ANALISI E PROGETTAZIONE DEL SOFTWARE

Laurea triennale in scienze informatiche

Andrea Broccoletti

Università degli studi di Milano Bicocca

A.A. 2022/2023

Indice

Capitolo 1

Analisi e progettazione orientata agli oggetti

1.1 Che cosa sono analisi e progettazione

L'**analisi** enfatizza un'investigazione del problema e dei requisiti, anzichè di una soluzione. La **progettazione** enfatizza una soluzione concettuale che soddisfa i requisiti, anzichè la relativa implementazione, spesso escluendo dettagli di basso livello o ovvi per coloro a cui è destinato.
L'analisi e la progettazione possono essere riassunte nell'espressione *fare la cosa giusta e fare la cosa bene*.

1.2 Che cosa sono analisi e progettazione orientata agli oggetti

Durante l'analisi orientata agli oggetti c'è un'enfasi sull'identificazione e la descrizione degli oggetti, o dei concetti, nel dominio del problema; mentre durante l'implementazione o la programmazione orientata agli oggetti, gli oggetti progettati vengono implementati.

Analisi e progettazione hanno obiettivi diversi, ma sono anche attività sinergiche e correlate alle altre attività dello sviluppo software.

1.3 Che cos'è UML

Unified modeling Language è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software. È uno

standard de facto per la notazione di diagrammi.

1.3.1 Tre modi per applicare UML

Ci sono tre modi di applicare UML:

- **come abbozzo** diagrammi informali e incompleti, utili per l'espressività dei linguaggi visuali
- **come progetto** diagrammi di progetto relativamente dettagliati, utilizzati per il reverse engineering, ovvero per visualizzare e comprendere meglio del codice esistente
- **come linguaggio di programmazione** specifica completamente eseguibile, con codice generato automaticamente: rappresenta quindi in modo pratico il comportamento e la logica

La **modellazione Agile** enfatizza il primo utilizzo.

1.3.2 Due punti di vista per applicare UML

UML descrive due tipi grezzi di diagrammi, ma non impone un particolare punto di vista di modellazione per l'uso di questi diagrammi.
Una stessa notazione può essere utilizzata secondo due punti di vista e tipi di modelli:

- **punto di vista concettuale** i diagrammi sono scritti e interpretati come descrizioni di oggetti del mondo reale o nel dominio di interesse
- **punto di vista software** i diagrammi descrivono astrazioni o componenti software

1.3.3 Il significato di "classe"

Nell'UML grezzo, le classi sono rappresentate da rettangoli e racchiudono una varietà di significati. In un modello di dominio, sono **concetti di dominio** o **classi concettuali** e rappresentano un concetto del mondo reale; quando i rettangoli sono disegnati nel modello di progetto, sono chiamati **classi di progetto** o **classi software** e rappresentano un componente software.

1.4 Vantaggi della modellazione visuale

I diagrammi aiutano a vedere o esaminare meglio il quadro generale e le relazioni tra gli elementi dell'analisi software, e allo stesso tempo ci permettono di ignorare o nascondere dettagli poco interessanti.

Capitolo 2

Iterativo ed evolutivo

2.1 Che cos'è UP

Un **processo per lo sviluppo software** descrive un approccio alla costruzione, al rilascio ed eventualmente alla manutenzione del software. **Unified Process** è un processo *iterativo* diffuso per lo sviluppo del software per la costruzione di sistemi orientati agli oggetti.

UP è molto flessibile e aperto, combina infatti delle best practice comunemente accettate, inoltre ci sono 3 motivi che lo rendono un'ottima scelta:

1. lo sviluppo iterativo è il modo migliore per applicare l'OOA/D
2. fornisce una struttura di esempio sia per eseguire che per spiegare l'OOA/D
3. può essere applicato usando un approccio leggero e *agile* che comprende pratiche di altri metodi agili

2.2 Processi per lo sviluppo software

Un processo per lo sviluppo software, definisce un approccio disciplinato per la costruzione, il rilascio e la manutenzione del software. UP ne è un esempio.
In generale, un processo descrive **chi** fa **che cosa**, **come** e **quando** per raggiungere un certo obiettivo. Ciò che caratterizza di più i diversi processi software sono le scelte che riguardano l'**organizzazione temporale** delle attività.

2.3 Il processo a cascata

Il processo software con ciclo di vita a cascata è basato su uno svolgimento sequenziale delle diverse attività dello sviluppo del software.
Questo processo è spesso mediocre e poco efficace, infatti è fortemente associato a una elevata percentuale di fallimenti e ad un basso indice di produttività.

2.3.1 Frequenti fallimenti

I frequenti fallimenti sono fortemente legati ad un presupposto erroneo fondamentale che sta alla base di molti progetti software poi falliti, ovvero che le specifiche siano prevedibili e stabili, e che possano essere definite correttamente sin dall'inizio, a fronte di un basso tasso nei cambiamenti. Ciò è falso, poichè lo sviluppo software è soggetto a cambiamento e instabilità: il *cambiamento* è una costante.

2.4 Lo sviluppo iterativo ed evolutivo

Una pratica moderna è lo **sviluppo iterativo**, sviluppo organizzato in una serie di mini-progetti brevim di lunghezza fissa, chiamati **iterazioni**. Il risultato di ogni iterazione è un sistema eseguibile, testato e integrato, anche se parziale, che fornisce un feedback alle fasi successive.
Il sistema cresce in modo incrementale nel tempo, poichè il feedback e l'adattamento fanno evolvere le specifiche e il progetto.

2.4.1 Come gestire i cambiamenti

Anzichè contrastare i cambiamenti che si verificano, lo sviluppo iterativo ed evolutivo si basa su un atteggiamento di accettazione del cambiamento e sull'adattamento come guide inevitabili ed essenziali. Dunque, il lavoro procede mediante una serie di cicli strutturati di costruzione, feedback, adattamento. Per questo, nelle iterazioni finali è difficile, ma non impossibile, che si verifichi un cambiamento significativo dei requisiti.

2.4.2 Feedback e adattamento

Il feedback può provenire dalle attività inziali di sviluppo, dai test e dagli sviluppatori, dal team e dal cliente o dal mercato. I feedback e l'adattamento sono ingredienti chiave di successo.

2.4.3 Vantaggi dello sviluppo iterativo

Lo sviluppo iterativo presenta i seguenti vantaggi:

- minore probabilità di fallimento del progetto
- riduzione precoce dei rischi maggiori
- progresso visibile fin dalle prime fasi
- coinvolgimento dell'utente e adattamento basato sui feedback, che portano ad un sistema che soddisfa meglio le esigenze reali
- gestione della complessità, evitando la "paralisi da analisi"
- ciò che si apprende nel corso di un'iterazione può essere utilizzato metodicamente per migliorare il processo di sviluppo stesso

2.4.4 Durata delle iterazioni e timeboxing

Molti metodi iterativi raccomandano una durata delle iterazioni da due a sei settimane. Iterazioni pià lunghe sono contrarie allo sviluppo e aumentano i rischi del progetto. Meglio puntare a iterazioni brevi.

Un'idea chiave è quella del **timeboxing**: le iterazioni hanno una durata fissata e non è consentito ritardare la data dell'iterazione. Se si rischia di farlo, si deve ridurre la portata, ovvero eliminare attività o requisiti dell'iterazione e includerli in una futura. Un'iterazione di durata fissa è detta **timeboxed**.

2.4.5 Flessibilità del codice di progetto

L'adozione dello sviluppo iterativo richiede che il software venga realizzato in modo *flessibile*, affinchè l'impatto sia basso. Il codice sorgente deve essere facilmente *modificabile*.

2.5 Guidata dal rischio e dal cliente

Lo sviluppo iterativo promuove la pianificazione guidata dal rischio e guidata dal cliente, ciò significa che gli obiettivi delle iterazioni iniziali vengono scelti per attenuare i rischi maggiori e per costruire e rendere visibili le caratteristiche a cui il cliente tiene di più.

2.6 Altre pratiche fondamentali di UP

Ci sono ulteriori best practice e concetti chiave di UP:

- affrontare le problematiche di rischio maggiore e valore elevato nelle iterazioni iniziali
- impegnare gli utenti continuamente sulla generazione dei feedback
- creare un architettura coesa
- verificare continuamente la qualità e testare spesso e in modo realistico

2.7 Le fasi di UP

Un progetto UP organizza il lavoro e le iterazioni in quattro fasi temporali principali successive:

1. **ideazione** visione approssimativa, con stime dei costi e dei tempi
2. **elaborazione** visione raffinata, implementazione iterativa del nucleo dell'architettura, identificazione della maggior parte dei requisiti
3. **costruzione** implementazione iterativa delgi elementi rimanenti, più facili e a rischio minore
4. **transizione** beta test e, infine, rilascio

2.8 Le discipline di UP

UP colloca le attività lavorative nell'ambito delle **discipline**, una disciplina è un insieme di attività e dei relativi elaborati in una determinata area. In UP, un **elaborato** è il termine generico che indica un qualsiasi prodotto di lavoro.
In UP, **implementazione** significa programmare e costruire il sistema, non rilasciarlo. La disciplina **infrastruttura** fa riferimento alla definizione degli strumenti, ovvero all'impostazione degli strumenti e del processo.

Capitolo 3

Agile

3.1 Metodi e atteggiamenti agili

Non è possibile dare una definizione precisa di metodo agile, poiché le pratiche adottate variano notevolmente da metodo a metodo. Tuttavia, una pratica di base è quella che prevede iterazioni brevi, con un raffinamento evolutivo dei piani, dei requisiti e del progetto. Inoltre, essi promuovo pratiche e principi che riflettono una sensibilità agile per la semplicità e la leggerezza.

3.2 Agile Modeling

Lo scopo della documentazione è principalmente quello di comprendere, non di modellare. Ciò significa che la modellazione serve per comprendere meglio un problema e spaziare tra le soluzioni, esplorando le alternative.
Questo modo di vedere è stato chiamato **modellazione agile** e si basa su un insieme di pratiche e di valori:

- non significa evitare di tutto la modellazione, ma si da una minore enfatizzazione
- lo scopo dei modelli è di agevolare la comprensione e la comunicazione
- è buona norma rimandare i problemi semplici o diretti alla fase di programmazione, utilizzando i modelli sono per problematiche difficili e insidiose
- va utilizzato per primo lo strumento più semplice possibile, che aiuti la creatività con il minimo dispendio, indipendentemente dalla tecnologia scelta (anche una lavagna va bene)

- la modellazione non va fatta in solitario, ma piuttosto a coppie
- non è importante essere rigidi e rigorosi nell'utilizzo dell'UML, purchè i modellatori si capiscano l'uno con l'altro
- tutti i modelli sono inevitabilmente imprecisi, e il codice o il progetto finale differiranno, a volte anche in modo notevole
- la modellazione OO dovrebbe essere eseguita dagli stessi sviluppatori che si occupano della programmazione

3.3 Che cos'è UP agile

UP, in chiave agile, deve preferire un insieme *piccolo* di attività e di elaborati UP, inoltre non esiste un piano dettagliato per l'intero progetto. Esiste un piano ad alto livello, **piano delle fasi**, che stima la data di fine del progetto e delle milestone principali. Un piano dettagliato, il **piano dell'iterazione**, pianifica in maggior dettaglio un'unica iterazione, ovvero quella successiva.

3.4 Che cos'è Scrum

Scrum è un metodo agile che consente di sviluppare e rilasciare prodotti software con il più alto valore per i clienti, nel più breve tempo possibile.
Si occupa principalmente dell'organizzazione del lavoro e della gestione dei progetti, con più disinteresse verso gli aspetti tecnici dello sviluppo software, lasciando agli sviluppatori la possibilità di scegliere le tecniche e le metodologie specifiche da utilizzare. Può essere facilmente combinato con altri metodi.

Scrum è un approccio iterativo e incrementale allo sviluppo del software. Ciascuna iterazione, chiamata uno **Sprint**, ha una durata fissata, per esempio di due settimane. Le iterazioni sono timeboxed, e dunque non vengono mai estese.

Ci sono solo tre ruoli: il **product owner**, il proprietario che definsce le caratteristiche del prodotto software, il **development team** è composto di solito da una manciata di persone, che possiedono le competenze necessarie per sviluppare software. Infine, lo **Scrum Master** aiuta l'intero gruppo ad apprendere e applicare Scrum, al fine di ottenere il valore desiderato. Queste tre figure, nel loro complesso, formano un **team scrum**.

Il metodo scrum inizia quando il product owner definisce le caratteristiche del prodotto da realizzare nel **product backlog**, che è un elenco di voci ordinato per priorità.
All'inizio di ciascuno Sprint, il Team seleziona dal product backlog un insieme di voci da sviluppare durante quell'iterazione, definendo uno **sprint goal**, ovver l'obiettivo di sviluppo del Team per questo Sprint.
Ogni giorno, all'inizio della giornata, il Team si riunisce brevemente in un **Daily Scrum** per verificare i progressi e decidere i passi successivi.
Il risultato di ciascuno Sprint deve essere un prodotto software funzionante, chiamato *incremento di prodotto pontenzialmente rilasciabile*.
Alla fine dello Sprint, nello **Sprint Review** il product owner e il Team presentano alle diverse parti interessate l'incremento di prodotto software che è stato sviluppato, e ne fanno una dimostrazione, al fine di ottenere un feedback.
Si procede così, di Sprint in Sprint, in modo iterativo, fino a quando l'intero prodotto non viene completato.

La caratteristica distintiva di Scrum tra i metodi agili è l'enfasi sull'adozione di team auto-organizzati e auto-gestiti. Inoltre, Scrum è basato su un insieme di elaborati ed eventi che hanno lo scopo di rendere visibili gli obiettivi e il progresso delle iterazioni e di favorire un adattamento evolutivo del processo di sviluppo.

Capitolo 4

Studi di caso

4.1 Di che cosa trattano gli studi di caso

In generale, le applicazioni comprendono elementi dell'interfaccia utente, la logica applicativa principale, l'accesso alla base di dati e collaborazioni con componenti software o hardware esterni.

Sebbene la tecnologia OO possa essere applicata in tutti gli strati, questa introduzione all'OOA/D si concentra su quello della logica applicativa principale, con alcune discussioni secondarie sugli altri strati.

4.2 Strategia degli studi di caso

L'analisi dei requisiti e l'OOA/D sono applicate agli studi di caso in più iterazioni. L'iterazione 0 illustra i requisiti; l'iterazione 1 riguarda l'analisi e la progettazione OO di alcune funzionalità principali, e le successive iterazioni prendono in considerazione altre funzionalità e capacità.

4.3 Esempio di caso: POS NextGen

Un sistema POS è un'applicazione software utilizzata per registrare le vendite e gestire i pagamenti.
Un sistema POS comprende componenti hardware, come un computer e un lettore di codici a barre, nonché del software. Si interfaccia a varie applicazioni di servizio, come un sistema per l'inventario o per la contabilità, che possono essere realizzati da terzi.
Un sistema POS deve essere relativamente tollerante ai guasti; ciò significa

che, anche se i servizi remoti sono temporaneamente non disponibili (come il sistema di inventario), il sistema deve essere comunque in grado di gestire le vendite, permettendo almeno il pagamento in contanti, in modo che l'attività di vendita non venga interrotta e danneggiata.
Un sistema POS deve essere in grado sempre più di supportare terminali e interfacce lato client multiple e diversificate.
Inoltre il POS da realizzare è un sistema commerciale, destinato a essere venduto a diversi clienti, con esigenze differenti rispetto alla gestione delle regole di business. Ciascun cliente vorrà un proprio insieme unico di logica da eseguire.
Utilizzando una strategia di sviluppo iterativo, si procederà attraverso i requisiti, l'analisi e la progettazione orientata agli oggetti e l'implementazione.

Capitolo 5

Iterazione 0: analisi dei requisiti

5.1 Iterazione 0 e i suoi obiettivi

L'iterazione 0 è un passo iniziale che permette di stabilire una visione comune e una portata di base per il progetto. Questa iterazione per gli studi di caso enfatizza i concetti fondamentali dell'analisi dei requisiti. Questo passo iniziale corrisponde alla fase di ideazione di UP. Si tratta infatti di una iterazione diversa dalle altre, con lo scopo di avviare le attività e gli elaborati tra cui anche l'analisi dei requisiti.

5.2 Processo: Ideazione

La maggior parte dei progetti richiede un breve passo iniziale in cui si esaminano delle domande fondamentali, sulle risorse e sui tempi.
In UP, l'ideazione è il passo iniziale che permette di definire la visione del progetto e di ottenere una stima dell'ordine di grandezza dei costi e dei tempi di sviluppo. A tal fine, si deve iniziare a esplorare i requisiti funzionali e i non funzionali più critici. Tuttavia, ***lo scopo non è quello di definire tutti i requisiti***, nè di generare una stima o un piano di progetto affidabili.

5.2.1 La durata dell'ideazione

Lo scopo dell'ideazione è stabilire una visione comune per gli obiettivi del progetto e deve essere una fase particolarmente breve, che comprende il primo workshop sui requisiti e la pianificazione per la prima iterazione, per poi passare direttamente all'elaborazione.

5.3 Elaborati iniziati durante l'ideazione

La Tabella 5.1 elenca gli elaborati iniziati nell'ideazione (o nell'elaborazione) e indica le problematiche affrontate. Si noti che l'ideazione può comportare un certo lavoro di programmazione, volto a creare dei prototipi "proof of concept".

Figura 5.1: alcuni elaborati per l'ideazione

Elaborato[1]	Commento
Visione e Studio economico	Descrive gli obiettivi e i vincoli di alto livello, lo studio economico, e fornisce un sommario del progetto.
Modello dei Casi d'Uso	Descrive i requisiti funzionali. Durante l'ideazione vengono identificati i nomi della maggior parte dei casi d'uso, e circa il 10% dei casi d'uso viene analizzato in modo dettagliato.
Specifiche supplementari	Descrivono altri requisiti, per lo più non funzionali. Durante l'ideazione è utile avere un'idea dei requisiti non funzionali fondamentali che avranno un impatto significativo sull'architettura.
Glossario	Terminologia chiave del dominio e dizionario dei dati.
Lista dei Rischi e Piano di Gestione dei Rischi	Descrive i rischi (aziendali, tecnici, di risorse, di calendario) e le idee per attenuarli o rispondervi.
Prototipi e proof of concept	Per chiarire la visione e validare tecniche.
Piano dell'Iterazione	Descrive che cosa fare nella prima iterazione dell'elaborazione.
Piano delle Fasi e Piano di Sviluppo del Software	Ipotesi (poco precise) riguardo alla durata e allo sforzo della fase di elaborazione. Strumenti, persone, formazione e altre risorse.
Scenario di Sviluppo	Una descrizione della personalizzazione dei passi e degli elaborati di UP per questo progetto; UP viene sempre personalizzato per il progetto.

5.3.1 La documentazione è troppa?

Si tenga presente che gli elaborati di UP vanno considerati opzionali. Bisogna scegliere di creare solo quelli che aggiungono realmente valore al progetto, e scartare quelli il cui valore non è dimostrato. In questa fase, è necessario avere solo documenti preliminari e approssimativi.
Inoltre, spesso lo scopo della creazione degli elaborati e dei modelli non è il documento o il diagramma in sé, ma il pensare, l'analizzare e la sollecitudine fattiva.

5.4 Ideazione e UML

Non è detto che si fa molto uso di diagrammi, tranne forse che dei semplici diagrammi UML dei casi d'uso. La maggior parte dei diagrammi di UML avverrà nella fase successiva, quella di elaborazione.

5.5 Processo: Envisioning (Scrum)

Le iterazioni di Scrum, a differenza di UP, non sono organizzate in fasi. In ogni caso, anche un progetto Scrum prevede un breve passo iniziale da svolgere prima del primo Sprint, chiamato **envisioning**, con obiettivi simili alla fase di ideazione di UP. VIene inoltre creata una versione iniziale del product backlog, con delle stime *approssimative*.

Capitolo 6

Requisiti evolutivi

6.1 Che cosa sono i requisiti

Ogni sistema software ha lo scopo di risolvere un determinato problema, il sistema deve di solito fornire un certo numero di *funzionalità*, relative alla gestione di alcune tipologie di *informazioni*. Inoltre, il sistema deve possedere alcune caratteristiche di *qualità*. Un **requisito** è una capacità o una condizione a cui il sistema, e più in generale il progetto, deve essere conforme.
I requisiti derivano da richieste degli utenti del sistema o di altri parti interessate, per risovlere dei problemi e per raggiungere degli obiettivi. Inoltre, i requisiti vengono di solito descritti anche in un contratto, uno standard, una specifica o altro documento formale, affinchè il sistema li soddisfi. Ci sono due tipi principali di requisiti:

- **requisiti funzionali** descrivono il comportamento del sistema, in termini di *funzionaliltà* fornite ai suoi utenti. Sono di solito orientati all'uso del sistema e possono essere rappresentati sotto forma di casi d'uso. Comprendono anche gli aspetti relativi alle *informazioni* che il sistema deve gestire

- **requisiti non funzionali** non riguardano le specifiche funzioni del sistema, ma sono relativi a proprietà del sistema nel suo complesso, come per esempio sicurezza, prestazioni, scalabilità, usabilità...

Una sfida primaria nell'analisi dei requisiti è trovare, comunicare e ricordare (il che di solito significa scrivere) ciò che è realmente necessario, in una forma che parli chiaramente al cliente e ai membri del team di sviluppo.

6.2 Requisiti evolutivi e a cascata

Un aspetto fondamentale dell'analisi dei requisiti è la gestione di requisiti *che cambiano*. In UP e in altri metodi evolutivi, si iniziano la programmazione di qualità produzione e il test molto tempo prima che la maggior parte dei requisiti siano stati analizzati o specificati.
Si può dimostrare che una minor progettazione ha effetti positivi, ma questo non implica che il modo giusto di procedere sia iniziare di gran carriera a scrivere codice il primo giorno del progetto, dimenticandosi dell'analisi o della scrittura dei requisiti. In realtà esiste una via intermedia: l'analisi dei requisiti iterativa ed evolutiva combinata con uno sviluppo anticipato, iterativo e timeboxed, nonché partecipazione, valutazioni e feedback dei risultati parziali frequenti da parte delle parti interessate.

6.3 Come trovati i requisiti

UP accoglie qualsiasi metodo di acquisizione dei requisiti che possa aggiungere valore e aumentare la partecipazione degli utenti. Anche la semplice pratica di XP delle "storie" è accettabile in un progetto UP, se può essere applicata in modo efficace (richiede la presenza a tempo pieno di un cliente o di un esperto; è una pratica eccellente, ma spesso difficile da realizzare).

6.4 Tipi e categorie di requisiti

Nella gestione dei requisiti, di solito è utile utilizzare un qualche schema di classificazione per la copertura dei requisiti. Ecco alcune categorie principali di requisiti:

1. **funzionale** requisiti funzionali, caratteristiche principali e capacità funzionali

2. **usabilità** riguarda aspetti legati alla facilità d'uso del sistema

3. **affidabilità** riguarda caratteristiche come la disponibilità, la tolleranza ai guasti e ai fallimenti

4. **prestazioni** riguarda caratteristiche come tempi di risposta, throughput, capacità e uso delle risorse

5. **sicurezza** rigurda la capacità del sistema di resistere ad usi non autorizzati, ma anche di poter essere utilizzato da utenti legittimi

6. **sostenibilità** è l'abilità del sistema di essere facilmente modificabile per consentire miglioramenti e riparazioni.

Alcuni dei requisiti non funzionali sono chiamati **attributi di qualità** (o **requisiti di qualità**) del sistema.

6.5 Requisiti ed elaborati di UP

UP offre diversi elaborati dei requisiti che, come molti degli elaborati di UP, sono opzionali. I principali sono:

- **modello dei casi d'uso** insieme di scenari tipici dell'utilizzo di un sistema. Usato principalmente per i requisiti funzionali
- **specifiche supplementari** essenzialmente, tutto quello che non rientra nei casi d'uso
- **glossario** definisce i termini significativi, avendo anche il ruole di **dizionario dei dati**
- **visione** riassume i requisiti ad alto livello che sono dettagliati nel modello dei casi d'uso
- **regole di business** descrivono di solito i requisiti o le politiche che trascendono un unico progetto software. Un ottimo esempio è dato dalle leggi fiscali dello stato.

6.5.1 Qual è il formato corretto degli elaborati?

In UP, tutti gli elaborati sono astrazioni di informazioni e possono essere memorizzate in un qualsiasi formato.

Capitolo 7

Casi d'uso

7.1 Cosa sono

I casi d'uso sono storie scritte, ampiamente utilizzati per scoprire e registrare i requisiti. Essi influenzano molti aspetti di un progetto, compresa l'analisi e la progettazione orientata agli oggetti.

7.2 Definizione: attore, scenario e caso d'uso

Inanzitutto, alcune definizioni preliminari: un **attore** è qualcosa o qualcuno dotato di comportamento, come una persona o un sistema.
Uno **scenario**, o **istanza di caso d'uso**, è una sequenza specifica di azioni e interazioni tra il sistema e alcuni attori.
Un **caso d'uso** è una collezione di scenari correlati, sia di successo che di fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo specifico.

7.3 Modello dei Casi d'Uso

Si tratta dell'insieme di tutti i casi d'uso scritti; è un modello delle funzionalità del sistema e del suo ambiente. **I casi d'uso sono documenti di testo, non diagrammi, e la modellazione dei casi d'uso è innanzitutto un atto di scrittura di testi, non di disegno di diagrammi.** Può comunque includere, opzionalmente, un diagramma UML dei casi d'uso che mostra i nomi dei casi d'uso e degli attori. I casi d'uso non sono orientati agli oggetti; quando li si scrive, non si sta facendo analisi OO.

7.4 Motivazione: perchè i casi d'uso

Molti metodi d'analisi sono troppo complessi, e vengono compresi dagli analisti ma creano confusione nell'uomo d'affari medio. Il mancato coinvolgimento dell'utente nei progetti software è quasi al primo posto tra i motivi di fallimento dei progetti. I casi d'uso sono un buon metodo per mantenere la semplicità e consentire agli esperti di dominio o a chi può contribuire ai requisiti di scrivere essi stessi i casi d'uso, o perlomeno partecipare alla loro scrittura. Un altro valore dei casi d'uso è che *mettono in risalto gli obiettivi degli utenti* e il loro punto di vista; i casi d'uso costituiscono una risposta alle domande.

7.5 I casi d'uso sono requisiti funzionali

I casi d'uso sono requisiti, soprattutto requisiti funzionali o comportamentali, che indicano che cosa deve fare il sistema. Oltre agli aspetti funzionali, i casi d'uso possono essere utilizzati anche per altri tipi di requisiti, soprattutto se questi sono fortemente correlati a un caso d'uso. Un punto di vista correlato è che un caso d'uso definisce un *contratto* relativo al comportamento di un sistema.

7.6 Tipi di attori

Un attore è qualcosa o qualcuno dotato di comportamento. Anche il sistema in discussione (o SuD, da *system under discussion*) stesso è considerato un attore, quando ricorre ai servizi di altri sistemi. Ci sono diversi tipi di attori:

- **attore primario** utilizza direttamente i servizi del SuD, affinchè vengano raggiunti degli obiettivi utente
- **attore finale** vuole che il SuD sia utilizzato affinchè vengano raggiunti dei suoi obiettivi
- **attore di supporto** offre un servizio al SuD. Spesso è un sistema informatico
- **attori fuori scena** ha un interesse nel comportamento del caso d'uso, ma non è un attore primario, finale o di supporto

7.7 Tre formati comuni per i casi d'uso

I casi d'uso possono essere scritti usando diversi formati e livelli di formalità:

- **formato breve** riepilogo conciso di un solo paragrafto, solitamente relativo al solo scenario principale di successo
- **formato informale** più paragrafi, scritti in modo informale, relativi a vari scenari
- **formato dettagliato** tutti i passi e le variazioni sono scritti nel dettaglio, definiendo anche pre-condizioni e post-condizioni

Figura 7.1: template di un caso d'uso

Sezione del caso d'uso	Commento
Nome del caso d'uso	Inizia con un verbo.
Portata	Il sistema che si sta progettando.
Livello	"Obiettivo utente" o "sottofunzione".
Attore primario	Usa direttamente il sistema; gli chiede di fornirgli i suoi servizi per raggiungere un obiettivo.
Parti interessate e Interessi	A chi interessa questo caso d'uso e che cosa desidera.
Pre-condizioni	Che cosa deve essere vero all'inizio del caso d'uso – e vale la pena di dire al lettore.
Garanzia di successo	Che cosa deve essere vero se il caso d'uso viene completato con successo – e vale la pena di dire al lettore.
Scenario principale di successo	Uno scenario comune di attraversamento del caso d'uso, di successo e incondizionato.
Estensioni	Scenari alternativi, di successo e di fallimento.
Requisiti speciali	Requisiti non funzionali correlati.
Elenco delle varianti tecnologiche e dei dati	Varianti nei metodi di I/O e nel formato dei dati.
Frequenza di ripetizione	Frequenza prevista di esecuzione del caso d'uso.
Varie	Altri aspetti, come per esempio i problemi aperti.

7.8 Sezioni di un caso d'uso dettagliato

Elementi di preambolo

Il preambolo è composto da tutto ciò che precede lo scenario principale e le estensioni. Contiene informazioni che è importante leggere prima degli scenari del caso d'uso

Portata

La portata descrive i confini del sistema (o dei sistemi) in via di progettazione. Normalmente un caso d'uso descrive l'utilizzo di un sistema software.

Livello

I casi d'uso vengono classificati con un livello; possibili livelli, tra gli altri, sono il livello obiettivo utente e il livello sottofunzione. Un caso d'uso **a livello di obiettivo utente** è il tipo più comune, che descrive gli scenari con cui un attore primario può portare a termine il suo lavoro, raggiungendo degli obiettivi. Un caso d'uso **a livello di sottofunzione** descrive invece dei sottopassi, richiesti come supporto al raggiungimento di un obiettivo dell'utente.

Attore finale e primario

L'attore finale è l'attore che vuole raggiungere un obiettivo, e questo richiede l'esecuzione dei servizi del sistema. L'attore primario è l'attore che usa direttamente il sistema. Di solito l'attore primario coincide con l'attore finale.

Elenco delle parti interessate e degli interessi

Questo elenco è più importante e pratico di quanto non possa sembrare a prima vista. Infatti le parti interessate e i loro interessi suggeriscono e limitano ciò che il sistema deve fare.

Pre-condizioni e garanzie di successo (post-condizioni)

Le **pre-condizioni** descrivono che cosa *deve essere sempre* vero prima di iniziare uno scenario del caso d'uso. Le pre-condizioni *non* vengono verificate all'interno del caso d'uso; piuttosto, si presume che siano condizioni vere. Le **garanzie di successo** (o **post-condizioni**) affermano che cosa deve essere vero quando è stato completato con successo il caso d'uso, ovvero lo scenario

principale di successo o un percorso alternativo. La garanzia deve soddisfare le esigenze di tutte le parti interessate.

Scenario principale di successo e i suoi passi (o Flusso di base)

Esso descrive un percorso di successo comune che soddisfa gli interessi delle parti interessate. Lo scenario principale è costituito da una sequenza di passi, che può contenere passi da ripetere più volte, ma che di solito non comprende alcuna condizione o diramazione.

Estensioni (o Flussi alternativi)

In genere un caso d'uso è composto da molti scenari, che possono essere decine, centinaia o anche più. Lo scenario principale è uno solo di questi scenari. Le estensioni hanno lo scopo di descrivere tutti gli altri scenari, sia di successo che di fallimento. In pratica si descrivono i diversi scenari sotto forma di diramazioni (ovvero, deviazioni temporanee) da uno scenario di base.

Requisiti speciali

Se un requisito non funzionale, un attributo di qualità o un vincolo si riferiscono in modo specifico a un caso d'uso, allora è bene scriverlo insieme al caso d'uso. Se un requisito non funzionale, un attributo di qualità o un vincolo si riferiscono in modo specifico a un caso d'uso, allora è bene scriverlo insieme al caso d'uso.

Elenco delle varianti tecnologiche e dei dati

Spesso ci sono variazioni tecnologiche nel modo in cui qualcosa deve essere fatto, ma non nella sostanza delle cose; è importante registrare tali varianti nel caso d'uso. Un esempio comune è un vincolo tecnico, imposto da una parte interessata, relativo alle tecnologie di input o di output.

7.9 Scrivere casi d'uso a scatola nera

Non descrivono il funzionamento interno del sistema, i suoi componenti o aspetti relativi al suo progetto. Piuttosto, il sistema è descritto come dotato di *responsabilità*.

7.10 Come trovare i casi d'uso

I casi d'uso hanno come scopo quello di soddisfare gli obiettivi degli attori primari. Quindi, la procedura di base per trovarli è la seguente:

1. scegliere i confini del sisema
2. identificare gli attori primari
3. identificare gli obiettivi di ciascun attore primario
4. definire i casi d'uso che soddisfano gli obiettivi degli utenti

7.11 Stabilire i confini del sistema

Per chiarire la definizione dei confini del sistema in corso di progettazione, è utile sapere che gli attori primari e gli attori di supporto sono considerati esterni al sistema. Una volta identificati gli attori esterni, i confini diventano più chiari. Il confine, è il sistema.

7.12 Verificare l'utilità di un caso d'uso

7.12.1 Test del capo

Il capo vi chiede: "Cosa avete fatto tutto il giorno?" e voi rispondete: "Il login!". Il vostro capo sarà felice? Se non lo è, il caso d'uso non supera il test del capo, il che significa che non è fortemente mirato a ottenere risultati il cui valore sia misurabile. Potrebbe essere un caso d'uso a un livello più basso, ma non al livello a cui è desiderabile concentrarsi durante l'analisi dei requisiti.

7.12.2 Tets EBP

La nozione di **Processo di Business Elementare** (Elementary Business Process o EBP) deriva dal settore dell'ingegneria dei processi di business, ed *è un'attività svolta da una persona in un determinato tempo e luogo, in risposta a un evento di business*.
La definizione non va presa in modo troppo letterale: un caso d'uso fallisce come EBP se sono necessarie due persone, oppure se una persona deve andare in giro? Probabilmente no, ma il tono della definizione è più o meno corretto.

7.12.3 Test della dimensione

Un caso d'uso è raramente costituito da una singola azione o passo; normalmente comprende diversi passi, e nel suo formato dettagliato spesso richiede da 3 a 10 pagine di testo. Un errore comune nella modellazione dei casi d'uso è definire un caso d'uso a sé formato da un singolo passo, all'interno di una sequenza di passi correlati.

7.13 Diagramma dei casi d'uso

UML fornisce una notazione dei diagrammi dei casi d'uso che consente di illustrare i nomi e gli attori dei casi d'uso, nonché le relazioni tra gli stessi. Un diagramma dei casi d'uso rappresenta un ottimo quadro del contesto del sistema; esso costituisce un buon **diagramma di contesto**, che consiste nel mostrare i confini del sistema, ciò che giace al suo esterno e come esso viene utilizzato.

7.14 Diagrammi di attività

UML comprende un tipo di diagrammi utile per visualizzare i flussi di lavoro e i processi di business: i diagrammi di attività. Poiché i casi d'uso comportano l'analisi dei processi e dei flussi di lavoro, questi diagrammi possono costituire un'alternativa utile o complementare alla scrittura del testo dei casi d'uso.

7.15 Elenchi delle caratteristiche

Può essere utile riassumere le funzionalità del sistema con un elenco di caratteristiche di alto livello, chiamato *caratteristiche di sistema*, come parte del documento di Visione.

Talvolta i casi d'uso non sono adatti; alcune applicazioni richiedono un punto di vista guidato dalle caratteristiche. I casi d'uso non sono idonei a questi tipi di sistemi software e al modo in cui essi si evolvono in termini di forze del mercato. Per questo si usano gli elenchi delle carattestiche.

7.16 Casi d'uso e metodi iterativi

UP incoraggia che la scrittura dei casi d'uso avvenga durante i workshop dei requisiti. Non tutti i casi d'uso vengono scritti nel formato dettagliato durante la fase di ideazione. Piuttosto, si supponga che venga svolto un workshop dei requisiti di due giorni all'inizio del progetto. La maggior parte dei casi d'uso interessanti, complessi o rischiosi viene scritta in formato breve, dedicando circa un paio di minuti alla scrittura di ciascun caso d'uso. In ogni breve workshop successivo viene adattata e raffinata la visione sui requisiti principali, che sarà instabile nelle prime iterazioni, ma si stabilizzerà nelle successive. Dunque, c'è un'interazione, mutua e iterativa, tra la scoperta dei requisiti e l'implementazione delle parti del software.

Capitolo 8

Altri requisiti

8.1 Glossario

Nella sua forma più semplice il Glossario è un elenco dei termini significativi e delle relative definizioni.
In UP, il Glossario svolge anche il ruolo di un **dizionario dei dati**, un documento che registra dati relativi ad altri dati, ovvero **metadati**.

8.2 Regole di Dominio

Le regole di dominio stabiliscono come possono funzionare un dominio o un business. Non si tratta di requisiti di una singola applicazione, anche se i requisiti di un'applicazione sono spesso influenzati dalle regole di dominio. Esempi di regole di dominio comuni sono le politiche aziendali, le leggi della fisica (per esempio, come scorre il petrolio sottoterra) e le leggi governative. Le regole di dominio vengono più comunemente chiamate **regole di business**.

Capitolo 9

Correlare i casi d'uso

9.1 La relazione di inclusione

La relazione di inclusione (*include*) è la relazione tra casi d'uso più comune e importante. È comune avere dei comportamenti parziali comuni tra diversi casi d'uso. Anziché duplicare il comportamento, è opportuno separarlo in un proprio caso d'uso a livello di sottofunzione e indicare la sua inclusione. Si tratta semplicemente di un refactoring e di un collegamento del testo per evitare la duplicazione.

Un altro utilizzo della relazione include è per descrivere la gestione di un evento asincrono, come quando un utente è in grado, in qualsiasi momento, di selezionare o passare a una finestra, una funzione o una pagina particolare web, oppure all'interno di una serie di passi.

9.2 Concreto, astratto, base e aggiunto

Un **caso d'uso concreto** viene iniziato da un attore ed esegue l'intero comportamento desiderato dall'attore. Al contrario, un **caso d'uso astratto** non viene mai istanziato per sé, si tratta di un caso d'uso sottofunzione che fa parte di un altro caso d'uso. Un caso d'uso che include un altro caso d'uso, oppure che è esteso o specializzato da un altro caso d'uso, si chiama **caso d'uso base**. D'altra parte, il caso d'uso che è un'inclusione, un'estensione o una specializzazione è chiamato **caso d'uso aggiunto**.

9.3 La relazione di estensione

Si supponga che il testo di un caso d'uso non debba essere modificato (almeno non in modo significativo) per qualche ragione, è stato stabilizzato come elaborato stabile e non può essere toccato. Come aggiungere qualcosa al caso d'uso senza modificarne il testo originale?
La relazione di estensione (*extend*) offre una risposta. L'idea è quella di creare un caso d'uso di estensione o aggiunto, e al suo interno descrivere dove e sotto quali condizioni esso estende il comportamento di qualche caso d'uso base.

Capitolo 10

Iterazione 1: concetti fondamentali

10.1 Requisiti per l'iterazione 1

L'iterazione 1 per questi studi di caso enfatizza un insieme di capacità fondamentali dell'analisi e della progettazione a oggetti.

10.2 Processo: Ideazione e Elaborazione

In termini di UP e degli studi di caso trattati, si immagini di aver terminato la fase di ideazione e che stia per iniziare la fase di elaborazione.

10.2.1 Cosa è successo durante l'ideazione

L'ideazione è un passo breve verso l'elaborazione. Essa determina la fattibilità, il rischio e la portata di base, al fine di stabilire se il progetto merita un'indagine più seria. Alcune attività ed elaborati probabili dell'ideazione sono i seguenti:

- un breve workshop dei requisiti
- definizione degli attori, scrittura della maggior parte dei casi d'uso e identificazione dei requisiti
- lista dei rischi, proof-of-concept, prototipi di interfaccia utente

10.2.2 Verso l'elaborazione

L'elaborazione è la serie iniziale di iterazioni durante le quali, in un progetto normale:

- viene programmato e verificato il nucleo, rischioso, dell'architettura software;
- viene scoperta e stabilizzata la maggior parte dei requisiti
- i rischi maggiori sono attenuati o rientrano

L'elaborazione è la serie iniziale di iterazioni durante le quali il team esegue un'indagine seria, implementa (scrivendo il codice e facendo test) il nucleo dell'architettura, chiarisce la maggior parte dei requisiti e affronta le problematiche di alto rischio; è spesso costituita da due o più iterazioni e durante questa fase non vengono creati prototipi "usa e getta"; al contrario, il codice e la progettazione sono parti di qualità produzione del sistema finale. In alcune descrizioni di UP, per descrivere il sistema parziale viene utilizzato il termine "**prototipo dell'architettura**".

10.3 Pianificare l'iterazione successiva

I requisiti e le iterazioni vanno organizzati in base a rischio, copertura e criticità.

- **rischio** comprende tanto la complessità tecnica quanto altri fattori, come l'incertezza dello sforzo o l'usabilità
- **copertura** implica che le iterazioni iniziali prendano in considerazione tutte le parti principali del sistema, probabilmente con un'implementazione "in ampiezza e poco profonda" di molti componenti
- **criticità** si riferisce alle funzioni che il cliente considera di elevato valore di business

Questi criteri sono utilizzati per collocare il lavoro nelle iterazioni. Ai casi d'uso o ai loro scenari viene dato un "voto" per l'implementazione; le iterazioni iniziali implementano scenari con voto elevato. Inoltre, alcuni requisiti sono espressi come caratteristiche di alto livello, non correlate a un particolare caso d'uso, come per esempio il servizio di logging. Anche alle caratteristiche viene dato un voto.

Capitolo 11

Verso l'analisi a oggetti

11.1 Analisi a oggetti

L'analisi è un'attività distinta dalla progettazione, che invece enfatizza una soluzione al problema, dunque va svolta in modo indipendente dalle possibili soluzioni progettuali.
L'analisi avviene solitamente modellando tre aspetti di un sistema: le informazioni da gestire, le funzioni e il comportamento. Pertanto, è utile analizzare e modellare le interazioni fra gli attori e il sistema, ovvero le **funzioni** che il sistema è chiamato a svolgere durante il suo uso, e il **comportamento** del sistema, ovvero i cambiamenti nelle informazioni associati a ciascuna funzione. Ogni applicazione software deve solitamente gestire dei dati di interesse nell'ambito di un determinato dominio applicativo. Pertanto è utile analizzare e modellare il **dominio informativo**, ovvero le tipologie di informazioni che il sistema deve rappresentare e gestire. Nell'analisi orientata agli oggetti, questi tre aspetti vengono modellati come segue:

- il dominio informativo è rappresentato mediante un modello a oggetti, che descrive i concetti del dominio e le relazioni
- le funzioni sono rappresentate dalle operazioni che il sistema è chiamato a svolgere
- il comportamento è descritto come l'effetto prodotto dall'esecuzione di ciascuna operazione di sistema

Durante l'analisi, funzioni e comportamento vengono modellati in un modo che, come vedremo più avanti, sarà utile per la progettazione a oggetti.

Capitolo 12

Modellazione di dominio

12.1 Che cos'è un modello di dominio

Un modello di dominio è una rappresentazione *visuale* di classi concettuali o di oggetti del mondo reale, nonchè delle relazioni tra di essi, in un dominio di interesse.
Nell'UP, è un modello incentrato sulla spiegazione di cose e prodotti importanti per un dominio di business. Ciò significa che è incentrato su un solo dominio.
Applicando la notazione UML, un modello di dominio può essere realizzato in pratica come uno o più **diagrammi delle classi** in cui non sono definite operazioni e adottando un *punto di vista concettuale*.

Una **classe concettuale** rappresenta una cosa o un concetto del dominio di interesse, ovvero un insieme di cose o oggetti con caratteristiche simili.

Un'**associazione** rappresenta una relazione ra gli oggetti di due classi.

Un **attributo** rappresenta una proprietà elementare degli oggetti di una classe.

12.2 Classi, associazioni e attributi

12.2.1 Classi concettuali

Una classe concettuale è, informalmente, un'idea, una cosa o un oggetto. Può essere formalizzata in termini del suo simbolo, intensione e estensione.

Per definirli:

- **simbolo** è una parola o un'immagine usata per rappresentare la classe concettuale
- **intensione** è la definizione della classe concettuale
- **estensione** è l'insieme degli oggetti descritti dalla classe concettuale

In UML una *classe* è definita, in generale, come "il descrittore per un insieme di oggetti che possiedono le stesse caratteristiche".
In un diagramma delle classi di UML, una classe viene mostrata come unrettangolo suddiviso in più sezioni, la più in alto riporta il nome della classe, la seconda sezione i suoi attributi.

12.2.2 Associazioni

Un'associazione è una relazione tra classi che indica una connessione significativa e interessante.
In un diagramma delle classi di UML, un'associazione viene mostrata come una linea che collega le classi partecipanti, il suo nome viene scritto accanto alla linea.
Le estremità di un'associazione possono contenere un'espressione di molteplicità, che indica le relazioni numeriche tra le istanze delle classi.

12.2.3 Attributi

Un attributo è un valore logico degli oggetti di una classe.
In un diagramma delle classi di UML, gli attributi sono mostrati nella seconda sezione del rettangolo per una classe.

12.3 I Ruoli

ciascuna estremità di associazione è anche chiamata un ruolo, che possono avere espressioni di molteplicità, nome e navigabilità.
La molteplicità di un ruolo definisce quante istanze di una classe possono essere associate a un'istanza di un'altra.

12.4 Aggregazione e composizione

L'**aggregazione** è, in UML, un tipo di associazione che suggerisce, in modo vago e approssimativo, una relazione intero-parte.

La **composizione**, nota anche come **aggregazione composta**, è un tipo forte di aggregazione intero-parte che è utile da mostrare in alcuni modelli. La composizione implica che ciascuna istanza della parte appartiene a una sola istanza del composto alla volta e che la vita delle parti è limitata da quella del composto.

12.5 Diagramma degli oggetti di dominio

Un diagramma degli oggetti è un modello che mostra un insieme di oggetti, con i loro attributi e le loro relazioni, in un dato momento. In pratica, un diagramma degli oggetti è simile a un diagramma delle classi, ma mostra oggetti (ovvero, istanze di classi) anziché classi, collegamenti (istanze di associazioni) anziché associazioni e valori (istanze di attributi) anziché attributi.

12.6 Modelli concettuali nello sviluppo software

Nello sviluppo del software l'uso dei modelli concettuali è piuttosto diffuso. Tuttavia, non sempre i modelli concettuali vengono creati con le stesse finalità o nello stesso modo dei modelli di dominio presentati in questo capitolo.

Capitolo 13

Operazioni di sistema e diagrammi di sequenza di sistema

13.1 Eventi e operazioni di sistema

I casi d'uso descrivono il modo in cui gli attori esterni interagiscono con il sistema software che interessa creare. Durante questa interazione, un attore genera degli **eventi di sistema**, che costituiscono un input per il sistema, di solito per richiedere l'esecuzione di alcune **operazioni di sistema**, che sono operazioni che il sistema deve definire proprio per gestire tali eventi.
Più precisamente, in UML un **evento** è qualcosa di importante o degno di nota che avviene durante l'esecuzione di un sistema. Inoltre, in UML un'**operazione** rappresenta una trasformazione oppure un'interrogazione che un oggetto o componente può essere chiamato a eseguire.

13.2 Diagrammi di sequenza di sistema

UML fornisce la notazione dei **diagrammi di sequenza** per illustrare interazioni tra attori e le operazioni iniziate da essi. Un **diagramma di sequenza di sistema** è una figura che mostra, *per un particolare scenario di un caso d'uso*, gli eventi generati dagli attori esterni al sistema, il loro ordine e gli eventi inter-sistema. Tutti i sistemi sono considerati a scatola nera.

13.3 Perchè disegnare un SSD

È utile sapere, *con precisione*, quali sono gli eventi esterni di input, ovvero gli **eventi di sistema**. Essi rappresentano una parte importante dell'analisi delle funzioni e del comportamento di un sistema.

13.4 Relazione tra SSD e casi d'uso

Un SSD mostra gli eventi di sistema *per un solo scenario di un caso d'uso*, e può essere generato per ispezione da tale scenario. In particolare, mostra l'attore primario, il sistema in discussione, nonchè i passi che rappresentano le interazioni tra il sistema e l'attore. Le interazioni iniziate dall'attore primario nei confronti del sistema sono mostrate come messaggi con parametri. Si noti che **un SSD mostra le interazioni implicate dal testo di un caso d'uso**, ma non mostra le azioni eseguite dal sistema.

13.5 Assegnare il nome a eventi e operazioni

Gli eventi di sistema dovrebbero essere espressi a un livello astratto, di intenzione, anziché in relazione al dispositivo di input fisico.
I parametri per gli eventi di sistema sono preferibilmente dei valori di un tipo di dato; per esempio, un numero, il codice identificativo di un articolo oppure una data. Bisogna invece evitare parametri che sono degli oggetti o dei concetti complessi, come per esempio una descrizione prodotto. Per quanto riguarda le risposte del sistema, vanno di solito mostrate come un elenco di dati o informazioni restituite dal sistema (e non come azioni o operazioni).

13.6 SSD e UP

Gli SSD fanno parte del Modello dei Casi d'Uso, dato che sono una visualizzazione delle interazioni implicate negli scenari dei casi d'uso.
La maggior parte degli SSD viene creata durante l'elaborazione, quando è utile identificare i dettagli degli eventi di sistema per chiarire quali sono le principali operazioni che il sistema deve essere in grado di gestire, scrivere i contratti delle operazioni di sistema e possibilmente supportare le stime.

Capitolo 14

Contratti delle operazioni di sistema

14.1 Contratto

I contratti delle operazioni usano pre-condizione e post-condizione per descrivere nel dettaglio i cambiamenti agli oggetti in un modello di dominio, come risultato dell'esecuzione di un'operazione di sistema. Il modello di dominio è il modello di OOA più diffuso, ma i contratti delle operazioni e i modelli a stati possono essere elaborati dell'OOA altrettanto utili.

14.2 Le sezioni di un contratto

Il seguente schema mostra una descrizione delle sezioni di un contratto.

- **operazione** nome e parametri dell'operazione
- **riferimenti** casi d'uso in cui può verificarsi questa operazione
- **pre-condizioni** ipotesi significative sullo stato del sistema o degli oggetti nel modello di dominio prima dell'esecuzione dell'operazione. Si tratta di ipotesi non banali, che dovrebbero essere comunicate al lettore
- **post-condizioni** è la sezione più importante. Descrive i cambiamenti di stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione.

14.3 Che cos'è un'operazione di sistema

Le operazioni di sistema sono operazioni che il sistema, considerato come un componente a scatola nera, offre nella sua interfaccia pubblica. Possono essere individuati negli SSD, dato che mostrano eventi di sistema. Un evento di sistema di input implica che il sistema definisca un'operazione di sistema per gestire quell'evento, così come, nella programmazione OO, un messaggio (che è un tipo di evento o segnale) viene gestito da un metodo (che è un tipo di operazione).
L'intero insieme delle operazioni di sistema, tra tutti i casi d'uso, definisce l'**interfaccia di sistema** pubblica, considerando il sistema come un singolo componente o una singola classe.

14.4 Utilità dei contratti

In UP, i casi d'uso sono il repository principale dei requisiti del progetto. Essi possono fornire tutti i dettagli, o la maggior parte di essi, necessari per sapere che cosa fare durante la progettazione; in questo caso i contratti non sono d'aiuto. Tuttavia ci sono situazioni in cui non è opportuno descrivere nei casi d'uso tutti i dettagli e la complessità dei cambiamenti di stato richiesti, poiché risulterebbero troppo dettagliati.

14.5 Come creare e scrivere contratti

Per creare i contratti si procede come segue:

1. identificare le operazioni di sistema dagli SSD
2. creare un contratto per le operazioni di sistema complesse o i cui effetti sono probabilmente sottili, o che non sono chiare dei casi d'uso
3. descrivere le post-condizioni suddividendole nelle categorie:
    - creazione o cancellazione di oggetto
    - formazioni o rottura di collegamento
    - modifica di attributo

14.5.1 Scrivere i contratti

Le post-condizioni vanno scritte in una forma dichiarativa, con una forma verbale al passato e passiva. Ci si ricordi inoltre di stabilire i collegamenti necessari tra oggetti esistenti e appena creati.

14.6 UML e contratti

UML definisce formalmente la nozione di operazione come una *specifica di una trasformazione o di un'interrogazione che un oggetto può essere chiamato ad eseguire*.
Per esempio, in termini di UML sono operazioni gli elementi di un'interfaccia. Un'operazione è un'astrazione, non un'implementazione. Al contrario, in UML un metodo è un'implementazione di un'operazione.
Nel metamodello di UML, un'operazione ha una firma (o signature, con nome e parametri) e, cosa più importante in questo contesto, è associata a un insieme di oggetti UML di tipo Constraint (Vincolo) classificati come pre-condizioni e post-condizioni che specificano la semantica dell'operazione.

Capitolo 15

Dai requisiti alla programmazione

15.1 Fare la cosa giusta e fare la cosa bene

Il lavoro successivo all'analisi dei requisiti si basa sul *fare la cosa bene*, ovvero sil progettare abilmente una soluzione che soddisfa i requisiti per questa iterazione.
Nello sviluppo iterativo, in ogni iterazione avviene il passaggio da un interesse centrato sui requisiti a uno incentrato su progettazione e implementazione.

15.1.1 Provocare il cambiamento all'inizio

I metodi iterativi ed evolutivi abbracciano il cambiamento, ma si cerca però di *provocare* questo cambiamento inevitabile nelle iterazioni *iniziali*, in modo da avere degli obiettivi più stabili per le iterazioni successive.

15.1.2 Richiesta di tempo

Quando sono state acquisite le capacità di scrittura dei casi d'uso, della modellazione di dominio e così via, il tempo richiesto per svolgere tutta la modellazione vera e propria esaminata finora è realisticamente di alcune ore o giorni.
Tuttavia, ciò non significa che siano trascorsi pochi giorni dall'inizio del progetto.

Capitolo 16

Architettura logica

L'architettura logica di un sistema software è l'organizzazione su larga scala delle classi software in package o namespace, sottoinsiemi e strati. È *logica* poichè non vengono prese decisioni su come questi elementi siano distribuiti. L'architettura di un sistema software può essere organizzata secondo diversi "stili", una architettura comune è quella **a strati**, e ciascuno strato è un gruppo di classi.

Gli strati di un'applicazione software comprendono normalmente i seguenti:

- **user interface** oggetto software per gestire l'interazione con l'utente
- **aplication logic** oggetti software che rappresentano concetti del dominio, che soddisfano i requisiti dell'applicazione
- **technical services** oggetti e sottoinsiemi d'uso generale che forniscono servizi tecnici di supporto, possono essere indipendenti dall'applicazione

In un'architettura **a strati stretta**, uno strato può solo richiamare i servizi dello strato immediatamente sottostante. Normalmente si utilizza però un'architettura **a strati rilassata**, in cui uno strato più alto può richiamare strati di diversi livelli più bassi.

16.1 Che cos'è un'architettura software

Viene definita come *l'insieme delle decisioni significative sull'organizzazione di un sistema software, la scelta degli elementi strutturali da cui è composto il sistema e delle relative interfacce, insieme al loro comportamento specificato dalle collaborazioni tra questi elementi, la composizione di questi elementi*

strutturali e comportamentali in sottosistemi via via più ampi, e lo stile architetturale che guida questa organizzazione.
Indipendentemente dalla definzione, si ha a che fare sempre con qualcosa in larga scala.

16.2 Diagramma dei package

L'architettura logica di un sistema può essere illustrata mediante un diagramma dei package di UML.
Il nome di un package può essere scritto sulla sua linguetta, se all'interno del package sono mostrati dei membri, oppure nella cartella principale, in caso contrario. È comune voler mostrare delle dipendenze tra i package, in modo che gli sviluppatori possano vedere l'accoppiamento su larga scala.
Un package UML rappresenta un **namespace**.

16.3 Progettazione con gli strati

Le idee dell'utilizzo degli strati sono semplici:

- organizzare la struttura logica su larga scala in strati separati con responsabilità distinte e correlate
- collaborazioni e accoppiamenti vanno dagli strati più alti a quelli più bassi, sono invece evitati accoppiamenti da strati bassi a strati più alti.

L'idea è descritta come **pattern layers** e produce un'**architettura a strati**.
L'uso degli strati contribuisce ad affrontare ed evitare diversi problemi, in sintesi l'obiettivo è la suddivisione di un sistema complesso in un insieme di elementi che possano essere sviluppati indipendentemente gli uni dagli altri.

16.4 Separazione Modello-Vista

Questo principio è costituito da almeno due parti:

1. gli oggetti non UI non devono essere connessi o accoppiati direttamente agli oggetti UI
2. non mettere logica applicativa nei metodi di un oggetto dell'interfaccia utente

In questo contesto, **modello** è un sinonimo per lo strato di oggetti del dominio, mentre **vista** è un sinonimo per gli oggetti del'interfaccia utente.

Il principio afferma che gli oggetti del modello non devono avere una conoscenza *diretta* degli oggetti della vista, almeno in quanto oggetti della vista. Un rilassamento legittimo di questo principio è il Pattern Observer, in cui gli oggetti del dominio inviano messaggi a oggetti della UI, visti solo indirettamente.

16.5 Legame tra SSD, operazioni di sistema e strati

Durante il lavoro di analisi, si abbozzano alcuni SSD per gli scenari dei casi d'uso. Sono stati identificati gli eventi di input nel sistema da parte di attori esterni, per richiedere l'esecuzione delle operazioni di sistema. Gli SSD mostrano queste operazioni di sistema, ma nascondono gli oggetti della UI. Ciò nonostante, saranno gli oggetti dello strato UI a catturare queste richieste di operazioni di sistema. In un'architettura a strati ben progettata, che sostiene una coesione alta e una separazione degli interessi, gli oggetti dello strato UI inoltreranno (o delegheranno) le richieste da parte dello strato UI allo strato del dominio (o allo strato Application, se presente), affinché vengano gestite.

Capitolo 17

Verso la progettazione a oggetti

17.1 Agile Modeling e il disegno leggero di UML

Alcuni degli scopi della modellazione agile sono *ridurre il costo aggiuntivo del disegno e modellare per comprendere e comunicare*, anzichè per documentare. È opportuno provare l'approccio semplice della modellazione agile.

17.2 Strumenti CASE per UML

I suggerimenti sull'abbozzo non devono sminuire l'importanza degli strumenti CASE per UML, poichè entrambi possono aggiungere valore.

17.3 Quanto tempo dedicare al disegno

Per un'iterazione con un timeboxing di tre settimane, si dedichino alcune ore o al massimo un giorno al disegno di UML, all'inizio dell'iterazione, insieme ai propri partner, "alla parete" o con uno strumento CASE per UML, per le parti difficili e creative della progettazione a oggetti dettagliata. Quindi ci si fermi e, nel caso di abbozzo, si scattino alcune fotografie digitali, si stampino le immagini e si passi alla codifica per il resto dell'iterazione, utilizzando i diagrammi UML come ispirazione da cui partire, ma riconoscendo che il progetto finale nel codice sarà diverso e dovrà essere migliorato. Nel corso dell'iterazione possono esserci altre sessioni di disegno/abbozzo più brevi.

17.4 Progettare gli oggetti: statico e dinamico

Ci sono due tipi di modelli per gli oggetti: dinamici e statici. I **modelli dinamici**, come i diagrammi di interazione aiutano a progettare la logica il comportamento del codice o il corpo dei metodi. I **modelli statici**, come quelli delle classi, aiutano a progettare la definizione dei package, dei nomi delle classi, degli attributi e delle firme dei metodi. C'è una relazione tra la modellazione statica e dinamica e la pratica della modellazione agile della *creazione di modelli in parallelo*.

17.5 Schede CRC

Una tecnica diffusa di modellazione orientata al testo è costituita dalle schede CRC, che sono dei piccoli fogli di cartoncino su cui vengono scritte le responsabilità e i collaboratori delle classi. Ciascuna scheda rappresenta una classe. Una sessione di modellazione CRC viene fatta da un gruppo di persone che discutono seguendo un modello "what if" ("cosa accadrebbe se") per gli oggetti, considerando che cosa devono fare e con quali altri oggetti devono collaborare.

Capitolo 18

Diagrammi di interazione

18.1 Interazioni e diagrammi di interazione

UML comprende i **diagrammi di interazione** per illustrare il modo in cui gli oggetti interagiscono attraverso lo scambio di messaggi.
Un diagramma di interazione mostra l'interazione tra un insieme di oggetti, basata sullo scambio di messaggi. Ma che cos'è un'**interazione**? *Un'interazione è una specifica di come alcuni oggetti si scambiano messaggi nel tempo per eseguire un compito nell'ambito di un certo contesto*.
In un diagramma, questo compito è rappresentato da un **messaggio trovato** che dà inizio all'interazione. Se il compito è complesso, l'oggetto interagisce e collabora con altri oggetti, chiamati **partecipanti**.

18.2 Di sequenza e di comunicazione

Il termine diagramma di interazione è una generalizzazione di due tipi più specifici di UML, di sequenza e di comunicazione.
Entrambi possono esprimere delle interazioni simili; i **diagrammi di sequenza** mostrano le interazioni in una specie di formato a steccato, in cui gli oggetti che partecipano all'interazione sono mostrati in alto, uno a fianco all'altro, come mostrato nella figura 18.1.
I **diagrammi di comunicazione** mostrano le interazioni tra gli oggetti in un formato a grafo o a rete, in cui gli oggetti possono essere posizionati ovunque nel diagramma, come mostrato nella figura 18.2.

Figura 18.1: Diagramma di sequenza

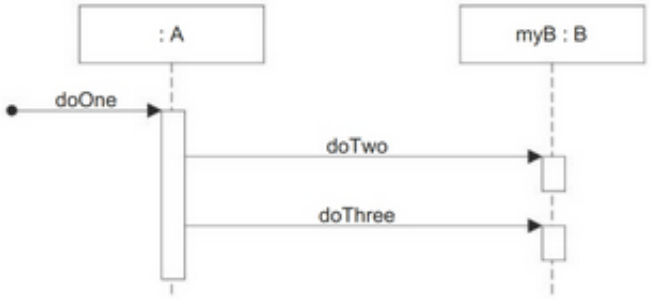
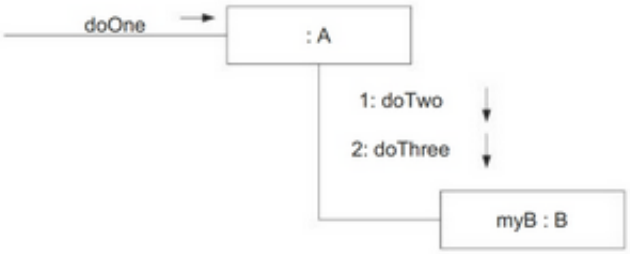


Figura 18.2: Diagramma di interazione



18.2.1 Punti di forza e di debolezza

I diagrammi di sequenza hanno alcuni vantaggi rispetto ai diagrammi di comunicazione. Prima di tutto la specifica di UML è maggiormente incentrata sui diagrammi di sequenza, e sono state dedicate più risorse a questa notazione e alla sua semantica. Per questo il supporto degli strumenti è migliore, e sono disponibili più opzioni di notazione. Inoltre, con i diagrammi di sequenza è più facile vedere la sequenza "call-flow" ("chiamata-flusso") poiché il tempo scorre dall'alto verso il basso, e quindi è sufficiente leggere i messaggi in quest'ordine.
D'altro canto, i diagrammi di comunicazione presentano dei vantaggi quando si applica "UML come abbozzo" per disegnare sulle pareti (una pratica di Agile Modeling), poiché sfruttano molto meglio lo spazio, in quanto i rettangoli possono essere posizionati o cancellati facilmente, sia orizzontalmente che verticalmente. Ne consegue inoltre che è più facile modificare gli abbozzi alla parete con i diagrammi di comunicazione; infatti è semplice (durante il lavoro di progettazione OO creativa, dove sono numerosi i cambiamenti) cancellare un rettangolo per un oggetto da una posizione, disegnarne un al-

tro in un altro punto e abbozzare una linea verso di esso. Al contrario, in un diagramma di sequenza i nuovi oggetti devono sempre essere aggiunti in alto, sulla destra, e ciò rappresenta un limite, poiché lo spazio orizzontale in una pagina (o su una parete) si consuma e si esaurisce molto rapidamente; lo spazio libero in verticale non viene usato in modo efficiente.

18.3 Diagrammi di interazione in UML

18.3.1 Linee di vita

Informalmente, sono i **partecipanti** all'interazione, ovvero delle parti correlate definite nel contesto di un qualche diagramma strutturale. Non è del tutto preciso dire che una linea di vita equivale a un'istanza di una classe ma, informalmente e in pratica, i partecipanti saranno spesso interpretati come tali.

18.3.2 Espressioni di messaggio

I diagrammi di interazione mostrano dei messaggi scambiati tra gli oggetti, UML ha una sintassi standard per queste espressioni messaggio:
`return = message(parameter : parameterType): returnType.`
Le parentesi vengono solitamente escluse se non ci sono parametri, anche se sono permesse.

18.3.3 Oggetti Singleton

Un oggetto singleton è una classe che contiene una sola istanza, in matematica un insieme che contiene un solo elemento.

18.4 Diagrammi di sequenza in UML

18.4.1 Linee di vita

Nei diagrammi di sequenza, una linea di vita comprende sia un rettangolo che una linea verticale che si estende sotto di esso.

18.4.2 Messaggi

Ogni messaggio, di solito sincrono, tra gli oggetti è rappresentato da un'espressione messaggio mostrata su una linea continua con una freccia piena tra le linee di vita verticali.

18.4.3 Barre di specifica dell'esecuzione

In un diagramma di sequenza, una barra di **specifica di esecuzione** mostra l'esecuzione di un'operazione da parte di un oggetto. Più precisamente, mostra il periodo impiegato nell'esecuzione dell'operazione, ovvero la sua attivazione, e che l'operazione è sullo stack del chiamante. La barra è opzionale.

18.4.4 Frame nei diagrammi di sequenza

Come supporto alle "istruzioni" di controllo condizionali e di ciclo (tra le altre cose), UML utilizza i frame (chiamati anche frame di interazione o frammenti combinati). I frame sono regioni o frammenti dei diagrammi, hanno un'**operazione** e una **guardia** nella forma booleana.

Figura 18.3: alcuni operatori

Operatore frame	Significato
alt	Frammento alternativo per logica mutuamente espressa nella guardia (un'istruzione *if-else* di Java o del C).
opt	Frammento opzionale che viene eseguito se la guardia è vera (un'istruzione *if*).
loop	Frammento da eseguire ripetutamente finché la guardia è vera (un'istruzione *while* o *for*). Si può anche scrivere *loop(n)* per indicare un ciclo da ripetere n volte. Può rappresentare anche l'istruzione *foreach* del C# o l'istruzione *for* "avanzata" di Java.
par	Frammenti che vengono eseguiti in parallelo.
region	Regione critica all'interno della quale può essere in esecuzione un solo thread.

18.4.5 Cicli

Un frame OPT può essere usato per notare un frame LOOP, che indica un ciclo.

18.4.6 Chiamate sincrone e asincrone

La notazione UML per le chiamate asincrone è un messaggio con una freccia con la punta non piena; le chiamate sincrone regolari (bloccanti) sono mostrate con una freccia a punta piena.

18.5 Diagrammi di comunicazione in UML

18.5.1 Collegamenti

Un collegamento è un percorso di connessione tra due oggetti, che indica che è possibile una qualche forma di navigazione e di visibilità tra gli oggetti. Si noti che su uno stesso collegamento possono scorrere più messaggi e messaggi in entrambe le direzioni. Non c'è una linea di collegamento per ciascun messaggio; tutti i messaggi scorrono sulla stessa linea, come in una strada su cui è consentito il traffico in entrambe le direzioni.

18.5.2 Messaggi

Ogni messaggio tra oggetti è rappresentato da un'espressione messaggio e da una piccola freccia che indica la direzione del messaggio. Viene aggiunto un numero di sequenza per mostrare l'ordine sequenziale dei messaggi nel thread di controllo corrente.

18.5.3 Creazione di istanze

Per indicare la creazione di un'istanza, UML prevede di utilizzare un messaggio chiamato *create*, anche se per questo scopo può essere utilizzato un messaggio qualsiasi. Se viene usato un altro nome di messaggio (meno ovvio), è utile annotare il messaggio con uno stereotipo UML, come *«create»*.

18.5.4 Numeri di sequenza dei messaggi

L'ordine dei messaggi è illustrato con i numeri di sequenza.

Capitolo 19

Diagramma delle classi

19.1 Notazione comune delle classi

UML comprende i diagrammi delle classi per illustrare le classi, le interfacce e le relative associazioni. Essi sono utilizzati per la modellazione statica degli oggetti.

19.2 Diagramma delle classi di progetto

I diagrammi delle classi possono essere utilizzati, da un punto di vista concettuale, per visualizzare un modello di dominio. Per la discussione occorre anche un termine univoco che chiarisca quando un diagramma delle classi è invece utilizzato da un punto di vista software o di progetto.

19.3 Classificatore

In UML, un classificatore è un "elemento di modello che descrive caratteristiche comportamentali e strutturali". I classificatori sono una generalizzazione di molti degli elementi di UML, comprese le classi, le interfacce, i casi d'uso e gli attori. Nei diagrammi delle classi, i due classificatori più comuni sono le classi e le interfacce.

19.4 Attributi delle classi e associazioni

In UML, le **proprietà strutturali** di un classificatore comprendono gli **attributi** della classe e le **estremità di associazioni** della classe. Le proprietà strutturali di una classe sono anche chiamate attributi della classe, inten-

dendo con questo termine sia gli attributi mostrati mediante la notazione testuale che le estremità di associazione.

19.5 Diagrammi degli oggetti software

Per favorire la comprensione di un diagramma delle classi di progetto, talvolta è utile disegnare un diagramma degli oggetti per descrivere la rappresentazione software di una certa realtà di interesse.

19.6 Operazioni

Una delle sezioni del rettangolo per una classe UML mostra le firme delle operazioni. Un'operazione *non* è un metodo. Un'operazione di UML è una *dichiarazione*, con un nome, dei parametri, un tipo di ritorno, un elenco di eccezioni e magari un insieme di *vincoli* di pre-condizioni e post-condizioni.

19.7 Parole chiave

Una parola chiave in UML è un decoratore testuale per classificare un elemento di un modello.

Figura 19.1: esempi di chiavi

Parola chiave	Significato	Esempio di uso
«actor»	il classificatore è un attore	nei diagrammi delle classi, sopra al nome di un classificatore
«interface»	il classificatore è un'interfaccia	nei diagrammi delle classi, sopra al nome di un classificatore
{abstract}	l'elemento è astratto; non può essere istanziato	nei diagrammi delle classi, dopo il nome di un classificatore o il nome di un'operazione
{ordered}	un insieme di oggetti ha un ordine predefinito	nei diagrammi delle classi, a un'estremità di associazione

19.8 Generalizzazione e astrazione

Una generalizzazione è mostrata in UML come una linea continua e una grossa freccia triangolare dalla sottoclasse verso la superclassi, in particola-

re è una *relazione tassonomica tra un classificatore più generale e un classificatore più specifico. Ogni istanza del classificatore più specifico è anche un'istanza indiretta del classificatore più generale*. La generalizzazione equivale all'ereditarietà nei linguaggi di programmazione orientati agli oggetti? Dipende. In un diagramma delle classi da un punto di vista concettuale, ovvero in un modello di dominio, la risposta è no.

19.9 Dipendenze

Le linee di dipendenza possono essere usate in qualsiasi diagramma, ma sono particolarmente comuni nei diagrammi delle classi e dei package. UML comprende una relazione di dipendenza generale che indica che un elemento **cliente** (di qualsiasi tipo, comprese le classi, i package, i casi d'uso e così via) è a conoscenza di un altro elemento **fornitore** e che un cambiamento nel fornitore potrebbe influire sul cliente.

19.10 Interfacce

UML offre diversi modi per mostrare l'implementazione di un'**interfaccia**, il fornire un'interfaccia ai propri clienti (un'**interfaccia fornita**) e la dipendenza da un'interfaccia (un'**interfaccia richiesta**). In UML, l'implementazione di un'interfaccia viene chiamata anche una **realizzazione di interfaccia**.

19.11 Relazione tra diagrammi di interazioni e delle classi

Quando si disegnano i diagrammi di interazione, dal processo di progettazione creativa della modellazione dinamica degli oggetti emergono un insieme di classi e i relativi metodi. Pertanto, è possibile generare le definizioni dei diagrammi delle classi dai diagrammi di interazione. Ciò suggerisce un ordinamento lineare secondo cui disegnare i diagrammi di interazione prima dei diagrammi delle classi. Tuttavia, in pratica, soprattutto quando si segue la pratica della modellazione agile di creare diversi *modelli in parallelo*, queste viste statica e dinamica complementari sono disegnate contemporaneamente.

Capitolo 20

Diagrammi di attività e modellazione

20.1 Diagramma di attività

Un diagramma di attività di UML mostra le attività sequenziali e parallele in un processo. Tali diagrammi sono utili per la modellazione di processi di business, flussi di lavoro (workflow), flussi dei dati e di algoritmi complessi.

20.1.1 Notazione di base

La notazione di base dei diagrammi di attività di UML include i concetti di **azione**, **arco** (flusso), **partizione**, **fork**, **join** e **nodo oggetto**. In sostanza, questo diagramma mostra un insieme di azioni, alcune delle quali vanno svolte in sequenza, mentre altre possono essere svolte in parallelo.
La maggior parte della notazione si spiega da sé. I punti cruciali della notazione di base sono i seguenti:

- Un'azione rappresenta un'attività elementare. Un arco rappresenta una relazione di sequenza tra azioni. Se ci sono due azioni in sequenza, quando la prima azione termina può automaticamente avere inizio l'azione successiva

- A un arco possono essere associati dei dati (rappresentati da nodi oggetto). Dunque un diagramma di attività può mostrare sia il flusso di controllo che il flusso dei dati

20.2 Come applicare i diagrammi di attività

Un diagramma di attività offre una notazione ricca per mostrare una sequenza di attività, comprese attività parallele. Può essere applicato a qualsiasi punto di vista o scopo, ma l'applicazione più diffusa è per la visualizzazione dei flussi di lavoro nonchè, quindi, dei casi d'uso.

20.2.1 Modellazione dei processi business

Anche se un processo può essere descritto sotto forma di testo (nel testo di un caso d'uso), in questo caso i diagrammi di attività rappresentano un esempio eccellente di come le immagini valgano mille parole. Si può infatti comprendere i propri processi di business complessi attuali, visualizzandoli. Le partizioni sono utili per vedere i vari partecipanti coinvolti e le azioni parallele svolte in un processo, e i nodi oggetti illustrano cosa si muove tutt'intorno. Dopo aver modellato il proprio processo corrente, essi esaminano visualmente le modifiche e le ottimizzazioni possibili.

20.2.2 Modellazione dei flussi di dati

A partire dagli anni settanta, i diagrammi dei flussi di dati (DFD, Data-Flow Diagram) sono divenuti un modo comune per visualizzare i passi e i dati principali coinvolti nei processi di un sistema software. Non è la stessa cosa della modellazione dei processi di business; anzi, i DFD erano usati di solito per mostrare i flussi dei dati in un sistema informatico, anche se teoricamente potevano essere applicati alla modellazione dei processi di business. I DFD erano utili per documentare i flussi di dati principali o per esaminare un nuovo progetto di alto livello in termini dei flussi di dati.

20.2.3 Modellazione di algoritmi paralleli e concorrenti

Benché i dettagli esulino da questa introduzione, gli algoritmi paralleli nei problemi di programmazione concorrente coinvolgono più partizioni, oltre alle operazioni fork e join. Lo spazio fisico complessivo è suddiviso in grossi blocchi, e vengono eseguiti molti thread (o processi) paralleli, uno per ciascun sottoblocco. In questi casi possono essere utilizzate le partizioni dei diagrammi di attività di UML per rappresentare i diversi thread o processi del sistema operativo. I nodi oggetto possono essere utilizzati per modellare gli oggetti e i dati condivisi o scambiati tra le partizioni.

20.3 Notazione a rastrello

Si può mostrare la chiamata di un'attività decomposta e descritta in un altro diagramma di attività, utilizzando il simbolo a forma di **rastrello**, che rappresenta una gerarchia e indica la chiamata a un'attività espansa in un altro diagramma di attività

20.4 Diagrammi di attività in UP

Una delle discipline di UP è la **Modellazione del business**; il suo scopo è quello di comprendere e comunicare "la struttura e la dinamica dell'organizzazione in cui deve essere rilasciato un sistema". Un elaborato fondamentale della disciplina di Modellazione del business è il **Modello degli Oggetti di Business**, che visualizza come funziona un business, utilizzando i diagrammi delle classi. Pertanto i diagrammi di attività sono particolarmente applicabili nella disciplina della Modellazione del business di UP.

Capitolo 21

Diagrammi di macchina a stati e modellazione

Come i diagrammi di attività, i diagrammi di macchina a stati di UML mostrano una vista dinamica. UML include la notazione per illustrare gli eventi e gli stati delle cose: transazioni, casi d'uso, persone e così via.
Un **diagramma di macchine a stati** illustra i possibili stati interessanti per un oggetto, insieme al comportamento dell'oggetto in termini di transizioni fra stati in risposta agli eventi durante la sua vita. Gli stati sono mostrati come rettangoli arrotondati. Le transizioni sono mostrate come frecce (archi) etichettate con il rispettivo evento. È comune includere uno pseudo-stato iniziale, con una transizione automatica a un altro stato quando l'oggetto viene creato.
Un diagramma di macchina a stati mostra il ciclo di vita di un oggetto: quali eventi sperimenta, le sue transizioni e gli stati in cui si trova tra questi eventi.

21.1 Evento, Stato e Transizione

Un **evento** è un avvenimento significativo o degno di nota, uno **stato** è la condizione di un oggetto in un certo intervallo di tempo, il tempo tra due eventi, infine una **transizione** è una relazione fra due stati che indica che quando si verifica un evento, l'oggetto passa dallo stato precedente allo stato successivo.

21.2 Come applicare i diagrammi di macchina a stati

21.2.1 Oggetti dipendenti e indipendenti

Se un oggetto risponde a un evento sempre nello stesso modo, viene considerato **indipendente dallo stato** (o non modale) rispetto a quell'evento. Per esempio, se un oggetto riceve un messaggio e il metodo rispondente fa sempre la stessa cosa, l'oggetto è indipendente dallo stato rispetto a quel messaggio. Se un oggetto reagisce sempre allo stesso modo per tutti gli eventi di interesse, è un **oggetto indipendente dallo stato**. Al contrario, gli **oggetti dipendenti dallo stato** reagiscono in modo diverso agli eventi a seconda del loro stato o modo.

21.2.2 Modellare oggetti dipendenti dallo stato

In senso ampio, le macchine a stati sono applicate in due modi:

1. per modellare il comportamento di un oggetto reattivo complesso in risposta agli eventi
2. per modellare le sequenze valide delle operazioni, ovvero specifiche di protocollo o di linguaggio

21.2.3 Oggetti reattivi complessi

Dispositivi fisici controllati dal software, **transazioni** e **oggetti di business** e **mutatori di ruolo**. Nell'ultimo caso la reattività può non essere esplicita, ma si pensi per esempio ad una Persona che cambia ruolo, da celibe a coniugato. Ciascun ruolo è rappresentato da uno stato.

21.3 Ulteriore notazione

21.3.1 Azioni di transizione e guardie

Una transizione può causare l'avvio di un'azione o attività. In un'implementazione software, ciò può rappresentare l'invocazione di un metodo della classe a cui è associato il diagramma di macchina a stati. Una transizione può anche avere una guardia condizionale, ovvero un test booleano. La transizione avviene solo se il test viene superato.

21.3.2 Stati annidati

Uno stato consente l'annidamento per contenere dei sottostati; un sottostato eredita le transizioni del proprio superstato.

21.4 I diagrammi di macchina a stati in UP

Non esiste nessun modello in UP chiamato "modello a stati". Tuttavia, qualsiasi elemento in qualsiasi modello (Modello di Progetto, Modello di Dominio, Modello degli oggetti di business e così via) può avere una macchina a stati per comprendere o comunicare meglio il proprio comportamento dinamico in risposta agli eventi.

Capitolo 22

GRASP: progettazione di oggetti con responsabilità

22.1 UML e principi di progettazione a confronto

UML è talvolta descritto come uno strumento di progettazione, ma questo non è del tutto corretto ***Lo strumento critico della progettazione per lo sviluppo software è una mente ben istruita sui principi di progettazione***.

22.2 Progettazione a oggetti

Questo paragrafo riassume un quadro generale di esempio per la progettazione di un metodo iterativo. In particolare, è opportuno capire come gli elaborati di analisi sono correlati alla progettazione a oggetti.

22.2.1 Quali sono gli input della progettazione oggetti

Vengno prima descritti gli input di processo:

- il primo **workshop dei requisiti della durata** di due giorni
- **tre dei molti casi d'uso**, quelli che sono i più significativi dal punto di vista dell'architettura e di elevato valore di business. UP raccomanda di analizzare il 10-20% dei requisiti in modo dettagliato
- **esperimenti di programmazione** hanno risolto le domande tecniche più eclatanti

- **il test dei casi d'uso** definisce il comportamento visibile che gli oggetti software devono in definita supportare
- **diagrammi di sequenza di sistema** identificano i messaggi per le operazioni di sistema, che sono i messaggi iniziali per i diagrammi di interazione di oggetti che collaborano da progettare
- **contratti delle operazioni** possono essere complementari al testo dei casi d'uso per chiarire che cosa devono compiere gli oggetti software in un'operazione di sistema.

Non tutti questi elaborati sono necessari. Si tenga infatti presente che in UP tutti gli elementi sono opzionali, e vengono creati possibilmente per ridurre dei rischi.

22.2.2 Quali sono le attività di progettazione oggetti

Dopo aver passato la fase di analisi, si inizia la fase di progettazione e modellazione. Dato uno o più degli input precedenti, gli sviluppatori possono iniziare immediatamente a codificare, progettando mentre si codifica, oppure iniziare una tecnica di modellazione prima di iniziare l'effettiva codifica. Nel caso si scelga UML, il punto cruciale non è UML stesso, ma la modellazione visuale che ci concede. La cosa più importante è che, durante le attività di disegno e codifica, vengano applicati i vari principi di progettazione OO, come i **pattern GRASP** e i **design pattern Gang-of-Four**. L'approccio complessivo al fare la modellazione per la progettazione OO si baserà sulla *mefatora* della **progettazione guidata dalle responsabilità**, ovvero pensare a come assegnare le responsabilità a degli oggetti che collaborano.

Durante il disegno UML va adottato l'atteggiamento realistico (promosso anche nella modellazione agile) secondo cui si disegnano i modelli soprattutto allo scopo di comprendere e comunicare, non di documentare.

22.2.3 Quali sono gli output della progettazione oggetti

Gli output sono in stretta relazione con gli input, infatti durante la progettazione è possibile fare riferimento agli input, prodotti dall'analisi.

22.3 La responsabilità

Un modo comune di pensare alla progettazione di oggetti software, ma anche di componenti su larga scala, è in termini di **responsabilità**, **ruoli** e **colla-**

borazioni. Questi aspetti fanno parte di un approccio più ampio chiamato RDD, ovvero **Responsibility-Driven Development**.
Nella RDD, gli oggetti software sono considerati come dotati di responsabilità; per responsabilità si intende un'astrazione di ciò che fa o rappresenta un oggetto o un componente software. Le responsabilità sono fondamentalmente di due tipi: *di fare* e *di conoscere*.

Le responsabilità **di fare** di un oggetto comprendono:

- fare qualcosa esso stesso, come per esempio creare un oggetto o eseguire un calcolo
- chiedere ad altri oggetti di eseguire azioni
- controllare e coordinare le attività di oggetti

Mentre le responsabilità **di conoscere** di un oggetto comprendono:

- conoscere i propri dati privati incapsulati
- conoscere gli oggetti correlati
- conoscere cose che può derivare o calcolare

Le responsabilità sono assegnate alle classe di oggetti durante la progettazione a oggetti. La traduzione delle responsabilità in classi e metodi è influenzata dalla *granularità* della responsabilità. Le responsabilità più grandi coinvolgono centinaia di classi e metodi, mentre le responsabilità minori possono coinvolgere un solo metodo.
L'idea di responsabilità è solo un'astrazione. Nel software non c'è niente che corrisponde direttamente a una responsabilità. In ogni caso, nel software vengono definite classi, metodi e variabili con lo scopo di soddisfare responsabilità.

Un'altra idea importante della RDD è la **collaborazione**. Le responsabilità sono implementate per mezzo di oggetti e metodi che agiscono da soli oppure che collaborano con altri oggetti e metodi.

La RDD è una metafora generale per pensare alla progettazione del software OO. Si pensi agli oggetti software come simili a persone che hanno delle responsabilità e che collaborano con altre persone per svolgere un lavoro. La RDD porta a considerare un progetto OO *come una comunità di oggetti con responsabilità che collaborano*.

Nella pratica, la progettazione guidata dalle responsabilità viene fatta, iterativamente, come segue:

- identificare le responsabilità e considerarle una alla volta
- chiedersi a quale oggetto software assegnare ogni responsabilità; potrebbe essere un oggetto tra quelli già identificati, oppure un nuovo oggetto
- chiedersi come fa l'oggetto scelto a soddisfare una responsabilità; potrebbe fare tutto da solo oppure collaborare. La collaborazione porta spesso ad identificare nuove responsabilità da assegnare.

22.4 Un approccio alla progettazione OO

È possibile dare un nome e spiegare i principi e i ragionamenti dettagliati richiesti per capire le basi della progettazione a oggetti e dell'assegnazione delle responsabilità agli oggetti.
I principi **GRASP** sono un aiuto per l'apprendimento dei principi essenziali della progettazione a oggetti e per l'applicazione dei ragionamenti di progettazione. Questo approccio alla comprensione e all'utilizzo dei principi di progettazione si basa su *pattern per l'assegnazione di responsabilità*.
Pertanto, i pattern GRASP sono importanti, ma d'altra parte rappresentano solo un aiuto per apprendere la struttura e dare un nome ai principi; una volta capiti gli aspetti fondamentali, i termini GRASP specifici non sono rilevanti.

22.5 GRASP e UML

Le decisioni sull'assegnazione delle responsabilità agli oggetti possono essere prese mentre si esegue la codifica oppure durante la modellazione. Nell'ambito di UML, il disegnare i diagrammi di interazione diventa l'occasione per considerare tali responsabilità, realizzate come metodi. Pertanto, mentre si disegna un diagramma di interazione di UML vanno prese delle decisioni riguardo all'assegnazione di responsabilità. Il disegno deve poi illustrare le scelte di progetto fatte.

22.6 Salto rappresentazionale basso

Un salto rappresentazionale basso sostiene l'assegnazione di responsabilità guidata dal modello mentale di dominio, e favorisce la comprensibilità

del progetto e del codice. In pratica, un modo di ottenere un salto rappresentazionale basso è quello di utilizzare il modello di dominio come fonte di ispirazione per la progettazione dello strato del dominio, ovvero scegleire nel progetto classi software il cui nome è ispirato a classi concettuali del modello di dominio.
Tuttavia, un salto basso non consente di rappresentare la complessità del sistema e gestirlo facilmente. Infatti, potrebbero esserci delle classi software *artificiali* che non hanno nessuna corrispndenza nel modello di dominio.

22.7 Che sosa sono i pattern

Si chiama **pattern** un repertorio contenente sia principi generali che soluzioni idiomatiche che guidino gli sviluppatori nella creazione del software, codificati in un formato strutturato che descrive il problema e la soluzione, e a cui è assegnato un nome.

Nella progettazione OO, un **pattern** è una descrizione, con un nome, di un problema di progettazione ricorrente e di una sua soluzione ben trovata e che può essere applicata a nuovi contesti. Idealmente, dà consigli su come applicare la sua soluzione in circostanze diverse e considera le forze e i compromessi.

22.7.1 I pattern hanno un nome

I pattern hanno un nome, e dare un nome a un pattern, a un'idea di progetto o a un principio presenta diversi vantaggi. sostiene la segmentazione e l'assimilazione di quel concetto nella nostra mente e facilita la comunicazione.
Quando a un pattern viene data un nome e viene ampiamente pubblicato, diventa possibile discutere un'idea di progettazione complessa in frasi o diagrammi più brevi, in virtù di astrazione.

22.7.2 Non esistono "nuovi" pattern

Il punto del design pattern *non* è quello di esprimere nuove idee della progettazione, ma tentare di codificare gli idiomi e i principi *esistenti*, di conoscenza comprovata e verificata. I pattern GASP non affermano nuove idee, ma assegnano un nome ai principi già ampiamente utilizzati.

22.7.3 Cosa sono i GRASP

I GRASP definiscono nove principi di progettazione OO di base o blocchi di costruzione elementari della progettazione.

22.8 Applicare i pattern GRASP

Ci sono nove pattern GRASP: Creator, Controller, Pure Fabrication, Information Expert, High Cohesion, Indirection, Low Coupling, Polymorphism, Proteced Variations. Il resto di questo capitolo esamina i primi 5, mentre i successivi saranno esaminati nei prossimi capitoli.

Le qualità del buon software, sono sostenute dalla **progettazione modulare**, principio presente nei pattern High Cohesion e Low Coupling, che sono sufficienti come pattern, ma sono valutativi è lunghi da applicare, quindi si ovvia a questa difficoltà comprendendo ulteriori pattern che consentono di prendere *più facilmente* delle decisioni di progettazione.

22.9 Creator

22.9.1 Problema

Chi deve essere responsabile della creazione di una nuova istanza di una classe?
La creazione di oggetti software è un'attività necessaria e anche molto comune nel software orientato agli oggetti. Di conseguenza, è utile avere un principio generale per l'assegnazione delle responsabilità di creazione.

22.9.2 Soluzione

Assegna alla classe B la responsabilità di creare un'istanza della classe A se una o più delle seguenti condizioni è vera:

- B contiene o aggrega con una composizione oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A, che saranno passati ad A al momento della sua creazione. Pertanto B è un Expert rispetto alla creazione di A

B è detto *creatore* di oggetti A. Se sono applicabili più opzioni, solitamente va preferita una classe B che *aggrega* o *contiene* la classe A.

22.9.3 Controindicazioni

Talvolta la creazione di oggetti è complessa, per esempio se bisogna usare delle istanze riciclate per migliorare le prestazioni, se la creazione di un'istanza va fatta in modo condizionale da una famiglia di classi simili sulla base del valore di una proprietà esterna, e così via. In questi casi è consigliabile delegare la creazione a una classe di supporto.

22.9.4 Vantaggi

Creator favorisce un accoppiamento basso, il che implica minori dipendenze di manutenzione e maggiori opportunità di riuso. Questo succede perchè la classe creata deve essere probabilmente già visibile alla classe creatore, grazie alle associazioni esistenti che ne hanno motivato la scelta come creatore.

22.10 information Expert (o Expert)

22.10.1 Problema

Qual è il principio generale nell'assegnazione di responsabilità agli oggetti? Un Modello di Progetto può definire centinaia o migliaia di classi software, e un'applicazione può richiedere centinaia o migliaia di responsabilità. Durante la progettazione a oggetti, quando vengono definite le interazioni tra gli oggetti, si effettuano delle scelte sull'assegnazione delle responsabilità alle classi software. Se le scelte sono buone, i sistemi tendono a essere più facili da comprendere, da mantenere e da estendere, e consentono maggiori opportunità di riuso dei suoi componenti.

22.10.2 Soluzione

Assegna una responsabilità all'esperto delle informazioni, ovvero alla classe che possiede *informazioni* necessarie per soddisfare la responsabilità.
La soddisfazione di una responsabilità spesso richiede informazioni che si trovano sparse tra varie classi di oggetti. Ciò implica che molti esperti delle informazioni "parziali" dovranno collaborare al compito.

22.10.3 Controindicazioni

In alcune situazioni, ci potrebbero essere dei problemi di accoppiamento e di coesione.
Sostenere una separazione degli interessi principali migliora l'accoppiamento e la coesione di un progetto. Pertanto, anche se in base a Expert si potrebbero trovare alcune giustificazioni nell'assegnare più funzionalità ad un'unica classe, per altre ragioni (di solito coesione e accoppiamento), si finirebbe per avere un progetto mediocre. Meglio preferire la separazione secondo i principi architetturali.

22.10.4 Vantaggi

- l'incapsulamento delle informazioni viene mantenuto, poichè gli oggetti usano le proprie informazioni per adempiere ai propri compiti
- il comportamento è distribuito tra tutte le classi che possiedono le informazioni richieste, incoraggiando in tal modo definizioni di classe più coese e leggere, più facili da comprendere e da mantenere. Di solito è sostenuta una coesione alta

22.11 Low Coupling

22.11.1 Problema

Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggior opportunità di riuso?
L'**accoppiamento** è una misura di quanto fortemente un elemento è connesso ad altri elementi, ha conoscenza di altri elementi e dipende da altri elementi. Un elemento con accoppiamento basso non dipende da troppi elementi. Per elementi si intendono classi, sottoinsiemi, sistemi e così via. Classi con un forte accoppiamento possono essere inopportune, e alcune di esse presentano dei problemi:

- i cambiamenti nelle classi correlate, da cui queste classi dipendono, obbligano a cambiamenti locali anche in queste classi
- queste classi sono più difficili da comprendere in isolamento, ovvero senza comprendere anche le classi da cui dipendono
- sono più difficili da riusare, poichè il loro uso richiede la presenza aggiuntiva delle classi da cui dipendono

22.11.2 Soluzione

Assegna una responsabilità in modo che l'accoppiamento rimanga basso. Usa questo principio per valutare le alternative. Low Coupling incoraggia ad assegnare una responsabilità in modo tale che la sua collocazione non faccia aumentare l'accoppiamento del progetto a un livello "troppo alto", tale da portare ai risultati negativi causati dall'accoppiamento alto.

22.11.3 Controindicazioni

Un accoppiamento alto con elementi stabili o con elementi pervasivi costituisce raramente un problema. Il problema non è l'accoppiamento alto di per sè, ma l'accoppiamento alto con elementi per certi aspetti *instabili*.

22.11.4 Vantaggi

- una classe o componente con un accoppiamento basso non è influenzata dai cambiamenti nelle altre classi e componenti
- è semplice da capire separatamente dalle altri classi e componenti
- è conveniente da riusare

22.12 High Cohesion

22.12.1 Problema

Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?
La **coesione** è una misura di quanto fortemente siano correlate e concentrate le responsabilità di un elemento, Un elemento con responsabilità altamente correlate che non esegue una quantità di lavoro eccessiva ha una coesione alta. Per elementi si intendono classi, sottoinsiemi e così via.

22.12.2 Soluzione

Assegna una responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare le alternative.
Una classe con una coesione bassa fa molte cose non correlate tra loro o svolge troppo lavoro. Tali classi non sono opportune, e presentano problematiche, sono difficili da: comprendere, mantenere, riusare e sono delicate.
Un elemento ha coesione alta se ha responsabilità (funzionali) altamente

correlate e se non svolge troppo lavoro. Viceversa, un elemento ha coesione bassa se fa molte cose scorrelate o se svolge troppo lavoro. High Cohesion incoraggia ad assegnare le responsabilità in modo tale che le loro collocazioni non facciano diminuire la coesione funzionale del progetto a un livello "troppo basso", tale da portare ai risultati negativi causati dalla coe-sione bassa. Non è possibile dare una misura assoluta di quando la coesione di un progetto sia troppo bassa. Infatti la coesione va intesa come una misura relativa, e non assoluta. L'importante è poter misurare il grado corrente di coesione e valutare la possibilità di eventuali problemi.

Alcuni scenari che illustrano vari gradi di coesione:

- *coesione molto bassa* una classe è la sola responsabile di molte cose in aree funzionali diverse
- *coesione bassa* una classe ha da sola la responsabilità di un compito complesso in una sola area funzionale
- *coesione alta* una classe ha responsabilità moderate in un'unica area funzionale e collabora con altre classi per svolgere i suoi compiti
- *coesione moderata* una classe ha, da sola, responsabilità leggere in poche aree diverse, che sono logicamente correlate al concetto rappresentato dalla classe, ma non l'una all'altra

Una classe con coesione alta è vantaggiosa poiché è più facile da mantenere, comprendere e riusare di una classe con coesione bassa. Infatti, l'elevato grado di correlazione delle funzionalità, combinato a un numero ridotto di operazioni, semplifica la manutenzione e l'evoluzione. Inoltre, avere funzionalità altamente correlate e a grana fine sostiene un maggiore potenziale di riuso.

22.12.3 Progettazione modulare

Nonostante tutto questo di cui sopra, bisogna comunque promuovere la **progettazione modulare**, ovvero la *proprietà di un sistema che è stato decomposto in un insieme di moduli coesi e debolmente accoppiati*.
Una coesione cattiva comporta di solito un accoppiamento cattivo, e viceversa.

22.12.4 Controindicazioni

In alcuni casi, è giustificabile accettare una coesione più bassa.
Un caso è il raggruppamento di responsabilità o di codice in una sola classe

o componente per semplificare la manutenzione da parte di una sola persona; occorre però tenere presente che questo raggruppamento potrebbe anche peggiorare la manutenzione.
Un altro caso di componenti con coesione più bassa è con gli oggetti distribuiti lato server. A causa delle implicazioni dell'overhead sulle prestazioni associate con gli oggetti remoti e la comunicazione remota, talvolta è opportuno creare meno oggetti server, più grandi e meno coesi, che forniscono un'interfaccia per molte operazioni. Questo approccio è anche correlato al pattern chiamato **Coarse-Grained Remote Interface**.

22.12.5 Vantaggi

- sostiene con maggiore chiarezza e facilità di comprensione del progetto
- spesso sostiene Low Coupling
- la manutenzione e i miglioramenti risultano semplificati
- maggiore riuso di funzionalità a grana fine e altamente correlate, poichè una classe coesa può essere usata per uno scopo molto specifico

22.13 Controller

22.13.1 Problema

Qual è il primo oggetto oltre lo strato UI che riceve e coordina ("controlla") un'operazione di sistema?
Le **operazioni di sistema** sono state inizialmente esaminate durante l'analisi degli SSD. Esse sono gli eventi di input principali nel sistema. Un **controller** è il primo oggetto oltre lo strato UI che è responsabile di ricevere o gestire un messaggio di un'operazione di sistema.

22.13.2 Soluzione

Assegna la responsabilità a una classe che rappresenta una delle seguenti scelte:

- rappresenta il "sistema" complessivo, un oggetto radice all'interno del quale viene eseguito il software, un punto d'accesso al software, o un sottosistema principale: sono tutte varianti di un *facade controller*

- rappresenta uno scenario di un caso d'uso all'interno del quale si verificano l'evento di sistema, spesso chiamato Handler, Coordinator o Session

Controller è semplicemente un pattern di delega. Gli oggetti dello strato UI catturano gli eventi di sistema generati dagli attori. In conformità al fatto che lo strato UI non deve contenere logica applicativa, gli oggetti dello strato UI devono delegare le richieste di lavoro a oggetti di un altro strato.

22.13.3 Tipi di controller

Il Facade Controller asseconda l'idea di scegliere un nome di classe che suggerisca una copertina o una facciata sopra agli altri strati dell'applicazione e che fornisca il punto di accesso principale per le chiamate dei servizi dello strato UI agli altri strati sottostanti. I Facade sono adatti quando non ci sono troppi eventi di sistema, o quando l'interfaccia utente non può riindirizzare i messaggi per gli eventi di sistema a più controller alternativi.

Se invece si sceglie un controller di caso d'uso, si avrà un controller diverso per ciascun caso d'uso. Lo si consideri un'alternativa quando la collocazione delle responsabilità in un facade controller porta a progetti con coesione bassa o accoppiamento alto, ovvero quando il facade controller diventa "gonfio" di eccessive responsabilità.

Una decisione "intermedia" tra facade controller e controller di caso d'uso è scegliere un controller diverso per ciascun attore del sistema software, che gestisce tutti i casi d'uso di quell'attore.

22.13.4 Applicazione dei Controller alle applicazioni web

Il pattern Controller può essere applicato nelle applicazioni web, attraverso il pattern MVC, che è diverso dal patter GRASP ma che, in ogni caso, può delegare le richieste di esecuzione di operazioni di sistema al controller GRASP del dominio.

22.13.5 Vantaggi

- *maggiore potenziale di riuso e interfacce inseribili*, l'applicazione di Controller assicura che la logica applicativa non sia gestita nello strato dell'interfaccia o della presentazione

- *opportunità di ragionare sullo stato del caso d'uso*, a volte è necessario assicurarsi che le operazioni di sistema si susseguano in una sequenza legale, oppure si desidera poter ragionare sullo stato corrente dell'attività e delle operazioni all'interno del caso d'uso in corso di esecuzione

22.13.6 Problemi e soluzioni

Se progettata in modo mediocre, una classe controller può avere una coesione bassa, se non è focalizzata e gestisce responsabilità in troppe aree. Si parla in questo caso di **controller gonfio**, che ha caratteristiche:

- nel sistema c'è un'unica classe controlelr che riceve *tutti* gli eventi di sistema
- un controller stesso svolge molti dei compiti necessari per soddisfare gli eventi di sistema, senza delegare lavoro
- un controller ha numerosi attributi, e conserva informazioni significative sul sistema o sul dominio.

Tra i rimedi, ci sono:

- progettare il controller in modo tale che esso, in primo luogo, deleghi la soddisfazione della responsabilità di ciascuna operazione di sistema agli altri oggetti
- aggiungere più controller, che alleggeriscano il solo Facade controller, ad esempio utilizzare invece più Controller di caso d'uso

22.13.7 Gestione di messaggi

Alcune applicazioni sono sistemi per la gestione di messaggi o server che ricevono richieste da altri processi. La progettazione del controller è in un certo modo diversa. Senza entrare nei dettagli, una soluzione consiste nell'usare i pattern Command e Command Processor. Quando un oggetto server riceve un messaggio remoto, crea un oggetto Command per quella richiesta, e lo consegna a un Command Processor, che può accodare, registrare, prioritizzare ed eseguire i comandi.

Capitolo 23

GRASP: altri oggetti con responsabilità

23.1 Introduzione

Esistono altri pattern GRASP, che ampliano lo spettro delle opzioni a disposizione del progettista per l'assegnazione di responsabilità agli oggetti software.
Lo scopo di questi quattro ultimi pattern GRASP, così come quello dei primi cinque, è sostenere alcune qualità importanti di un buon sistema software, di modo che esso sia facile da comprendere, da mantenere e da modificare, e che ci siano delle buone opportunità di riuso dei suoi componenti. Inoltre, una volta che questi pattern saranno stati illustrati, si avrà a disposizione una terminologia ricca e condivisa con cui discutere i progetti.

23.2 Pure Fabrication

23.2.1 Problema

Quale oggetto deve avere la responsabilità quando non si vogliono violare High Cohesion e Low coupling, o altri obiettivi, ma le soluzioni suggerite da Expert (per esempio) non sono appropriate?
Ci sono molte situazioni in cui l'assegnare responsabilità solo a classi software ispirate a classi concettuali del modello di dominio provoca problemi in termini di coesione o accoppiamento mediocri, o basso potenziale di riuso.

23.2.2 Soluzione

Assegna un insieme di responsabilità altamente coeso a una classe artificale o di convenienza, che non rappresenta un concetto del dominio del problema, ma piuttosto è una classe inventata, per sostenere coesione alta, accoppiamento basso e riuso.

23.2.3 Tipi di decomposizione

Nella progettazione a oggetti vengono usati, in linea di massima, due tipi di oggetti:

1. oggetti scelti per decomposizione rappresentazionale
2. oggetti scelti per decomposizione comportamentale

La decomposizione rappresentazionale è una strategia comune nella progettazione a oggetti e sostiene l'obiettivo di un salto rappresentazionale basso. Tuttavia, talvolta si desidera assegnare delle responsabilità per raggruppare dei comportamenti oppure in base a un algoritmo, senza alcuna preoccupazione di creare una classe con un nome o uno scopo correlato a un concetto del mondo reale. Si creano quindi classi di supporto o di convenienza, che corrispondono a una *decomposizione comportamentale*.

23.2.4 Vantaggi

- sostiene High Cohesion, poichè la responsabilità sono divise in una classe a grana fine focalizzata solo su un insieme molto specifico di compiti correlati
- il potenziale di riuso può aumentare grazie alla presenza di classi a grana fine le cui reponsabilità possono essere usare in altre applicazioni

23.2.5 Controindicazioni

A volte, è utilizzata in modo eccessivo, creando troppe classi che rappresentano funzioni o algoritmi, ne consegue un eccesso di oggetti comportamentali e, come sintomo correlato, la maggior parte dei dati all'interno degli oggetti viene passata ad altri oggetti che devono ragionare sui dati.

23.3 Polymorphism

23.3.1 Problema

Come gestire alternative basate sul tipo? come creare componenti software inseribili?

- *alternative basate sul tipo* se un programma è progettato utilizzando la logica delle istruzioni condizionali, nel caso in cui si presenti una nuova variazione sarà necessario modificare la logica condizionale, spesso in molti punti
- *componenti software inseribili* com'è possibile sostituire un componente server con un altro, senza ripercussioni sui client?

23.3.2 Soluzione

Quando le alternative o i comportamenti correlati variano con il tipo, allora assegna la responsabilità del comportamento ai tipi per i quali il comportamento varia, utilizzando operazioni polimorfe.

Polymorphism è un principio fondamentale nella progettazione dell'organizzazione di un sistema che deve gestire variazioni simili. Un progetto basato sull'assegnazione di responsabilità in base a Polymorphism può essere esteso facilmente per gestire nuove variazioni. Si noti che l'estensione di un progetto basato su Polymorphism avviene di solito aggiungendo nuove classi, e non modificando le classi esistenti.

23.3.3 Controindicazioni

A volte si specula sulle variazioni, progettando in visione di variazioni sconosciute. Non è raro che sia applicato uno sforzo inutile per il "future proofing" con la progettazione con iul polimorfismo in punti di variazione di fatto improbabili, che in realtà non si verificheranno mai.

23.3.4 Vantaggi

- le estensioni richieste per le nuove variazioni sono facili da aggiungere
- è possibile introdurre nuove implementazioni senza influire sui client

23.4 Indirection

23.4.1 Problema

Dove assegnare una responsabilità, per evitare l'accoppiamento diretto tra due o più elementi? Come disaccoppiare degli oggetti in modo da sostenere un accoppiamento basso e mantenere alto il potenziale di riuso?

23.4.2 Soluzione

Assegna la responsabilità a un oggetto intermediario per mediare tra altri componenti o servizi, in modo che non ci sia un accoppiamento direatto tra di essi. L'intermediario crea una *indirezione* tra gli altri componenti.

23.4.3 Vantaggi

L'unico vantaggio preponderante ed evidente è un accoppiamento più basso tra i componenti.

23.5 Proteced Variation

23.5.1 Problema

Come progettare oggetti, sottosistemi e sistemi in modo tale che le variazioni o l'instabilità in questi elementi non abbiano un impatto indesiderato su altri elementi?

23.5.2 Soluzione

Identifica i punti in cui sono previste variazioni o instabilità; poi assegna delle responsabilità per creare un'interfaccia stabile attorno a questi punti.

La variazione protetta, è un principio *molto* importante e fondamentale nella progettazione del software: è un principio fondamentale che motiva la maggior parte dei meccanismi e dei pattern, nella programmazione e nella progettazione, per fornire flessibilità e protezione dalle variazioni, tra i quali:

- **progettazione guidata dai dati** riguarda un vasto insieme di tecniche che comprende la lettura di codici, valori, percorsi, allo scopo di cambiare in qualche modo il comportamento del sistema, parametrizzandolo

- **lookup dei servizi** comprende tecniche quali l'utilizzo di servizi di naming o directory o di service discovery per ottenere un servizio
- **progettazione guidata da interprete** comprende l'uso di interpreti per regole che eseguono le regole lette da una sorgente esterna, gli interpreti di script o di un linguaggio che leggono ed eseguono i programmi, le macchine virtuali, i motori per reti neurali che eseguono le reti, i motori di logiche su vincoli che leggono e ragionano su insiemi di vincoli, e così via
- **meta-livello** il sistema è protetto dall'impatto delle variazioni sulla logica o nel codice esterno tramite algoritmi riflessivi che usano i servizi di introspezione e di meta-linguaggio
- **linguaggi standard** gli standard per linguaggi ufficiali, come SQL, forniscono una protezione dalla proliferazione di linguaggi variabili
- **principio di sostituzione di Liskov** formalizza il principio di protezione dalle variazioni nelle differenti implementazioni di un'interfaccia o nelle estensioni di una super-classe
- **structure-hiding** evitare di creare progetti che attraversano percorsi lunghi nella struttura degli oggetti e inviano messaggi a oggetti lontani e indiretti

23.5.3 Controindicazioni

Prima di tutto, si definiscono due tipologie di punti di cambiamento:

- **punto di variazione** una variazione nel sistema o nei requisiti correnti
- **punto di evoluzione** un punto di variazione speculativo che potrebbe sorgere in futuro, ma che non è presente nei requisiti correnti

PV si applica ad entrambi, ma talvolta il costo del "future proofing" speculativo nei punti di evoluzione supera i costi di un progetto semplice, più "fragile", che viene rielaborato quando necessario in risposta alle vere pressioni di cambiamento. Ciò significa che il costo per la realizzazione della protezione nei punti di evoluzione può essere più elevato di quello della rielaborazione di un progetto semplice.

Il punto non è quello di sostenere la causa della rielaborazione e dei progetti

fragili. Se la necessità di flessibilità e di protezione dal cambiamento è realistica, allora l'applicazione di PV è giustificata. Ma se le ragioni sono quelle di un "future proofing" speculativo o di un "riuso" speculativo con probabilità molto dubbia, è utile ricorrere alla moderazione e al pensiero critico.

23.5.4 Vantaggi

- le estensioni richieste per nuove variazioni sono facili da aggiungere
- è possibile introdurre nuove implementazioni senza influire sui client
- accoppiamento più basso
- è possibile ridurre l'impatto o il costo dei cambiamenti

Capitolo 24

Applicare i design pattern GoF

24.1 Design pattern GoF

I design pattern Gang-of-Four e la loro considerevole influenza, sono stati citati in precedenza. Ciascun design pattern descrive una soluzione progettuale comune a un problema di progettazione ricorrente. Sono classificabili in base al loro scopo.

24.2 Adapter (GoF)

Per rispondere al problema "come gestire interfacce incompatibili, o fornire un'interfaccia stabile a componenti simili ma con interfacce diverse?" si propone di convertire l'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio. Un adattatore può anche essere considerato un oggetto Facade, poichè avvolge l'accesso al sottosistema o al sistema con un oggetto singolo.

24.3 Principi GRASP come generalizzazione di altri pattern

L'uso precedente del pattern Adapter può essere visto come una specializzazione di alcuni dei principi elementari GRASP. Esistono tantissimi tipi di pattern, è importante per un progettista esperto conoscere nel dettaglio almeno 50 tra i pattern più importanti. Sfida non complessa considerando che la maggior parte dei design pattern può essere vista come una specializzazione di alcuni principi GRASP di base.

24.4 Factory

L'uso dell'adattatore fa sorgere un nuovo problema nella progettazione: chi crea gli adattatori? E chi stabilisce quali creare?
Se vengono creati da un oggetto di dominio, le responsabilità dell'oggetto di dominio vanno oltre la logica applicativa pura. Ciò significa modularizzare o separare interessi distinti in aree diverse, in modo che ciascuno abbia uno scopo coeso. Pertanto, la scelta di un oggetto di dominio per creare gli adattatori non sostiene l'obiettivo della separazione degli interessi, e riduce la coesione.

Un'alternativa comune in questo caso consiste nell'applicare il design pattern creazionale **Factory**, in cui viene definito un oggetto factory per creare gli oggetti.

Il problema nasce nel chiedersi chi deve essere responsabile della creazione di oggetti quando ci sono delle considerazioni speciali, come una logica complessa.

La soluzione è creare un oggetto Pure Fabrication che gestisce la creazione. Gli oggetti Factory presentano diversi vantaggi:

- separano le responsabilità delle creazioni complesse in oggetti di supporto coesi
- nascondono la logica di creazione pontenzialmente complessa
- consentono l'introduzinoe di strategie per la gestione della memoria che possono migliorare le prestazioni, come il caching o il riciclaggio di oggetti

24.5 Singleton

In alcune occasioni è opportuno supportare la visibilità globale o un punto di accesso singolo a una istanza singola di una classe, anzichè un'altra forma di visibilità.

Il problema nasce chiedendosi se è consentita (o richiesta) una sola istanza di una classe, ovvero un "Singleton". Gli altri oggetti hanno bisogno di un punto di accesso globale e singolo a questo oggetto.

La soluzione è definire un metodo statico di classe della classe che restituisce l'oggetto singleton.
Però, i metodi statici non sono polimorfi e non consentono la ridefinizione nelle sottoclassi, nella maggior parte dei linguaggi.

24.6 Strategy

Per rispondere alla domanda "come progettare per gestire un insieme di algoritmi o politiche variabili ma correlati? Come progettare per consentire di modificare questi algoritmi o politiche?", si definisce ciascun algoritmo, politica, strategia in una classe separata, con un'interfaccia comune.
Intuitivamente, gli algoritmi di cui si parla nel pattern Strategy sono correlati nel senso che risolvono tutti uno stesso problema.

Nel pattern Strategy l'esecuzione di una strategia è basata su una collaborazione tra un oggetto strategia e un **oggetto contesto**. L'oggetto contesto è l'oggetto a cui va applicato l'algoritmo. L'oggetto contesto è associato a un **oggetto strategia**, che è un oggetto che implementa un algoritmo. Inoltre è comune (e di fatto necessario) che l'oggetto contesto passi un riferimento a se stesso (*this*, in Java) all'oggetto strategia, in modo che la strategia abbia una visibilità per parametro nei confronti dell'oggetto contesto, per ulteriori collaborazioni.

24.6.1 Creare una Strategy con una Factory

Ci sono diversi algoritmi o strategie di determinazione del prezzo, che cambiano nel corso del tempo. Chi deve creare la strategia? Un approccio semplice è applicare di nuovo il pattern Factory: una classe può essere responsabile della creazione di tutte le strategie (tutti gli algoritmi o le politiche inseribili o variabili) richieste dall'applicazione.

24.7 Composite e principi di progettazione

Come gestire il caso di politiche di determinazione? Si deve definire una **strategia di risoluzione del conflitto**.
Possono coesistere più strategie, ma la domanda base a cui rispondere è "come trattare un gruppo o una struttura composta di oggetti dello stesso tipo nello stesso modo di un oggetto non composto?". Si devono definire le classi per gli oggetti composti e atomici in modo che implementino la stessa interfaccia.

24.7.1 Dall'ID all'oggetto

Una pratica comune nella progettazione ad oggetti è trasformare le chiavi e gli identificatori degli oggetti in oggetti veri e propri. Questa trasformazione avviene spesso poco dopo che un ID o una chiave entrano nello strato del dominio del Modello di Progetto dallo strato UI.

24.8 Facade

In alcuni casi, è richiesta un'interfaccia comune e unificata per un insieme disparato di implementazioni o interfacce, come per definire un sottosistema. Può verifiarsi un accoppiamento indesiderato, a molti oggetti nel sottosistema, oppure l'implementazinoe del sottosistema può cambiare. Che cosa fare?
Si deve definire un punto di contatto singolo con il sottosistema, ovvero un oggetto facade che copre il sottosistema. Questo oggetto facade presenta un'interfaccia singola e unificata ed è responsabile della collaborazione con i componenti del sottosistema.
Una Facade è un oggetto "front-end" che rappresenta il punto di entrata singolo ai servizi di un sottosistema; l'implementazione e gli altri componenti del sottosistema sono privati e non possono essere visti dai componenti esterni. Una Facade fornisce Protected Variations da cambiamenti nell'implementazione di un sottosistema. Spesso agli oggetti Facade si accede attraverso Singleton.

24.9 Observer/Publish-Subscribe/Delegation Event Model

L'idea alla base del design pattern comportamentale **Observer**, è il principio di separazione Modello-Vista. Ci si pone come problema: diversi tipi di oggetti *subscriber* sono interessati ai cambiamenti di stato o agli eventi di un oggetto *publisher*. Ciascun subscriber vuole reagire in un suo modo poprio quando il publisher genera un evento. Inoltre, il publisher vuole mantenere un accoppiamento basso verso i subscriber. Che cosa fare?
Bisogna definire un'interfaccia *subscriber* o *listener*. Gli oggetti *subscriber* implementano questa interfaccia. Il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi, e li avvisa quando si verifica un evento.

24.9.1 Origine della nomenclatura

È stato chiamato observer poichè il listener o il subscriber sta osservando l'evento; il termine è stato reso popolare per questo motivo, anche se è largamente conosciuto anche come Publish-subscribe.

Capitolo 25

Progettare per la visibilità

25.1 Visibilità tra oggetti

I progetti creati per le operazioni di sistema illustrano lo scambio di messaggi tra oggetti. Affinchè un oggetto possa inviare un messaggio a un altro oggetto, è necessario che il destinatario sia visibile al mittente, ovvero che l'oggetto mittente conosca un qualche tipo di riferimento o un puntatore all'oggetto destinatario.
Quando si crea un progetto di oggetti che interagiscono, è necessario garantire che siano presenti tutte le visibilità necessarie per consentire le interazioni tra gli oggetti.

25.2 Che cos'è la visibilità

La **visibilità** è la capacità di un oggetto di "vedere" o avere un riferimento a un altro oggetto. È relativa all'aspetto del campo di azione (o scope). Ci sono quattro modi comuni con cui è possibile ottenere la visibilità di un oggetto A a un oggetto B:

- **per attributo** B è un attributo di A
- **per parametro** B è un paramentro di un metodo di A
- **locale** B è un oggetto locale in un metodo di A
- **globale** B è in qualche modo sibile globalmente

25.2.1 Visibilità per attributo

La visibilità per attributo da A e B esiste quando B è un attributo di A, detto in altro modo se B è memorizzato da una variabile d'istanza di A, oppure se B è un elemento di una collezione memorizzata da una variabile d'istanza di A. Si tratta di una visibilità relativamente duratura, poichè permane finantochè esistono A e B.

25.2.2 Visibilità per parametro

La visibilità per parametro da A a B esiste quando B viene passato come parametro a unn metodo id A. Si tratta di una forma di visibilità temporanea, poiché persiste solo nell'ambito dell'attivazione e del campo d'azione del metodo.

25.2.3 Visibilità locale

Da A a B esiste quando B è un oggetto locale nell'ambito di un metodo di A. Si tratta di una forma di visibilità relativamente temporanea, poiché persiste solo nell'ambito dell'attivazione e del campo d'azione del metodo. Dopo la visibilità per attributo e quella per parametro, è la terza forma di visibilità più comune nei sistemi orientati agli oggetti.

25.2.4 Trasformare la visibilità

Un aspetto importante della progettazione per la visibilità è la possibilità di poter trasformare le visibilità, da una forma a un'altra.

25.2.5 Visibilità globale

Da A a B esiste quando B è globale per A. Si tratta di una forma di visibilità relativamente permanente, poichè persiste fintantochè esistono A e B. È la forma meno comune di visibilità nei sistemi orientati agli oggetti. Talvolta è poco desiderabile, poiché può favorire un accoppiamento alto. Un modo per ottenere la visibilità globale è assegnare un'istanza a una variabile globale.

Capitolo 26

Trasformare i progetti in codice

26.1 Programmazione e sviluppo iterativo ed evolutivo

La creazione di codice in un linguaggio OO, non fa parte dell'OOA/D, tuttavia è un obiettivo finale. Gli elaborati creati nel Modello di Progetto forniscono alcune delle informazioni necessarie per generare il codice.
Un punto di forza dei casi d'uso uniti all'OOA/D e alla programmazione OO è che tutti insieme forniscono un percorso completo, dai requisiti fino al codice. I vari elaborati alimentano i successivi elaborati in modo utile e tracciabile, e culminano alla fine in un'applicazione eseguibile.

26.2 Trasformare in codice

L'implementazione in un linguaggio orientato agli oggetti richiede la scrittura di codice sorgente per definizione: di classi e interfacce, di variabili d'istanza, di metodi e costruttori.

26.3 Definizioni di classi dai DCD

I DCD descrivono i nomi delle classi e delle interfacce, le superclassi, le firme e gli attributi delle classi. Ciò è sufficiente a creare una definizione base delle classi.

Il metodo *create* è spesso escluso dal diagramma delle classi perchè è molto comune, e per le diverse interpretazioni che comporta a seconda del linguaggio di destinazione.

26.4 Metodi dai diagrammi di interazione

La sequenza dei messaggi in un diagramma di interazione si traduce in una serie di istruzioni nelle definizioni di metodi e costruttori.

In sintesi, ogni messaggio in sequenza all'interno di un metodo, come mostrato nel diagramma di interazione, è trasformato in un'istruzione nel metodo Java.

26.5 Uso di collezioni

Le relazioni uno-a-molti sono comuni. Nei linguaggi di programmazione OO, queste relazioni sono implementate di solito con l'introduzione di un oggetto **collezione**, come una *list* o una *map*, o talvolta anche un semplice array.
La scelta del tipo specifico è naturalmente influenzata dai requisiti.

26.6 Eccezioni ed errori

Finora, nello sviluppo di una soluzione, la gestione delle eccezioni è stata ignorata. Ciò è stato fatto intenzionalmente, per concentrarsi sulle questioni di base dell'assegnazione delle responsabilità e della progettazione a oggetti. Tuttavia, nello sviluppo delle applicazioni, è bene considerare le strategie di gestione delle eccezioni su larga scala durante la modellazione per la progettazione.
In breve, in termini di UML, le eccezioni possono essere indicate nelle stringhe di proprietà dei messaggi e delle dichiarazioni delle operazioni.

26.7 Ordine di implementazione

Le classi possono essere implementate in modo e in ordine diverso, non c'è un ordine specifico come ad esempio si può implementare dalla meno accoppiata alla più accoppiata.

26.8 Sviluppo guidato dai test o precedeuto dai test

Un'ottima pratica applicabile all'UP è lo **sviluppo guidato dai test** o **sviluppo preceduto dai test**. In questa pratica, il codice dei test unitari viene

scritto *prima* del codice che deve essere testato, e lo sviluppatore scrive il codice dei test unitari per *tutto* il codice di produzione.

Il ritmo di base è scrivere un po' di codice di test, poi scrivere un po' di codice di produzione, fare in modo che questo passi il test, quindi scrivere ancora un po' di codice di test e così via.

Capitolo 27

Sviluppo guidato dai test e refactoring

27.1 Sviluppo guidato dai test

Un'ottime pratica promossa dal metodo iterativo e agile è lo **sviluppo guidato dai test** (TDD), noto anche come **sviluppo preceduto dai test**. Questa introduzione è centrata soprattutto sull'applicazione del TDD per i **test unitari**.
Il TDD è basato su un'idea semplice: il codice di test è scritto *prima* del codice da verificare. In questi stile, lo sviluppatore scrive il codice dei test unitari per quasi *tutto* il codice di produzione.

Tra i vantaggi del TDD ci sono i seguenti:

- **i test unitari vengono effettivamente scritti** attività che spesso non viene considerata importante
- **la soddisfazione del programmatore porta a una scrittura più coerente dei test** il debugging non è informale, ma guidato dai test che invogliano il programmatore a porsi in sfida rispetto alla risoluzione positiva del test
- **chiarimento dell'interfaccia e del comportamento dettagliati** il TDD impone che lo sviluppatore adotti non solo il punto di vista di chi implementa gli oggetti, ma anche quello di chi li utilizza
- **verifica dimostrabile, ripetibile e automatica** avere centinaia o migliaia di test unitari che si accumulano nel corso delle settimane for-

nisce una verifica significativa della correttezza. Oltretutto la verifica è facile, poiché i test unitari possono essere eseguiti automaticamente

- **fiducia nei cambiamenti** i test sono un modo per verificare che il sistema funzioni a dovere, anche dopo aver implementato dei cambiamenti

27.1.1 Test unitari

In generale, il TDD prevede diversi tipi di test:

- **unitari** per verificare il funzionamento di piccole parti
- **di integrazione** per verificare la comunicazione tra specifiche parti
- **end-to-end** per verificare il collegamento complessivo tra tutti gli elementi del sistema
- **di accettazione** per verificare il funzionamento complessivo del sistema

Un'**unità** è una piccola parte verificabile di un'applicazione o di un sistema, dotata di funzionamento autonomo. L'obiettivo di un **test unitario** è di verificare che l'unità funzioni correttamente. Un metodo di test unitario è logicamente composto da quattro parti:

- **preparazione** crea l'oggetto e prepara altri oggetti o risorse necessarie
- **esecuzione** fa fare qualcosa alla fixture, viene richiesto lo specifico comportamento da verificare
- **verifica** valuta che i risultati ottenuti corrispondano a quelli previsti
- **rilascio** opzionalmente rilascia o ripulisce gli oggetti e le risorse utilizzate

Il framework per test unitari più diffuso è la famiglia **xUnit**, ma per Java la più diffusa è la sua versione **JUnit**.

27.1.2 Il ciclo del TDD

In particolare, il TDD per i test unitari si basa su cicli di lavorazione molto brevi, con semplici regole:

1. scrivere un test unitario che fallisce, per dimostrare la mancanza o di una funzionalità o di codice
2. scrivere il codice più semplice possibile per far passare il test
3. riscivere o ristrutturare il codice, migliorandolo, oppure passare a scrivere il prossimo test unitario.

27.1.3 Organizzazione dei test

Il TDD implica la definizione di numerosi test, e pertanto è utile capire come organizzare le classi di test e i metodi di test. Il metodo più semplice è costruire una classe di test per ciascuna classe. Questa classe di test conterrà uno o più metodi di test per ciascun metodo pubblico della classe da verificare. Il nome di un metodo di test deve descrivere sia il metodo verificato che il caso in cui viene verificato.

Quando i metodi di test sono molto numerosi, si può definire una classe di test per ciascuna *feature* da verificare.

Un'ulteriore alternativa è definire una classe di test per ciascuna *fixture* (ovvero oggetto o gruppo di oggetti da verificare) se ci sono molti metodi di test che opreano su una fixture identica.

27.2 Refactoring

Il **refactoring** è un metodo *strutturato* e *disciplinato* per riscrivere o ristrutturare codice esistente senza però modificarne il comportamento esterno, applicando piccoli passi di trasformazione in combinazione con la ripetizione dei test ad ogni passo. L'essenza del refactoring è applicare piccole *trasformazioni che preservano il comportamento*, una alla volta. Dopo ciascuna trasformazione, i test unitari vengono eseguiti nuovamente per dimostrare che il refactoring non abbia provocato una regressione (un fallimento). Pertanto, c'è una relazione tra il refactoring e il TDD: tutti i test unitari sostengono il processo di refactoring.

Preservare il comportamento durante il refactoring è importante. Se ciò

non fosse vero, ovvero se lo sviluppatore modificasse il comportamento mentre ristruttura il codice, allora i test unitari esistenti si rivelerebbero poco utili. Infatti, la modifica del comportamento causerebbe probabilmente il fallimento di alcuni test, e lo sviluppatore non potrebbe capire facilmente se un test fallisce per la modifica del comportamento oppure per la ristrutturazione del codice. Piuttosto, nel refactoring il codice viene ristrutturato, ma senza modificarne il comportamento, in modo che i test possano sostenere il lavoro dello sviluppatore.

Il codice su cui è stato eseguito un buon refactoring è breve, conciso, chiaro e senza duplicazioni. Si dice che un codice che non ha queste qualità abbia un "cattivo odore". In altre parole, il progetto è mediocre. I cattivi odori del codice (code smell) sono una metafora del refactoring; sono indicazioni che qualcosa nel codice potrebbe non essere corretto. Alcuni cattivi odori del codice sono i seguenti:

- codice duplicato
- metodo di grosse dimensioni
- classe con molte variabili d'istanza
- classe con molto codice
- sottoclassi estremamente simili
- scarso uso delle interfacce nel progetto
- elevato accoppiamento tra molti oggetti
- tanti altri modi in cui un codice può essere di scarsa qualità...

Il rimedio per i cattivi odori è il refactoring. Per avere un'idea generale, si riportano i principali metodi di refactoring:

- **rename** per cambiare il nome di una classe, un metodo o un campo, per rendere più evidente il suo scopo
- **extract method** trasforma un metodo lungo in uno più breve, estraendone una aprte in un metodo di supporto
- **extract class** crea una nuova classe e vi sposta alcuni campi e metodi da un'altra classe
- **extract constant** sostituisce una letterale costante con una variabile costante

- **move method** crea un nuovo metodo, con un corpo simile, nella classe che lo usa di più
- **introduce explaning variable** mette il risultato dell'espressione, o di una parte dell'espressione, in una variabile temporanea con un nome che ne spiega lo scopo
- **repalce constructor call with factory method** in java sostituisce l'uso dell'operazione *new* e la chiamata di un costruttore con l'invocazione di un metodo di supporto che crea l'oggetto

Più in generale, il refactoring offre i seguenti vantaggi:

- **miglioramento continuo del codice** è importante poter migliorare la struttura e l'organizzazione del codice anche *dopo* che il codice è stato scritto. Senza un processo di miglioramento continuo, invece, il codice tende a diventare disordinato e a "deteriorarsi". Il refactoring è uno strumento per perseguire questo obiettivo. Esso sostiene la possibilità di cambiare scelte di progetto
- **preparazione al cambiamento** il refactoring consente di preparare il codice all'introduzione di nuove funzionalità, nonché all'applicazione di cambiamenti. Poiché queste sono due attività estremamente comuni e frequenti nello sviluppo iterativo, il refactoring costituisce una pratica fondamentale