Implementazione To-Do List in Java con MVC, Singletone e Observer Pattern

Lorenzo Cappetti

Gennaio 2024

Contents

1 Introduzione

Durante il mio percorso di studi in Ingegneria Informatica, ho avuto l'opportunità di lavorare su diversi progetti, uno dei quali riguardava la realizzazione di una To-Do List in C++. Questa esperienza mi ha motivato a esplorare ulteriormente il concetto e ad implementare una versione simile utilizzando il linguaggio Java. In questo nuovo progetto, ho deciso di arricchire l'implementazione introducendo alcuni design pattern, MVC (Model-View-Controller), Observer e Singleton. Inoltre ho aggiunto la possibilità di salvare le task in un file di testo e di poterle caricare da esso.

1.1 Obiettivo del Progetto

Il progetto mira a fornire un'applicazione To-Do List in Java che sfrutti i principi del design pattern MVC, del pattern Observer e del Singleton. La To-Do List consentirà agli utenti di gestire le proprie attività quotidiane attraverso operazioni come l'aggiunta, la rimozione, la modifica e il segnare come completate delle diverse attività.

1.2 Motivazioni e Sfide

L'iniziale implementazione in C++ ha fornito una buona base, ma l'esplorazione di nuove tecnologie e linguaggi come Java ha rappresentato una sfida interessante. L'introduzione dei 3 design pattern ha l'obiettivo di migliorare la struttura dell'applicazione e rendere il codice più flessibile e manutenibile.

1.3 Strumenti

L'applicazione è scritta in Java, come già detto, nell'ambiente di sviluppo Visual Studio Code; i grafici e i diagrammi sono generati con STAR UML.

2 Diagrammi UML

Andremo a rappresentare all'inizio il diagramma generale e poi nello specifico i vari pattern.

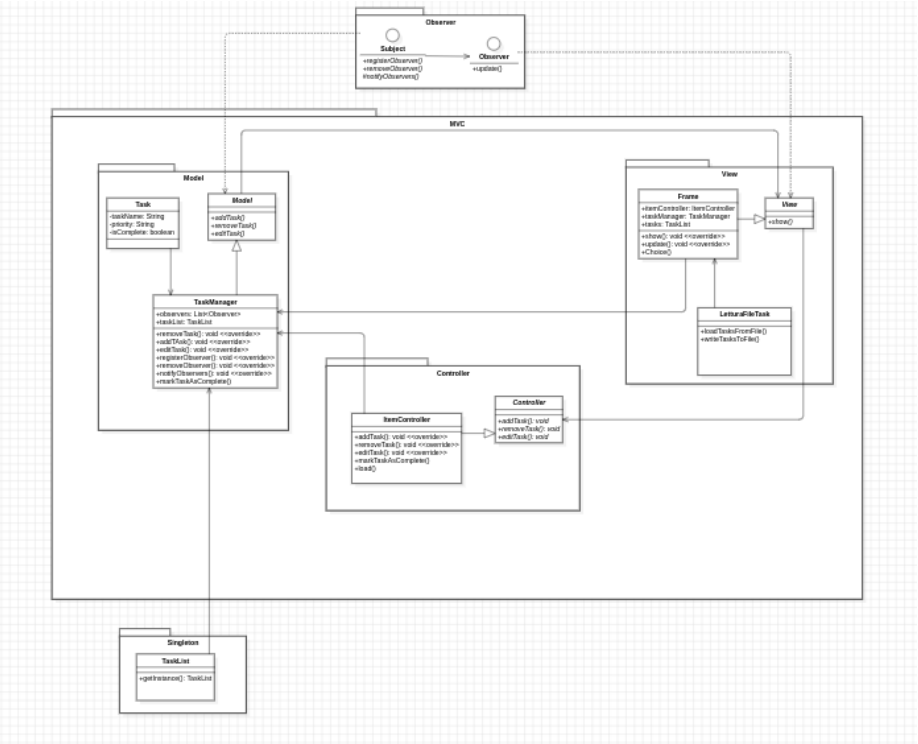

Figure 1: Diagramma UML completo.

2.1 Observer

Il pattern Observer è un design pattern comportamentale che facilita la comunicazione tra oggetti in un sistema, introducendo una dipendenza uno-a-molti tra un oggetto principale, chiamato soggetto, e una serie di oggetti osservatori. L'obiettivo è permettere agli osservatori di ricevere notifiche automatiche quando lo stato del soggetto cambia, consentendo loro di reagire di conseguenza.

Nel contesto dell'implementazione del pattern Observer in linguaggi di programmazione come Java, è comune definire due classi principali: Observer e Subject.

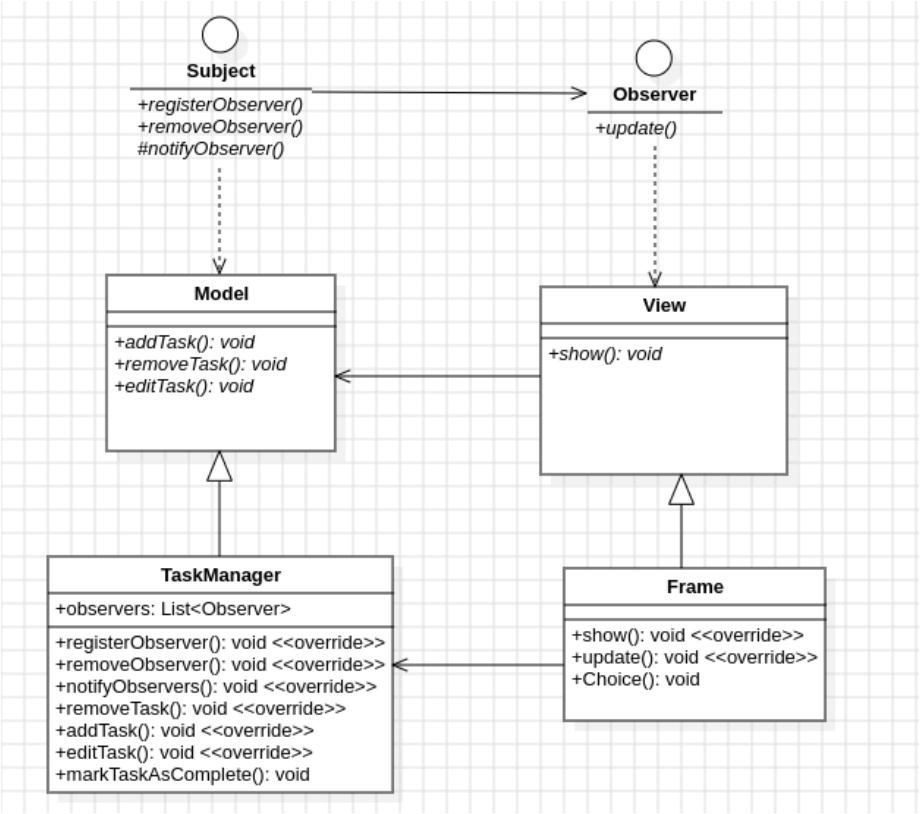

Figure 2: Diagramma UML del pattern Observer. Nell'immagine non è implementato tutto, ma solamente la parte che ci serve per capire il funzionamento di Observer e Subject.

La classe Subject rappresenta l'oggetto principale che è soggetto a cambiamenti. Essa mantiene una lista di osservatori, tramite una list, registrati e fornisce metodi per aggiungere, rimuovere e notificare gli osservatori, che sono tutti scritti nell'UML soprastante. Quando lo stato del Subject cambia, esso notifica tutti gli osservatori registrati chiamando i loro metodi di aggiornamento. In particolare, nel nostro caso, il concreteSubject è TaskList e il concreteObserver è Frame. Entrambi estendono le classi sovrastanti, TaskList estende Subject e Model, Frame estende Observer e View. Per estenderli, chiaramente, entrambe le classi implementano i metodi virtuali delle due classi.

La classe Observer è una classe astratta che fornisce un'interfaccia per gli oggetti che intendono osservare il Subject. Essa presenta almeno un metodo di aggiornamento che verrà implementato dalle classi concrete degli osservatori. Questo metodo consente agli osservatori di ricevere e reagire alle notifiche del cambiamento di stato del Subject.

Questo approccio rende il sistema estensibile e flessibile, consentendo l'aggiunta

di nuovi osservatori senza dover modificare il codice del Subject. Inoltre, il pattern Observer promuove una separazione chiara tra il soggetto e gli osservatori, migliorando la manutenibilità e la chiarezza del codice.

La classe Observer si presenta così:

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

La classe Subject si presenta così:

```
public interface Subject {

    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Come spiegato sopra le classi poi sono estese e i metodi implementati da altre classi.

2.2 MVC (Model-View-Controller)

La struttura MVC suddivide l'applicazione in tre componenti principali, ciascuno con un ruolo specifico: il Model, che rappresenta i dati dell'applicazione; la View, responsabile della presentazione dei dati all'utente; e il Controller, che gestisce le interazioni utente e coordina le operazioni tra Model e View. L'obiettivo principale di questo pattern è separare in modo netto le responsabilità, facilitando così la manutenibilità e l'estensione del codice.

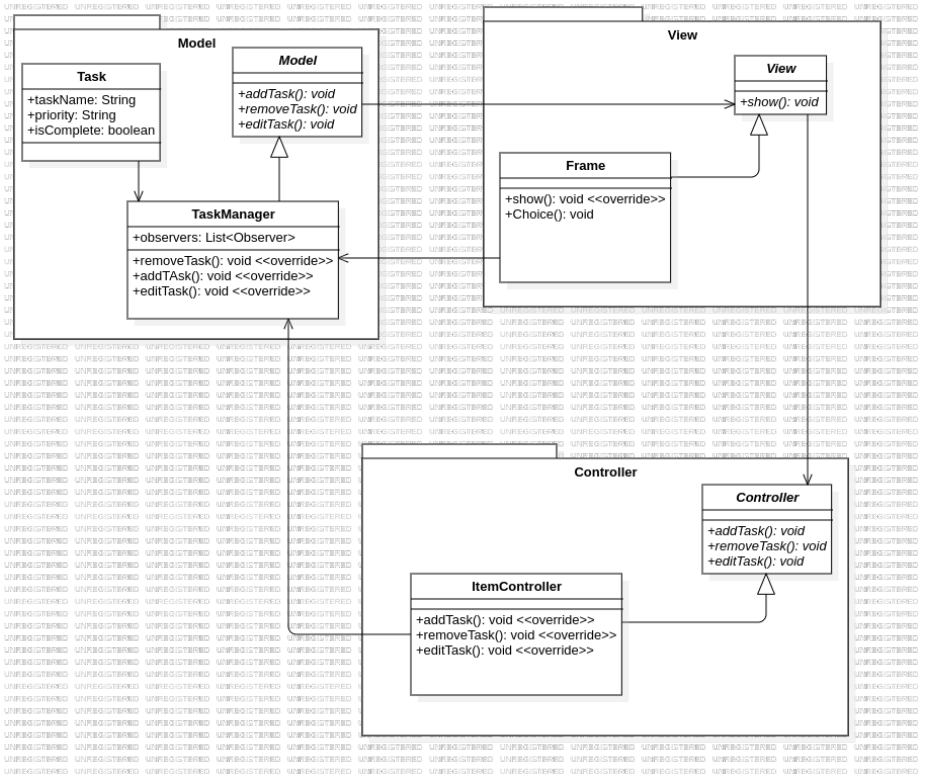

Figure 3: Diagramma UML del pattern MVC. Nell'immagine non è implementato tutto, ma solamente la parte che ci serve per capire il funzionamento di Model-View-Controller

Nel mio caso, in particolare, abbiamo le 3 classi che sono virtuali e quindi hanno delle classi concrete che ne implementano i metodi virtuali. Per Model abbiamo TaskList, che estende i suoi metodi e ne aggiunge alcuni per lavorare sulle task, inoltre contiene una lista per registrare gli observer da aggiornare poi. Per View abbiamo Frame, che implementa show() che viene da View e altri metodi, tra cui anche quelli dell'Observer. Per Controller abbiamo ItemController, che implementa tutti i metodi per aggiungere, rimuovere e modificare e completare le task; il suo compito è reagire agli input che Frame raccoglie e trasformarli in azioni su TaskList.

Le classi astratte si presentano così:

Model:

```
1 public interface Model {
2
3     public abstract void addTask(String title, String
          priority, boolean completed);
4     public abstract void removeTask(int index);
```

```
5       public abstract void editTask(int index, String
            name, String priority);
6
7   }
```

View:

```
1   public abstract class View implements Observer {
2
3       public abstract void show( String input);
4   }
```

Controller:

```
1   public abstract class Controller {
2
3   public abstract void addTask(String title, String
        priority);
4
5   public abstract void removeTask(int index);
6
7   public abstract void editTask(int index, String name,
        String priority);
8
9   }
```

Come sopra detto, le tre sono classi astratte e vengono estese da altre classi e vengono implementati anche i vari metodi.

2.3 Singleton

Il Singleton è un design pattern creazionale che assicura che una classe abbia una sola istanza e fornisce un punto globale di accesso a tale istanza, rendendolo particolarmente utile in situazioni in cui è essenziale garantire che solo un'unica istanza di una classe sia presente nel sistema.

Nel nostro caso, viene utilizzato per inizializzare una sola lista di task, in modo da non avere più istanze e correre il rischio di fare azioni su liste diverse.

```
1   public class TaskList {
2       private static TaskList instance = null;
3       private ArrayList<Task> tasksList = new ArrayList<
            Task>();
4
5       private TaskList() {
6       }
7
8       public static TaskList getInstance() {
9           if (instance == null) {
```

```
10                instance = new TaskList();
11            }
12            return instance;
13        }
14    }
```

La classe chiaramente non termina qui, ci sono altri metodi, tra cui dei get/set, per far si che il codice funzioni, ma che ritengo inutili da riportare qui per far capire l'utilità del pattern.

3 UseCase Diagram

Lo Use Case Diagram è uno strumento essenziale nell'analisi e nella progettazione dei sistemi software, mirato a rappresentare visivamente le interazioni tra gli attori e i casi d'uso all'interno di un sistema. Questo diagramma fornisce una panoramica chiara e comprensibile delle funzionalità offerte dal sistema, focalizzandosi sulle prospettive degli utenti finali e degli altri attori coinvolti.

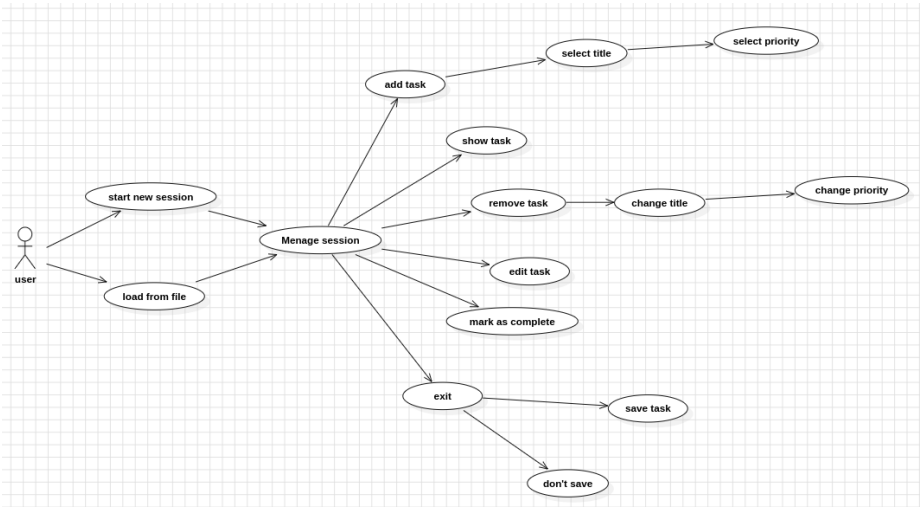

Figure 4: Diagramma UserCase completo.

Nel nostro caso abbiamo la possibilità di caricare le task da un file o da iniziare una nuova sessione nella quale non abbiamo task. Dopo aver superato questo step possiamo fare diverse azioni.

- aggiungere uno nuova task, immettendo un nuovo titolo e una nuova priorità
- visualizzare le task che abbiamo scritto fino a questo momento
- rimuovere una task tra quelle esistenti

- modificarne una di quelle esistenti, quindi cambiando titolo e priorità
- segnare come completata una task o rimetterla da completare
- terminare la sessione e scegliere se salvare o meno quanto fatto

4 Unit Test per la Classe `TaskManager`

In questa sezione, verranno presentati e descritti i test di unità sviluppati per la classe `TaskManager` al fine di garantire la corretta funzionalità delle operazioni fondamentali di gestione dei compiti. Ho testato solo questa classe perché è l'unica che contiene i dati, quindi l'unica vera da testare.

4.1 Test `AddTask`

4.1.1 Scopo

Questo test verifica che l'aggiunta di un nuovo compito aumenti la dimensione della lista dei compiti e che i dettagli del compito siano correttamente registrati.

4.1.2 Procedura

```
1 // Aggiunta di una task.
2 taskManager.addTask("Task1", "High", false);
```

Qui aggiungiamo una task con titolo "Task1", priorità "High", stato "false".

4.1.3 Verifica

```
1 // Verifica che la dimensione della lista dei compiti
     sia aumentata di uno.
2 assertEquals(1, taskManager.tasksList.size());
3 // Verifica che i dettagli del compito aggiunto
     corrispondano alle specifiche.
4 assertEquals("Task1", taskManager.tasksList.get(0).
     getTaskName());
5 assertEquals("High", taskManager.tasksList.get(0).
     getPriority());
6 assertFalse(taskManager.tasksList.get(0).getIsComplete
     ());
```

4.2 Test `RemoveTask`

4.2.1 Scopo

Questo test assicura che la rimozione di un compito dalla lista comporti la corretta riduzione delle dimensioni e che i compiti rimanenti mantengano la loro integrità.

4.2.2 Procedura

```
// Aggiunta di due compiti.
taskManager.addTask("Task1", "High", false);
// Rimozione del primo compito.
taskManager.addTask("Task2", "Low", false);
taskManager.removeTask(0);
```

4.2.3 Verifica

```
// Verifica che la dimensione della lista dei compiti
   sia diminuita di uno.
assertEquals(1, taskManager.tasksList.size());
// Verifica che il secondo compito nella lista
   mantenga i dettagli originali.
assertEquals("Task2", taskManager.tasksList.get(0).
   getTaskName());
assertEquals("Low", taskManager.tasksList.get(0).
   getPriority());
assertFalse(taskManager.tasksList.get(0).getIsComplete
   ());
```

4.3 Test `EditTask`

4.3.1 Scopo

Questo test verifica che la modifica di un compito nella lista aggiorni correttamente i suoi dettagli.

4.3.2 Procedura

```
// Aggiunta di un compito.
taskManager.addTask("Task1", "High", false);
// Modifica del compito aggiunto.
taskManager.editTask(0, "Task2", "Low");
```

4.3.3 Verifica

```
// Verifica che la dimensione della lista dei compiti
   non sia cambiata.
assertEquals(1, taskManager.tasksList.size());

// Verifica che i dettagli del compito modificato
   corrispondano alle nuove specifiche.
assertEquals("Task2", taskManager.tasksList.get(0).
   getTaskName());
assertEquals("Low", taskManager.tasksList.get(0).
   getPriority());
assertFalse(taskManager.tasksList.get(0).getIsComplete
   ());
```

4.4 Test `MarkTaskAsComplete`

4.4.1 Scopo

Questo test verifica che la marcatura di un compito come completato imposti correttamente la proprietà `isComplete`.

4.4.2 Procedura

```
// Aggiunta di un compito.
taskManager.addTask("Task1", "High", false);
// Marcatura del compito come completato.
taskManager.markTaskAsComplete(0);
```

4.4.3 Verifica

```
// Verifica che la propriet  isComplete del compito
   sia stata impostata correttamente a true.
assertTrue(taskManager.tasksList.get(0).getIsComplete
   ());
```

5 Conclusioni

Il progetto è stato un assaggio di come si può programmare utilizzando pattern e rendendo il codice più facile da gestire e da maneggiare.

Certamente l'uso di questi pattern è eccessivo per il codice che sono andato a scrivere, ma è un buon inizio per imparare a usarli a dovere.

Al termine di questo lavoro ho imparato tutti i vantaggi di questi modelli e perchè ha senso utilizzarli.