Capitolo 11
Affrontare il design

Uno dei problemi fondamentali con cui ci si confronta quando si inizia una progettazione a oggetti, è quello dell'identificazione della struttura iniziale del programma. Si tratta in altre parole di identificare gli oggetti che entrano in considerazione nella prima versione del prodotto.

Abbiamo già visto che l'approccio agile alla progettazione e allo sviluppo permette di relativizzare l'importanza del primo design. Questo non ci impedisce però di citare alcuni metodi che aiutano a passare dalla prima analisi dei requisiti a un design che ci permetta di implementare le prime funzionalità.

Terminologia dei requisiti

Un primo approccio sufficientemente semplice e in un certo senso "oggettivo", consiste nell'analizzare il testo scaturito dall'analisi dei requisiti. Ricordiamoci che alla base di tutto c'è la necessità di identificare gli oggetti. La strategia adottata prevede di isolare i nomi e le frasi nominali incontrati nella descrizione del sistema. Questi nomi saranno i primi candidati per la suddivisione del sistema in classi e oggetti.

Un nome comune rappresenta il nome di una classe di elementi. Per esempio, *persona* è un nome comune che descrive una classe di elementi, cioè le persone. I nomi propri sono invece i nomi di elementi specifici. Un nome collettivo o astratto è solitamente il nome di una quantità o di un'attività. Per esempio, *società* è un nome collettivo che si riferisce a un insieme di elementi.

Nel contesto della progettazione a oggetti, seguendo questo criterio, un nome comune porta spesso a identificare una classe di oggetti (un'astrazione di dati). Un nome proprio rappresenterà un'istanza di una classe (oggetto). Un nome collettivo o astratto servirà a indicare le

caratteristiche dei vincoli o raggruppamenti di oggetti e classi specifici del problema. È utile ricordare due cose. Una parola può essere un nome comune in un contesto, un nome proprio in un altro e, in alcuni casi, un nome collettivo o astratto in un terzo contesto. Inoltre non tutti i nomi o frasi nominali saranno d'interesse per la realizzazione software finale. Alcuni oggetti risiedono fuori dai confini dello spazio della soluzione software. Altri oggetti, invece, sebbene rilevanti per il problema, potranno rivelarsi ridondanti quando la soluzione viene raffinata.

Una volta identificati gli oggetti dello spazio della soluzione, si può passare alla selezione dell'insieme delle operazioni che agiscono sugli oggetti. Le operazioni vengono identificate esaminando tutti i verbi e i predicati contenuti nella descrizione iniziale. I verbi servono quindi a determinare le operazioni da associare alle classi; in altre parole, permettono di identificare i metodi.

Responsabilità e collaborazioni

Un approccio diverso usato per determinare le classi che compongono un programma prevede l'analisi delle responsabilità di ogni singolo elemento e delle collaborazioni tra classi.

Si tratta di un metodo che richiede una certa esperienza, ma che porta a una suddivisione che considera fin dall'inizio i criteri di buona coesione e basso accoppiamento, importanti per il resto del programma e per ogni progetto. Abbiamo in parte già visto questo tipo di approccio nell'esempio sviluppato nella prima parte del libro. La posizione di metodi e variabili e la presenza o meno di classi, dipende dal tipo di responsabilità da delegare e dal tipo di collaborazione che si deve instaurare fra gli oggetti.

Un mezzo semplice e utile in un approccio di questo tipo consiste nelle CRC Cards (Class Responsibility Collaboration Cards). Si tratta di compilare per ogni classe una scheda, nella quale si inseriscono nome, responsabilità e collaborazioni previste per la classe in questione.

Per compilare ogni scheda sono necessarie sia un'analisi "locale" (collaborazioni), sia un'analisi "globale" (ruolo e responsabilità della classe in un contesto più ampio). Questo porta a una suddivisione dei ruoli più equilibrata e, contemporaneamente, all'identificazione delle classi coinvolte nel sistema. Seguendo questo procedimento si arriva direttamente al diagramma di classi. Più avanti vedremo un esempio di utilizzo semplice delle CRC Cards.

Coesione e accoppiamento

È strano come certi concetti relativamente semplici siano a volte difficili da spiegare. Forse perché non hanno una vera misura oggettiva, forse poiché, proprio in quanto semplici, si tende a sottovalutarli. Il fatto è che gli studenti sembrano fare una fatica enorme a recepire il significato di coesione a accoppiamento, alla base di un buon design, non solo object-oriented.

Per quanto riguarda la coesione, riporto qui la definizione trovata nel libro su UML di Vetti Tagliati [Vetti-Tagliati-2001], che a sua volta si riferisce a Booch: "un modulo presenta un'elevata coesione quando tutti i componenti collaborano tra loro per fornire un ben preciso comportamento". Naturalmente questo può essere applicato a diversi livelli di dettaglio (package, classi, metodi), anche se a noi interessa particolarmente il livello di classe.

Sempre Vetti Tagliati definisce la coesione come la misura della correlatività delle proprietà strutturali (attributi e relazioni con le altre classi) e comportamentali (metodi) di una classe. La

coesione può però essere anche misurata con il grado di responsabilità delle singole classi, e in questo caso il metodo delle CRC Cards ci è senz'altro di aiuto. Evidentemente si deve tendere a un livello elevato di coesione.
L'accoppiamento è invece la misura della dipendenza tra componenti software, nel nostro caso principalmente tra classi. Questa, al contrario della coesione, va mantenuta ai minimi termini. Vedremo in seguito che la dipendenza minore si ottiene basando sull'interfaccia la relazione tra due classi (la cosa corrisponde anche con il cosiddetto "primo principio del design object-oriented riutilizzabile"). Nella programmazione a oggetti, la relazione di ereditarietà è quella che crea maggiore accoppiamento.

Diagrammi di sequenza

L'utilizzo di metodi di progettazione come UML porta a combinare più "viste" e aspetti di un sistema. Ogni vista ha il suo scopo e la sua specificità. Il diagramma di classi serve a descrivere l'architettura statica del sistema; il diagramma di sequenza viene usato per descrivere le fasi di un'operazione che comprende l'interazione tra più oggetti; i diagrammi per i casi d'uso servono a descrivere le funzionalità del sistema, e così via. Quando questi diversi elementi vengono messi in relazione tra di loro, si può usare un aspetto per realizzarne un altro. Così un diagramma di sequenza, che ci obbliga a specificare quali oggetti entrano in gioco in una determinata operazione, può essere sfruttato per estrarre le classi necessarie al design del sistema.

Design evolutivo

Potremmo chiamare "design by refactoring" il design che scaturisce a partire dal refactoring e "design by testing" quello che scaturisce dai test. I metodi agili, eXtreme Programming in particolare, propongono di iniziare la codifica dal design minimo indispensabile per realizzare le prime funzionalità del sistema. Solo in seguito, inserendo nuove funzionalità, il design va rivisto e rifattorizzato tenendo conto dei nuovi requisiti da aggiungere al sistema.

Il design può evolversi a partire da azioni di refactoring finalizzate alla preparazione del codice per nuove funzionalità, oppure da azioni di test. Il test può servire a specificare la API di un sistema (accesso esterno al sistema) e quindi indirettamente contribuisce all'evoluzione del design. Sia nella parte relativa al test, nel capitolo sul "test-driven development", sia in quella relativa ai metodi di refactoring, avremo modo di vedere in pratica questo tipo di approccio allo sviluppo e al design. Volendo descrivere questo procedimento in termini di UML, si potrebbe dire che si arriva al design partendo dai casi d'uso (*use cases*).

Ruolo dei pattern

Qual è dunque il ruolo dei pattern in questo contesto? I design pattern aiutano nell'identificare le astrazioni necessarie e nel definire le classi che le implementano. Le astrazioni create durante la fase di design sono fondamentali per la flessibilità del sistema. Le soluzioni astratte

sono le più flessibili perché facilitano le modifiche a livello di implementazione, isolando i dettagli. Con lo studio e l'utilizzo dei pattern impariamo a manipolare oggetti attraverso classi astratte e *interface*. Torniamo ai concetti di tipo e classe visti in precedenza. La classe definisce il modo in cui un oggetto viene implementato, lo stato interno e l'implementazione dei metodi. Un tipo definisce invece l'interfaccia di un oggetto, le richieste a cui è in grado di rispondere. Abbiamo detto che un oggetto può avere più tipi. Più importante, nel contesto in cui ci troviamo, è però il fatto che più oggetti di classi diverse possono avere lo stesso tipo. La manipolazione attraverso classi astratte e interfacce viene chiamata "programmazione generica"; riduce la dipendenza di implementazione tra sottosistemi e comprende i seguenti punti:

- Il client (l'oggetto che "usa") non conosce (e non deve conoscere) il tipo specifico dell'oggetto che utilizza.

- Il client non dichiara variabili come istanze di una classe particolare, ma solo attraverso classi astratte (o *interface*).

Abbiamo già incontrato questo approccio più volte. Abbiamo visto l'ultimo caso con l'utilizzo del pattern Iterator. Ecco il modo in cui l'oggetto client utilizza un iteratore:

```
Iterator iter = new ConcreteIterator(…);
iter.rewind();
while(iter.hasMoreElements()){
  System.out.println(iter.nextElement());
}
…
```

Si tratta di mettere in evidenza il fatto che la variabile iter venga definita di tipo Iterator, anche se l'oggetto concreto associato sarà di una sottoclasse di Iterator. Questo porta alla genericità del sistema e ci conduce a quello che nel libro sui pattern [Gamma-1995] viene descritto come il "primo principio del design object-oriented riutilizzabile".

Primo principio: "Program to an interface, not to an implementation"

Utilizziamo la parte generica degli elementi (tipo), quando specifichiamo le interazioni tra gli oggetti. La parte specifica di implementazione va inserita il più tardi possible e isolata dal contesto di collaborazione.

Ereditarietà e composizione

Determinare le responsabilità dei singoli elementi di design e stabilire le collaborazioni, significa anche sfruttare al meglio i meccanismi a disposizione per mettere in relazione classi (struttura statica) e oggetti (runtime).

Se consideriamo le due tecniche principali di contatto tra classi, ovvero ereditarietà e composizione, possiamo analizzare vantaggi e svantaggi, in parte già anticipati nel capitolo sulla programmazione a oggetti, quando sono stati introdotti i concetti di riutilizzo *white-box* e riutilizzo *black-box*.

Ereditarietà

L'ereditarietà è una relazione statica, definita in compilazione. A parte in pochi linguaggi altamente dinamici, non è possibile durante l'esecuzione di un programma modificare la struttura gerarchica di una classe. Questo è generalmente uno svantaggio, anche se il controllo rigido eseguito in compilazione garantisce una maggiore affidabilità del sistema.

Generalmente il meccanismo di eredità è supportato direttamente dal linguaggio. Questo significa che definire una gerarchia di classi è da considerare un'attività relativamente semplice e naturale, con linguaggi di programmazione ad oggetti.

Le modifiche di comportamento possono essere realizzate in modo molto semplice, attraverso il meccanismo chiamato *overriding*, che permette di riutilizzare la funzionalità di un'intera classe, modificando solo ciò che non fa esattamente al caso.

Con l'ereditarietà, quindi con il riutilizzo di tipo white-box, si rischia più facilmente di infrangere il principio dell'incapsulamento. C'è una relazione più diretta tra le classi. Se si modifica la superclasse, si rischia di dover adattare anche le sottoclassi. Questa dipendenza può limitare fortemente il riutilizzo. Un modo per evitare questo consiste nell'ereditare unicamente da classi astratte.

Composizione

Siccome l'oggetto componente è accessibile unicamente attraverso la sua interfaccia, non si infrange il principio dell'incapsulamento e questo è da considerare un vantaggio. Ogni oggetto della composizione può essere rimpiazzato a runtime con un altro oggetto che abbia la stessa interfaccia (stesso tipo), cioè che sia in grado di rispondere agli stessi messaggi. Questo aggiunge un grosso fattore di dinamicità al meccanismo, se confrontato con l'ereditarietà. Il fatto che il nuovo oggetto abbia la stessa interfaccia può essere controllato in compilazione (in linguaggi come Java), oppure non viene controllato affatto (in linguaggi maggiormente dinamici, come Smalltalk e Lisp).

Se il riferimento viene definito attraverso l'interfaccia, si ottiene una minore dipendenza di implementazione tra classi, visto che può entrare in considerazione come componente un qualsiasi oggetto della gerarchia che parte dall'interfaccia.

Una dipendenza di questo tipo tra due classi si traduce a runtime nella creazione di due oggetti, mentre la relazione di ereditarietà è puramente strutturale, quindi l'oggetto in esecuzione è uno solo (senza considerare oggetti ausiliari, dipendenti dall'implementazione del sistema).

Sulla base di queste considerazioni, arriviamo a quello che nel libro sui pattern viene definito il "secondo principio del design object-oriented riutilizzabile".

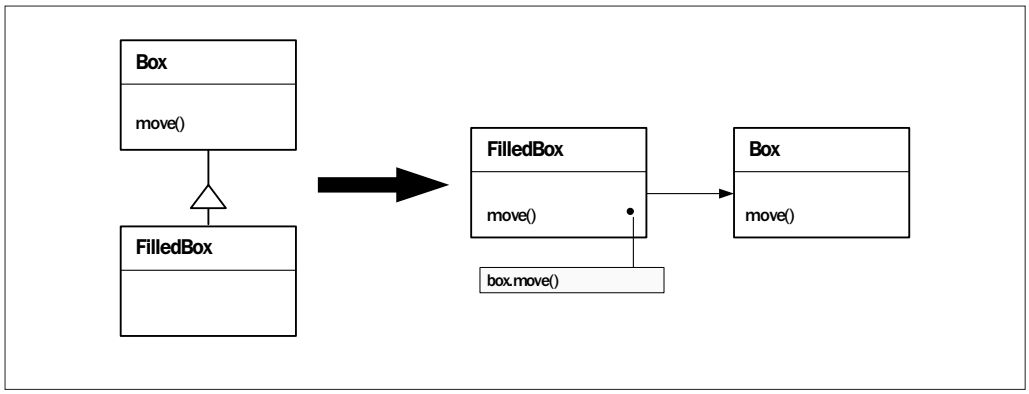

Figura 11.1 – Trasformazione di relazione.

Secondo principio: "Favor object composition over class inheritance"

Quando ci si trova di fronte al dubbio se scegliere ereditarietà o composizione, meglio scegliere composizione, per i suoi aspetti maggiormente dinamici visti nel confronto precedente. Se poi a questo aggiungiamo l'utilizzo del primo principio, otteniamo un sistema flessibile, con elementi sostituibili e dipendenze specificate unicamente da interfacce.

Delegation

Il meccanismo di composizione, usato in contrapposizione all'ereditarietà viene anche chiamato *delegation*, termine più astratto, che esula dal meccanismo di implementazione effettivamente utilizzato. L'idea è unicamente quella di un oggetto che delega a un altro determinati compiti. Quando si sceglie tra ereditarietà e delegation, si deve considerare che entrambi i tipi di relazione possono essere trasformati nell'altro. Una relazione *is-a* (ereditarietà) può essere trasformata in ana relazione *has-a* e viceversa. Nella parte dedicata alle metodologie di refactoring vedremo che ce ne sono due che hanno proprio questo scopo. I loro nomi sono, non a caso, "Replace Delegation with Inheritance" e "Replace Inheritance with Delegation". Vediamo in figura 11.1 uno schema semplice di trasformazione da ereditarietà a delegation.

Ci sono alcune conseguenze, come la necessità di definire dei metodi *wrapper* in `FilledBox` (ad esempio `move()`) per mettere a disposizione la funzionalità di `Box`, ma la trasformazione non modifica la funzionalità.

Ereditarietà nella programmazione a oggetti

Chi ha sempre pensato che l'ereditarietà fosse l'elemento più importante della programmazione a oggetti può rimanere disorientato dall'enunciazione dei due principi del design object-oriented

riutilizzabile. Questi infatti considerano la composizione come elemento di maggiore flessibilità rispetto alla derivazione. In parte è vero, in parte si tratta di contestualizzare.
È vero che l'ereditarietà come riutilizzo di implementazione viene messa in secondo piano dalla composizione. D'altra parte la vera flessibilità, nella composizione, si raggiunge riferendosi non a un elemento specifico, bensì a un'intera gerarchia, rappresentata possibilmente da una classe astratta, come abbiamo già osservato con il pattern Strategy usato nell'esempio della Parte I. Si utilizza perciò ancora il meccanismo di ereditarietà, per sfruttare però il suo elemento essenziale, cioè il polimorfismo.

Design for change

I design pattern favoriscono il cosiddetto "design for change". Bisogna ricordarsi che un design che non considera cambiamenti complica la manutenzione del programma. La chiave di un buon riutilizzo consiste sia nel favorire refactoring per preparare il programma a nuove aggiunte, sia nell'anticipare nel design un certo numero di possibili cambiamenti.

I design pattern sono di aiuto in entrambi gli aspetti. La flessibilità introdotta definendo parti variabili e indipendenti facilita il redesign, rendendo il sistema "robusto" verso certi tipi di cambiamento. Inoltre si deve considerare che un pattern rappresenta un design già studiato, più volte riutilizzato e ben documentato, quindi facilmente comprensibile. Si tratta perciò del caso migliore in cui si possono eseguire ulteriori passaggi di refactoring, perché il design è chiaro e pulito. Se poi queste ulteriori iterazioni di refactoring portano ad altri pattern, tanto meglio.

Cause di redesign

Vediamo però quali sono le cause che portano a considerare il redesign in un sistema e quali sono i pattern da considerare.

Creazione di oggetti attraverso il nome di una classe

Si tratta di eliminare la dipendenza tra creazione di un oggetto e nome specifico della classe coinvolta nella creazione dello stesso. Quando creiamo oggetti sappiamo con precisione dove e quando questi oggetti vengono creati, ma spesso non sappiamo con altrettanta precisione a quali classi specifiche questi oggetti appartengono.

I pattern coinvolti sono: Abstract Factory, Factory Method e Prototype.

Dipendenza da operazioni specifiche

Quando il codice dipende da operazioni specifiche, significa che per applicare altre operazioni dovremmo riscrivere codice identico, ma con chiamate a operazioni diverse. Per parametrizzare il codice con queste operazioni, per passare le operazioni da una parte all'altra del codice sotto forma di oggetti o per suddividere il codice secondo il tipo di responsabilità

che si prende rispetto a una determinate operazione, vengono coinvolti i pattern Chain of Responsibility e Command.

Dipendenza da piattaforme software o hardware

Quando si desidera isolare dipendenze software o hardware dal resto del programma, per essere in grado di sostituire in modo semplice gli stessi elementi per piattaforme diverse, si fa capo ai pattern Abstract Factory e Bridge.

Dipendenza da rappresentazioni o implementazioni di oggetti

In generale quando si desidera isolare rappresentazioni specifiche dal resto del programma, per facilitare eventuali modifiche, quando si hanno implementazioni che si desidera separare dal resto del programma, o quando si vuole avere un livello intermedio in più di astrazione, sono a disposizione i pattern Abstract Factory, Bridge, Memento e Proxy.

Dipendenza algoritmica

Quando si desidera generalizzare algoritmi, creare serie di algoritmi interscambiabili, oppure più semplicemente isolare implementazioni per centralizzare eventuali modifiche, sono da considerare Builder, Iterator, Strategy, Template e Visitor.

Associazioni troppo strette

Quando si desidera allentare associazioni tra classi, ritenute troppo strette e problematiche, oppure quando si intende separare elementi o creare nuove astrazioni, intervengono i pattern Abstract Factory, Bridge, Chain of responsibility, Command, Mediator, Façade e Observer.

Estensione di funzionalità attraverso la gerarchia

Quando si cercano metodi diversi per estendere la funzionalità in modo che il tutto possa rimanere il più trasparente possibile per l'oggetto client, intervengono Bridge, Chain of responsibility, Composite, Decorator, Observer e Strategy.

Impossibilità di modificare classi in modo conveniente

Quando si devono combinare elementi diversi, senza la possibilità di modificare la loro interfaccia, perché ci sono altre dipendenze da considerare, si ricorre ai pattern Adapter, Decorator, Visitor.

Livelli di riutilizzo

Prima di terminare questo capitolo, vediamo di fare chiarezza sui progressivi livelli di riutilizzo del software che è possibile individuare. La portata del riutilizzo, infatti, varia da livelli più ristretti a possibilità più ampie e complesse.

Applicazione

Quando scrivo un'applicazione specifica, sviluppo classi e metodi per questa applicazione. Non penso al riutilizzo, se non in termini interni di metodi e gerarchie di classi, per fattorizzare al meglio il codice ed eliminare eventuali ripetizioni. Ho quindi un riutilizzo prettamente interno.

Biblioteche di classi

Il livello ulteriore consiste nell'implementazione di biblioteche di classi, i cosiddetti *toolkits*. Si realizza in questo caso una programmazione per terzi, perché si sviluppa funzionalità da mettere a disposizione di più programmi, completamente diversi uno dall'altro. Si parla di riutilizzo esterno. Sviluppare un toolkit, se consideriamo il punto di vista del riutilizzo, è più complesso, perché bisogna essere in grado di prevedere richieste che arrivano da applicazioni con esigenze molto diverse.

Framework

Il terzo livello è quello del framework, un tipo di approccio diverso dal toolkit, in un certo senso completamente inverso. Mentre un toolkit mette a disposizione funzioni di utilità da richiamare all'interno di un programma principale (applicazione), il framework object-oriented mette a disposizione un programma principale generico, che va adattato specificando singoli elementi che vengono chiamati automaticamente all'interno del framework. Si parla di programmazione generica e riutilizzo di design. Se parliamo di difficoltà, possiamo dire che realizzare un framework sufficientemente generico richiede senza dubbio più preparazione che sviluppare un'applicazione specifica. I design pattern vengono anche definiti come "mini-framework", proprio perché rappresentano la generalizzazione di singoli aspetti della programmazione a oggetti e, come i framework, permettono il riutilizzo di design.

Un framework rappresenta una soluzione astratta. Si tratta di realizzare un funzionamento e un'architettura basati su classi astratte, che definiscono lo scheletro di base del programma. Una nuova applicazione verrà realizzata come istanza del framework. Si tratterà di adattare lo scheletro di base secondo le esigenze specifiche dell'applicazione, specificando le classi concrete, che sfruttano i collegamenti esistenti a livello astratto. La parte critica di un framework consiste nelle interfacce. Proprio perché la funzionalità del framework (e quindi dei programmi che con questo framework vengono creati) si basa su interazioni tra classi specificate a livello di interfaccia, queste devono essere più stabili possibili. Un cambiamento di interfaccia avrebbe conseguenze in tutte le applicazioni realizzate con il framework.

Il framework Jacaranda

Verso la fine del 2000, durante un periodo sabbatico in Sudafrica, a Pretoria, presso la locale università, ho avuto l'occasione di sviluppare il framework Jacaranda [Jacaranda]. Si tratta di un framework usato per implementare macchine a stati finiti. Il programma legge la descrizione della macchina a stati da un file, si costruisce i collegamenti tra stati in memoria ed è quindi in grado di accettare o rifiutare una sequenza di input.

Esistono innumerevoli implementazioni di macchine a stati finiti, alcune realizzate dal prof. Bruce Watson, con il quale ho avuto modo di collaborare a Pretoria. Quali sono allora i vantaggi che si ottengono da una realizzazione con framework? Sono soprattutto la flessibilità e l'adattabilità del programma. Usare come base di realizzazione un framework significa non solo poter creare macchine a stati diverse modificando il contenuto del file con la descrizione degli stati e dei collegamenti tra stati, come in quasi tutte le altre realizzazioni, ma molto di più.

Usare come base di realizzazione un framework significa poter adattare il framework in modo da permettere nuovi tipi di stato (interi, caratteri, nuovi tipi di oggetto); significa adattarlo per trasformarlo da automa a stati finiti tradizionale (utlilizzato per accettare o meno sequenze di input), a *transducer*, cioè automa in grado di accettare sequenze in input e contemporaneamente generare nuove sequenze in output (utilizzo del pattern Command); significa poter liberamente riscrivere e sostituire l'algoritmo di controllo delle sequenze, cioè la strategia di attraversamento della struttura dati caricata in memoria (utilizzo del pattern Strategy), e così via.

Ogni volta che si adatta un framework, si dice che si crea una nuova istanza dello stesso. Perché questo? Perché adattare un framework non significa modificare il codice esistente, ma vuol dire aggiungere nuovo codice che si integra nel framework per creare il nuovo programma (istanza). Il nuovo codice che viene aggiunto consiste in sottoclassi di classi esistenti, di cui si riscrivono alcuni metodi per dare al framework un nuovo comportamento in determinate circostanze. Ecco perché si dice che il framework mette a disposizione il flusso principale del programma e chi lo adatta deve implementare solo alcune singole parti.

Le macchine a stati finiti sono considerate, per efficienza e spazio di memoria richiesto, lo stato dell'arte per la rappresentazione di informazioni linguistiche di tipo morfologico, come la gestione delle varie declinazioni di un aggettivo o le varie coniugazioni di un verbo. Diverse istanze del framework Jacaranda vengono utilizzate dal portale Canoo.net ([Canoo.net], il maggiore portale lessicografico per la lingua tedesca) per realizzare la struttura di transducer necessaria alla gestione dei milioni di collegamenti e connessioni di informazioni lessicografiche e morfologiche presenti nel sito e a libera disposizione dell'utente.

Ultima curiosità: Jacaranda è il nome di una pianta caratteristica di Pretoria (anche se importata dall'America Latina, più precisamente dall'Argentina) che fiorisce in ottobre, durante la primavera sudafricana, il periodo, appunto, in cui mi trovavo a Pretoria. Durante quel mese, le ottantamila e più piante presenti in quella che viene anche chiamata Jacaranda City colorano l'intera città di viola.

Il framework controlla completamente il flusso, ma le sue chiamate si riferiranno a una sottoclasse piuttosto che a un'altra, a seconda dell'applicazione, cioè a dipendenza delle classi concrete effettivamente a disposizione, realizzate da chi ha sviluppato l'applicazione finale. L'idea di programmazione generica consiste nel fatto che chi scrive il framework specifica un

flusso di operazioni che solamente chi in seguito realizzerà l'applicazione sarà in grado di sfruttare. Realizzare un'applicazione attraverso un framework può essere complesso se non si conoscono i meccanismi e le interazioni tra classi "nascoste". Questo è senza dubbio uno dei lati negativi dei framework, se questi non sono ben documentati. Diventa quindi un compito essenziale documentare bene il modo in cui usare al meglio le interazioni messe a disposizione.

Restando a Java, un framework sicuramente noto è la gestione di eventi presente in AWT, utilizzata anche in Swing. Quando vogliamo ascoltare gli eventi del mouse, registriamo un oggetto come `Listener` ("ascoltatore") degli eventi del mouse; poi i metodi dell'oggetto `Listener` vengono chiamati automaticamente dal sistema, ad ogni evento specifico.
Quando riutilizziamo un pattern, non facciamo che ricreare una struttura astratta, per poi adattarla alle esigenze specifiche della nostra applicazione. Utilizziamo quindi un framework, che però è molto piccolo (come detto, i pattern vengono anche definiti come mini-framework) e quindi può essere compreso in modo approfondito.

Principio di Hollywood

Quando si parla di framework, capita di incontrare il termine "principio di Hollywood". Di cosa si tratta? Si tratta della frase tipica, formulata dai produttori cinematografici di Hollywood agli attori speranzosi, non affermati, in cerca di parti e fama: "Non chiamarci, ti chiameremo noi…".
Un framework si comporta in modo simile. È lui che gestisce il flusso di chiamate. Le classi concrete implementate per le nuove applicazioni non specificano nuove chiamate, si comportano invece in modo passivo e vengono attivate unicamente dalle richieste che arrivano dal framework.

In questo capitolo…

Abbiamo appena superato il capitolo probabilmente più importante del libro. Abbiamo trattato la problematica del passaggio dai requisiti al design, con gli eventuali passaggi intermedi.

Anche se abbiamo più volte detto che l'approccio agile alla progettazione e allo sviluppo permette di relativizzare l'importanza del primo design, questo non vuole assolutamente significare che il design non sia importante. Al contrario, proprio perché rappresenta l'architettura, è l'elemento fondamentale ed è quindi utile poter riutilizzare vecchi design (design pattern), poter creare design flessibili (design for change) e poter rivedere in ogni momento l'architettura dell'applicazione (redesign).