Capitolo 13
Pattern per il Web

All'elenco iniziale dei pattern, che rappresenta tuttora un punto di riferimento più che valido, si sono aggiunti altri elenchi e altre pubblicazioni che analizzano varianti di architetture note, oppure modelli completamente nuovi, specifici per i diversi ambiti di applicazione.

Architetture Web

Un campo sempre più percorso, anche a livello di architetture software, è quello delle applicazioni Web. Nel suo libro sul tema dei pattern per applicazioni enterprise [Fowler-2002] Martin Fowler ha dedicato a questo tipo di applicazioni una serie di sei modelli.

Ne vedremo due, legati all'esempio sviluppato e discusso nella Parte I, in cui si trattava di generare il contenuto della fattura di un noleggio in formato HTML. Prima di presentare i due pattern è però necessaria una breve introduzione sull'architettura Model – View – Controller (MVC) relativa al Web.

Architettura MVC

Questo tipo di modello, non nuovo, introdotto già con Smalltalk e ripreso da altri framework, ad esempio Swing, considera tre ruoli: *model*, *view* e *controller*. Il controller legge l'input dell'utente e lo passa al modello, che gestisce i dati dell'applicazione. Da qui le modifiche vengono propagate alla view per essere mostrate all'utente.

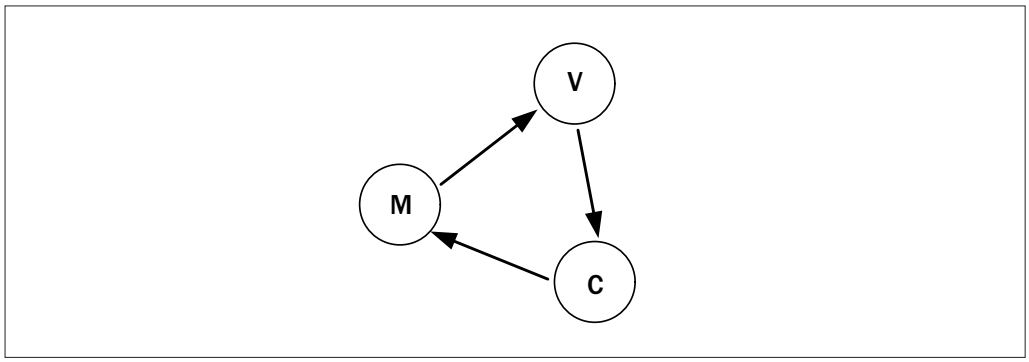

Figura 13.1 – Schema MVC.

Il modello è quindi un elemento che non si vede, ma che può essere rappresentato in interfaccia da una sua "vista" (*view*), o da più viste, anche diverse tra di loro nel modo in cui presentano il modello. La view è qualcosa di passivo. L'interazione con l'utente viene gestita dal controller, che quindi, assieme alla view, costituisce l'interfaccia (GUI).

Separare il modello dalla view è senza dubbio uno degli argomenti chiave nel design del software. Si tratta infatti di due elementi con scopi e contesti diversi, che è meglio implementare in maniera individuale. Anche le modalità di test sono diverse ed è quindi più semplice verificare i due elementi in modo indipendente. Il modello gestisce dei dati, mentre la view li mostra all'utente. Inoltre potrebbe essere utile rappresentare gli stessi dati (modello) in vari modi, completamente diversi tra di loro. Creare una dipendenza troppo forte tra modello e vista sarebbe un impedimento. La relazione tra modello e view viene spesso implementata con il pattern Observer, visto in precedenza, che permette la propagazione di un evento di modifica dal modello alla vista (oggetto observer).

Controller per architetture Web

In un'architettura Web con contenuto dinamico, le viste sono le pagine HTML mostrate dal browser, il modello sono i dati gestiti sul server (*business logic*) e il controller è l'intero meccanismo di gestione del passaggio da una pagina all'altra, visto che l'interazione con client gestiti da un browser avviene seguendo il paradigma a pagine.

Se parliamo in termini di Java, il controller può essere fatto corrispondere alla servlet che gestisce la richiesta dell'utente. Quando quest'ultimo invia una richiesta, la servlet si occupa di propagare l'informazione ai dati dell'applicazione (modello) e in seguito seleziona (o genera) il tipo di visualizzazione da inviare al browser. Vediamo un semplice esempio.

```
public class Controller extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
```

```
        throws IOException, ServletException{
      String searchArgument = request.getParameter("search");
      if (searchArgument == null){
        getServletContext().getRequestDispatcher("/noArgError.html").forward(req,res);
      }else{
        String htmlDBResultView = createHtmlView(queryDB(searchArgument));
        res.setContentType("text/html");
        res.getWriter().println(htmlDBResultView);
      }
    }
}
```

La classe Controller rappresenta la servlet principale che gestisce l'input che arriva dal client. Ogni servlet gestisce in doGet() le richieste HTTP di tipo GET, quindi a Controller basta ereditare da HttpServlet e riscrivere doGet() per essere interpellata durante una richiesta di questo tipo.

Nel caso specifico, la servlet controlla se è stato inviato l'argomento search da utilizzare per un'interrogazione a un database inglobata nella sequenza di richieste ipotetiche createHtmlView() (view) e queryDB() (model). Se invece l'argomento non è stato inviato alla servlet, quest'ultima invia al client il contenuto del file noArgError.html.

Pattern Transform View

Questo pattern presenta un modo per gestire la trasformazione di dati che arrivano dal modello in dati HTML (view) per l'utente finale.

Il programma ha il compito di navigare attraverso la struttura del modello, riconoscere ogni singolo elemento del modello e generare la sequenza di tag HTML che rappresenti la visualizzazione dei dati. Si tratta quindi di prevedere un modo per trasformare un modello in HTML partendo da una struttura nota a entrambi.

Grafica o programmazione

Molti modelli di programmazione Web sono nati con lo scopo di separare nel miglior modo possibile il lavoro dei grafici da quello degli sviluppatori. È un'esigenza riconosciuta che ha portato a tecnologie basate sulla view, come JSP, PHP, ColdFusion e altro. L'idea di base è quella di permettere al Web designer di eseguire modifiche di tipo grafico senza dover chiedere modifiche nel codice a chi gestisce lo sviluppo dell'applicazione. L'applicazione nasce come sequenza di pagine HTML, all'interno delle quali si inseriscono elementi di interazione con il resto dell'applicazione.

Questo tipo di approccio non è però esente da rischi. Il più grave è quello di inserire troppo codice applicativo nelle view. Questo codice, eseguito sul server prima che il contenuto HTML venga inviato al browser del client, dovrebbe essere ridotto ai minimi termini e fungere unicamente da collegamento tra la parte grafica e il programma vero e proprio che gira sul server. Spesso però, proprio perché l'applicazione, dal punto di vista del workflow, si sviluppa attorno alla

> sequenza di videate HTML, esiste la tentazione di inserire sempre più funzionalità nel codice delle pagine, con i problemi di test e di manutenzione che ben si possono immaginare.
> Il pattern Transform View si sviluppa invece attorno ai dati, piuttosto che attorno alla view, lasciando il programma completamente allo sviluppatore. La separazione tra grafica e programmazione può essere gestita dai meccanismi di trasformazione.

Di regola ci sono diverse trasformazioni da eseguire all'interno di una struttura dati. Ci si può quindi immaginare un ciclo nel quale venga generata una trasformazione per ogni elemento incontrato nel modello. Nonostante l'idea di trasformazione sia del tutto neutra, quindi realizzabile con qualsiasi linguaggio di programmazione, al momento la tecnica più diffusa è rappresentata da XSLT. XSLT (eXtensible Stylesheet Language Transformation) è un linguaggio funzionale che serve a invocare una funzione di trasformazione su ogni elemento riconosciuto del dominio di partenza. Esistono parecchi motori XSLT che eseguono trasformazioni a partire da una struttura XML e una descrizione delle trasformazioni (XSLT style sheet).

L'idea è quindi quella di utilizzare motori come XSLT per applicare trasformazioni al documento XML di input (che descrive la struttura, il modello) e generare HTML (view) che può poi essere inviato come risposta ad una richiesta HTTP. Il fatto di utilizzare un meccanismo di trasformazione consente di mantenere una chiara separazione tra model e view, permettendo di eseguire modifiche di impostazione grafica in modo indipendente dai dati dell'applicazione. Inoltre diventa più semplice scrivere metodi di test, visto che non serve un'infrastruttura completa (Web server), per generare l'output e verificarne la correttezza.

Esempio

Torniamo all'esempio della generazione della fattura, sia in formato testo che in formato HTML, trattato nella Parte I. Nella soluzione proposta in precedenza abbiamo utilizzato il pattern Template per descrivere gli elementi in comune (metodo `valore()`) e delegare a sottoclassi eventuali differenze (metodi `intestazione()`, `dettaglioNoleggio()` e `chiusura()`).

```
abstract class Resoconto
{
  public String valore(Cliente cliente){
    String testo = intestazione(cliente);
    Iterator iterNoleggi = cliente.getNoleggi().iterator();
    while(iterNoleggi.hasNext()){
      Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
      testo += dettaglioNoleggio(ogniNoleggio);
    }
    testo += chiusura(cliente);
    return testo;
  }

  abstract String intestazione(Cliente cliente);
```

```
    abstract String dettaglioNoleggio(Noleggio noleggio);

    abstract String chiusura(Cliente cliente);
}
```

Guardando bene, il pattern Template, oltre che descrivere il metodo di base, ci ha permesso di specificare la struttura della fattura, condivisa tra tutte le diverse view (plain text e HTML). Non è quindi difficile descrivere gli elementi della fattura in XML, specificando unicamente il modello, senza entrare nei dettagli degli elementi grafici. Potremmo ad esempio usare i seguenti tag, già applicati a un output di esempio:

```
<intestazione> Resoconto di noleggio per Pinco Pallino </intestazione>

<dettaglio>
  <singolo> Stöckli 2B   55.0 </singolo>
  <singolo> Völkl BX  49.5 </singolo>
</dettaglio>

<chiusura>
<totale> Ammontare dovuto:  Fr. 104.5 </totale>
<bonus> Ha guadagnato 2 punti di bonus </bonus>
</chiusura>
```

Riutilizzando il codice previsto per i resoconti in testo semplice e HTML, potremmo implementare una nuova classe ResocontoXml, da utilizzare come unico generatore di dati del modello. I dati, generati sulla base dei noleggi effettivi, vengono poi passati al trasformatore che ne genera una view nel formato desiderato.

Vediamo l'esempio di generazione HTML, tramite l'adattamento del main() già mostrato nella prima parte.

```
public class Main
{
  public static void main(String[] args){
    Noleggio noleggio1 = new Noleggio(new Sci("Stöckli 2B", 170, Sci.NORMALE),4);
    Noleggio noleggio2 = new Noleggio(new Sci("Völkl BX", 110, Sci.BAMBINI),5);
    Cliente cliente = new Cliente("Pinco Pallino");
    cliente.addNoleggio(noleggio1);
    cliente.addNoleggio(noleggio2);
    XsltTransformer trasformatore = new XsltTransformer("toHtml.xsl");
    String fattura = trasformatore.esegui(new ResocontoXml().valore(cliente));
    System.err.println(fattura);
  }
}
```

Applicazione Web

Questo codice si integra bene in un'applicazione Web, in cui l'elemento servlet rappresenta il controller, come già mostrato in precedenza.

```
public class Controller extends HttpServlet
{
  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    Cliente cliente = estraiDati(req)
    if (cliente != null){
      XsltTransformer trasformatore = new XsltTransformer("toHtml.xsl");
      String fattura = trasformatore.esegui(new ResocontoXml().valore(cliente));
      res.setContentType("text/html");
      res.getWriter().println(fattura);
    }else{
      getServletContext().getRequestDispatcher("/error.html").forward(req,res);
    }
  }
}
```

Pattern Two Step View

L'evoluzione naturale del pattern precedente è rappresentata dal pattern Two Step View. In questo caso la trasformazione avviene in due passaggi: il primo ha lo scopo di preparare la struttura logica della pagina, il secondo ha lo scopo di trasformare la pagina in HTML.

Questo doppio passaggio si rivela importante quando l'applicazione ha un *look* consistente in tutte le sue pagine. Una modifica alla visualizzazione dell'intera applicazione può essere eseguita intervenendo unicamente nel secondo passaggio di trasformazione. Modificando il passaggio da pagina logica a HTML (passaggio di rendering o formattazione grafica), si riesce a ottenere un'unica modifica globale del *look & feel* dell'applicazione.

D'altra parte, questa separazione in due passaggi permette di specificare più look & feel per la stessa applicazione, di nuovo, senza intervenire nella struttura logica delle pagine. Basta prevedere più tipi di trasformazione da applicare al secondo passaggio.

Descrizione dei passaggi

Il primo passaggio crea la struttura logica di una schermata, senza però aggiungere elementi HTML. Raccoglie i dati necessari per la visualizzazione e li inserisce in un concetto logico, come potrebbe essere una pagina con tabelle di menu item, intestazione, fondo pagina, e così via. Si tratta evidentemente di una descrizione orientata alla presentazione su schermo, ma non ha ancora nessun elemento grafico specifico. Questa struttura è diversa per ogni pagina estratta dal modello, all'interno del workflow seguito dall'applicazione.

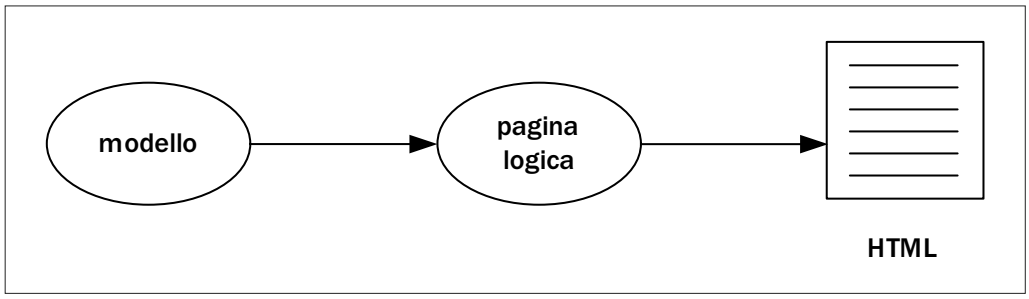

Figura 13.2 – Passaggi di Two Step View.

Il secondo passaggio trasforma la struttura logica in HTML. Conosce ogni elemento della descrizione logica e sa come trasformarlo nella presentazione finale. Come abbiamo detto, questo secondo passaggio può essere definito un'unica volta per tutta l'applicazione, garantendo un look & feel consistente.

La realizzazione può essere prevista con due trasformazioni XSLT in sequenza (come applicare il pattern Transform View due volte), oppure creando dal modello una nuova struttura di classi orientata alla presentazione, dalla quale in seguito far generare il codice specifico HTML per ogni elemento coinvolto.

Vantaggi

Il vantaggio dato dalla separazione in due passaggi è la facilità con la quale è possibile apportare modifiche globali all'applicazione. Nel caso di utilizzo di Transform View, ogni pagina ha la sua trasformazione specifica che mappa il contenuto specifico (descritto ad esempio in XML) in HTML.

Con Two Step View la trasformazione finale in HTML non è specifica per ogni pagina (o per ogni serie di tag XML che descrivono il contenuto particolare di ogni pagina), ma è globale, basata su una descrizione strutturale utilizzata allo stesso modo in ogni pagina. Se torniamo all'esempio di descrizione con XML, possiamo dire che prima dell'ultimo passaggio i tag di XML utilizzati sono gli stessi per ogni pagina, perché hanno già superato la trasformazione da XML che descrive il contenuto (modello), a XML che descrive la struttura logica della pagina (primo passaggio).

Nel caso in cui volessimo fornire la stessa applicazione con diversi look & feel, la cosa diventa ancora più chiara. Consideriamo l'esempio di un'applicazione che presenta dieci diverse pagine come interfaccia Web, di cui si vogliono fornire due diverse versioni, con diversi look & feel.

Se utilizzassimo il pattern Transform View, dovremmo fornire venti passaggi di trasformazione, cioè due diverse trasformazioni HTML per ogni pagina. Scegliendo invece Two Step View possiamo limitarci ad aggiungere una trasformazione in più come secondo passaggio, passando da 10 (primo passaggio) + 1 (secondo passaggio) a 10 + 2.

Svantaggi

I vantaggi dell'utilizzo del pattern Two Step View sono importanti, ma si deve partire dal presupposto che la struttura dopo il primo passaggio sia veramente utile per la trasformazione finale in HTML.

In un'applicazione in cui la struttura logica è direttamente legata alla presentazione finale (ad esempio ogni pagina ha un suo look & feel specifico), il pattern in questione non sarebbe adeguato. Se non esiste un comune denominatore tra le diverse schermate, va perso anche il senso del doppio passaggio, che diventerebbe al contrario un elemento eccessivo nel processo di trasformazione.

Un ulteriore punto a sfavore di questo approccio può essere identificato negli utensili a disposizione per descrivere le trasformazioni. Esistono parecchi tool a disposizione di non programmatori per generare HTML (approccio view-based, come ColdFusion, JSP, PHP, e altri), ma è più difficile pretendere che non programmatori creino elementi renderer che permettano di trasformare in view (HTML) la propria struttura logica. Inoltre questa idea di doppio passaggio di trasformazione è più complessa da imparare e mettere in pratica.

Una variante interessante è comunque quella di prevedere un secondo passaggio diverso per ogni nuovo device. Si avrebbe così un secondo passaggio per un browser, un secondo per device tipo smart phone o palmari e così via. Anche qui c'è comunque un limite: si presuppone che tutte le view debbano seguire la stessa struttura logica e questo può diventare un limite troppo importante.

In questo capitolo…

Non poteva mancare un capitolo sui pattern per il Web, visto il grande numero di applicazioni Web esistenti e l'elevato miscuglio di tecnologie usate, che favoriscono un'anarchia di design.

In questo capitolo abbiamo mostrato due modi diversi per trasformare in presentazione l'informazione presente nel modello di un'applicazione Web, per gestire cioè la visualizzazione di informazione in formato HTML.