Capitolo 9
Definizioni di design pattern

Abbiamo visto cosa si intende con riutilizzo di design. Ma come capitalizzare l'esperienza maturata in passato? Soprattutto, come condividerla? Come memorizzarla? Come descriverla? Ecco quindi la necessità di redigere un catalogo di schemi e modelli adottati in passato, ridotti all'essenziale, pronti a essere applicati e adattati ai bisogni specifici di un'applicazione.

Questi sono i design pattern!

Definizioni

Vediamo alcune altre definizioni:

- Ogni pattern (modello) dà un nome, spiega e valuta un tipo di design ricorrente nella programmazione OO.
- I pattern facilitano e promuovono il riutilizzo di design e di architetture software.
- Documentando nuove architetture, i pattern le rendono più accessibili.
- I pattern facilitano eventuali scelte tra più alternative di design.

- I pattern facilitano la manutenzione del software.
- I pattern avvicinano la progettazione alla programmazione. Con i pattern il design diventa "programmazione astratta".

Descrizione nel catalogo

È possibile realizzare un catalogo di pattern, descrivendo in esso alcuni elementi fondamentali per ciascun pattern. Vediamo una breve descrizione di tali elementi.

Scopo
Scopo principale che porta all'utilizzo di un dato modello in un'architettura.

Altri nomi
Se esistono, altri nomi attraverso i quali vengono descritte e sono riconoscibili architetture simili all'interno di sistemi software noti.

Motivazione
Perché e quando è utile e consigliabile far uso del pattern.

Applicabilità
Come va applicato un certo pattern. Quali sono gli elementi tipici del pattern e quali sono invece gli elementi che vanno adattati alla specificità del problema.

Struttura
Si tratta fondamentalmente della descrizione dell'architettura fornita attraverso un diagramma di classi.

Descrizione dei partecipanti
Si intende descrizione di ogni singola classe coinvolta nella struttura di classi.

Collaborazioni
Descrizione delle collaborazioni fra le diverse classi coinvolte nell'architettura definita con il pattern.

Conseguenze
Cosa bisogna considerare nel momento in cui si decide di usare questo pattern.

Tecniche di implementazione

Esempi di codice

Utilizzi noti

In quali sistemi conosciuti e a quale scopo sono stati utilizzati modelli basati sul pattern in questione.

Pattern simili

Quali altri pattern condividono scopi e motivazione e potrebbero quindi entrare in considerazione per risolvere il problema di implementazione.

Un esempio dal catalogo: il pattern Proxy

La cosa migliore a questo punto è quella di mostrare un pattern, così come è possibile trovarlo nel catalogo. Il modello scelto per questo primo esempio è il pattern Proxy.

Scopo

Lo scopo del pattern Proxy è quello di mettere a disposizione come riferimento il sostituto di un altro oggetto, in modo da poter controllare l'accesso all'oggetto originale.

Altro nome

Surrogate

Motivazione

Una ragione per controllare l'accesso a un oggetto è quella di ritardarne il costo della creazione e dell'inizializzazione fino a quando abbiamo veramente bisogno di utilizzarlo.

Consideriamo un editor in grado di gestire documenti con testi e immagini. Alcuni oggetti grafici possono essere costosi e lunghi da creare per essere visualizzati. Questo contrasta con la necessità di poter aprire un documento in modo veloce. Bisogna quindi evitare che tutti gli oggetti grafici vengano creati durante l'apertura del documento. Del resto non serve crearli tutti, visto che non saranno visualizzati sullo schermo contemporaneamente.

Questa restrizione indica che è meglio creare ognuno di questi oggetti unicamente quando questi devono effettivamente essere mostrati all'utente. Ma cosa usiamo nel documento al posto dell'immagine? E, soprattutto, come nascondiamo il fatto che l'oggetto reale viene creato *on demand* senza con questo complicare l'implementazione dell'editor?

La soluzione è quella di utilizzare un oggetto "surrogato", *proxy* appunto, che per il resto dell'applicazione rappresenti l'immagine, senza esserlo. Questo oggetto deve anche preoccuparsi di gestire la creazione dell'immagine al momento giusto, cioè al momento in cui questa diventa realmente necessaria. Inoltre deve mantenere un riferimento all'immagine stessa per passare eventuali altre richieste che arrivano dall'applicazione. In figura 9.1, è mostrato uno schema che illustra tale relazione.

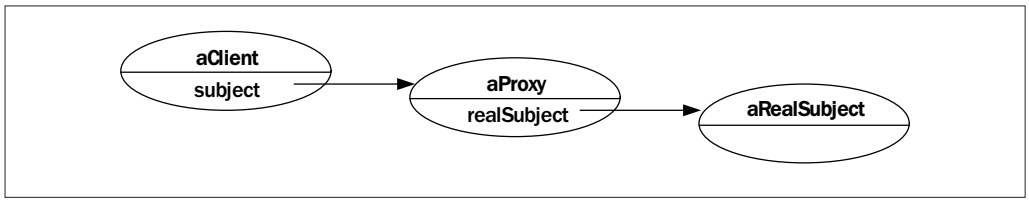

Figura 9.1 – Relazioni tra oggetti in runtime.

Applicabilità

Proxy è applicabile quando si rende necessario un riferimento a un oggetto più versatile e sofisticato che un semplice puntatore all'oggetto stesso.

Esempi

- Remote proxy: rappresentazione locale di un oggetto remoto.
- Virtual proxy: crea l'oggetto solo al momento in cui quest'ultimo è veramente richiesto (come visto nella motivazione).
- Protection proxy: controlla l'accesso all'oggetto.
- Smart reference: esegue operazioni supplementari a ogni accesso.

Struttura

In figura 9.2, è possibile osservare la struttura di classi come è presentata nel catalogo.

Partecipanti

Le classi (o interfacce) coinvolte in questo pattern sono tre: Subject, Proxy e RealSubject. L'elemento Subject determina l'interfaccia di RealSubject. In altre parole stabilisce a quali messaggi un oggetto di tipo RealSubject deve rispondere. L'elemento Proxy si posiziona tra il resto dell'applicazione e RealSubject, allo scopo di "filtrare" le richieste a RealSubject. Per questioni di trasparenza, l'oggetto Proxy appartiene allo stesso tipo di RealSubject, cioè a Subject.

Collaborazioni

A dipendenza dal tipo di proxy, lo stesso manda avanti la richiesta a RealSubject

Conseguenze

Il pattern Proxy introduce un livello di indirezione verso un oggetto. L'utilizzo di questa indirezione dipende dal tipo di proxy.

Esempi di codice

Per mostrare un esempio di codice dobbiamo prima descrivere il problema. Consideriamo un caso simile a quello menzionato nel paragrafo riguardante la motivazione.

Si tratta di un programma usato per visualizzare dei testi. Il programma deve gestire informazioni ottenute dai diversi file dove i testi considerati vengono archiviati. Queste informazioni sono la dimensione in byte del contenuto, il nome del file e il contenuto stesso dei file, con possibilità di accesso riga per riga.

Per l'implementazione della funzionalità vera e propria realizziamo una classe che si occupi di caricare il file e che svolga le operazioni descritte dalla seguente interfaccia.

```
public interface FileInfo
{
  public String getFileName();
  public int getSize();
  public String getContent();
  public String getLine(int lineNr);
}
```

Come si vede, le operazioni sono: fornire il nome del file, leggere la sua dimensione, fornire il suo contenuto e leggere una qualsiasi riga del file, specificata attraverso il numero di linea.

Vediamo ora l'implementazione della classe che realizza i metodi dichiarati in FileInfo.

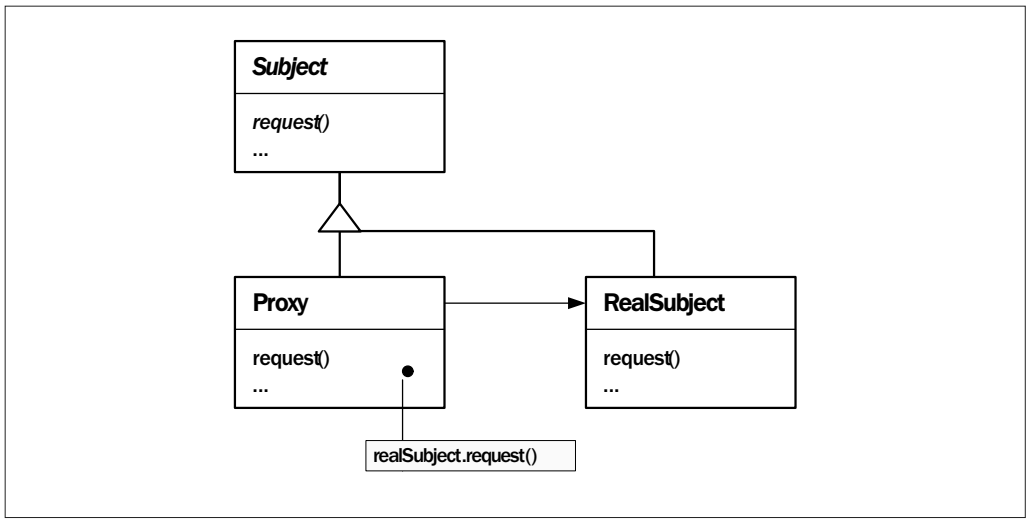

Figura 9.2 – Struttura di classi del pattern Proxy

```
public class RealFileInfo implements FileInfo
{
  private String fFileName;
  private String fContent;
  private int fByteContentSize;

  public RealFileInfo(String fileName){
    fFileName = fileName;
    try{
      FileInputStream file = new FileInputStream(fFileName);
      fByteContentSize = file.available();
      byte[] byteContent = new byte[fByteContentSize];
      file.read(byteContent);
      file.close();
      fContent = new String(byteContent);
    } catch(Exception e){
      e.printStackTrace();
    }
  }

  public String getFileName() {
    return fFileName;
  }

  public String getContent() {
    return fContent;
  }

  public int getSize() {
    return fByteContentSize;
  }

  public String getLine(int lineNr){
    BufferedReader reader = new BufferedReader(new StringReader(fContent));
    String line = null;
    int currentLineNr = 1;
    try {
      if (lineNr > 0){
        while (((line = reader.readLine())!= null) && (currentLineNr != lineNr)){
          currentLineNr++;
        }
      }
    } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        return line;
    }
}
```

Ecco un esempio di utilizzo della classe:

```
public class FileInfoTry
{
    public static void main(String[] args){
        FileInfo fileInfo = new RealFileInfo(args[0]);
        System.err.println("size: " + fileInfo.getSize());
        System.err.println("name: " + fileInfo.getFileName());
        System.err.println("line: 4 " + fileInfo.getLine(4));
        System.err.println("line: 4 " + fileInfo.getLine(4));
        System.err.println("line: 6 " + fileInfo.getLine(6));
    }
}
```

La classe RealFileInfo è sicuramente utile dato che fornisce tutte le funzionalità richieste dall'applicazione. Nonostante ciò, il suo uso è inefficiente perché se si desidera accedere unicamente al nome del file non è necessario aver caricato tutto il suo contenuto in memoria. Un altro fattore di inefficienza si presenta nel caso in cui si facciano più richieste dello stesso numero di riga: ogni volta si ripete la ricerca effettuata in precedenza.

Il pattern Proxy ci aiuta a migliorare questa situazione, senza modificare l'implementazione della classe RealFileInfo, che è comunque necessaria alla funzionalità.

Il pattern in questione ci suggerisce l'implementazione di una nuova classe (ProxyFileInfo) che offra la stessa interfaccia della classe originale contenente la funzionalità vera e propria (RealFileInfo), che sia però in grado di risolvere le richieste più semplici pervenute dall'oggetto client, senza dover utilizzare inutilmente le risorse, ad esempio senza dover caricare inutilmente l'intero file per restituirne il nome. Solo al momento in cui ricevesse una richiesta più complessa il proxy andrebbe a creare il vero RealFileInfo per inoltrare a esso le richieste. In questo modo gli oggetti più pesanti vengono creati solo quando sono realmente necessari. Il tipo di proxy con questa finalità viene chiamato *virtual proxy*, come già visto in precedenza.

Vediamone una prima implementazione:

```
public class ProxyFileInfo implements FileInfo
{
    private FileInfo fReal = null;
    private String fFileName;

    public ProxyFileInfo(String fileName) {
```

```
        fFileName = fileName;
    }

    private FileInfo getReal(){
        if (fReal == null){
            fReal = new RealFileInfo(fFileName);
        }
        return fReal;
    }

    public String getFileName(){
        return fFileName;
    }

    public String getContent(){
        return getReal().getContent();
    }

    public int getSize() {
        return getReal().getSize();
    }

    public String getLine(int lineNr){
        return getReal().getLine(lineNr);
    }
}
```

In questo primo esempio ci sono due aspetti da sottolineare. Il primo consiste nel fatto che ProxyFileInfo gestisce anche lui il nome del file, per poterlo restituire senza chiamare il costruttore di RealFileInfo. Potremmo realizzare la funzionalità di getFileName() sia di ProxyFileInfo che di RealFileInfo con una classe astratta in comune, magari sostituendo l'*interface* FileInfo con una classe astratta. La cosa importante è comunque la possibilità di restituire il nome del file senza doverne caricare il contenuto, come avviene invece utilizzando direttamente RealFileInfo. Il secondo aspetto è invece quello di centralizzare nel metodo privato getReal() l'accesso a RealFileInfo (e la sua eventuale creazione, che viene quindi chiamata una sola volta).

L'utilizzo di un proxy del genere può però suggerire altri miglioramenti. Per rendere più efficiente l'accesso diretto alle singole linee, può essere utile gestire in ProxyFileInfo una cache sottoforma di tabella hash, che immagazzini le linee già viste. Una tabella del genere permetterebbe così di controllare prima nella hash se una linea è già stata letta ed eventualmente restituire quella letta dalla tabella. Vediamo le modifiche alla classe precedente allo scopo di integrare la cache e realizzare una sorta di *cache proxy*.

```
public class ProxyFileInfo implements FileInfo
{
  FileInfo fReal = null;
  String fFileName;
  Map fCache = new HashMap();

  public ProxyFileInfo(String fileName) {
    fFileName = fileName;
  }

  private FileInfo getReal(){
    if (fReal == null){
      fReal = new RealFileInfo(fFileName);
    }
    return fReal;
  }

  public String getFileName(){
    return fFileName;
  }

  public String getContent(){
    return getReal().getContent();
  }

  public int getSize() {
    return getReal().getSize();
  }

  public String getLine(int lineNr){
    String line = (String)fCache.get(new Integer(lineNr));
    if (line == null){
      line = getReal().getLine(lineNr);
      fCache.put(new Integer(lineNr),line);
    }
    return line;
  }
}
```

Alla richiesta getLine(), il metodo cerca prima nella cache il valore corrispondente al numero di linea. Se lo trova, restituisce quello; altrimenti accede al file attraverso RealFileInfo e poi, prima di restituirla come valore di return, inserisce l'informazione trovata nella cache, per una eventuale richiesta successiva.

Il programma di chiamata delle funzionalità implementate è identico a quello che prevedeva l'utilizzo diretto di RealFileInfo, con la differenza che in questo caso l'oggetto istanziato è del tipo ProxyFileInfo.

```
public class FileInfoTry
{
  public static void main(String[] args){
    FileInfo fileInfo = new ProxyFileInfo(args[0]);
    System.err.println("size: " + fileInfo.getSize());
    System.err.println("name: " + fileInfo.getFileName());
    System.err.println("line: 4 " + fileInfo.getLine(4));
    System.err.println("line: 4 " + fileInfo.getLine(4));
    System.err.println("line: 6 " + fileInfo.getLine(6));
  }
}
```

Utilizzi noti di Proxy

ET++ text building block classes.

NEXTSTEP utilizza Proxy come rappresentazione locale di oggetti che possono essere distribuiti.

(…)

Pattern simili

Adapter, Decorator

In questo capitolo…

Dopo una serie di definizioni di pattern, siamo passati al catalogo e al tipo di descrizione per ogni modello contenuto nel catalogo stesso.

Come esempio di descrizione abbiamo esaminato il pattern Proxy, di cui abbiamo spiegato e approfondito gli aspetti principali, compreso un esempio di implementazione.