Sviluppo Applicazioni Software 13/14 - Design Pattern Spike

A Team: Bono, Cerrato, Vinci

4 giugno 2014

Testo della consegna

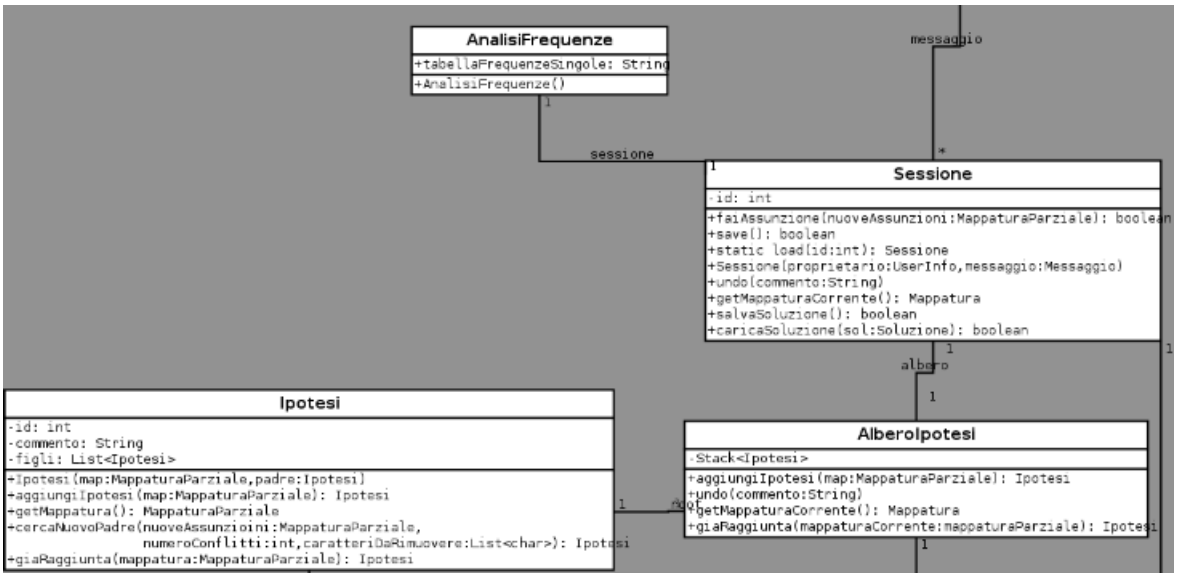
Si tratta di un esercizio di implementazione di un Design Pattern GOF a vostra scelta fra: Decorator, Composite, Observer, Visitor, State, Proxy

Dovete immaginare una situazione pratica di utilizzo dei pattern (volendo presa dal nostro progetto o da qualche sua evoluzione che immaginate possa avere nel futuro, ma anche completamente diversa se non vi viene in mente nulla). Quindi dovete disegnare lo schema delle classi nel pattern istanziato al caso che avete pensato, e realizzare uno scheletro di implementazione che faccia vedere l'utilizzo del pattern stesso. Per scheletro si intende la struttura delle classi con campi e metodi rilevanti, e un abbozzo di linee di codice all'interno dei metodi che sintetizzano l'uso degli oggetti coinvolti nel pattern. Naturalmente non è necessario che il risultato sia compilabile e/o eseguibile.
Se per caso però uno di questi pattern è entrato di diritto nel codice del vostro progetto, potete consegnarci direttamente le classi relative così come le avete implementate, corredate dalla porzione di modello del progetto che descrive la struttura del pattern.

Il pattern da noi selezionato è il pattern **Proxy**.

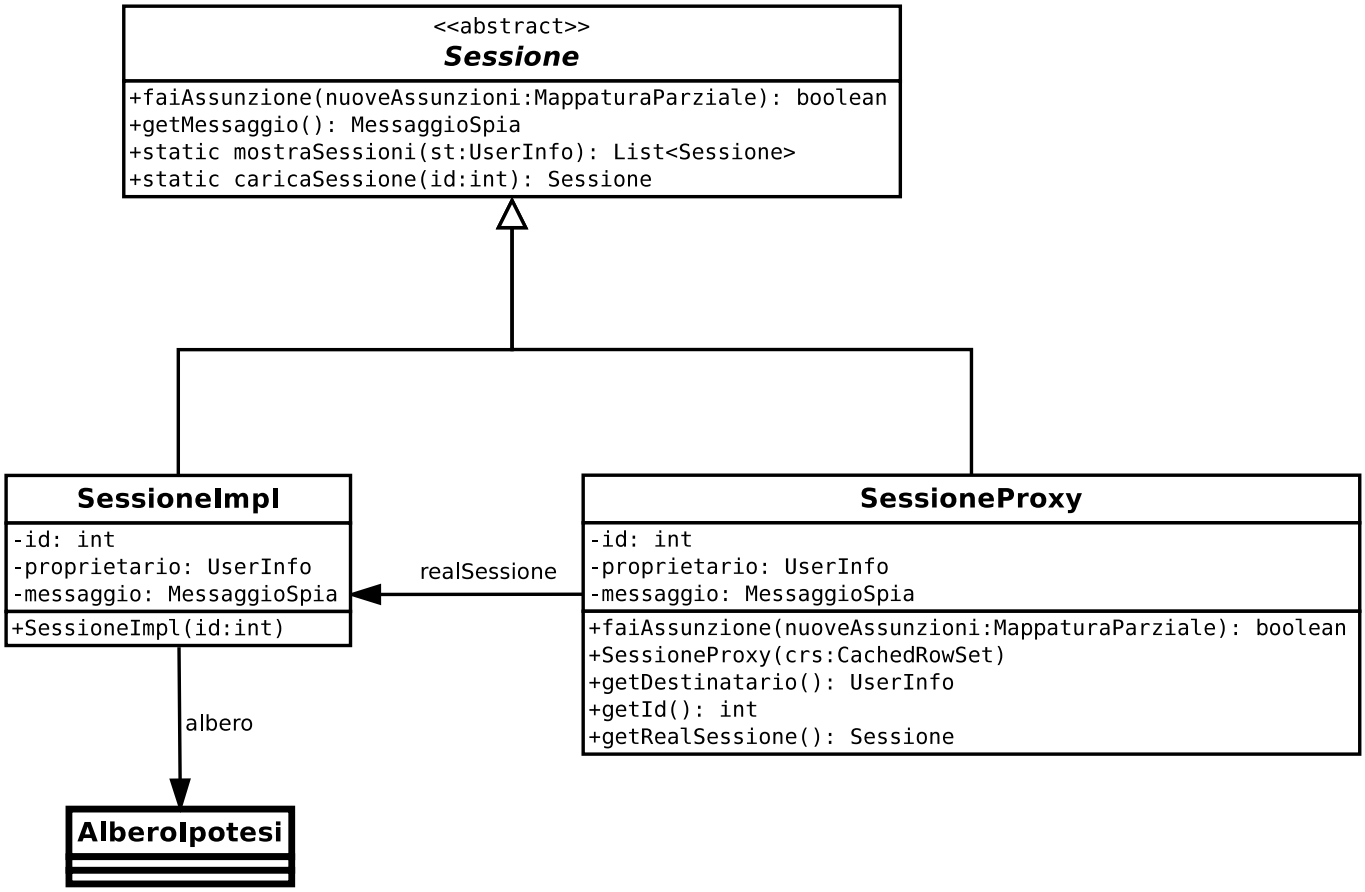
Descrizione dell'utilizzo del Pattern

Abbiamo definito all'interno del nostro Modello di Progetto una classe *Sessione*. Tale classe racchiude all'interno di sè le varie operazioni che la Spia vorrà svolgere durante la decifratura di un messaggio, e mantiene riferimenti agli oggetti utili a questo scopo.



Il nostro oggetto Sessione contiene, tra le altre cose, il riferimento all'oggetto *AlberoIpotesi*: tale oggetto è di tipo ricorsivo, in cui varie Ipotesi sono aggiunte via via che la Spia compone le sue assunzioni tentando di decifrare il Messaggio. Si tratta di un oggetto complicato e pesante, che non desideriamo caricare a meno che non sia strettamente necessario.
È d'altra parte necessario che l'utente possa visualizzare le Sessioni che sta effettuando, potendo scegliere quale continuare o eliminare. In questa operazione, che nei nostri contratti abbiamo chiamato *mostraSessioni*(), bisogna necessariamente caricare dal DB l'intero oggetto *Sessione*, completo di *AlberoIpotesi*. Desideriamo evitare questo scenario, scarsamente efficiente, utilizzando il pattern **Proxy**, in particolare secondo il concetto di **Virtual Proxy**.

Schema del Pattern nel contesto



Scheletro di Implementazione

Listing 1: Sessione.java

```
1  public abstract class Sessione {
2
3    public static List<Sessione> mostraSessioni(UserInfo st) {
4      DBController dbc = DBController.getInstance();
5      CachedRowSet crs = dbc.execute("SELECT id, proprietario, messaggio "
6                                     +"FROM Sessione "
7                                     +"WHERE proprietario = ?", st.getNome());
8      List<Sessione> result = null;
9      while(crs.next()) {
10       result.add(new SessioneProxy(crs));
11     }
12     return result;
13   }
14
15   public static Sessione caricaSessione(int id) {
16     DBController dbc = DBController.getInstance();
17     CachedRowSet crs = dbc.execute("SELECT id, proprietario, messaggio "
18                                    +"FROM Sessione "
19                                    +"HERE id = ?", id);
20     return new SessioneProxy(crs);
21   }
22
23   public MessaggioSpia getMessaggio();
24   public faiAssunzione(Mappatura nuovaAssunzione);
25 }
```

Listing 2: SessioneProxy.java

```java
public class SessioneProxy extends Sessione {
  private int id;
  private final UserInfo proprietario;
  private final MessaggioSpia messaggio;
  private Sessione realSessione;

  public SessioneProxy(CachedRowSet crs) {
    this.id = crs.getInt("id");
    this.messaggio = Messaggio.load(crs.getInt("messaggio"));
    this.proprietario = UserInfo.load(crs.getInt("proprietario"));
  }

  public UserInfo getDestinatario() {
    return destinatario;
  }

  public int getId() {
    return id;
  }

  public boolean faiAssunzione(Mappatura nuoveAssunzioni) {
    return getRealSessione().faiAssunzione(nuoveAssunzioni);
  }

  public Mappatura getMappaturaCorrente() {
    return getRealSessione().getMappaturaCorrente();
  }

  private Sessione getRealSessione() {
    if (realSessione == null) {
      realSessione = new SessioneImpl(id);
    }
    return realSessione;
  }
}
```

Listing 3: SessioneImpl.java

```java
public class SessioneImpl extends Sessione {
  private int id;
  private final UserInfo proprietario;
  private final Messaggio messaggio;
  public AlberoIpotesi albero;

  public SessioneImpl(Int id) {
    DBController dbc = DBController.getInstance();
    CachedRowSet crs = dbc.execute("SELECT * FROM Sessione WHERE id = ?", id);
    this.id = crs.getInt("id");
    this.messaggio = Messaggio.load(crs.getInt("messaggio"));
    this.proprietario = UserInfo.load(crs.getInt("proprietario"));
    Blob bl = crs.getBlob("albero");
    byte[] buf = bl.getBytes(1, (int) bl.length());
    ObjectInputStream objectIn;
    objectIn = new ObjectInputStream(new ByteArrayInputStream(buf));
    this.albero = ((AlberoIpotesi) objectIn.readObject());
  }

  public boolean faiAssunzione(Mappatura nuoveAssunzioni) {
    // ... aggiunge una ipotesi ...
  }
}
```