15 descrive la sequenza di operazioni invocate per eseguire una query e restituire il risultato all'utente: è la servlet (e non il client) che si preoccupa di interagire con tutti i DataManager presenti nel sistema. In particolare, si noti che vengono invocate tante executeQuery quanti sono i DataManager registrati presso l'Index. Per illustrare questo aspetto è stato introdotto un frame Loop. L'elenco dei DataManager è recuperato dalla servlet all'atto della sua inizializzazione (metodo init) e per questo motivo quest'operazione non è illustrata nella figura.

**Figura A.14** Module view del componente Query.

274 Appendice A

**Figura A.15** Interrogazione dei siti.

Si può notare che gli oggetti `Query` e `DbManagerImpl` sono descritti nei class diagram precedenti. Il metodo `Naming.lookup()` è uno dei servizi di RMI e quindi non è presente nel class diagram, così come non lo è il `Client` perché esso è costituito solo da una form HTML.

Infine, la Figura A.16 mostra l'organizzazione delle classi in package. Il package `Beni` è usato sia dal package `Query` che contiene la servlet `Query`, sia dal package `DbManager` che contiene, tra le altre, la classe `DbManagerImpl`. La prima usa le classi del package `Beni` per comporre e visualizzare correttamente il risultato della ricerca; la seconda utilizza quello stesso package per gestire le interrogazioni sul database e comporre il risultato da restituire alla servlet.

A.3.3 Deployment view

Tramite la module view sono state descritte in dettaglio le classi che compongono la soluzione e che, una volta implementate, daranno origine al codice sorgente. Il passo successivo è individuare come saranno distribuiti i componenti definiti e le classi da cui sono formati. Nel caso di studio considerato, la deployment view è una parte molto

A.3 Progetto della soluzione 275

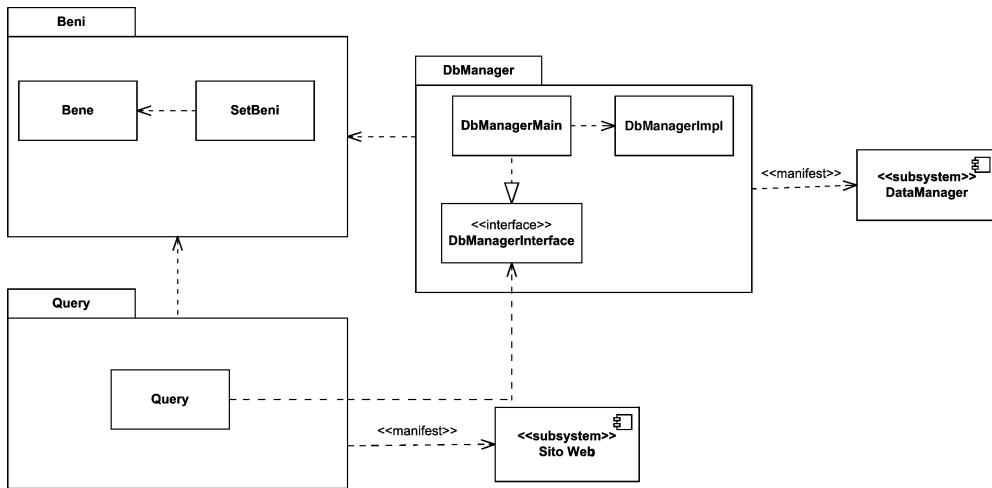


Figura A.16 Package diagram delle classi della soluzione.

importante della progettazione della soluzione a causa dello stile architettonale scelto. Si ricordi, infatti, che per le esigenze e le caratteristiche del problema è stata scelta un'architettura distribuita, e in particolare un p2p ibrido.

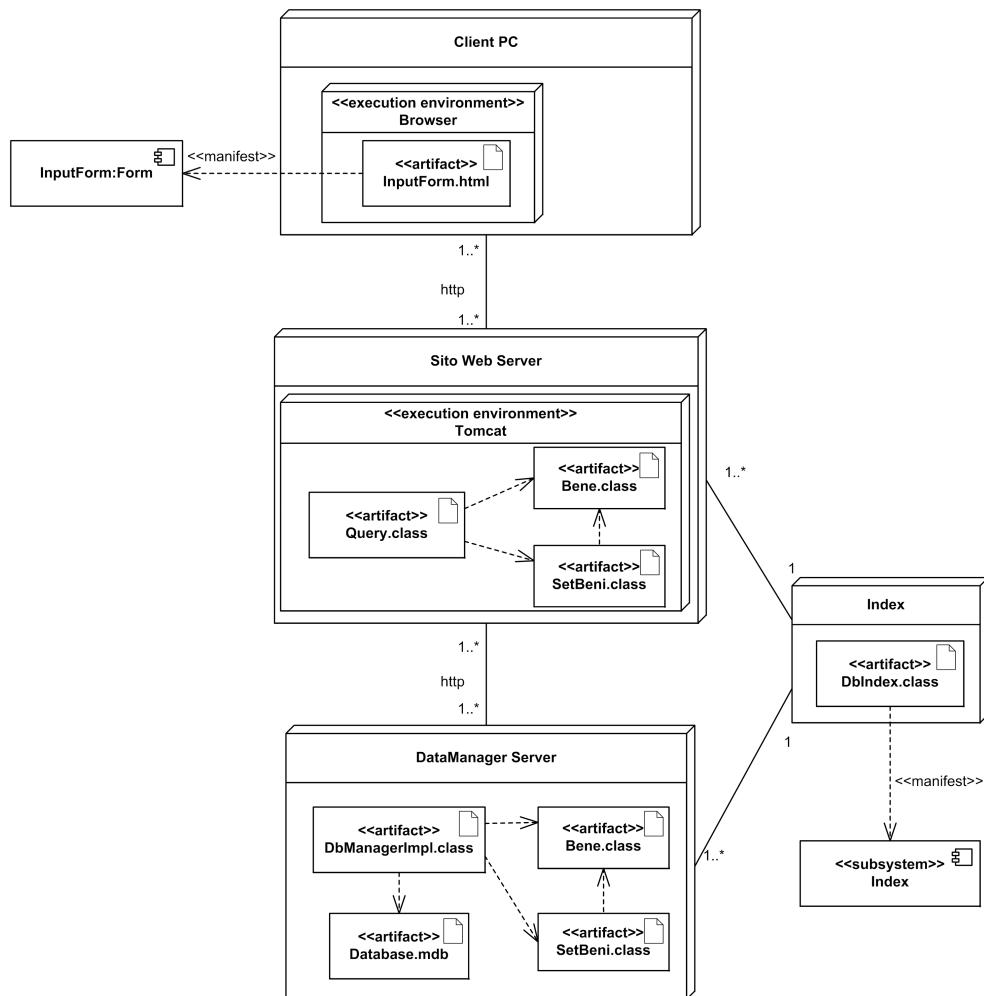
La Figura A.17 identifica i diversi livelli dell'architettura proposta, ipotizzando la presenza di più siti, ciascuno dotato del proprio **DataManager** e **Sito Web**. Si ipotizza, inoltre, che anche dal punto di vista fisico, si sia in presenza di tre livelli: una o più macchine su cui sarà installato il **Client**; una o più macchine con l'installazione del componente **Sito Web**; una o più macchine con la corrispondente installazione del componente **DataManager**. È presente, inoltre, la descrizione di una macchina con l'installazione di **Index**. La Figura A.17, in realtà, mostra una delle possibili distribuzioni del codice, vale a dire il caso in cui ogni componente è installato su macchine diverse. È possibile però, effettuare altre scelte di distribuzione del codice: ad esempio, si può scegliere di avere i due componenti del sito, **Sito Web** e **DataManager**, sulla stessa macchina, oppure di avere **Index** su una macchina non dedicata, ma utilizzata per gestire uno dei siti del sistema.

In particolare, gli artifact presentati nel deployment manifestano ciascuno una delle classi descritte nel class diagram e realizzate di conseguenza nel codice.

A.3.4 Execution view

La parte finale della progettazione della soluzione prevede l'analisi e la descrizione di ciò che avviene durante l'esecuzione del codice che costituisce la soluzione. I nomi dei processi e dei thread identificano i componenti a partire dai quali sono stati creati. In particolare, la Figura A.18 mostra cosa succede durante l'esecuzione della registrazione del **DataManager** presso l'indice e durante l'esecuzione della richiesta del **DataManager** presente da parte del **Sito Web** che deve eseguire un'interrogazione.

276 Appendice A

**Figura A.17** Deployment diagram.

La Figura A.19 mostra i processi attivati quando un Sito Web esegue un'interrogazione tra i siti dell'ambiente collaborativo, e nello specifico, tra i componenti DataManager dei siti. La prima colonna mostra la richiesta da parte del Client e l'interrogazione al DataManager, mentre la seconda colonna mostra il ritorno dei risultati forniti da un particolare DataManager: è possibile notare che il ritorno dei risultati prevede la composizione di una form HTML di output dove saranno descritti i Beni trovati dalla ricerca. Un esempio di tale form è già stato mostrato nella Figura A.9.

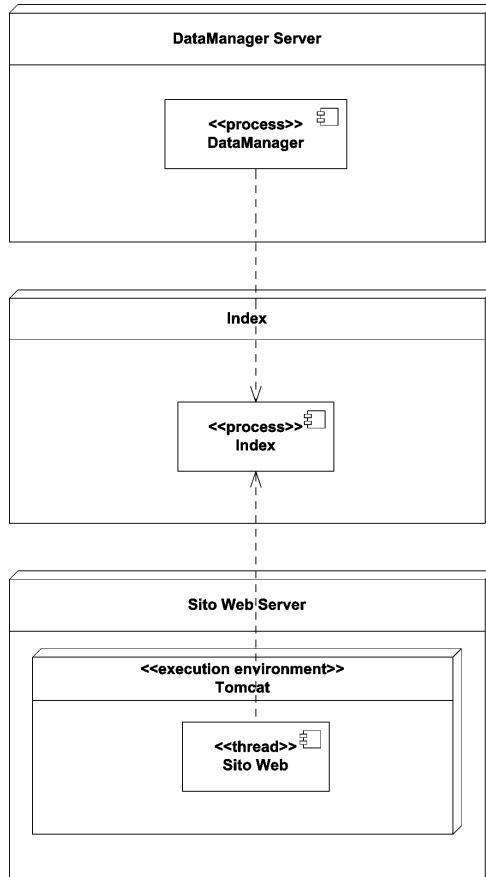


Figura A.18 Execution view delle interazioni con l'indice.

A.4 Il codice

Di seguito è mostrato il codice delle due classi su cui ci si è soffermati in precedenza: la classe `Query` e la classe `DbManagerImpl`. In questo modo è possibile notare la coerenza tra quanto progettato e il codice effettivamente sviluppato. In particolare, si può notare come siano presenti tutti gli attributi e i metodi definiti e descritti nelle classi `Query` e `DbManagerImpl` del class diagram della soluzione. L'ingegnere, nel caso considerato, ha creato le classi presenti nel diagramma della struttura della soluzione e ha implementato ogni singola classe inserendo gli attributi definiti e le intestazioni dei metodi identificati. Il codice dei metodi, vale a dire ciò che effettivamente fanno i singoli metodi, è stato implementato sulla base dei sequence diagram realizzati in fase di progettazione della soluzione.

278 Appendice A

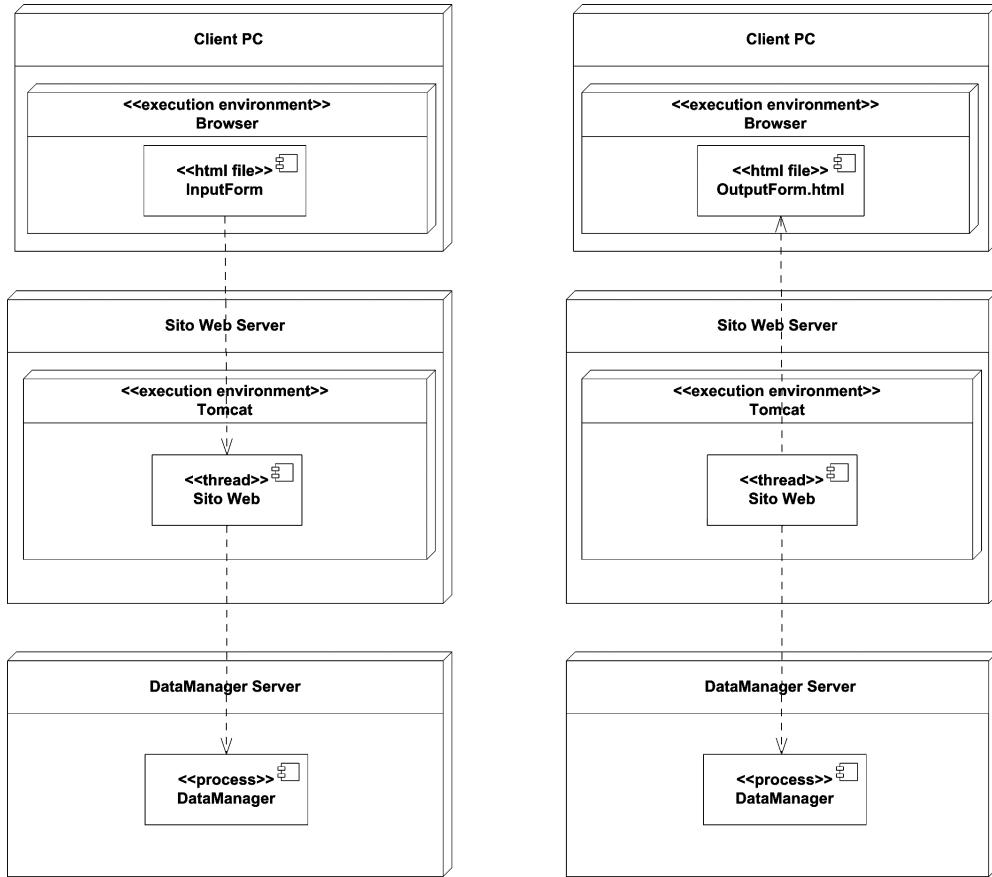


Figura A.19 Execution view di un'interrogazione a regime.

Nel codice riportato ci sono tutte le intestazioni dei metodi presenti nella classe `Query`, che corrispondono a quelli definiti nel diagramma. Per maggior leggibilità e comprensione, si è scelto di includere l'implementazione dei soli metodi presenti anche nel sequence diagram della Figura A.15, in modo da mostrare comunque i passaggi nella realizzazione del codice. Alcune istruzioni sono state rimosse per facilitare la lettura.

```

package Query;

// Clausole di import per accedere alle classi utilizzate nel seguito

import javax.servlet.*;
import javax.servlet.http.*;

```

```
...  
  
import Beni.*;  
import DbManager.DbManagerInterface;  
import IndexDb.DbIndexInterface;  
import java.util.Vector;  
  
// Query è una servlet  
public class Query extends HttpServlet {  
  
    //Definizione delle variabili  
    private static final String CONTENT_TYPE = "text/html";  
  
    private Vector indexDb;  
    private int cardIns = 0; //numero di parole chiave che caratterizza la ricerca  
    ...  
  
    // Metodo di inizializzazione della servlet  
    public void init(ServletConfig config) throws ServletException  
    {  
        ...  
    }  
  
    // Compone il testo della query a partire dai contenuti della form  
    // Ritorna come risultato la query da eseguire  
    private String createQuery()  
    {  
        String q; // Conterrà il testo della query  
        int i;  
        // L'if considera le varie opzioni di ricerca e compone la query secondo  
        // gli input che l'utente inserisce nella form di richiesta  
        if (c.equals("")) && insieme.equals("Nessuno"))  
            // Non ci sono chiavi con le quali fare la ricerca. Non è stato  
            // selezionato alcun insieme predefinito. La ricerca è diretta.  
        {  
            // Composizione della query per ricerca diretta  
            q = "SELECT DatiId.Autore, DatiId.Titolo, Descrizioni.Descrizione " +
```

280 Appendix A

```
    "FROM DatiId INNER JOIN Descrizioni ON" +
    " (DatiId.Titolo = Descrizioni.Titolo) " +
    "AND (DatiId.Autore = Descrizioni.Autore)";
}

else // Ci sono chiavi con le quali fare la ricerca
{
    // In questo ramo else (omesso per semplificare la lettura) è
    // assegnato un valore alla variabile cardIns pari al numero di parole
    // chiave che caratterizza la query
    ...
    /// end dell'else che compone la query secondo le parole chiave
q = q + " ORDER BY DatiId.Autore, DatiId.Titolo, Descrizioni.Descrizione";
return q;
} // end del metodo createQuery

// Il metodo doPost (la servlet vera e propria) interroga i DataManager
// presenti nell'indice e compone la pagina dei risultati.
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    ...
    //chiamata al metodo createQuery per comporre l'interrogazione
q = createQuery();

try {
    // Stampa dell'intestazione della pagina con i risultati
    out.println("<html>");
    out.println("<head><title>Risultato della ricerca</title></head>");
    out.println("<body>");
    out.println("<h1>Beni localizzati</h1>");
    if (!insieme.equals("Nessuno"))
        out.println("<h2>Insieme predefinito: "+insieme+"</h2>");

    // Identificazione del nome del computer su cui
    // gira il processo corrente
    serverName = java.net.InetAddress.getLocalHost().getHostName();

    ...
}
```

A.4 Il codice 281

```
// Ciclo su tutti i registrati nell'indice
for (k = 0; k <indexDb.size(); k++)
{
    //Nome del DataManager considerato
    nome = (String) indexDb.elementAt(k);
    try {
        db=(DbManagerInterface) Naming.lookup("//"+serverName+"/"+ nome);
        //Chiamata al metodo del DataManager che esegue l'interrogazione
        //sul database del sito considerato
        sb=db.executeQuery(q);
        // Stampa dei dati del sito sulla pagina dei risultati
        out.println("<h2>Sito: "+ nome+"</h2>");
        if (sb.getNumeroBeni()==0 && insieme.equals("Nessuno"))
        {
            out.println("<p>Nessun dato dal database "+nome+"</p>");
            continue;
        }
        // Stampa dei risultati
        if ( (!insieme.equals("Nessuno")) || r != null) {
            // Stampa un insieme predefinito eliminando i duplicati
            Stampainspred(sb); }
        else {
            //Stampa tutti beni ritornati come risultato, senza duplicati
            ...
        } // Fine if-then-else sull'insieme predefinito
    } // Fine try interno sul singolo database
    catch (Exception e)
    {
        e.printStackTrace();
        out.println("<p>Il database "+nome+" non è raggiungibile.</p>");
    } // Fine catch legato al try interno sul singolo database
} // Fine ciclo su DataManager
} // Fine del try esterno su tutti i DataManagers
catch (Exception e)
{
    e.printStackTrace();
} // Fine catch legato al try interno sul singolo database
```

282 Appendix A

```
out.println("</body></html>");

} // Fine dopost

// Metodo che, dato un insieme di beni, li stampa sulla pagina dei risultati
// eliminando le informazioni duplicate.

private void StampaInsPred(SetBeni s)
{
    int i,j,nd, cardCorr = 0;
    String dcurr, dprev=null; //Variabili relative alle descrizioni di
                             //un bene
    Bene curr;
    boolean stampato = false;
    //Ciclo di tutti i beni dell'insieme dato in ingresso
    for (i=0; i < s.getNumeroBeni(); i++)
    {
        curr = s.getBeneAt(i);
        a = curr.getAutore();
        t = curr.getTitolo();
        nd = curr.getNumeroDescr();
        if (nd >= cardIns)
        {
            //ciclo for per eliminare i duplicati
            for (j = 0; j < cardIns; j++)
            {
                dcurr = curr.getDescrAt(j);
                if (j == 0) cardCorr = 1;
                else
                {
                    if (dcurr.equals(dprev)) cardCorr++;
                }
                dprev = dcurr;
                if (cardCorr == cardIns)
                {
                    StampaBeneIns(curr,nd);
                    stampato = true;
                }
            }
        }
    }
}
```

```
        } // Fine stampa di un bene
    } // Fine ciclo for sull'insieme di beni
    if (!stampato) {
        out.println("<p>Nessun dato dal database "+nome+"</p>");}
} // Fine del metodo StampaInsPred

public void StampaBeneIns(Bene b, int nd)
{
    ...
}

...
}

// Fine della classe Query
```

La classe `DbManagerImpl` è la classe che esegue l'interrogazione composta da `Query` sul relativo database. Anche in questo caso sono state omesse tutte quelle righe di codice non strettamente connesse con il sequence diagram mostrato nella Figura A.15. Il metodo `executeQuery` riceve in ingresso la stringa `query` e restituisce come risultato l'insieme dei beni le cui caratteristiche soddisfano l'interrogazione, così come definito nel diagramma delle classi della module view. Per comprendere meglio il metodo `executeQuery`, occorre precisare che il risultato dell'interrogazione sul singolo database non è un insieme costituito da beni, ma una serie di elementi costituiti dai dati autore, titolo, descrizione. Il metodo, quindi, comprende una serie di istruzioni e cicli che hanno lo scopo di organizzare e raggruppare i dati in oggetti di tipo `Bene` e poi di inserire questi oggetti nell'oggetto di tipo `SetBeni`.

```
package DbManager;
// Clausole di import per accedere alle classi utilizzate nel seguito

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import Beni.*;
import java.util.Vector;
import IndexDb.*;

public class DbManagerImpl extends UnicastRemoteObject
    implements DbManagerInterface
```

284 Appendice A

```
{ //Definizione delle variabili
    private String dbName;

    //Metodo costruttore della classe
    public DbManagerImpl(String name) throws RemoteException {
        ...
    }

    // Metodo chiamato dalla classe Query, che esegue l'interrogazione passata
    // come parametro
    public SetBeni executeQuery(String query)
    {
        // Creazione dell'insieme di beni (all'inizio è vuoto)
        SetBeni sb = new SetBeni();
        Bene tempb, previous = null; //Variabili temporanee
        String autore, titolo, descrizione; //Dati del singolo bene
        ResultSet rs;
        try {
            //Connessione al database
            String driver ="sun.jdbc.odbc.JdbcOdbcDriver";
            Class.forName(driver);
            Connection con = DriverManager.getConnection("jdbc:odbc:"+dbName);
            Statement st = con.createStatement();
            //Esecuzione dell'interrogazione "query"
            rs = st.executeQuery(query);
            // Ciclo di scansione di tutti i beni ritornati come risultato
            // della query
            // Ad ogni iterazione sono considerati i dati di ogni elemento di rs.
            // Il risultato dell'interrogazione nono ritorna un insieme di beni,
            // ma una serie di elementi formati da: autore, titolo, descrizione.
            // Nel seguito è necessario creare gli oggetti bene e
            // aggiungerli al risultato.
            while(rs.next())
            {
                //Per ogni elemento di rs si ricavano i dati autore, titolo,
                //descrizione
                autore = rs.getString(1);
```

```
titolo = rs.getString(2);
descrizione = rs.getString(3);

if (previous == null)
{
    //Primo bene considerato.
    //Si crea un nuovo oggetto Bene con i dati in possesso
    tempb = new Bene(autore,titolo);
    tempb.addDescrizione(descrizione);
    previous = tempb;
}
else
{
    // Analisi degli altri beni
    if ( autore.equals(previous.getAutore())
        &&
        titolo.equals(previous.getTitolo()) )
    {
        // Le informazioni considerate riguardano lo stesso bene
        // di prima.
        // All'oggetto bene precedentemente creato si aggiunge la sola
        // descrizione considerata.
        previous.addDescrizione(descrizione);
    }
    else
    {
        // Le informazioni considerate riguardano un nuovo bene
        // Si aggiunge all'oggetto insiemeDiBeni il bene precedente
        // e se ne crea uno nuovo con i nuovi dati e la nuova
        // descrizione
        sb.addBene(previous);
        tempb = new Bene(autore, titolo);
        tempb.addDescrizione(descrizione);
        previous = tempb;
    } // Fine else sul nuovo bene
} // Fine else sui beni diversi dal primo
} // Fine while di scansione di tutti i beni
```

286 Appendice A

```

        // inserimento dell'ultimo bene considerato nell'oggetto setBeni
        if (previous != null) sb.addBene(previous);
    } catch (Exception e) {e.printStackTrace();}
    // l'oggetto setBeni è ritornato come risultato del metodo
    return sb;
} // Fine metodo executeQuery

...
} // Fine della classe DbManagerImpl

```

Infine, è riportato di seguito il codice della form iniziale, che nel corso della progettazione è stata definita `InputForm`. Questo è il codice sorgente della pagina mostrata nella Figura A.8. Si noti la clausola `form` che definisce quale servlet chiamare. L'indicazione `method = "post"` fa sì che sia invocato il metodo `doPost` della servlet relativa. In particolare, la servlet è indicata inserendo l'indirizzo al quale è possibile trovarla: tale indirizzo è composto dal nome del computer su cui risiede, la porta dedicata, la directory in cui si trova il package della servlet e il nome stesso del package che contiene la servlet `Query`. La struttura dell'indirizzo è definita dal web server usato nel progetto, vale a dire Tomcat.

```

<html> <head> <title> prova </title> </head>
<body>
<h1>Ricerca beni su siti</h1>

<form method = "post" action = "http://localhost:8080/DirectoryProgetto/Query">
<p>Autore <input type="text" name="Autore" size="20"></p>
<p>
Titolo <input type="text" name="Titolo" size="20"><p>
Chiavi <input type="text" name="Chiavi" size="80"><p>
<input type="checkbox" name="Ricerca" value="OFF">Tutte le chiavi<p>
Insieme predefinito<p>
<input type="radio" value="Nessuno" checked name="Insieme">Nessuno<p>
<input type="radio" name="Insieme" value="Rinascimento">Pittura
rinascimentale<p>
<input type="radio" name="Insieme" value="Sacra">Pittura sacra<p>
&nbsp;<p>
<input type = "submit"> <input type = "reset"> </p> </p>
<p> Gruppo <input type="text" name="Gruppo" size="20">
</form> </body> </html>

```

A.5 I casi di test

Il test dell'applicazione (si veda il Capitolo 9) è stato svolto utilizzando un approccio in tre passi.

1. Nel primo passo si è provato il singolo `DataManager` sconnesso da tutti gli altri componenti. Per fare ciò si è creato un driver in grado di inviare una query al `DataManager` e di mostrare i risultati ottenuti.
2. Nel secondo passo, è stato integrato nel prototipo ottenuto al passo 1 anche il componente `Index` per verificare la registrazione presso lo stesso di tutti i `DataManager`.
3. Nel terzo passo si è aggiunta la servlet con l'interfaccia web-based.

I casi di test sono stati derivati direttamente dalla descrizione del problema. In particolare, sono stati ideati per verificare il corretto funzionamento delle principali operazioni di ricerca: ricerca diretta, ricerca guidata, ricerca per insiemi predefiniti. Nella definizione dei casi di test sono stati considerati i requisiti specifici di ogni operazione, secondo quanto spiegato nel Capitolo 9. In particolare, come punto di partenza per la definizione dei casi di test sono stati adottati i sequence diagram presentati in precedenza. Tali diagrammi, infatti, descrivono in maniera precisa le richieste e le esigenze espresse dal cliente nella fase iniziale di definizione dei requisiti. Inoltre è stato definito un caso di test mirato a verificare l'affidabilità del sistema in presenza di malfunzionamenti.

Nel seguito si riportano i test black box effettuati sul sistema complessivo. La correttezza dei risultati è stata verificata rispetto al contenuto dei database di prova (che non sono stati allegati, anche se ciò non impedisce di comprendere la struttura dei casi di test). In realtà, per motivi di spazio non sono presenti tutti i casi derivabili dalla descrizione del problema. Il completamento è lasciato al lettore come esercizio.

A.5.1 Casi di test per la ricerca diretta

I casi di test da usare per la ricerca diretta sono stati definiti analizzando il sequence diagram della Figura A.5. Il test dell'operazione si concentra sul metodo `trovaBene(autore, titolo)`. In particolare, dal sequence diagram si nota che ci sono diverse alternative di esecuzione della ricerca. I casi di test devono essere definiti considerando attentamente tutte le alternative mostrate nel diagramma. In particolare, per testare il funzionamento della prima alternativa `trovaBene(a, t)`, in cui ai metodi sono forniti entrambi i parametri `autore` e `titolo`, è necessario definire un caso di test per ogni possibile combinazione di questi parametri. In questo caso le combinazioni sono quattro: entrambi i parametri sono corretti; entrambi i parametri sono sbagliati; è sbagliato solo l'autore; è sbagliato solo il titolo. Di seguito sono mostrati i casi di test derivanti dalle prime due combinazioni. Il primo caso serve per verificare che a fronte di dati corretti, l'operazione restituiscia i risultati attesi; il secondo caso per controllare che il sistema si comporti come progettato a fronte della ricezione di dati errati (ci si riferisce ai campi della form di input).

288 Appendice A**Caso di test 1**

- Descrizione: vengono inseriti nei campi “Autore” e “Titolo” i dati di un’opera presente nei database. Dovranno essere visualizzate tutte le informazioni inerenti all’opera desiderata presenti sul sito che la pubblica.
- Input: Botticelli, La Primavera.
- Risultato: corretto.

Caso di test 2

- Descrizione: vengono inseriti i dati identificativi di un bene che non esiste nel database. Per ogni database raggiunto dalla query viene visualizzato un messaggio che avverte del mancato ritrovamento di dati sull’opera desiderata.
- Input: Caravaggio, La conversione di San Paolo.
- Risultato: corretto.

Caso di test 3

- Descrizione: viene inserito solo il titolo di un’opera presente nei database. Dovranno essere visualizzate tutte le informazioni sulle opere che hanno quel titolo.
- Input: La Primavera.
- Risultato: corretto.

Caso di test 4

- Descrizione: viene inserito nel campo “Autore” il nome di un autore presente nel database. Dovranno essere visualizzate tutte le opere presenti nei database che hanno come autore quello indicato.
- Input: Botticelli.
- Risultato: corretto.

Caso di test 5

- Descrizione: non viene inserito alcun termine nei campi presenti. Vengono visualizzate tutte le opere presenti nei database.
- Input: nessuno.
- Risultato: corretto.

A.5.2 Casi di test per la ricerca guidata

I casi di test relativi alla ricerca guidata sono stati definiti analizzando il sequence diagram della Figura A.6. In particolare, si tratta di sollecitare in modo sistematico il sistema analizzando la struttura dell'operazione rappresentata dal metodo `trovaInsiemeDiBeni(paroleChiave, and)`

In generale, nel testare quest'operazione è necessario considerare due casistiche:

1. le parole chiavi memorizzate in `parolaChiave` sono presenti nel database;
2. una o più delle chiavi memorizzate in `parolaChiave` non sono presenti nel database.

I casi di test devono essere generati combinando queste due situazioni con l'uso della variabile `and` (si vuole che i beni presenti nella risposta siano qualificati con tutte le parole chiave o meno) e con il fatto che il database di prova contenga o meno opere con le caratteristiche richieste. I casi di test che ne derivano sono numerosi. Nel seguito ne sono elencati alcuni. Il lettore è invitato a completare la lista con gli altri mancanti.

Caso di test 6

- Descrizione: viene inserito un unico termine nel campo “Chiavi” e non viene marcata la casella di controllo. Verranno visualizzate tutte le opere che sono associate alla parola chiave selezionata.
- Input: 1400.
- Risultato: corretto.

Caso di test 7

- Descrizione: viene inserito un termine nel campo “Chiavi” che non è presente nel database come chiave e non viene marcata la casella di controllo. Non verrà visualizzata alcuna opera.
- Input: 2500.
- Risultato: corretto.

Caso di test 8

- Descrizione: vengono inseriti nel campo “Chiavi” più termini tutti corretti e presenti nei database e non viene marcata la casella di controllo.
- Input: Pittura, 1400.
- Risultato: corretto.

290 Appendice A**Caso di test 9**

- Descrizione: vengono inseriti nel campo “Chiavi” più termini di cui alcuni sbagliati o non presenti nei database e non viene marcata la casella di controllo. Vengono visualizzate le sole opere che sono associate alle parole chiave corrette.
- Input: 1400, “pitura”, Madonna, 2055.
- Risultato: corretto.

Caso di test 10

- Descrizione: vengono inseriti nel campo “Chiavi” più termini tutti errati o non presenti nei database come parole chiave e non viene marcata la casella di controllo. Non viene visualizzata alcuna informazione su nessuna opera.
- Input: “pitura”, 2500, paperino.
- Risultato: corretto.

Caso di test 11

- Descrizione: vengono inseriti due termini corretti nel campo “Chiavi” e viene marcata la casella di controllo. Vengono visualizzate le informazioni sulle opere che presentano come parole chiave tutte quelle digitate.
- Input: 1400, Pittura.
- Risultato: corretto.

Caso di test 12

- Descrizione: vengono digitati termini corretti nel campo “Chiavi” e viene marcata la casella di controllo, ma non vi sono opere che presentano le parole chiave indicate. Viene segnalato che non sono state trovate opere con tutte le parole chiave desiderate.
- Input: religiosa, 1700.
- Risultato: corretto.

Caso di test 13

- Descrizione: vengono digitati dei termini errati o non presenti nei database nel campo “Chiavi” e viene marcata la casella di controllo. Viene segnalato che non sono state trovate opere con tutte le parole chiave desiderate.

- Input: 2000, scultura.
- Risultato: corretto.

Caso di test 14

- Descrizione: vengono inseriti nel campo “Chiavi” alcuni termini presenti nel database e altri errati o non presenti e viene marcata la casella di controllo. Vengono considerati solo i termini corretti e su questi viene fatta la ricerca.
- Input: 1400, Allegria (al posto di Allegoria).
- Risultato: corretto.

Caso di test 15

- Descrizione: viene inserita un'unica parola chiave nel campo “Chiavi” e viene marcata la casella di controllo. Vengono visualizzate le informazioni sulle opere che hanno la parola chiave desiderata.
- Input: Pittura.
- Risultato: corretto.

A.5.3 Casi di test per la ricerca per insiemi predefiniti

I casi di test da usare per la ricerca per insiemi predefiniti sono stati determinati analizzando il sequence diagram della Figura A.7.

`trovaInsiemeDiBeni(paroleChiave, true)`

Per testare il funzionamento dell'operazione sono stati definiti due casi. Il primo caso verifica che, selezionando un insieme predefinito, vengano mostrate tutte le opere in esso contenute; il secondo caso verifica che passando come parametro di ricerca un insieme predefinito, qualunque altra informazione specificata nella form è ignorata.

Caso di test 16

- Descrizione: viene scelto un insieme predefinito di opere tra quelli presenti. Vengono visualizzate le opere facenti parte di tale insieme. Se un database non possiede opere dell'insieme, viene visualizzato un messaggio di avviso.
- Input: Pittura sacra.
- Risultato: corretto.

292 Appendice A***Caso di test 17***

- Descrizione: viene selezionato un insieme predefinito e contemporaneamente vengono inseriti dei dati nei campi "Autore", "Titolo", "Chiavi". Vengono visualizzate le opere appartenenti all'insieme predefinito scelto mentre gli altri dati inseriti vengono ignorati.
- Input: Pittura rinascimentale.
- Risultato: corretto.

A.5.4 Caso di test per l'affidabilità del sistema

Per testare l'affidabilità dell'operazione di ricerca di un bene in caso di malfunzionamento di una parte del sistema, questo caso di test verifica che, qualora il gestore della base di dati sia inattivo, la situazione di malfunzionamento venga segnalata esplicitamente all'utente. Ovviamente, questo non è l'unico caso di test che è possibile eseguire per verificare l'affidabilità del sistema. Tra gli altri possibili, si può ad esempio considerare il caso in cui l'indice non è disponibile. Il lettore è invitato ad aggiungere i casi di test mancanti.

Caso di test 18

- Descrizione: la query viene correttamente formulata e inviata, ma i database sono inattivi. La situazione di malfunzionamento viene segnalata esplicitamente all'utente.
- Input: una qualunque query corretta. Prima che la query sia inviata i database vengono terminati per poter simulare la situazione di malfunzionamento voluta.
- Risultato: corretto.

Bibliografia

- Ahern, D.M., Clouse, A., et al., *CMMI: A Practical Introduction to Integrated Process Improvement*, 2nd Edition, Addison-Wesley, 2003.
- Arlow, J., Neustadt, I., *UML 2 and the Unified Process*, Addison-Wesley, 2005.
- Arnold, K., Gosling, J., et al., *The Java™ Programming Language*, 4th Edition, Addison-Wesley, 2006.
- Bandinelli, S., Fuggetta, A., et al., "Modeling and Improving an Industrial Software Process", *IEEE Transactions on Software Engineering*, 21(5), pp. 440-454, 1995.
- Beck, K., *Extreme Programming Explained*, Addison-Wesley, 2000.
- Boehm, B., "Spiral Development: Experience, Principles, and Refinements", *Workshop on Sprial Development* (a cura di W.J. Hansen), Special Report CMU/SEI-00-SR-08, ESC-SR-00-08, 2000.
- Boehm, B., Abts, C., et al., *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.
- Brooks, Jr., F.P., *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition, Addison-Wesley, 1995.
- Burdy, L., Cheon, Y., et al., "An overview of JML tools and applications", *International Journal on Software Tools for Technology Transfer*, 7(3), pp. 212-232, 2005.
- Buschmann, F., Meunier, R., et al., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.
- Carrano, F.M., Pritchard, J.J., *Data Abstraction and Problem Solving with Java™*, 2nd Edition, Addison-Wesley, 2006.
- Cattaneo, F., Fuggetta, A., et al., "Pursuing coherence in software process assessment and improvement", *Software Process: Improvement and Practice*, 6(1), pp. 3-22, 2001.
- Cockburn, A., *Agile Software Development*, Pearson Education, 2002.
- Conradi, R., Fuggetta, A., "Improving Software Process Improvement", *IEEE Software*, 19(4), pp. 92-99, 2002.
- Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.
- Di Nitto, E., Rosenblum, D.S., "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures", *Atti della International Conference on Software Engineering (ICSE 1999)*, pp. 13-22, 1999.
- Eriksson, H., Penker, M., et al., *UML 2 Toolkit*, Wiley Publishing Inc., 2004.

294 Bibliografia

- Fagan, M.E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering*, 12(7), pp. 744-751, 1986.
- Feldman, S., "Make-A computer program for maintaining computer programs", *Software-Practice and Experience*, 9(4), pp. 255-265, 1979.
- Fielding, R.T., Taylor, R.T., "Principled design of the Modern Web Architecture", *ACM Transactions on Internet Technology*, 2, 2, pp. 115-150, 2002.
- Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996.
- Fowler, M., *UML Distilled*, 3rd Edition, Addison-Wesley, 2004, (trad. it. *UML Distilled*, 3° Edizione, Pearson Education Italia, 2004).
- Fuggetta, A., "A Classification of CASE Technology", *IEEE Computer*, 26(12), pp. 25-38, 1993.
- Fuggetta, A., "Software process: a roadmap", *Atti della International Conference on Software Engineering (ICSE 2000), Future of Software Engineering Track*, pp. 25-34, 2000.
- Fuggetta, A., Lavazza, L., et al., "Applying GQM in an Industrial Software Factory", *ACM Transactions on Software Engineering and Methodology*, 7(4), pp. 411-448, 1998.
- Fuggetta, A., Picco, G.P., et al., "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, 24(5), IEEE Computer Society, pp. 342-361, 1998.
- Fuggetta, A., Sfardini, L., "Software Engineering Methods and Technologies", *Software Engineering Volume 2: The Supporting Process* (a cura di R.H. Thayer e M. Dorfman), John Wiley & Sons, pp. 313-320, 2005.
- Gamma, E., Helm, R., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- Garlan, D., Allen, R., et al., "Architectural mismatch: why reuse is so hard", *IEEE Software*, 12(6), pp. 17-26, 1995.
- Ghezzi, C., Jazayeri, M., et al., *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2003, (trad. it. a cura di Giacomo Ghezzi e Sam Guinea, *Ingegneria del software*, 2° Edizione, Pearson Education Italia, 2004).
- Hass, A.M.J., *Configuration Management Principles and Practice*, Addison-Wesley, 2002.
- Hofmeister, C., Nord, R., et al., *Applied Software Architecture*, Addison-Wesley, 2000.
- IEEE, *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Software Engineering Standard n. 830-1998, <http://standards.ieee.org>.
- ISO, *Software Engineering - Product quality - Part 1: Quality model*, International Standard ISO/IEC 9126-1, 2001, <http://www.iso.org>.
- ISO, *Software Engineering - Product quality - Part 2: External metrics*, International Standard ISO/IEC TR 9126-2, 2003, <http://www.iso.org>.

- ISO, *Software Engineering - Product quality - Part 3: Internal metrics*, International Standard ISO/IEC TR 9126-3, 2003, <http://www.iso.org>.
- ISO, *Software Engineering - Product quality - Part 4: Quality in use metrics*, International Standard ISO/IEC TR 9126-4, 2004, <http://www.iso.org>.
- Jackson, M., *Software Requirements and Specifications*, Addison-Wesley, 1995.
- Jackson, M., *Problem Frames*, Addison-Wesley, 2001.
- Laney, R., Barrica, et al., "Composing Requirements Using Problem Frames", *Requirements Engineering Conference*, Kyoto (Japan), 2004.
- Leavens, G.T., *The JML Home Page*, <http://www.cs.iastate.edu/~leavens/JML>.
- Lee, R., Seligman, S., *JNDI API Tutorial and Reference: Building Directory-Enabled JavaTM Applications*, Addison-Wesley, 2000.
- Myers, G.J., Sandler, C., et al., *The Art of Software Testing*, 2nd Edition, John Wiley & Sons, 2004.
- Object Management Group (OMG), *UMLTM Resource Page*, <http://www.uml.org>.
- Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of ACM*, 15(12), pp. 1053-1058, 1972.
- Parnas, D.L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, 2(1), pp. 1-9, 1976.
- Parnas, D.L., "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, 5(2), pp. 128-138, 1979.
- Perry, D.E., Wolf, A.L., "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, 1992.
- Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005.
- PMI, *Project Management Institute*, <http://www.pmi.org>.
- Reiss, S.P., "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, 7(4), pp. 57-66, 1990.
- Royce, W., *Software Project Management*, Addison-Wesley, 1998.
- Sebesta, R.W., *Programming the World Wide Web*, Addison-Wesley, 2005.
- Software Engineering Institute (SEI), *CMMI[®] Web Site*, <http://www.sei.cmu.edu/cmmi>.
- Sommerville, I., *Software Engineering*, 7th Edition, Pearson Education, 2004.
- Sun Microsystems, *Java.sun.com: The Source for Java Developers*, <http://java.sun.com>.
- Tichy, W., "RCS – A system for version control", *Software Practice and Experience*, 15(7), pp. 637-654, 1985.
- Tracz, W., "DSSA (Domain-Specific Software Architecture): pedagogical example", *ACM SIGSOFT Software Engineering Notes*, 20(3), pp. 49-62, 1995.
- UDDI, *OASIS UDDI Web Site*, <http://www.uddi.org>.

296 Bibliografia

- W3C, *World Wide Web Consortium*, <http://www.w3.org>.
- Wikipedia, *Client-server*, <http://en.wikipedia.org/wiki/Client-server>.
- Wikipedia, *Earned value management*,
http://en.wikipedia.org/wiki/Earned_value_management.
- Wikipedia, *Model-view-controller*,
http://en.wikipedia.org/wiki/Model_view_controller.
- Wikipedia, *Peer-to-peer*, <http://en.wikipedia.org/wiki/Peer-to-peer>.
- Wikipedia, *Waterfall model*, http://en.wikipedia.org/wiki/Waterfall_model.
- Woodcock, J., Davies, J., *Using Z*, Prentice Hall, 1996.

Postfazione

Avendo speso 11 capitoli e centinaia di pagine per illustrare un numero non piccolo di concetti, tecniche, metodi ed esempi, quali sono i principali messaggi che il lettore dovrebbe trarre da questa mole di informazioni? Come organizzare e ricordare in modo logico e sintetico quanto letto? Come procedere nell'approfondimento dei temi trattati?

Non ha certo senso ripetere ancora una volta quanto già presentato e discusso in precedenza. Tuttavia, è possibile provare a fornire una sorta di sintesi finale, che ripercorra il lavoro fatto, evidenziandone i punti di svolta e gli aspetti maggiormente critici.

Un primo punto chiave è quello richiamato dal titolo stesso del testo: *l'ingegneria del software è una combinazione di creatività e metodo*. Il software è una tecnologia complessa e per certi versi unica. Va conosciuta, va approfondito lo studio delle tecnologie che ne permettono lo sviluppo e l'uso, vanno studiati i metodi che ne garantiscono una progettazione organica e coerente con i bisogni dell'utenza. Ma l'ingegneria del software è anche una disciplina dove l'ingegno viene sollecitato a identificare e sviluppare soluzioni a problemi. Il problem-solving non è un'attività che può essere svolta in modo meccanico. È per definizione qualcosa che richiede il genio, l'inventiva, la creatività delle persone. Bravissimi "cultiori della materia" che conoscano in modo asettico tecnologie e metodi non saranno mai in grado di costruire soluzioni vincenti. Il motto di una famosa azienda dice "la passione ci guida" e sintetizza proprio ciò che deve complementare le competenze tecnologiche e manageriali di un ingegnere del software: la passione, la creatività, l'intuito, il genio. Per questo motivo, in diverse occasioni nel testo si è ricordato che la scelta di un metodo, di una tecnica o di una specifica soluzione deve essere basata non solo su astratti e asettici criteri di natura "meccanicistica": l'ingegnere dovrà saper valutare di volta in volta la soluzione più opportuna, in alcuni casi rifacendosi a esperienze del passato, in altri creando ex-novo risposte innovative e anche non convenzionali in grado di cogliere la complessità delle situazioni che devono essere affrontate e risolte.

Un secondo aspetto chiave, certamente collegato al precedente, concerne *il ruolo del software nella società moderna e, conseguentemente, lo sviluppo della professionalità dell'ingegnere del software*. Il software non è più semplicemente un fenomeno di nicchia per funzioni e servizi particolari. È penetrato in tutti i servizi e prodotti. Non esiste settore industriale e della vita economica e sociale in genere che non sia stato toccato dalle tecnologie informatiche.

Un primo esempio è il mondo delle tecnologie per le telecomunicazioni. Ciò a cui si sta assistendo è la progressiva "informatizzazione" delle telecomunicazioni. I fornitori di apparati di rete, di centrali di commutazione, di telefoni cellulari e telefoni in senso

298 Postfazione

stretto stanno sempre più diventando aziende di software. Le vecchie centrali telefoniche si sono trasformate in sofisticati sistemi software (*softswitch*). Persino la gestione delle fibre ottiche e dei fasci di luce nella trasmissione digitale avviene oggi grazie a complesse applicazioni informatiche. Il mondo della telefonia cellulare è scosso alle radici dall'avvento di nuovi dispositivi (smartphone e PDA) per i quali si delineano i nuovi concorrenti del futuro: non solo i tradizionali costruttori di cellulari europei, americani e asiatici, ma anche e soprattutto Microsoft e Symbian, cioè i produttori del software di base. Skype (un software p2p nato con investimenti relativamente piccoli) sta scuotendo alle fondamenta il mondo dei servizi voce e immagini. Lo sviluppo di IPTV (cioè servizi televisivi trasmessi via rete grazie a software e banda larga) costituisce una grandissima rivoluzione che potrebbe sconvolgere il mercato televisivo tradizionale.

Ma la penetrazione delle tecnologie del software non si limita al mondo delle telecomunicazioni. I componenti per elaborazioni digitali sono ormai presenti in moltissimi oggetti di uso comune: dai telepass ai milioni di elettrodomestici prodotti e venduti in Italia e nel mondo. Si pensi che a fronte di 3,5 milioni di PC venduti in Italia, la sola industria italiana di elettrodomestici produce decine di milioni di pezzi, sull'80% dei quali vi sono uno o due microprocessori, un sistema operativo e software applicativo e di controllo. I moderni sistemi di infomobilità per le quattro e due ruote sono basati su sistemi ICT e, soprattutto, moderne applicazioni informatiche. Il mondo delle macchine utensili, dove l'Italia è ancora un leader a livello mondiale, è sempre più basato su sistemi informatici di controllo e gestione remota. Le applicazioni di monitoraggio ambientale e di supporto al mondo della protezione civile e della sicurezza sono centrate sul ruolo dell'ICT e del software.

Persino settori legati nell'immaginario collettivo a quanto di più convenzionale esista, come l'agricoltura, si stanno rapidamente evolvendo grazie al software e, in generale, all'ICT. Le industrie che producono trattori agricoli utilizzano sofisticati sistemi infotelematici sia per il controllo della trazione del mezzo (che grazie ad essi può ruotare su se stesso) sia per l'automazione tramite GPS delle operazioni sui campi (l'aratura in automatico). Le tecnologie dei RFID permettono già ora la costruzione di soluzioni e applicazioni molto sofisticate e innovative nel campo della logistica e del controllo alimentare.

In sintesi, *non esistono settori della nostra società che non siano permeati da prodotti innovativi basati sull'ICT e soprattutto sul software. È il software che permette di fornire "l'intelligenza" ed è il software che abilita anche lo sviluppo di servizi innovativi.* Per esempio, recentemente sono stati annunciati sistemi infotelematici per le due ruote che grazie alla localizzazione satellitare e all'uso di applicazioni informatiche specifiche rendono possibile lo sviluppo di servizi assicurativi basati sul pay-per-use.

Il software è quindi decisivo. Ma non basta semplicemente affermare questa centralità. *Servono imprese capaci di svilupparlo.* E soprattutto *servono professionisti che siano in grado di dominare e condurre efficacemente i processi di sviluppo del software.* *L'ingegneria del software non è dunque una disciplina riservata a coloro che operano nei settori più strettamente e storicamente legati all'ICT (produttori di computer, società di telecomunicazione, software house).* *Tutte le imprese moderne dovranno in un qualche modo usare, applicare, in-*

tegrare e persino sviluppare software, direttamente o interagendo con un fornitore esterno. Ciò significa che concetti come descrizione del problema, verifica e validazione, pianificazione e gestione di progetto dovranno essere conosciuti e dominati da una fascia sempre crescente di professionisti e imprese.

Infine, terzo aspetto chiave, è essenziale che siano noti e conosciuti i principi, le tecniche, i metodi e le tecnologie dell'ingegneria del software.

- *Il software serve per risolvere problemi.* È essenziale che lo sviluppo di una soluzione informatica si fondi su un profondo e dettagliato, anche se a volte incrementale, studio del problema. *Descrivere bene il problema è un primo decisivo passo verso la costruzione di una soluzione vincente.* Per fare ciò, è essenziale saper ascoltare i clienti, gli utenti e coloro che conoscono il dominio applicativo nel quale il sistema da sviluppare dovrà andare a collocarsi.
- Un software efficace deve essere pensato “bene”: *occorre saper progettare il software, valutando e combinando le diverse soluzioni architettoniche disponibili.*
- Nello studiare il problema e nel concepire la soluzione è *importante saper riusare le conoscenze ed esperienze pregresse* (problem frame, stili architettonici, design pattern ecc.). È inutile e dannoso reinventare le cose. È altrettanto essenziale *saper trasmettere agli altri quanto di originale si è concepito o sviluppato.*
- Tutti sbagliano e nessuno può immaginare di sviluppare una qualunque applicazione informatica senza introdurre alcun errore. *Verificare e validare tutto quanto si produce è essenziale per garantire il successo di un progetto di sviluppo di software.*
- Essendo un processo complesso, *lo sviluppo di un sistema informatico deve essere condotto applicando con rigore e sistematicità le tecniche, i metodi e gli strumenti di gestione di progetti e prodotti:* dal configuration management al controllo di avanzamento di progetto.
- Nessuno è perfetto: tutti possono migliorare. Avere software di qualità bassa è un rischio che, in alcuni casi, può persino mettere in pericolo vite umane. *Chi sviluppa software deve sempre perseguire il miglioramento della qualità del prodotto e del processo,* proprio perché il software è ormai d'importanza vitale in moltissimi ambiti della nostra società.
- Infine, l'ingegneria del software richiede persone di qualità, creative, capaci di lavorare con i colleghi, con gli utenti e con i partner. *Servono professionalità, intelligenza e talento.* Nessuna tecnologia potrà mai sopperire a una carenza o mancanza di queste qualità.

Questo testo vuole essere un primo passaggio nello studio dell'ingegneria del software. Nei capitoli che lo compongono si è cercato di illustrare non solo le singole tecniche o i concetti astratti, ma di spiegare come l'ingegnere del software deve sviluppare le proprie attività, valorizzando competenze, conoscenze e tecnologie. Ovviamente, un singolo testo non potrà mai soddisfare appieno tutte le esigenze. Serviranno approfondimenti e ul-

300 Postfazione

teriori studi che vadano alla radice delle tante tematiche discusse. Taluni possono vivere tutto ciò come una frustrazione: “non si finisce mai”. Per altri, è uno stimolo continuo a crescere, imparare, confrontarsi con nuove sfide, migliorare.

A proposito delle imprese difficili o persino “impossibili”, un detto molto saggio recita così: “Come può un singolo uomo mangiare un elefante? Un boccone alla volta.”

Indice analitico

#

«delegate», 42
 «include», 14
 «instantiate», 25
 «interfacciahardware», 85

A

accuratezza, 55
 activity diagram, 28
 Actual Cost (AC), 169
 Actual Cost for Work Perfomed (ACWP), 169
 adattabilità, 60
 adeguatezza, 55
 aderenza agli standard, 57
 affidabilità, 58, 63, 247
 aggregazione, 24
 Agile Software Developmen, 157
 Extreme Programming, 158
 ambiente di esecuzione, 49
 ambito della descrizione, 11
 analista di mercato, 74
 analizzabilità, 60
 application logic, 131
 apprendibilità, 58
 approcci evolutivi, 109
 approccio jo-jo, 242
 architectural mismatch, 241
 architettura, 127
 composita, 142
 hardware, 128
 software, 8, 128, 235
 arco, 29
 artifact, 43
 assert, 250
 asserzioni, 250
 association class, 18, 22

associazione, 19

binaria, 20
 n-aria, 20
 riflessiva, 21

attività, 159
 critica, 164
 interne, 35

attore, 12
 attrattività, 58
 attributo, 16
 azione, 28, 35

B

baseline, 174
 bottom-up, 241
 brainstorming, 84
 branch, 174
 budget, 167
 Budgeted Cost for Work Performed (BCWP), 169
 Budgeted Cost for Work Scheduled (BCWS), 169
 building, 176
 byte code, 8

C

cammino critico, 164
 Capability Maturity Model Integration (CMMI), 191
 caratteristiche statiche, 115
 casi di test, 10, 181
 caso d'uso, 13
 check-in, 175
 check-out, 175
 ciclo di vita, 7, 148, 231
 a cascata, 149
 a spirale, 151-152
 basato su componenti, 154
 iterativo, 151
 Unified Process, 155

302 Indice analitico

- ciclo di vita a cascata, 109
- class diagram, 16
- classe, 16
 - astratta, 18
- client, 130
- client-server, 8, 130
 - due livelli, 131
- COCOMO, 233
- COD, 138
- code on demand, 138
- codice, 7
 - intermedio, 8
 - mobile, 138
 - oggetto, 8
- coding standard, 244
- coerenza esterna, 5
- coerenza interna, 4
- coesione, 63-64
- coesistenza, 61
- committente, 73
- completezza, 227
- component diagram, 42
- componente, 10, 42, 127
- componenti business tier, 211
- componenti client tier, 209
- componenti di sistema, 225
- componenti web tier, 209
- comportamento temporale, 59
- composite structure diagram, 44
- composizione, 24
- comprensibilità, 57
- comunicazione, 83
- condivisione, 24
 - di prototipo, 84
- condizione, 35
- Configuration Management (CM), 173
 - politiche di locking, 175
- configurazione, 174
- conformità, 58-61
- connettore, 47
- conoscenza di dominio, 6, 230
- consumo di risorse, 59
- container, 211
- context diagram, 69
 - designed domain, 69
 - dominio applicativo, 69
 - given domain, 69
 - interfaccia, 69
- machine domain, 69
- problem domain, 69
- Controller, 145
- controlli run-time, 252
- copertura del codice, 182
- CORBA, 201
- Cost Performance Index (CPI), 171
- Cost Variance (CV), 171
- costo, 159
 - marginale, 1

- D**
- data management, 131
- decision, 29
- defensive programming, 245, 247, 251
- deliverable, 148, 150
- deployment diagram, 49
- deployment view, 110, 120
- descrizione, 3
 - del dominio, 234
 - dominio applicativo, 7
 - interfaccia del sistema, 7
 - requisiti, 7
- Design By Contract, 247, 251
- design pattern, 9, 129, 143
- diagramma di Gantt, 164
- diagramma PERT, 162
- dimensione, 62
- directory, 135
- direzione sistemi informativi, 73
- disaccoppiamento, 63, 65
- dispatcher, 137
- dispositivo, 49
- distribuzione fisica, 112
- documentation standard, 245
- documentazione di prodotto, 53
- documentazione di progetto, 9
- documentazione di supporto, 53
- Domain Specific Software Architecture, 239
- dominio applicativo, 5, 67-68, 228
- driver, 9
- durata del progetto, 159

- E**
- Earned Value (EV), 169
- Earned Value Management (EVM), 168
- efficacia, 61
- efficienza, 59

effort, 160
ensures, 252
Enterprise Java Bean (EJB), 215
 Deployer Descriptor, 215
 EJB Class, 215
 EJBHomeInterface, 215
 EJBObjectInterface, 215
Entity EJB, 216
errore, 181
 di stima, 232
esperti di dominio, 6, 231
esperto di dominio, 69
etichetta, 49
eventi, 137
evento, 35
execution view, 122
extreme programming, 245

F

fenomeni, 5
fenomeni condivisi, 5
flusso, 29
focus group, 84
fork, 29
formalità, 227
frame, 40
functional decomposition, 236
funzionalità, 55, 63
funzione parziale, 248
funzione totale, 248

G

generalizzazione, 12, 22
 tra use case, 13
generatori, 225
gestione delle configurazioni, 235
Goal/Question/Metric (GQM), 196
guasto, 58

I

IEC, 54
implementazione, 236
indentazione, 244
information hiding, 63-64, 236
installabilità, 61
interazione indiretta, 84

interfaccia, 18, 85, 236
 del sistema, 67
 fornita, 18
 richiesta, 18
 software, 85
 utente, 85
interoperabilità, 56
intervista, 83
invariant, 252
invariante, 248
invarianti, 250
ISO, 54
ISO 9000, 193
 Ente di accreditamento, 193
 Ente di certificazione, 193
 Ente di standardizzazione, 193
 Standard, 193
istanza di componente, 42
iterazione, 151

J

Java 2 Enterprise Edition (J2EE), 209
Java DataBase Connectivity (JDBC), 217
Java Messaging Service (JMS), 206
 connessione publish-subscribe, 208
 connessione punto-punto, 207
Java Naming and Directory Interface (JNDI), 205
Java Server Pages (JSP), 214
JDBC
 Application, 218
 Data Source, 218
 Driver, 218
 Driver Manager, 218
JML, 251
JMS, 240
join, 29
JVM, 203

L

layer, 141
libreria, 8
linguaggi, 223
 non object-oriented, 243
 object-oriented, 242
linguaggio di descrizione, 4, 233
link, 25

304 Indice analitico

linking, 8
 lock-in, 234
 logical-functional view, 110-111

M

MA, 138
 macchina, 67
 malfunzionamento, 58, 181
 manutenibilità, 59
 maturità, 58
 merge, 30, 175
 Message-Driven Bean, 217
 messaggio, 38
 asincrono, 38
 di ritorno, 40
 sincrono, 38
 meta-metodi, 223
 metodi formali, 227
 metodo, 2
 object-oriented, 237
 metrica, 196
 metriche del software, 62
 Function Point, 63
 numero di classi e interfacce, 62
 numero di errori per linea di codice, 63
 numero di linee di codice, 62
 standard ISO/IEC 9126, 62
 middleware, 201, 236, 240
 mobile agent, 138
 model, 144
 model-view-controller, 9, 143
 modificabilità, 60
 modularità, 63-64
 del codice, 236
 module view, 110, 115
 modulo, 7, 64
 molteplicità, 17
 multi-tier, 133

N

naming standard, 244
 navigabilità, 20
 nodo, 49
 fine, 28
 inizio, 28
 numerosità, 47

O

object diagram, 25
 oggetto, 25
 operabilità, 58
 operazione, 17
 organizzazione del team di progetto, 235

P

p2p, 239
 ibrido, 135
 puro, 135
 package, 26
 annidati, 27
 diagram, 26
 parte, 46
 partizioni, 32
 path di comunicazione, 50
 peer, 135
 peer-to-peer, 135
 pianificazione, 159
 piano di progetto, 159, 231-232
 piattaforme per componenti software, 225
 pipe-and-filter, 141
 Planned Value, 169
 port, 42
 portabilità, 60
 postcondizione, 248
 postcondizioni, 250
 precondizione, 248
 precondizioni, 250
 presentation, 131
 principi, 222
 di progettazione, 63
 problem diagram, 75
 annotazione sulle interfacce, 75
 requisito, 75
 problem frame, 91
 commanded behaviour, 93
 composite frame, 106
 information display, 96
 required behaviour, 92
 simple workpieces, 100
 transformation, 102
 problema, 2
 processo, 147
 di sviluppo per componenti, 109
 processo software, 147

processo software
 controllabilità, 54
 predicibilità, 54
prodotto software, 53, 147
produttività, 62-63
progettazione architetturale, 109
progettazione di dettaglio, 109
programma sorgente, 7
programmi eseguibili, 8
programming idiom, 9, 129
project management, 225
protocolli, 225
pseudostato finale, 36
pseudostato iniziale, 36
publish-subscribe, 137, 239

Q

qualità, 1
 caratteristiche, 55
 descrizione, 4
 esterna, 55
 fasi del processo software, 54
 in uso, 55
 interna, 54
 modello di qualità in uso, 55, 61
 modello di qualità interna ed esterna, 55
prodotto software, 53
soluzione software, 53
sottocaratteristiche, 55
standard ISO/IEC 9126, 54
Quality Improvement Paradigm (QIP), 195

R

RE, 138
relazione manifest, 120
release, 176
remote evaluation, 138
Remote Method Invocation (RMI), 202
Repository, 174
requires, 252
requisito, 73, 229
 componente di implementazione, 81
 documento di specifica, 81
 fisico, 74
 fonte, 81
 funzionale, 74
 interfaccia, 74

licenza, 75
matrice, 75
performance, 74
tecnologico, 74
tracciatura, 81
tracciatura orizzontale, 81
tracciatura verticale, 81
usabilità, 74
vincolo, 75
rigore, 227
ripristinabilità, 59
rischio, 232
risorsa, 159
robustezza, 247
RPC, 201
runtime view, 110

S

scelte architetturali, 235
scelte progettuali, 238
Schedule Performance Index (SPI), 171
Schedule variance (SV), 171
schemi, 2
segnaletica, 32
 inviatore, 32
 ricevuto, 32
 temporale, 32
sequence diagram, 37
server, 130
Service Provider, 220
Service Requestor, 220
servlet, 212
Session EJB, 215
 Stateful, 216
 Stateless, 216
sicurezza, 57, 62
Simple Object Access Protocol (SOAP), 221
sistema informatico, 11
sistemi federati, 136
Skype, 136
soddisfazione, 62
software inspections, 188
 checklist, 188
soluzione, 3, 109
 software, 53
sostituibilità, 61
spazio del problema, 67

306 Indice analitico

spazio della soluzione, 5, 67
specifica, 7, 229
specificità dell'interfaccia, 67, 85, 89
 descrizione dinamica, 85
 descrizione statica, 85
stabilità, 60
stakeholder, 73
standard di documentazione, 249
standard ISO/IEC 9126, 54, 62
state machine diagram, 35
stato, 35
 concorrente, 36
stepwise refinement, 236
stereotipo, 13
stile architettonale, 8, 129
stili architettonici, 238
stili e pattern, 223
stima dei costi, 232
Structured Query Language (SQL), 218
stub, 9
superstato, 36

T

task region, 151
tecniche di analisi manuali, 180
tecniche di sviluppo, 223
 formali, 223
 informali, 223
 semiformali, 223
tecniche formali, 4
tecnologie, 234
 di processo, 225
 di sviluppo del software, 225
test black box, 186
test di integrazione, 187
test di modulo, 187
 driver, 187
 stub, 187
test di regressione, 188
test di sistema, 187
test white box, 182
 boundary condition, 184
 Condition coverage, 184
 Edge coverage, 182
 Path coverage, 184
 Statement coverage, 182

testabilità, 60
testing, 180-181
thread, 9
tolleranza ai guasti, 59
tool, 223
toolset, 224
top-down, 241
total cost of ownership, 234

U

UML, 4, 11, 242, 246
unità di deployment, 127
unità di riuso, 127
Universal Description Discovery and
Integration (UDDI), 221
usabilità, 57
use case diagram, 11
utente, 73

V

validazione, 180
verifica, 180
versione, 173
 dot notation, 173
view, 145
vincoli, 21
vincolo, 73
 {OR}, 21
virtual machine, 8
visibilità, 16
viste architettoniche, 110

W

walkthrough, 189
WBS, 233
 summary task, 161
 workpackage, 161
web service, 219
Web Service Description Language
(WSDL), 221
Work Breakdown Structure (WBS), 161
workspace, 175