Observer & MVC

Alessandro Midolo, Ph.D. Student
alessandro.midolo@phd.unict.it

Tutorato Ingegneria del Software

A.A. 2021/2022

Observer

Intento

- Definire una dipendenza uno a molti fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e **aggiornati automaticamente**

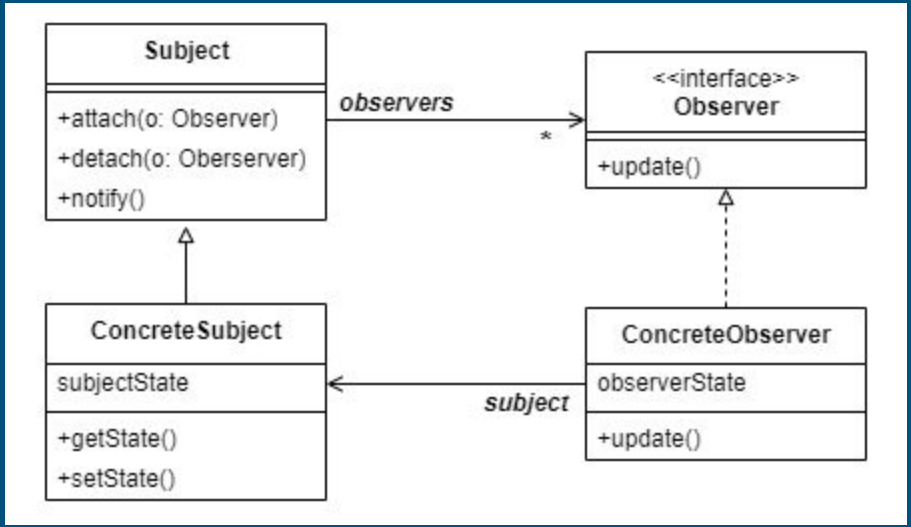
Problema

- Mantenere la *consistenza* fra oggetti che hanno relazioni in un sistema partizionato
- Un cambiamento su un oggetto richiede il cambiamento di altri
- E' necessario notificare altri oggetti senza fare assunzioni su chi sono tali oggetti

Soluzione

- **Subject:** conosce gli osservatori. Implementa operazioni di aggiunta/rimozione/notifica di *Observers*
- **Observer:** definisce un'interfaccia comune a tutti gli oggetti che necessitano la notifica
- **ConcreteSubject:** tiene lo stato interessato dagli osservatori e li notifica quando il suo stato cambia.
- **ConcreteObserver:** ha un riferimento al *ConcreteSubject*, e tiene lo stato consistente con quello del Subject. Implementa *Observer* per ricevere notifiche dei cambiamenti di *Subject*

Diagramma UML Observer



Conseguenze Observer

- Il **Subject** conosce solo la classe **Observer**, quindi non ha bisogno di conoscere le classi **ConcreteObserver**. Inoltre, **ConcreteSubject** e **ConcreteObserver** non sono accoppiati → facile riuso
- Il **ConcreteSubject** manda la notifica a tutti i **ConcreteObserver** iscritti senza sapere quanti questi sono. E' semplicemente un evento che indica il completamento di un'operazione, gli osservatori non sanno cosa sia cambiato
- L'aggiornamento del **Subject** può far avviare tante operazioni sugli **Observer** e altri oggetti collegati. La notifica non dice agli **Observer** cosa è cambiato nel **ConcreteSubject**

Reactive Streams

- Quando il **ConcreteSubject** chiama *notify()*, questo richiama *update()* che verrà eseguito sul **thread del chiamante** → aspettare l'esecuzione di *update()* di ciascun ***ConcreteObserver***
- Con i *ReactiveStre*ams si è cercato di risolvere il problema, senza bloccare il publisher e senza inondare il subscriber
- A partire da Java 9, la libreria java.util.concurrent fornisce le interfacce *Publisher<T>, Subscriber<T>, Subscription* e la classe *SubmissionPublisher*

  Quest'ultima implementa un *Publisher* dedicando un thread per mandare ciascun messaggio ai *Subscriber*

Model View Controller (MVC)

E’ un pattern architetturale per le applicazioni interattive

Problema

- Le interfacce utente possono cambiare, poiché funzionalità, dispositivi o piattaforme cambiano
- Le stesse informazioni sono presentate in finestre differenti
- Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati
- I cambiamenti all’interfaccia utente dovrebbero essere facili
- Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali

Soluzione

- **Model:** incapsula le funzionalità principali e i dati dell’applicazione. E’ indipendente dalla rappresentazione degli output e dagli input. Registra **View** e **Controller**. Li avvisa dei cambiamenti
- **View:** mostra i dati all’utente. Generalmente ci sono tante *View*, ognuna è associata ad un **Controller**. Mostra i dati che legge da **Model** e inizializza il proprio **Controller**
- **Controller:** riceve gli input dall’utente sotto forma di eventi. Traduce gli eventi in richieste di servizio per **Model** o avvisa **View**

Diagramma UML MVC