Capitolo 23
Refactoring complessi

Dopo aver visto nel capitolo precedente due esempi relativamente semplici di refactoring, vediamone uno un po' più complesso in questo capitolo.

Consideriamo complesso un refactoring che abbia un forte impatto sul design finale. Ho già avuto modo di dire nel corso del libro che la relazione che intercorre tra design pattern e refactoring consiste nel fatto che l'introduzione di un pattern nel design dell'applicazione è spesso il risultato di un refactoring complesso.

In questo capitolo vedremo un caso concreto. Tratteremo con un esempio i passaggi contenuti nel refactoring Separate Domain From Presentation.

Modello MVC

Lo scopo dichiarato del refactoring scelto è quello di separare in un programma la logica dell'applicazione dall'interfaccia utente. Prima di addentrarci nell'esempio concreto, dobbiamo riprendere brevemente il paradigma Model – View – Controller (MVC). Diventato famoso con Smalltalk all'inizio degli anni Ottanta, il modello MVC rappresenta un modo per suddividere un'applicazione, o anche solo parte di essa, in tre elementi: il modello, la visualizzazione (o le diverse visualizzazioni) del modello e il controller.

Lo spunto era quello di creare una sequenza del tipo *input* —> *processing* —> *output* anche per il mondo delle interfacce grafiche. Così si è arrivati alla corrispondente sequenza *controller* —> *model* —> *view*. L'input dell'utente, la modellizzazione del mondo esterno e la visualizzazione del modello vengono così separate.

Il controller interpreta i segnali in arrivo dall'utente attraverso mouse o tastiera, e li trasforma in comandi inviati al modello. Il modello gestisce la struttura dati e risponde all'input dell'utente con le modifiche richieste, notificando poi i cambiamenti avvenuti all'interfaccia (o alle diverse interfacce che rappresentano lo stesso modello), che si adatterà alla nuova situazione interna.

È il modello che rappresenta e approssima nel programma gli oggetti del mondo reale. La view è invece un modo per rappresentare il modello, attraverso il rendering dello stesso sul display del device grafico scelto. Lo stesso modello può essere rappresentato contemporaneamente da più visualizzazioni. È importante che il modello sia in grado di informare tutti i suoi device di visualizzazione ogni volta che avviene un cambiamento interno. Il paradigma MVC permette la separazione tra dati, la loro manipolazione e la loro visualizzazione.

MVC, in quanto modello architetturale, può essere considerato a sua volta un pattern, anche se è una combinazione di diversi pattern (modelli del genere vengono anche chiamati *patterns of patterns*). La relazione che ci interessa maggiormente è quella esistente tra modello e sue visualizzazioni, ovvero tra logica del programma e rappresentazione degli elementi. Questa relazione viene descritta dal pattern Observer.

Pattern Observer

Il pattern Observer, già affrontato in capitoli precedenti, definisce una dipendenza *1-m* tra oggetti: quando l'elemento centrale (soggetto, modello) modifica il suo stato, gli altri (observer, listener, view) ne vengono informati. Riportiamo di nuovo l'architettura del pattern in figura 23.1.

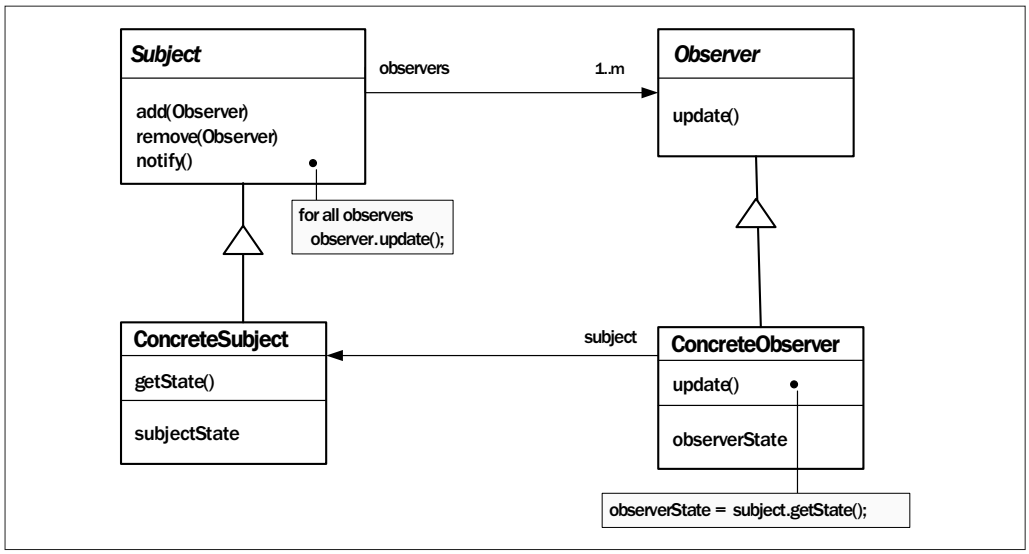

Figura 23.1 – Schema di classi del pattern Observer.

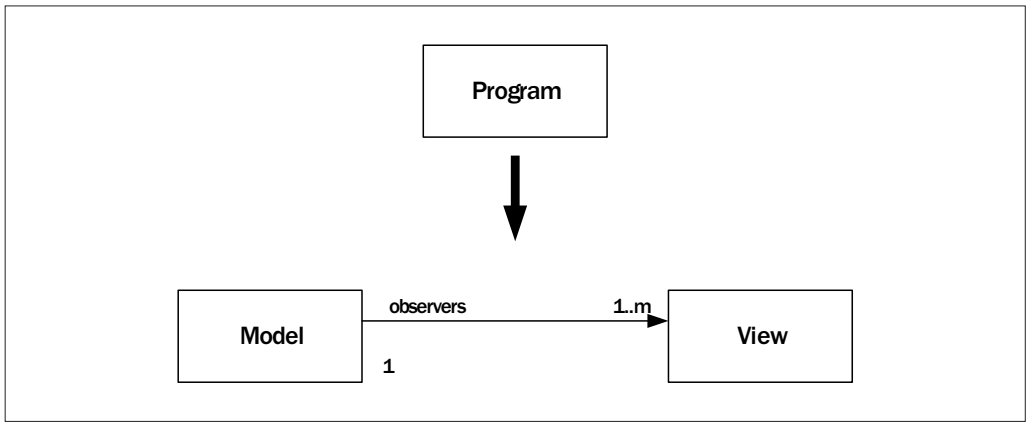

Figura 23.2 – Struttura di "Separate Domain From Presentation".

Si utilizza Observer quando la modifica di un oggetto rende necessarie le modifiche di altri oggetti, ma non si sa quanti sono. Inoltre questo pattern è particolarmente utile quando questa dipendenza è dinamica, quando cioè un elemento deve notificare i suoi cambiamenti, senza dover necessariamente conoscere gli altri oggetti.

Queste sono le conseguenze dell'utilizzo di Observer:

- Observer e Subject possono essere modificati in modo indipendente. La loro relazione è di tipo *has-a* ("favor object composition over class inheritance").
- Nuovi elementi di tipo Observer possono essere aggiunti dinamicamente, senza conseguenze per l'elemento Subject.
- Gli oggetti di tipo Observer possono essere parte di qualsiasi gerarchia di classe.
- Il modello supporta l'idea di broadcasting (limitato agli elementi che l'elemento Subject gestisce nella sua lista). Del resto questa idea è intrinseca nella relazione Model – View, modellata dal pattern Observer.

Esempio

Nel nostro esempio andremo a modificare un programma trasformandolo al suo interno da struttura unica a struttura modulare, in cui modello e presentazione (view) siano separati. Il refactoring Separate Domain From Presentation ci permetterà di passare da una struttura "massiccia", anche detta *legacy*, a una struttura più modulare, con la presenza di modello e view come entità separate (figura 23.2).

Passaggi

Elenchiamo di seguito quali sono i passaggi da usare per la trasformazione desiderata. Cercheremo poi di applicarli nel nostro esempio pratico.

- Modificare la vecchia GUI (quella nella soluzione legacy) in modo che diventi observer del modello (separazione *observer/observable*).
- Compilare ed eseguire i test.
- Isolare l'accesso ai dati del modello all'interno della GUI con metodi `get/set`.
- Compilare ed eseguire i test.
- Creare i nuovi campi nel modello con i metodi di accesso `get/set` e spostare nel modello le funzioni dell'applicazione.
- Compilare ed eseguire i test.

Descrizione

L'esempio non è particolarmente originale, ma serve perfettamente a dare l'idea di come vadano eseguite le trasformazioni. Supponiamo che la situazione di partenza sia quella di un programma con un'interfaccia grafica contenente tre campi di testo. I contenuti dei campi di testo sono in relazione tra di loro in questo modo: il valore intero del terzo campo corrisponde alla differenza tra il primo e il secondo.

Già da questa breve descrizione si intuisce qual è la logica del programma (relazione tra valori interi dei campi di testo) e qual è la presentazione (campi di testo).

Vediamo una possibile, seppur parziale, implementazione della funzionalità richiesta, utilizzando Swing come framework di interfaccia grafica.

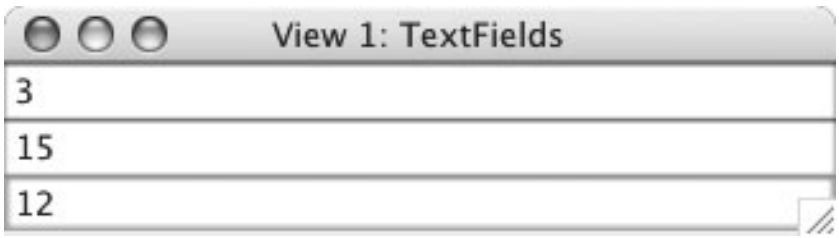

Figura 23.3 – Interfaccia grafica con campi di testo.

```
public class Program extends JFrame
{
  private JTextField fStartField, fEndField, fDifferenceField;

  Program(String title){
    super(title);
    ...
  }

  ...

  void startCheck(){
    ...
    computeDifference();
  }

  void endCheck(){
    ...
    computeDifference();
  }

  void differenceCheck(){
    ...
    computeEnd();
  }

  void computeDifference(){
    int start = Integer.parseInt(fStartField.getText());
    int end = Integer.parseInt(fEndField.getText());
    fDifferenceField.setText(String.valueOf(end - start));
  }

  void computeEnd(){
    int start = Integer.parseInt(fStartField.getText());
    int difference = Integer.parseInt(fDifferenceField.getText());
    fEndField.setText(String.valueOf(start + difference));
  }
}
```

La classe Program, rappresentante l'intero programma legacy, è una finestra (JFrame) che contiene i tre campi di testo (JTextField). I metodi computeDifference() e computeEnd() servono a calcolare il nuovo valore, rispettivamente, dei campi fDifferenceField (ultimo campo) e fEndField. Quando l'utente modifica il contenuto di fStartField, oppure il contenuto di fEndField, bisogna

chiamare computeDifference() (le chiamate sono eseguite dai metodi startCheck() e endCheck() che verranno attivati dagli eventi che definiremo). Quando invece l'utente modifica il campo fDifferenceField, bisogna chiamare computeEnd() (attraverso differenceCheck(), attivato dall'evento associato alla modifica del campo in questione). Come attivare queste funzioni? Si tratta di sapere come Swing gestisce gli eventi. Distinguiamo due possibilità:

1. Dopo che l'utente ha modificato il valore nel campo di testo, preme il tasto "enter" della tastiera. In questo caso Swing associa un evento di tipo ActionEvent al campo di testo. Significa che per attivare un'azione bisogna implementare una classe che erediti da ActionListener e sovrascriva il metodo actionPerformed(). Ad ognuno dei tre campi di testo bisognerà associare l'azione desiderata. Consideriamo come esempio l'implementazione di ActionListener da associare al campo fEndField.

2. Dopo che l'utente ha modificato il valore nel campo di testo, esce dal campo utilizzando il mouse. In questo caso l'evento attivato da Swing è il FocusEvent. Per associare un'azione dobbiamo perciò implementare una classe che implementi FocusListener (o che erediti da FocusAdapter) che sovrascriva il metodo focusLost().

Mostriamo entrambi i casi nel codice che segue, con le classi necessarie per il campo fEndField. Da notare che entrambe le classi sono interne a Program, e possono così chiamare senza problemi il metodo endCheck(), responsabile dell'azione di controllo del campo fEndField. Analogamente ci saranno due classi StartFocus e StartAction che chiameranno nei loro metodi di attivazione startCheck(), e due classi DifferenceFocus e DifferenceAction che chiameranno differenceCheck().

```
class EndFocus extends FocusAdapter
{
  public void focusLost(FocusEvent event){
    endCheck();
  }
}

class EndAction implements ActionListener
{
  public void actionPerformed(ActionEvent event){
    endCheck();
  }
}
```

Primo passaggio di trasformazione

Siamo pronti per un primo passaggio di trasformazione. Il primo punto prevede quanto segue: "Modificare la vecchia GUI (quella nella soluzione legacy, cioè la classe Program) in

modo che diventi observer del modello (separazione *observer/observable*)". Diventa perciò necessario crearsi un modello. Il framework Java mette a disposizione due elementi pensati proprio per questo scopo: la classe Observable e l'interfaccia Observer. Sarebbe consigliabile usare questi due elementi ogni volta che si desidera implementare una relazione di questo tipo, anche perché vengono gestiti eventuali conflitti dovuti all'utilizzo di più thread.

Per mostrare in modo più esplicito cosa succede, vale però la pena, unicamente in questo caso, implementare noi questo tipo di funzionalità. Iniziamo con la classe Model, che deve poter gestire una lista di Observer.

```
public class Model
{
  private List fObservers = new ArrayList();

  public void addObserver(Observer observer){
    fObservers.add(observer);
  }

  protected void  notifyObservers(){
    Iterator iter = fObservers.iterator();
    while(iter.hasNext()){
      Observer ob = (Observer)iter.next();
      ob.update(this);
    }
  }
}
```

Ogni nuovo Observer viene aggiunto alla lista tramite una chiamata ad addObserver(). Quando il modello viene modificato, chiama il suo metodo notifyObservers(), che ha quale scopo l'aggiornamento degli oggetti Observer tramite l'attivazione del meccanismo di update presente in ognuno. Definiamo invece di tipo Observer l'interfaccia di tutti gli elementi che rispondono alla chiamata update().

```
public interface Observer
{
  public void update(Model model);
}
```

Ogni view deve implementare Observer e di conseguenza il metodo update(), che avrà quale compito quello di informarsi sulle modifiche avvenute nel modello e aggiornare di conseguenza la propria rappresentazione. La nostra classe Program (che ora chiameremo View) viene modificata leggermente, inserendo il riferimento al modello.

```
public class View extends JFrame implements Observer
```

```
{
  private JTextField fStartField, fEndField, fDifferenceField;
  private Model fModel;

  View(String title, Model model){
    super(title);
    ...
    fModel = model;
    fModel.addObserver(this);
    update(model);
  }

  public void update(Model model){
  }
}
```

Secondo passaggio di trasformazione

Il secondo passaggio di trasformazione prevede quanto segue: "Isolare l'accesso ai dati del modello all'interno della GUI con metodi get/set".

Creazione dei metodi

Nel nostro caso il modello è ancora rappresentato nei campi di testo fStartField, fEndField, fDifferenceField. Si tratta, in un primo passaggio, di isolare con get/set l'accesso a questi dati, in modo che sia poi possibile spostare la logica senza conseguenze dovute a dipendenze dirette con i campi di testo.

Detto in altre parole: invece di accedere al contenuto del campo di testo fEndField attraverso l'accesso alla variabile e al suo metodo getText(), come segue

```
fEndField.getText()
```

definiamo i metodi di accesso getEnd() e setEnd():

```
String getEnd(){
  return fEndField.getText();
}

void setEnd(String arg){
  fEndField.setText(arg);
}
```

Sostituzione dell'accesso diretto

Il passaggio successivo è quello di sostituire con chiamate a set/get tutti gli accessi diretti alle variabili coinvolte. Consideriamo i due metodi computeDifference() e computeEnd(), mostrati in precedenza, e inseriamo le modifiche. Questo è quanto risulta:

```
void computeDifference(){
  int start = Integer.parseInt(getStart());
  int end = Integer.parseInt(getEnd());
  setDifference(String.valueOf(end - start));
}

void computeEnd(){
  int start = Integer.parseInt(getStart());
  int difference = Integer.parseInt(getDifference());
  setEnd(String.valueOf(start + difference));
}
```

Aggiornamento con input dell'utente

C'è un ulteriore passaggio, in questa seconda trasformazione, che va affrontato. Quando l'utente esegue una modifica in interfaccia, ad esempio interviene sul contenuto del campo fEndField, per ora la modifica avviene contemporaneamente sia in interfaccia che nel modello, visto che sono la stessa cosa. Il modello non ha quindi nessuna necessità di informare la view di un avvenuto cambiamento. Prevedendo che view e model diventeranno due cose distinte, possiamo già prevedere il passaggio di informazione, inserendo l'assegnazione del valore in endCheck(), metodo della classe View, chiamato dai gestori di evento ActionListener o FocusAdapter.

```
void endCheck(){
  setEnd(fEndField.getText());
  computeDifference();
}
```

Momentaneamente viene assegnato a sé stesso il valore di fEndField. Più avanti la cosa cambierà. Infatti la chiamata setEnd() andrà a modificare il campo del modello, al quale si passerà il nuovo valore scelto dall'utente, prima di calcolare la nuova differenza.

Terzo passaggio di trasformazione

Il terzo passaggio prevede: "Creare i nuovi campi nel modello con i metodi di accesso get/set e spostare nel modello le funzioni dell'applicazione". È il passaggio che serve a creare il modello vero e proprio. Finora il modello era vuoto. La classe Model non aveva campi, se escludiamo la lista utilizzata per gestire gli oggetti Observer.

Nuovi campi

La prima azione è dunque quella di inserire i campi nel modello.

```
class Model
{
   private List fObservers = new ArrayList();
   private String fStart, fEnd, fDifference;

   Model(){
      fStart = "0";
      fEnd = "0";
      fDifference = "0";
   }

   …
}
```

Per comodità gestiamo come stringhe le informazioni nel modello. Ora si tratta di definire anche qui i metodi di accesso. Vediamo unicamente l'esempio per il campo fEnd. I nomi dei metodi (getEnd() e setEnd()) sono volutamente gli stessi già usati per i get/set della classe View, per evitare di dover adattare i metodi che contengono la chiamata, che verranno spostati da View a Model.

```
String getEnd(){
   return fEnd;
}

void setEnd(String arg){
   fEnd = arg;
   notifyObservers();
}
```

Da notare che il metodo setEnd(), oltre che aggiornare il valore del campo fEnd, chiama notifyObservers(), visto in precedenza, che ha il compito di iniziare il meccanismo di aggiornamento degli oggetti Observer.

Spostamento della logica

La logica dell'applicazione è rappresentata dai metodi computeDifference() e computeEnd(), che possono essere spostati direttamente nel modello, senza apportare modifiche, visto che ritrovano l'implementazione dei metodi get/set appena specificati nel modello. Lo spostamento della logica dalla view al modello ha però come conseguenza la necessità di "collegare" le chiamate get/set della view con quelle del modello.

Riprendiamo l'esempio del campo fEndField di View. Questi sono i get/set precedenti:

```
String getEnd(){
  return fEndField.getText();
}

void setEnd(String arg){
  fEndField.setText(arg);
}
```

Queste sono le nuove versioni, che al loro interno fanno riferimento ai corrispondenti get/set del modello:

```
String getEnd(){
  return fModel.getEnd();
}

void setEnd(String arg){
  fModel.setEnd(arg);
}
```

Una trasformazione analoga avviene per i metodi computeEnd() e computeDifference() di View, che conterranno una chiamata di delega ai corrispondenti metodi spostati nel modello. Ora possiamo inserire nella view l'implementazione di update(), cioè la reazione che deve avere una view ad ogni notifica di cambiamento del modello:

```
public void update(Model model){
  fStartField.setText(getStart());
  fEndField.setText(getEnd());
  fDifferenceField.setText(getDifference());
}
```

Ulteriori presentazioni

I passaggi di refactoring sono terminati. A questo punto abbiamo un'architettura divisa in due livelli: view e model. Per mostrare l'utilità di questo approccio, implementiamo un secondo tipo di view, che mostra all'utente lo stesso modello, ma in altro modo. Se il primo tipo di view prevedeva tre campi di testo da editare, questo secondo tipo presenterà i dati sotto forma di *slider* come mostrato nell'immagine di figura 23.4.

La cosa interessante è che il modello non va affatto modificato, visto che è perfettamente indipendente dalla sua presentazione. Va quindi implementata una seconda view che sia in grado unicamente di gestire i suoi campi di visualizzazione e che implementi una sua versione del metodo update() di aggiornamento.

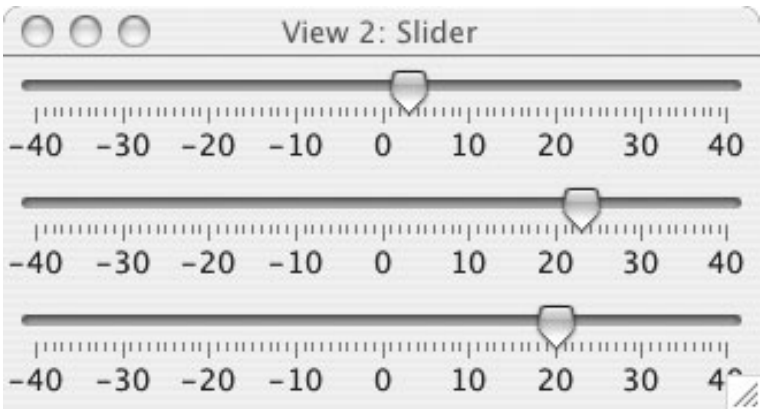

Figura 23.4 – Interfaccia grafica con slider.

```
public class View2 extends JFrame implements Observer
{
  private JSlider fStartSlider, fEndSlider, fDifferenceSlider;

  View2(String title, Model mod){
    …
  }

  public void update(Model model){
    fStartSlider.setValue(Integer.parseInt(getStart()));
    fEndSlider.setValue(Integer.parseInt(getEnd()));
    fDifferenceSlider.setValue(Integer.parseInt(getDifference()));
  }

  void startCheck(){
    setStart(String.valueOf(fStartSlider.getValue()));
    computeDifference();
  }

  void endCheck(){
    setEnd(String.valueOf(fEndSlider.getValue()));
    computeDifference();
  }
```

```
  void differenceCheck(){
    setDifference(String.valueOf(fDifferenceSlider.getValue()));
    computeEnd();
  }
```

La gestione dell'evento avviene qui non con ActionListener o FocusAdapter, ma con ChangeListener, associato all'oggetto slider corrispondente. Anche qui lo scopo è quello di attivare i metodi startCheck(), endCheck() e differenceCheck().

```
class StartChange implements ChangeListener
{
  public void stateChanged(ChangeEvent e) {
    startCheck();
  }
}

class EndChange implements ChangeListener
{
  public void stateChanged(ChangeEvent e) {
    endCheck();
  }
}

class DifferenceChange implements ChangeListener
{
  public void stateChanged(ChangeEvent e) {
    differenceCheck();
  }
}
```

In questo capitolo…

In questo capitolo abbiamo trattato un refactoring più complesso dei precedenti, per mostrare che è possibile introdurre un design pattern all'interno di un design preesistente, mantenendo i rischi sotto controllo. Con il pattern Observer abbiamo diviso il codice legacy di un'applicazione e realizzato un meccanismo model – view pulito, che ci ha permesso in un secondo tempo di aggiungere un tipo diverso di visualizzazione senza dover metter mano alla logica (modello) dell'applicazione.