12. Proxy

(GoF pag. 207)

12.1. Descrizione

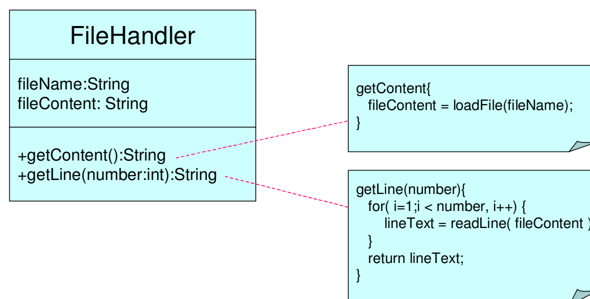
Fornisce una rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l'accesso o creazione. Il Proxy consente di posticipare l'accesso o creazione al momento in cui sia davvero richiesto

12.2. Esempio

Un programma di visualizzazione di testi deve gestire informazioni riguardanti file dove i testi vengono archiviati, come il contenuto stesso dei file. Il programma potrebbe essere in grado di visualizzare il nome di un file, il testo completo, o trovare e visualizzare una singola riga.

Si pensi che per l'implementazione si ha progettato un oggetto che al momento della sua istanziazione fa il caricamento del file, e che svolge le operazioni descritte nel seguente modo:

- Restituire nome del file: restituisce una stringa contenente il nome del file dentro il file system.
- Restituire il testo completo: restituisce una stringa contenente il testo.
- Restituire una singola riga di testo: riceve come parametro un numero valido di riga (si considera CR + LF come separatore di riga), e tramite un algoritmo di ricerca, restituisce una stringa contenente il testo corrispondente.
-



Questa classe diventa utile dato che fornisce tutte le funzionalità richieste dall'applicativo. Nonostante ciò, il suo uso è inefficiente perché se solo si vuole accedere al nome del file non è necessario aver caricato tutto il suo contenuto in memoria. Un altro caso di inefficienza si presenta nel caso in cui si fanno due richieste successive dello stesso numero di riga, che determinano la ripetizione della ricerca appena effettuata.

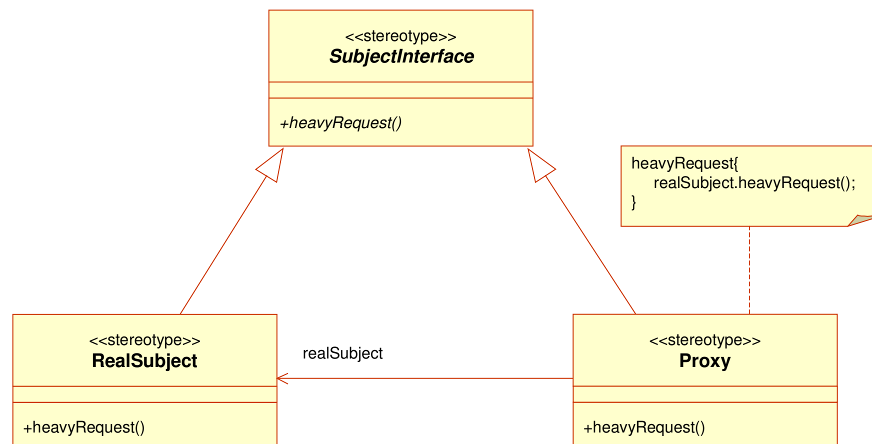
E' di interesse poter gestire più efficientemente questa classe, senza modificare la sua implementazione.

12.3. Descrizione della soluzione offerta dal pattern

Il "Proxy" pattern suggerisce l'implemetazione di una classe (ProxyFileHandler) che offra la stesa interfaccia della classe originale (FileHandler), e che sia in grado di risolvere le richieste più "semplici" pervenute dall'applicativo, senza dover utilizzare inutilmente le risorse (ad esempio, restituire il nome del file). Solo al momento di ricevere una richiesta più "complessa" (ad esempio, restituire il testo del file), il proxy andrebbe a creare il vero FileHandler per inoltrare a esso le richieste. In questo modo gli oggetti più pesanti sono creati solo al momento di essere necessari. Il proxy che serve a questa finalità spesso viene chiamato "*virtual proxy*".

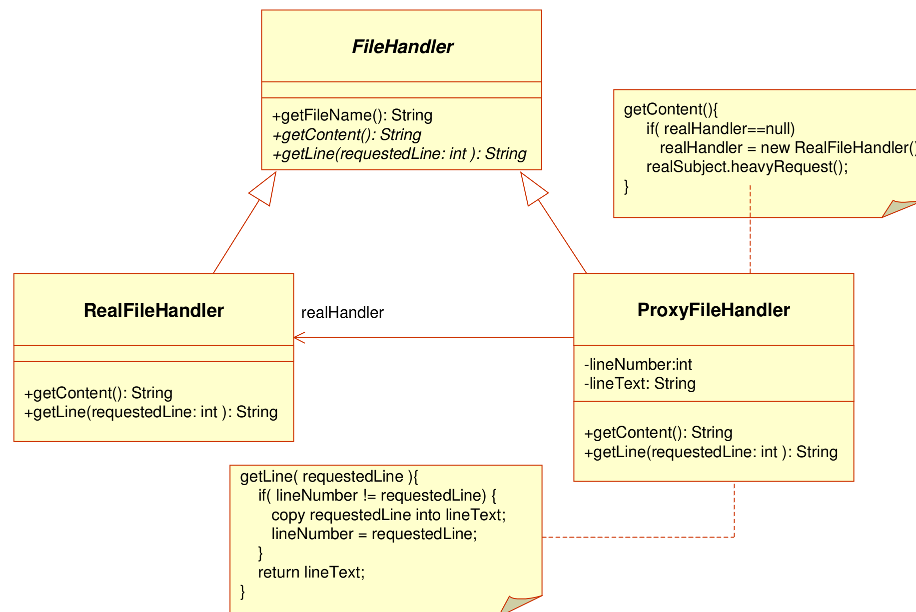
Come altra funzionalità, a questo proxy potrebbe essere aggiunta la possibilità di imagazinare temporaneamente l'ultima riga restituita dal metodo "getLine", in modo che due richieste successive della stesa linea comportino solo una ricerca, riducendo lo spreco di tempo di elaborazione. Il proxy che offre questa funzionalità viene chiamato "*caché proxy*".

12.4. Struttura del Pattern



12.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Proxy**: classe ProxyFileHandler.
  - Mantiene un riferimento per accedere al **RealSubject**.
  - Implementa una interfaccia identica a quella del **RealSubject**, in modo che può sostituire a esso.
  - Controlla l'acceso al RealSubject, essendo responsabile della sua istanziazione e gestione di riferimenti.
  - Come *virtual proxy* pospone la istanziazione del **RealSubject**, tramite la gestione di alcune informazioni di questo.
  - Come *caché proxy* immagazzina temporaneamente il risultato di alcune elaborazioni del **RealSubject**, in modo di avere delle risposte pronte per i clienti.
- **Subject**: classe FileHandler.
  - Fornisce l'interfaccia comune per il **RealSubject** e il **Proxy**, in modo che questo ultimo possa essere utilizzato in ogni luogo dove si aspetta un **RealSubject**.
- **RealSubject**: classe RealFileHandler.
  - Implementa l'oggetto vero e proprio che il **RealSubject** rappresenta.

Descrizione del codice

Si crea la classe `FileHandler` che rappresenta l'interfaccia che la classe di gestione dei file (`RealFileHandler`) e il suo proxy

(`ProxyFileHandler`) devono implementare. Questa classe offre la funzionalità di gestione del nome del file da aprire.

```
public abstract class  FileHandler {

    protected String fileName;

    public FileHandler(String fName) {
        fileName = fName;
    }

    public String getFileName() {
        return fileName;
    }

    public abstract String getContent();

    public abstract String getLine( int requestedLine );

}
```

La classe `RealFileHandler` estende `FileHandler`. Nel costruttore viene fornito il codice di caricamento del file di testo in memoria. Questa classe offre i metodi di restituzione del testo del file come una singola stringa, e di ricerca e restituzione di una singola riga di testo, a partire del suo numero.

```
import java.io.*;

class RealFileHandler extends FileHandler {

   private byte[] content;

   public RealFileHandler( String fName ){
      super(fName);
        System.out.println( "(creating a real handler with file
                                                            content)" );
      FileInputStream file;
      try{
         file = new FileInputStream(fName);
           int numBytes = file.available();
           content = new byte[ numBytes ];
           file.read( content );
      } catch(Exception e){
          System.out.println( e );
      }
   }

   public String getContent(  ){
      return new String( content );
   }


   public String getLine( int requestedLine ){
        System.out.println( "(accessing from real handler)" );
        int numBytes = content.length;
        int currentLine = 1;
        int startingPos = -1;
        int lineLength   = 0;
        for(int i=0;i<numBytes; i++) {
           if( ( currentLine == requestedLine ) &&
               ( content[i] != 0x0A ) ) {
             if( startingPos == -1)
               startingPos = i;
             lineLength++;
           }
           if( content[i] == 0x0D )
             currentLine++;
        }
        String lineText = "";
        if(startingPos != -1)
          lineText = new String( content, startingPos, lineLength-1 );
        return  "\"" + lineText + "\"";
```

```
    }
}
```

Si noti che nell'implementazione della classe `RealFileHandler` il testo del file viene caricato in memoria appena viene istanziata, e che l'algoritmo di ricerca di una riga viene eseguito ad ogni chiamata del metodo di restituzione di righe.

La classe `ProxyFIleHandler` implementa il proxy per il `RealFileHandler`. Questa classe, eredita la sua interfaccia e la gestione del nome del file associato, dalla superclasse `FileHandler`. Si noti che un oggetto della classe `ProxyFileHandler` crea un una istanza di `RealFileHandler` solo al momento in cui diventa indispensabile caricare in memoria il contenuto del file, cioè la prima volta che viene richiesto il suo contenuto completo, o quello di una singola riga.

```
public class ProxyFileHandler extends FileHandler {

    private RealFileHandler realHandler;
    private int lineNumber;
    private String lineText;

    public ProxyFileHandler ( String fName ){
      super( fName );
        System.out.println( "(creating a proxy cache)" );
    }

    public String getContent(){
        if( realHandler == null )
        realHandler = new RealFileHandler( fileName );
      return realHandler.getContent();
    }

    public String getLine(int requestedLine){

      if( requestedLine == lineNumber ) {
          System.out.println( "(accessing from proxy cache)" );
          return lineText;
        } else {
        if( realHandler == null )
           realHandler = new RealFileHandler( fileName );
          lineText = realHandler.getLine( requestedLine );
          lineNumber = requestedLine;
        }
      return lineText;
    }

}
```

Si noti che in entrambi classi sono stati inseriti dei messaggi di testo che servono a monitorare le chiamate di ogni metodo, del `RealFileHandler` e del `ProxyFileHandler`.

Il codice dell'applicazione, presentato di seguito, istanzia un `ProxyFileHandler` con l'indicazione di aprire il file "`Files/Secret.txt`" e fa delle invocazioni ai diversi metodi. Si noti che particolarmente prima fa una invocazione al metodo di restituzione del nome del file, e dopo fa una doppia invocazione al metodo di restituzione del contenuto, poi una doppia invocazione al metodo `getLine` per ottenere il testo della seconda riga, e finalmente una invocazione allo stesso metodo, ma adesso per la restituzione della quarta riga.

```
public class ProxyExample{
    public static void main(String[] args){
    FileHandler fh=new ProxyFileHandler( "Files/Secret.txt" );
    System.out.println( "** The name of the file is: " );
    System.out.println( fh.getFileName());
    System.out.println( "** The content of the file is: " );
    System.out.println( fh.getContent() );
    System.out.println( "** The content of the file is (again):" );
    System.out.println( fh.getContent() );
    System.out.println( "** The content of line 2 is: " );
    System.out.println( fh.getLine( 2 ) );
    System.out.println( "** The content of line 2 is (again): " );
    System.out.println( fh.getLine( 2 ) );
    System.out.println( "** The content of line 4 is: " );
    System.out.println( fh.getLine( 4 ) );

    }
}
```

Osservazioni sull'esempio

Questo esempio dimostra l'utilizzo del proxy pattern in due ambiti diverse: come virtual proxy e come caché proxy. In entrambi casi l'obbiettivo del proxy è ridurre lo spreco di risorse, in particolare, di memoria e di potenza di calcolo. E' questo obbiettivo quello che crea realmente una distinzione tra questo pattern e il Decorator.

Esecuzione dell'esempio

Si offre di seguito i risultati dell'esecuzione del programma di prova. Le stampe dei messaggi di controllo inseriti nel codice rivelano che l'oggetto `RealFileHandler` è creato soltanto quando viene invocato per prima volta il metodo `getContent`, intanto che la precedente invocazione al metodo `getName` viene risolta completamente dal `ProxyFileHandler`. La successiva invocazione al metodo `getLine` con parametro `2`, è inoltrata dall'oggetto `ProxyFileHandler` all'oggetto `RealFileHandler`, la cui risposta viene immagazzinata nella caché del proxy, prima di essere trasmessa al chiamante. In questo modo, la seguente invocazione al metodo `getLine` con lo stesso valore come parametro, viene risposta direttamente dal proxy. Invece, l'ultima invocazione a `getLine`, con un valore diverso come parametro, comporta l'invocazione al rispettivo metodo del `RealFileHandler`.

```
C:\Design Patterns\Structural\Proxy>java ProxyExample

(creating a proxy cache)
** The name of the file is:
Files/Secret.txt

** The content of the file is:
(creating a real handler with file content)
One reason for controlling access to an object
is to defer the full cost of its creation and
initialization until we actually need to use it.
Proxy is appicable whenever there is a need for
a versatile or sophisticated reference to an
object than a simple pointer.

** The content of the file is (again):
One reason for controlling access to an object
is to defer the full cost of its creation and
initialization until we actually need to use it.
Proxy is appicable whenever there is a need for
a versatile or sophisticated reference to an
object than a simple pointer.

** The content of line 2 is:
(accessing from real handler)
"is to defer the full cost of its creation and "

** The content of line 2 is (again):
(accessing from proxy cache)
"is to defer the full cost of its creation and "

** The content of line 4 is:
(accessing from real handler)
"Proxy is appicable whenever there is a need for "
```

12.6. Osservazioni sull’implementazione in Java

La Remote Method Invocation (RMI) che consente l’interazione di oggetti Java distribuiti, utilizza il proxy pattern per l’interfacciamento remoto di oggetti. In questo caso i proxy sono chiamati “stub”.