Capitolo 15
A caccia di pattern

Abbiamo già parlato di framework a oggetti e del fatto che un pattern possa essere considerato una sorta di mini-framework. Ogni pattern rappresenta un meccanismo generico che deve essere adattato (o meglio "istanziato", per restare alla terminologia object oriented) esattamente come va fatto con un framework a oggetti.

Una combinazione di pattern, proprio grazie a questa loro intrinseca genericità, rappresenta un framework. Vale anche il contrario, cioè ogni framework può essere descritto attraverso un'interazione tra pattern. È proprio quest'ultima peculiarità che approfondiremo nel presente capitolo, che riprende due miei articoli pubblicati su MokaByte nei mesi di marzo e aprile del 2001 [Pedrazzini-2001], mostrando come la comprensione e l'utilizzo di elementi di framework e librerie di classi sarebbero nettamente semplificati se i pattern coinvolti venissero menzionati in modo esplicito. Andiamo allora a caccia di pattern nelle interazioni esistenti tra le classi presenti nella distribuzione standard di Java.

Gestione degli eventi

Consideriamo come primo esempio la gestione degli eventi con il cosiddetto Delegation Model presente nelle classi AWT e riutilizzato in Swing. Il modello si basa sulla separazione tra oggetto che genera l'evento e "ascoltatori" (*listener*) dell'evento stesso. Essenzialmente un evento rappresenta per un oggetto la possibilità di comunicare con un altro oggetto inviando un messaggio. Ad esempio, un pulsante desidera notificare al programma il fatto di essere stato premuto dall'utente: la notifica avviene attraverso un evento.

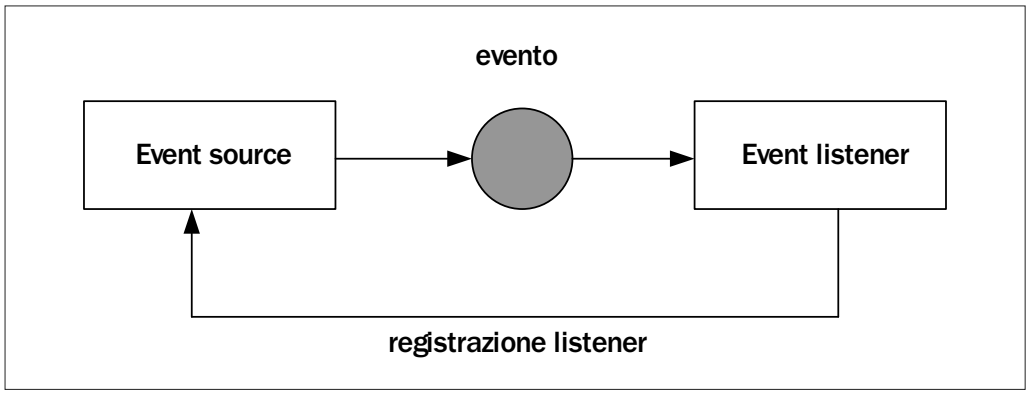

Figura 15.1 – Delegation Model.

Esiste una separazione logica tra oggetto che fa partire l'evento (*source*) e oggetto che si trova in ascolto (*listener*) per reagire all'evento. I due (che in alcuni casi sono lo stesso oggetto) sono legati attraverso la registrazione del *listener*. Quando l'evento viene passato da un oggetto all'altro causa la chiamata di un metodo nell'oggetto listener. L'evento stesso è un oggetto e può quindi mantenere alcune informazioni sul suo stato.

Registrarsi come *listener* significa implementare un'interfaccia specifica predefinita e chiamare un metodo particolare che permetta al nuovo oggetto listener di ricevere un certo tipo di eventi dall'oggetto *source*. Se ad esempio un oggetto si registra come `MouseListener`, questo significa che riceverà solo gli eventi legati al mouse.

Ci sono tre passaggi da eseguire quando si intende utilizzare il Delegation Model:

- Decidere a quali eventi un oggetto è interessato e trovare quali sono le interfacce *listener* corrispondenti.
- Implementare i metodi definiti nei *listener* scelti, creando nuove classi.
- Registrare le classi appena implementate all'oggetto sorgente di eventi.

Documentazione con pattern

Quanto tempo si perde inizialmente per capire il del sistema di eventi? È vero che il tutto non viene per niente facilitato dagli esempi "didattici" forniti in molti tutorial, dove, pensando di semplificare la spiegazione, l'oggetto che genera l'evento e l'unico listener vengono quasi sempre realizzati nella stessa classe. Sarebbe comunque più semplice specificare che la collaborazione tra oggetto che genera l'evento e i vari listener ricalca il modello descritto dal pattern Observer (anche chiamato paradigma Model–View). Questo non solo spiegherebbe il design e il funzionamento, ma metterebbe in evidenza altri aspetti, come l'esatta collaborazione e dipen-

denza dei singoli elementi, il contesto in cui questi elementi vengono utilizzati, vantaggi, svantaggi, ecc. Insomma, con un'unica identificazione si potrebbe associare all'architettura un'intera serie di spiegazioni, valide anche per parti simili del framework.

Lo schema in figura 15.2 mostra l'architettura di base del pattern Observer, che abbiamo già visto nel capitolo riguardante l'applicazione dei pattern. Gli elementi Subject e Observer sono astratti (classi astratte o, nel caso di Observer, *interface*) e servono a definire le collaborazioni principali.

Se andiamo a leggere il catalogo dei pattern [Gamma-1995], troviamo che lo scopo del pattern Observer è quello di definire una dipendenza *1-m* tra oggetti, in modo tale che quando uno modifica il suo stato, gli altri possano essere informati. Il suo utilizzo è utile quando la modifica di un oggetto rende necessaria la modifica di altri, ma non si sa quanti siano gli altri. La dipendenza tra oggetti è inoltre dinamica, può cioè cambiare a runtime.

Da notare che la dipendenza *subject* tra Observer e Subject, può essere realizzata attraverso un riferimento, ma anche con il passaggio di un parametro in update() che permetta di risalire a Subject, come effettivamente accade nel Delegation Model, mostrato in figura 15.3, in cui event è il parametro ricevuto nei metodi di KeyListener.

Lo schema mostra la relazione tra classi per un certo tipo di listener: il KeyListener. Il modello è comunque valido per tutti gli altri tipi di Listener (MouseListener, MouseMotionListener, ActionListener, ChangeListener, e così via).

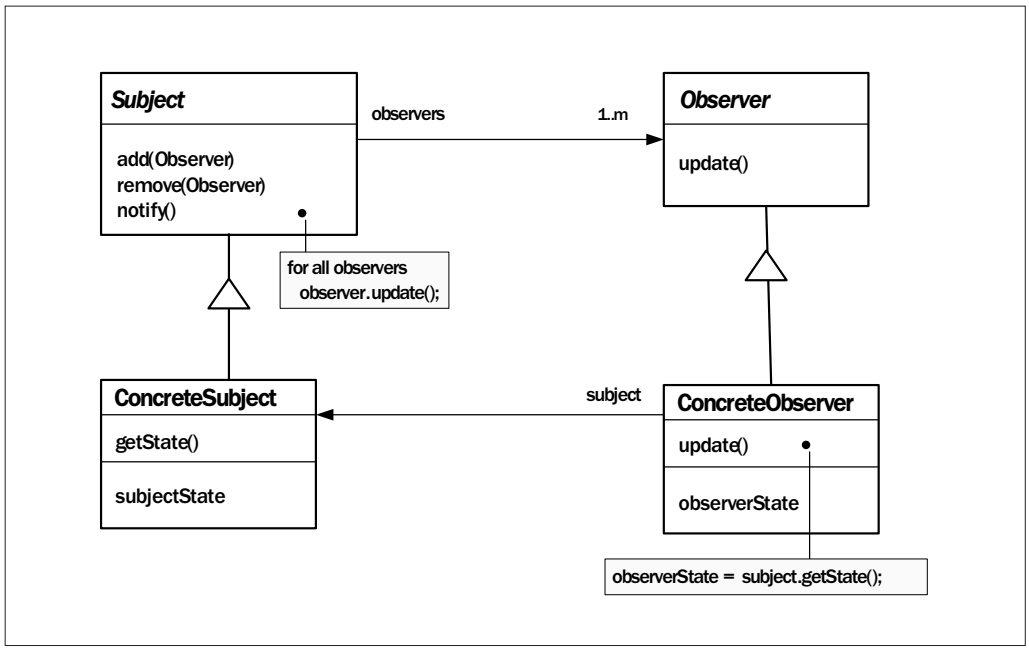

Figura 15.2 – Schema di classi per Observer.

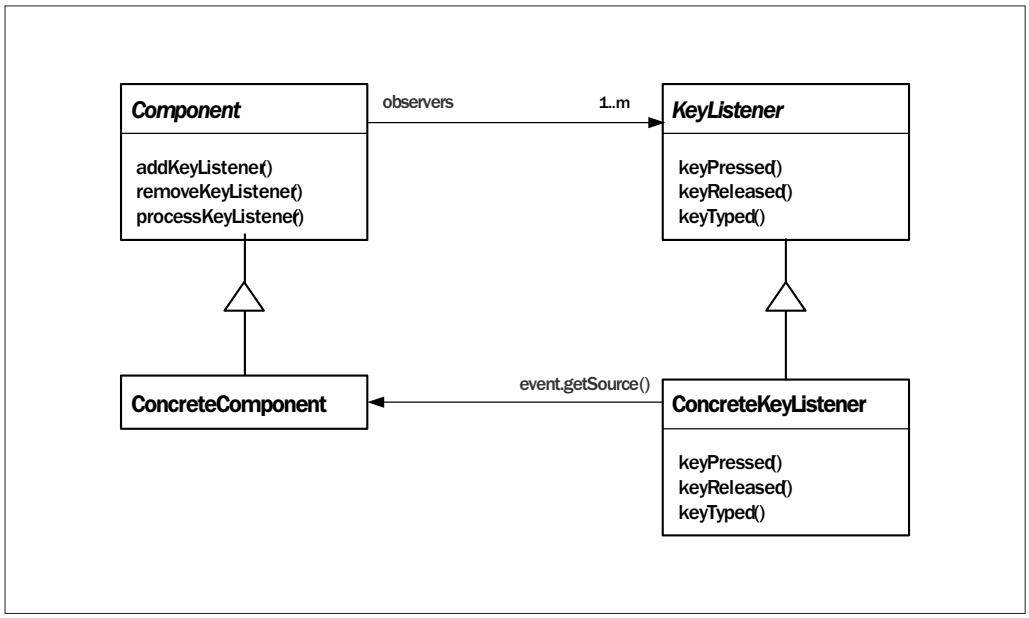

Figura 15.3 – Schema del Delegation Model.

Observer

L'interfaccia KeyListener rappresenta l'elemento Observer, che verrà notificato a ogni evento di tipo KeyEvent registrato in Component. I metodi da concretizzare sono in questo caso tre e corrispondono ai tre tipi di evento possibili attraverso l'utilizzo della tastiera. Si tratta dei metodi keyReleased(), keyTyped() e keyPressed().

Ognuno di questi metodi riceve come parametro un oggetto di tipo KeyEvent, che permette a Observer, tra le altre cose, di ottenere un riferimento sull'oggetto che ha mandato l'evento, attraverso la chiamata getSource() di KeyEvent. Questo realizza implicitamente il collegamento "subject" che nel pattern Observer è stato definito in modo esplicito.

```
void keyPressed(KeyEvent event){
  ...
  event.getSource();
  ...
}
```

Nel design non si considera il fatto che per comodità la classe concreta potrebbe implementare il Listener attraverso una classe intermedia chiamata Adapter (in questo caso KeyAdapter), presente nelle classi della distribuzione Java.

Subject

L'elemento Subject in questo schema è rappresentato da Component. Da notare che Component contiene tutta una serie di metodi addXYZListener(), removeXYZListener() e processXYZEvent() che permettono la gestione di diversi meccanismi paralleli di Observer: per ogni evento esiste una serie di metodi add/remove/process per Observer. Si può parlare anche in questo caso di pattern. Lo chiameremo "add/remove/process" pattern. Permette la realizzazione di più elementi Subject del pattern Observer all'interno della stessa classe (Component).

Supponendo di avere in una nostra applicazione due classi concrete *Listener* (MyKeyListener e MyMouseListener), la loro registrazione all'interno dell'oggetto Subject avverrebbe così:

```
addKeyListener(new MyKeyListener());
addMouseListener(new MyMouseListener());
```

Realizzazione in Java

Vorrei da ultimo far notare che l'utilizzo del pattern Observer all'interno di un'applicazione Java viene facilitato dall'esistenza degli elementi Observable e Observer. Il primo è una classe che implementa alcuni metodi di utilità, tra cui addObserver(), deleteObserver() e notifyObservers() (che potremmo ricondurre al nostro add/remove/process pattern, se non ci fosse un'inconsistenza nella terminologia).

La seconda è la seguente interfaccia, che definisce lo schema di base per un elemento Observer.

```
public interface Observer
{
  public void update(Observable o, Object arg);
}
```

Si può notare che anche in questo caso il legame subject tra Observer e Subject viene realizzato attraverso l'arrivo di un riferimento a Subject (in questo caso rappresentato da Observable) come parametro di update().

Gestione del layout

Continuando la ricerca di design pattern all'interno delle classi Java, ci imbattiamo di nuovo nel pattern Strategy, utilizzato dal sistema di gestione del layout.
In questo caso la conoscenza del pattern è meno determinante per l'utilizzo delle classi associate, visto che la funzionalità di LayoutManager viene sfruttata dal sistema e non direttamente dallo sviluppatore. Interessante è comunque conoscere l'architettura, perché attraverso questa è possibile capire proprio perché non serve conoscere i dettagli del funzionamento o degli algoritmi di layout usati. Inoltre, ovviamente, servirebbe nel caso in cui si desiderasse implementare un proprio layout, concretizzando la classe LayoutManager.

Il pattern Strategy, come descritto nel catalogo e già osservato più volte nei capitoli precedenti, viene mostrato in figura 15.4. Il suo scopo è quello di definire una famiglia di algoritmi interscambiabili. Il client (Context) utilizza uno di questi algoritmi attraverso un'interfaccia comune. L'utilizzo si rivela appropriato quando si desidera diversi algoritmi in tempi diversi con una configurazione dinamica. I vantaggi consistono soprattutto nel fatto che la classe che chiama l'algoritmo non deve conoscere i dettagli dell'implementazione e non deve inserire controlli

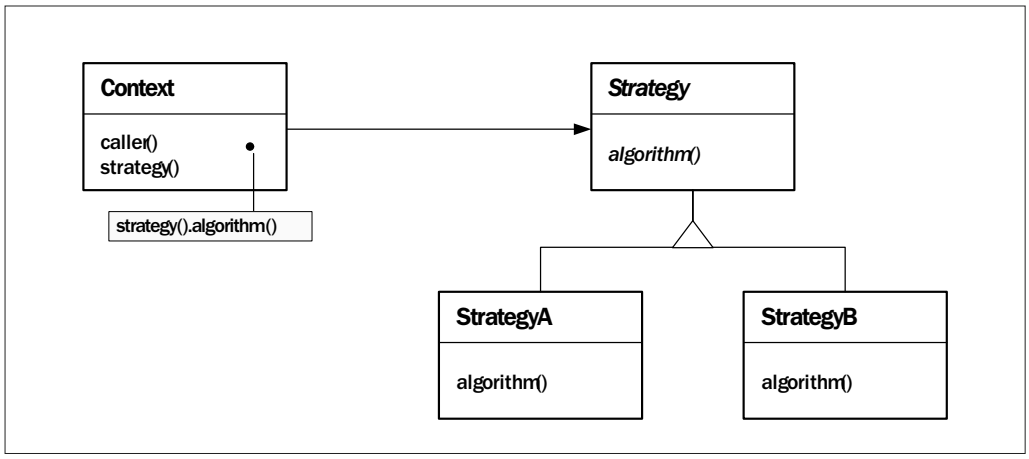

Figura 15.4 – Schema di classi per Strategy.

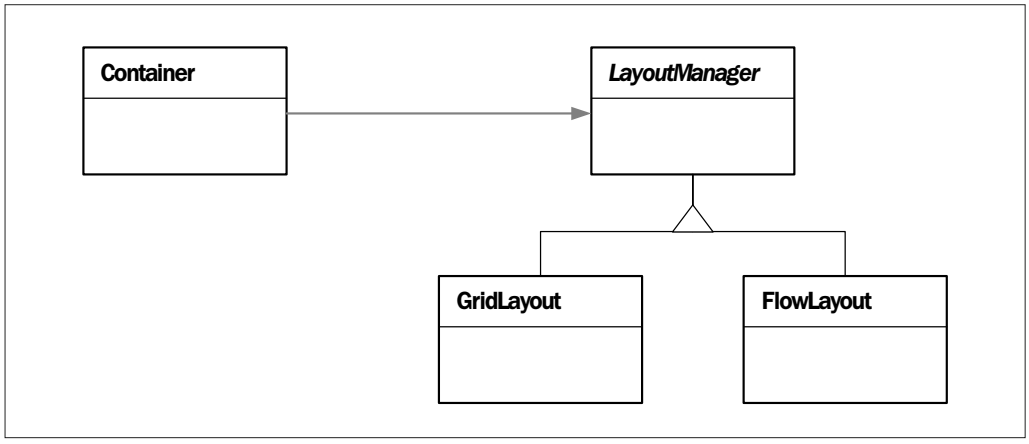

Figura 15.5 – Strategy nel contesto del layout manager.

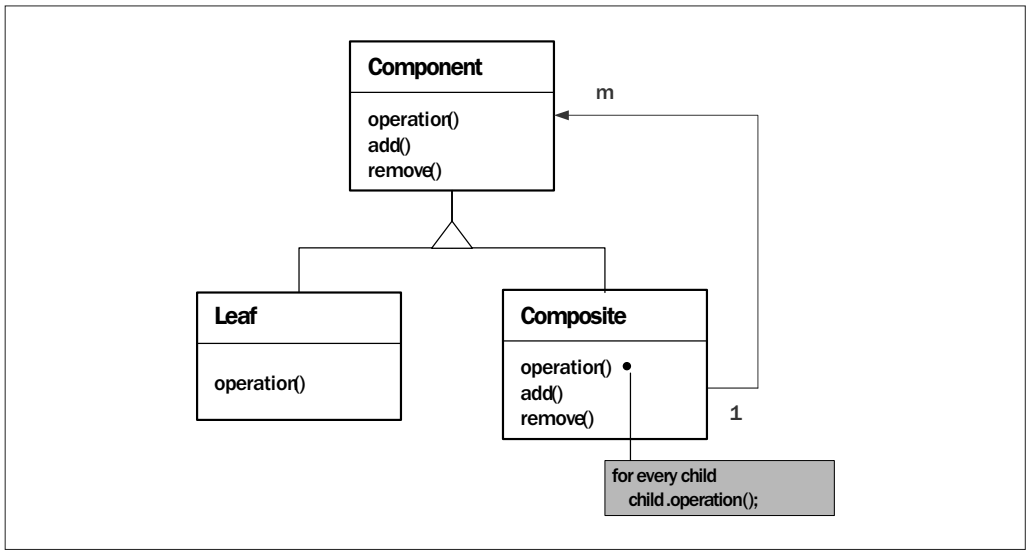

Figura 15.6 – Schema delle classi del pattern Composite.

di condizione per sapere quale algoritmo chiamare: in caso di cambio di algoritmo la classe concreta di Strategy viene semplicemente sostituita. In figura 15.5, il modo in cui la gestione dei layout manager sfrutta il pattern Strategy.

Component e Container

Rimaniamo alle interazioni specificate nel package java.awt per parlare del rapporto tra le classi Component e Container, dalla quale eredita anche JComponent, classe base per tutti i widget di Swing. Anche in questo caso il pattern associato ci permette di comprendere meglio la collaborazione tra le singole classi. Il pattern coinvolto in questo caso è il Composite: lo schema delle classi è illustrato in figura 15.6.

Questo design permette di trattare in modo uniforme oggetti individuali e composizioni di oggetti. È esattamente quanto capita con elementi di tipo Container. Ogni Container è un Component e può contenere liste di Component, che a loro volta possono essere elementi finali della gerarchia di widget (Button, Checkbox, e così via) oppure di nuovo Container, continuando in modo ricorsivo la composizione.

I vantaggi principali di una visione uniforme consistono nel fatto che il codice del client rimane semplice, senza necessità di switch e flag per determinare con quale singolo componente ha a che fare. Inoltre nuovi componenti possono essere aggiunti in modo del tutto tra-

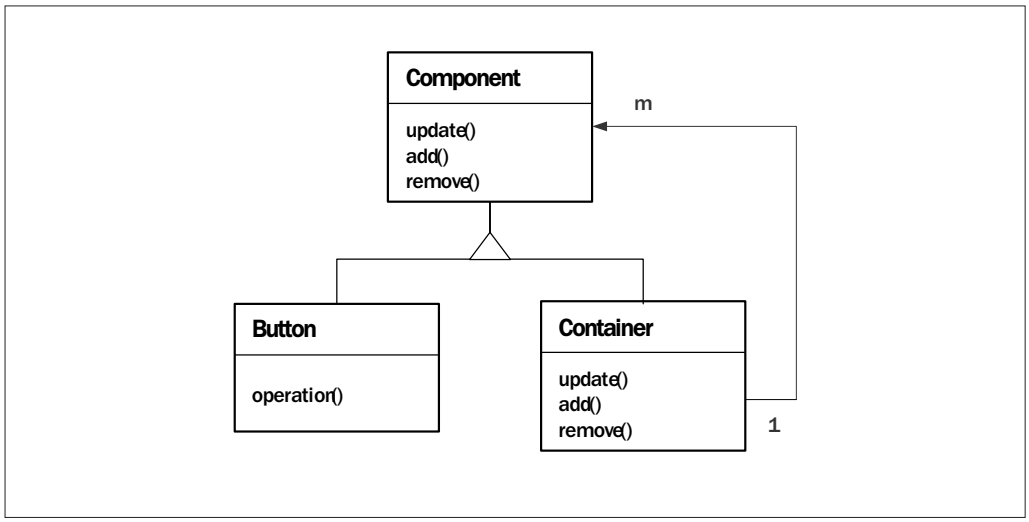

Figura 15.7 – Schema di classi per Composite.

sparente per il client. In figura 15.7, il pattern è applicato alla gerarchia di classi per widget e finestre in java.awt.

Ulteriori pattern

Parecchi altri pattern del catalogo possono essere estratti dalle classi presenti nella distribuzione standard di Java. Oltre a quelli appena visti, mi limito a citarne ancora un paio, senza entrare nel dettaglio dell'architettura.

Il primo è il pattern Bridge utilizzato per separare le due gerarchie di widget nel modello AWT: quella definita con elementi ComponentPeer, cioè le classi che interagiscono direttamente con il windowing system della piattaforma, e quella definita con elementi Component. Questo design è fondamentale per i cosiddetti componenti *heavyweight* (tipici di AWT, contrapposti ai componenti *lightweight* presenti in Swing), perché permette l'adattamento, e di conseguenza il porting, delle classi AWT su diverse piattaforme, senza dover modificare le classi della gerarchia Component.

Anche nel pattern Bridge si nota l'utilizzo della composizione contrapposta all'ereditarietà. Lo scopo è quello di rendere maggiormente dinamica la relazione tra classi. La relazione gerarchica viene usata unicamente per ottenere il polimorfismo e poter quindi definire relazioni più generiche.

Collegato direttamente a questa relazione tra Component e ComponentPeer c'è il meccanismo di creazione dei widget heavyweight che passa dall'implementazione (specifica per ogni piattaforma) di sottoclassi della classe astratta Toolkit. Questa centralizza le operazioni di creazione

dei widget, fungendo da Abstract Factory (nome del pattern), mettendo a disposizione un'interfaccia per creare famiglie di oggetti correlati, senza ricorrere direttamente al nome della classe specifica a cui appartengono gli oggetti.

Nuovi pattern

Finora ci siamo limitati ad estrarre pattern seguendo come modello di riferimento il catalogo iniziale sui design pattern. In realtà qualsiasi modello, prassi di programmazione o architettura ricorrente può essere interpretata come pattern.

Ho già fatto riferimento alla ricorrenza del modello add/remove/process utilizzato in `Component` per permettere alla classe di agire come `Subject` del pattern Observer a più livelli.

Qualcosa di simile si ha con il cosiddetto "get/set" pattern, cioè la prassi di chiamare `getXyz()` e `setXyz()` i metodi pubblici che permettono di accedere a variabili `xyz` con visibilità minore all'interno della stessa classe. Questa prassi è stata introdotta sistematicamente con classi di Java soprattutto per questioni di compatibilità con il meccanismo dei Beans, componenti software Java che hanno la necessità di "mostrarsi" in modo uniforme a programmi esterni che ne vogliano fare uso. L'utilizzo sistematico del pattern get/set permette al JavaBean di comunicare il nome e il tipo di variabile a cui il programma esterno ha accesso.

Navigando all'interno della gerarchia di classi di Swing è possibile identificare un'ulteriore architettura ricorrente, denominata "interface/implementation" pattern. Si tratta di un'archi-

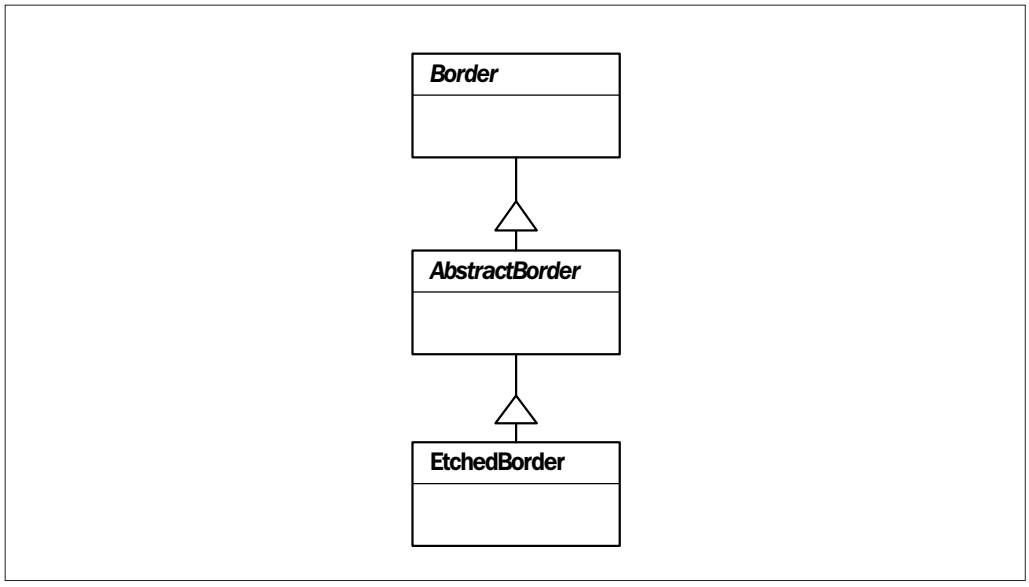

Figura 15.8 – Gerarchia di Border.

tettura interessante, usata nello sviluppo di framework in Java, utile, anche a scopi didattici per capire la differenza nell'utilizzo di interfacce e classi astratte.

Consideriamo subito l'esempio della classe EtchedBorder in Swing (figura 15.8).

L'interfaccia Border è la responsabile di tutte le interazioni con il resto del framework. Ogni componente del framework che utilizza un bordo lo fa attraverso l'astrazione Border, che rappresenta quindi la API di utilizzo per ogni bordo.

Si ottiene con ciò una chiara separazione tra interfaccia e implementazione, visto l'uso di *interface*. L'interfaccia rappresenta la parte stabile del framework. Deve essere fissa perché una sua modifica rappresenterebbe una catena di cambiamenti e adattamenti all'interno del framework e delle applicazioni derivate. È l'uso dell'interfaccia che permette il riutilizzo del design.

La classe astratta AbstractBorder rappresenta invece un livello intermedio, una sorta di implementazione minima in comune tra tutte le classi concrete. Non solo: mette a disposizione un'implementazione di default, permettendo quindi allo sviluppatore che volesse realizzare la classe concreta di concentrarsi sui metodi veramente necessari per differenziare la sua implementazione da altre. C'è a questo livello un riutilizzo parziale sia di design che di implementazione.

L'ultimo livello è quello dell'implementazione vera e propria, che in un framework è costituita da un certo numero di classi cosiddette "out of the box", cioè da classi direttamente utilizzabili dallo sviluppatore di applicazioni. Oltre a EtchedBorder si può trovare EmptyBorder, LineBorder, CompoundBorder, e varie altre. La prassi per chi intendesse implementare una nuova classe di Border sarebbe di ereditare da AbstractBorder. Niente impedirebbe comunque di ereditare da una classe concreta esistente, nel caso si trattasse di una modifica minima.

Siccome un pattern del genere può rischiare di generare un numero molto elevato di tipi, sarebbe da utilizzare unicamente per astrazioni chiave del sistema, come del resto viene fatto in Swing. Un altro esempio in Swing consiste nell'astrazione TableModel – AbstractTableModel – DefaultTableModel.

In questo capitolo…

In questo capitolo si è voluto mostrare che in ogni framework e, più in generale, in ogni architettura, è possibile isolare una serie di pattern. Questo permette di comprendere meglio le collaborazioni e le interazioni tra le classi che compongono il design e contribuisce alla comprensione dell'architettura nel suo insieme, migliorando notevolmente la documentazione del sistema.