10. Design Patterns

Andrea Polini

Ingegneria del Software
Corso di Laurea in Informatica

Problemi

Ci focalizziamo nelle problematiche riguardanti la progettazione di moduli architetturali

- Aspetto Economico: Progettazione è un'attività costosa (tempo e budget)
- Aspetto Tecnico: Progettazione di nuovi moduli inevitabilmente porta con se possibilità di errori nel progetto
- Aspetto Antropologico: sistemi prodotti da altri sono sempre difficili da capire e architetto tendenzialmente prova una sensazione di disagio/malessere nell'attività di apprendimento di lavoro fatto da altri

Obiettivi

Le soluzioni tipicamente passano attraverso la definizione di qualche forma di riuso di esperienze già maturate o prodotti già esistenti:

- Aspetto Economico: capacità di trarre vantaggio da attività precedentemente svolte riduce certamente i costi della progettazione e le corrispondenti incertezze sui tempi di progetto
- Aspetto Tecnico: riuso di soluzioni già attuate riduce rischi di errore e fornisce maggiori garanzie sul corretto funzionamento di quanto prodotto
- Aspetto Antropologico: riuso sistematico riduce tempi e difficoltà di apprendimento della struttura del progetto da parte di terzi

Molte possono essere le dimesioni del riuso: CBSE, Aspect Oriented Programming, Generative programming, Librerie . . .

Design Patterns

Definizione: un pattern descrive un problema che ricorre spesso e propone una possibile soluzione in termini di organizzazione di classi/oggetti che generalmente si è rilevata efficace a risolvere il problema stesso.

Design Pattern sono caratterizzati da quattro elementi principali:

- Nome - riferimento mnemonico che permette di aumentare il vocabolario dei termini tecnici e ci permette di identificare il problema e la soluzione in una o due parole
- Problema - descrizione del problema e del contesto a cui il pattern intende fornire una soluzione
- Soluzione - descrive gli elementi fondamentali che costituiscono la soluzione e le relazioni che intercorrono tra questi
- Conseguenze - specifica le possibili conseguenze che l'applicazione della soluzione proposta può comportare. Si riferiscono ad esempio a possibili problemi di spazio o efficienza della soluzione, oppure ad applicabilità con specifici linguaggi di programmazione

Design Patterns e documentazione

Una volta ben definito e generalmente accettato da una comunità di sviluppatori il pattern diventa anche un'ottimo strumento per documentare il software.

Esistono esempi di applicazioni interamente descritte attraverso l'uso di design pattern (e.g. JUnit)

Riferimento:
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Pattern: elements of reusable Object Oriented software
Addison Wesley

Design Pattern

perché è una soluzione efficace?

Si riferisce ad un riuso di un'attività di progettazione dunque:

- Aspetto Economico: riduce i tempi ed i costi di progetto dei singoli moduli
- Aspetto Tecnico: riduce i rischi di progetto e possibilità di errori nel progetto stesso. Soluzione attuata e riutilizzata più volte e di cui si possono prevedere le caratteristiche a priori (e.g. comportamento non-funzionale)
- Aspetto Antropologico: semplifica la comprensione del progetto da parte di terzi fornendo livello di astrazione più chiaro

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

La definizione di collezioni di pattern in uno specifico dominio applicativo sono il primo passo per poter stabilire un contesto di riuso
Nel libro della GoF i pattern sono descritti attraverso i seguenti punti:

- Nome: sintetizza l'essenza del pattern. Obiettivo è ampliare il vocabolario di progetto.
- Intento: Descrizione che cerca di rispondere alla domanda "cosa fa il DP?" "A quale problama di progettazione si rivolge?"
- Aka (Also known as): eventuali altri nomi che identificano il pattern
- Motivazioni: Scenario che illustra il problema e come il pattern fornisce una soluzione
- Applicabilità: situazioni di applicabilità e come poter riconoscere tali situazioni

Come definire un pattern in modo da facilitarne il riuso

Descrizione dei Design Pattern secondo la *GoF*

- Struttura: rappresentazione grafica tramite UML delle classi coinvolte e delle loro relazioni
  - Partecipanti: classi che partecipano al pattern, loro relazioni e responsabilità (class diagram, object ed activity diagram)
  - Collaborazioni: Come collaborano le varie classi per raggiungere gli obiettivi (sequence diagram)
- Conseguenze: vantaggi e svantaggi dell'uso del pattern e possibili effetti collaterali nell'uso del pattern
- Implementazione: tecniche che e suggerimenti per l'implementazione con riferimento anche a specifici linguaggi di programmazione
- Codice sorgente di esempio: frammenti di codice che forniscono una guida per l'implementazione
- Usi noti: esempi di uso in sistemi esistenti
- Patterns correlati: differenze e relazioni più importanti con altri pattern. Tipico uso concomitante.

Classificazione dei Pattern

Elenco di pattern può essere corposo (GoF ha definito 23 pattern generali) risulta importante cercare di identificare caratteristiche per la loro classificazione. Ciò semplifica anche lo studio e la comprensione dei pattern stessi. Sono stati idendificati due criteri principali:

- A cosa si riferisce il pattern
  - Oggetti: relazioni fra oggetti che possono modificarsi a tempo di esecuzione
  - Classi: riguardano relazioni tra classi e sottoclassi
- Cosa fa il pattern (scopo)
  - Creazionali: riguarda il processo di creazione di oggetti
  - Strutturali: riguarda la composizione di classi e di oggetti
  - Comportamentali: definisce come classi e oggetti interagiscono e distribuiscono fra loro delle responsabilità

Certamente è possibile classificarli secondo altri criteri

Object vs. Class pattern

Pattern creazionali di classe risolvono il problema della creazione di oggetti attraverso la delega a sottoclassi.

Per contro pattern di oggetti creazionali risolvono il problema attraverso la definizione di specifiche interazioni tra oggetti.

Pattern comportamentali di classe usano ereditarietà per definire algoritmi e flussi di controllo.

Per contro i pattern di oggetti comportamentali definiscono interazioni tra oggetti che permettano di svolgere una determinata attività

GoF classificazione

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	***Adapter (class)***	Interpreter Template Method
	Object	***Abstract Factory*** Builder Prototype ***Singleton***	Adapter (object) Bridge Composite Decorator Façade Flyweight ***Proxy***	Chain of Responsibility Command Iterator Mediator Memento ***Observer*** State ***Strategy*** Visitor

In che modo i DP contribuiscono a semplificare la progettazione

- Identificare gli oggetti necessari: scomporre sistema in oggetti è compito difficile. Pattern aiutano nell'identificazione di oggetti che rappresentano astrazioni non banali a partire dalla definizione del problema che si vuole risolvere.
- Determinare granularità degli oggetti: l'applicazione di un pattern porta con se scelte riguardanti la granularità. Ci sono pattern che permettono di definire oggetti come composizione di oggetti più semplici oppure pattern che permettono di rappresentare sottosistemi completi con semplici oggetti
- Definizione delle interfacce: l'applicazione di un pattern porta con se le scelte riguardanti le interfacce ed i meccanismi di interazione tra gli oggetti
- Definire implementazione: applicare un pattern comporta specifiche scelte implementative. In particolare vi sono pattern che si focalizzano su ereditarietà mentre altri si focalizzano su meccanismi di composizione

Principi generali di buona progettazione OO

Ereditarietà e polimorfismo

Ereditarietà definisce una classe in termini di un'altra. Principalmente meccanismo di riuso del codice di tipo white-box. Il programmatore deve conoscere il codice della classe base

Principio di sostituzione di Liskov: un oggetto di una sottoclasse deve poter essere utilizzato dove sia atteso un'oggetto della superclasse

Meccanismo molto potente ma presenta alcune caratteristiche da maneggiare con cura:

- Ridefinizione di superclassi può rompere "contratto" delle sottoclassi
- Ereditarietà multipla di classe ed il problema del diamante

È preferibile applicare ereditarietà al livello delle interfacce

Principi generali di buona progettazione OO

Meccanismo della delega

Due tipi di oggetti delegante che rinvia lo svolgimento di un determinato compito ad un oggetto delegato (relazione analoga a classe e sottoclasse)

Il meccanismo della delega (has-a vs is-a) permette di ottenere le stesse caratteristiche di riuso garantite dall'ereditarietà

Caso di una classe Window e di una classe Rettangolo per definirne forma e dimesioni. Un "Window" è un rettangolo oppure ha un rettangolo?

Dunque attraverso il meccanismo della delega si può ottenere lo stesso riuso ottenibile con ereditarietà di classe ma con il vantaggio di avere una maggiore dinamicità a run-time.

Principi generali di buona progettazione OO

Composizione di oggetti consiste nell'assemblare differenti oggetti al fine di ottenere il comportamente desiderato. Composizione richiede definizione di precise interfacce ma non richiede conoscenza di dettagli implementativi.

Principi generali di buona progettazione:

- Programmare riferendosi alle interfacce e non alle implementazioni
- Favorire la composizione di oggetti rispetto all'ereditarietà di classe

DP creazionali

Astraggono il meccanismo di generazione di istanze di una classe con l'obiettivo di rendere il sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati. La creazione degli oggetti viene tipicamente gestita da apposite "strutture".

UN DP creazionale che opera su classi userà ereditarietà per definire la classe da instanziare. Mentre DP che operano su oggetti useranno meccanismi di delega.

In generale semplificano la composizione di classi e la favoriscono rispetto all'uso dell'ereditarietà.

I pattern creazionali forniscono molta flessibilità nel cosa, il come, il chi ed il quando della creazione di oggetti

Saranno considerati due pattern di questo tipo:

- Abstract Factory
- Singleton

Abstract Factory

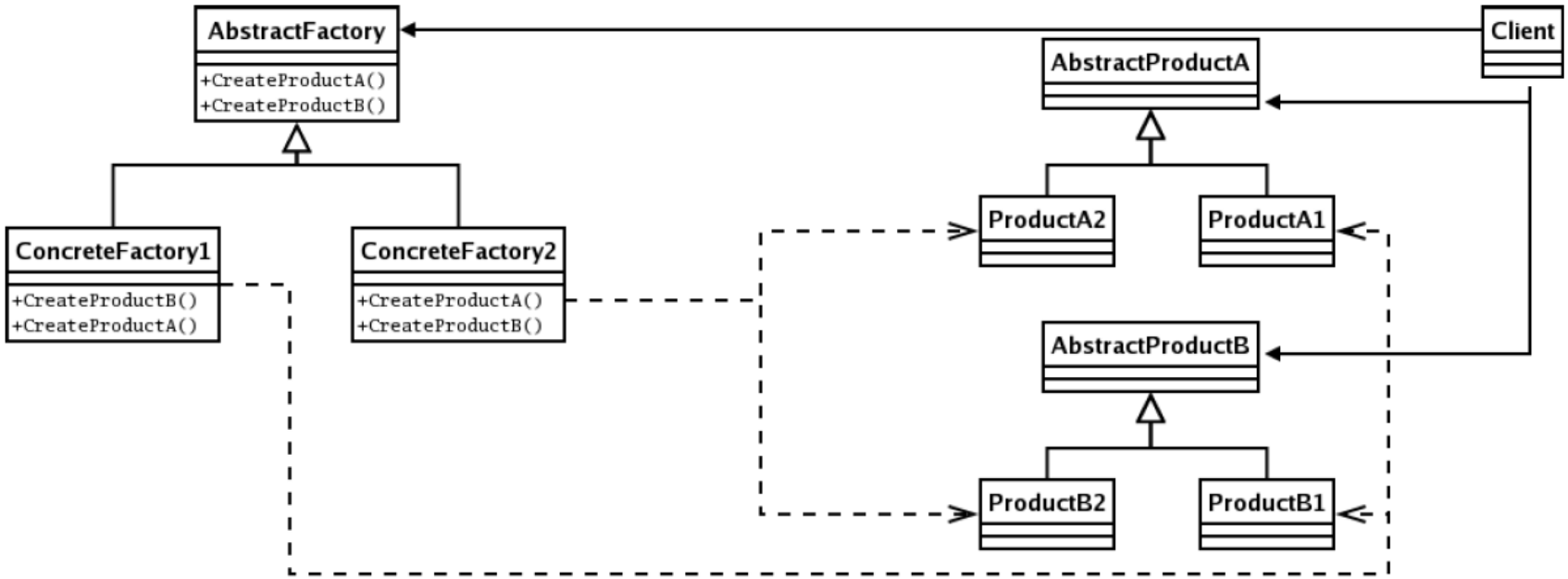
- Scopo: definire un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare quali siano le classi concrete.
- Motivazione: spesso ci si trova di fronte al problema di voler istanziare un oggetto senza specificare precisamente la classe. Comportamento chiaramente non possibile con meccanismi di creazione tipo `new`. Caso tipico delle interfacce grafiche e delle interfacce che mantengono coerenza con il tema in uso.
- Applicabilità: utilizzato quando:
  - sistema indipendente dalle modalità di creazione, composizione;
  - sistema configurabile dipendentemente dalle caratteristiche di una tra piú tipologie di oggetto
  - libreria di classi fornendo solo interfaccia ma non implementazioni specifiche

Abstract Factory

- Conseguenze:
  - isolamento delle classi concrete. Gli oggetti non contengono dettagli che si riferiscono ai meccanismi di creazione degli oggetti. La creazione di un particolare oggetto è visto come un normale servizio fornito da classi preposte allo scopo.
  - Risulta semplice la modifica/aggiunta di classi rappresentanti factory concrete. L'uso di queste risulta estremamente localizzato. Dunque prodotti diversi si ottengono modificando la factory.
  - Il supporto a nuove tipologie di prodotto che richiederebbero di modificare l'abstract factory risulta complessa in quanto richiede di modificare tutte le classi concrete.

Abstract Factory

- Struttura:



Abstract Factory

- Partecipanti:
  - ***AbstractFactory***: Dichiara un'interfaccia per le operazioni di creazione di oggetti astratti
  - ***ConcreteFactory***: implementa le operazioni di creazione di oggetti concreti
  - ***AbstractProduct***: Dichiara un'interfaccia per una tipologia di oggetti
  - ***ConcreteProduct***: definisce un oggetto che dovrà essere creato dalla corrispondente factory concreta
  - ***Client***: utilizza soltanto le interfacce dichiarate sopra
- Collaborazioni:
  - spesso esiste una sola istanza di una ConcreteFactory a run-time per le diverse tipologie
  - La classe AbstractFactory delega la creazione di oggetti alle sue sottoclassi.

Abstract Factory

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: associato ad altri pattern creazionali in particolare al Singleton per avere classi concrete che non ammettono istanze multiple.

Abstract Factory - Codice di esempio

Codice di esempio per i creare differenti livelli di un videogioco

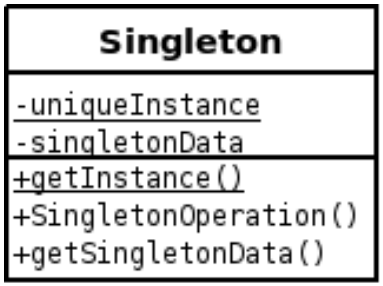
```
public interface MazeFactory {
  Maze MakeMaze();
  Wall MakeWall();
  Door MakeDoor();
  Room MakeRoom();
}

public class MazeGame(MazeFactory mf) {
  Maze aMaze = mf.MakeMaze(); Room r1 = mf.MakeRoom(1);
  Room r2 = mf.MakeRoom(2); Door aDoor = mf.MakeDoor(r1,r2);
  aMaze.AddRoom(1); aMaze.addRoom(2);
  r1.SetSide(N,mf.MakeWall()); r1.SetSide(S,mf.MakeWall());
  r1.SetSide(E,mf.MakeWall()); r1.SetSide(O,aDoor);
  r2.SetSide(N,mf.MakeWall()); r2.SetSide(S,mf.MakeWall());
  r2.SetSide(E,aDoor); r2.SetSide(O,mf.MakeWall());
  return aMaze;
}
```

Singleton

- Scopo: Fare in modo che a run-time esista al più una sola istanza di una classe e fornire un punto globale di accesso a tale istanza
- Motivazione: É spesso importante avere una sola istanza di una classe a run-time. Si consideri ad esempio il caso del File System Manager o del Window Manager di un sistema. (Kdm o Gdm per chi usa Linux). Dunque come possiamo fare in modo da garantire che a run-time non sia possibile avere più istanze di una stessa classe e che tale istanze sia facilmente accessibile ai potenziali utilizzatori?
- Applicabilità: il pattern singleton puè essere usato quando:
  - deve esistere una sola istanza di una classe e punto di accesso noto agli utilizzatori
  - quando l'unica istanza deve poter essere estesa attraverso definizione di sottoclassi ed i client non devono essere modificati come conseguenza di ciò

Singleton

- Conseguenze: controllo degli accessi all'istanza, spazio dei nomi ridotto, possibilità di estensione ad aver un numero n di istanze, facile manutenzione
- Struttura:



- Partecipanti:
  - ***Singleton***: definisce un'operazione `Instance` che permette ai client di accedere all'unica istanza disponibile della classe. La detta operazione deve essere un'operazione di classe ovvero in Java dovrà essere marcata dalla parola chiave `static`

Singleton

- Collaborazioni: I client possono accedere ad un'istanza di singleton soltanto invocando l'operazione `Instance`
- Conseguenze: Il pattern Singleton offre importanti benefici:
  - Accesso controllato ad un'unica istanza
  - Riduzione dello spazio dei nomi
  - Raffinamento delle operazioni e della rappresentazione interna
  - Gestione di un numero variabile di istanze
- Implementazione: si possono fare alcune considerazioni sui differenti meccanismi messi a disposizione dai differenti linguaggi OO per la definizione di operazioni di classe o per altri meccanismi che possono influenzare l'implementazione con un dato linguaggio.
- Utilizzi noti: ...
- Pattern correlati: altri pattern creazionali potrebbero usare il pattern singleton per esempio per risolvere gli specifici problemi "creazionali" associandoli alla necessità di avere una sola istanza.

Singleton - Codice di esempio

Quanto segue è un esempio di codice Java che permette di implementare una classe Singleton:

```
public class Singleton {
  private static Singleton instance = null;
  private static int counter = 0;

  ..... // Objects specific attributes and methods

  private Singleton() {...}
  public static Singleton Instance() {
    if (counter == 0) {
      instance = new Singleton();
      counter++;
    }
    return instance;
  }
}
```

Dunque un oggetto che abbia bisogno di poter utilizzare l'oggetto singleton potrà farlo con un invocazione del tipo:

```
Singleton.Instance()
```

Pattern Strutturali

I pattern strutturali descrivono come comporre oggetti o classi per creare strutture complesse.

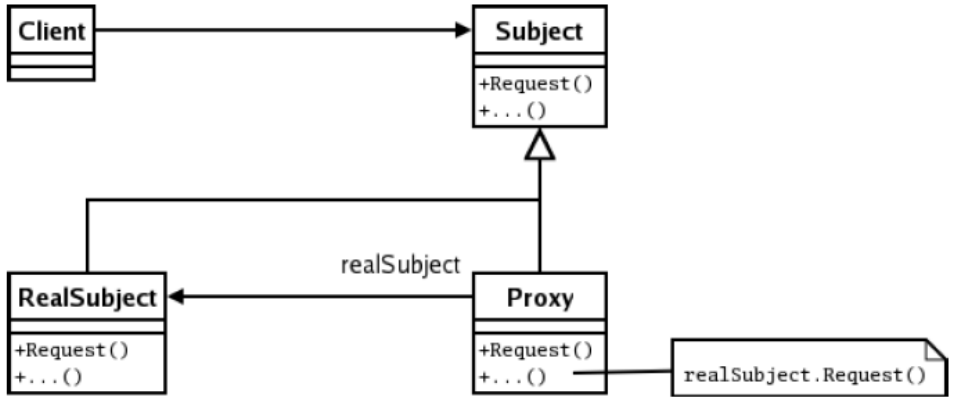
L'unico pattern strutturale che vedremo è quello del Proxy

Proxy

- Scopo: fornire un surrogato o un segnaposto per un altro oggetto al fine di controllare l'accesso all'oggetto stesso. Allo stesso tempo può essere utile per rimandare la creazione di un oggetto ad un momento successivo al fine di ridurre uso di risorse.
- Motivazione: motivo può essere differire il costo di creazione ed inizializzazione dell'oggetto ad un momento successivo quando questo è effettivamente necessario
- Applicabilità: il pattern proxy puè essere usato secondo diverse tipologie:
  - remote proxy: fornisce rappresentazione locale per un oggetto residente in un diverso spazio di indirizzamento
  - virtual proxy: gestisce su richiesta la creazione di oggetti costosi
  - protection proxy: controlla l'accesso ad un oggetto. Si rivela utile quando possone esserci diversi diritti di accesso allo stesso oggetto.
  - smart reference: sostituisce quello che è un puntatore puro con un puntatore dalle funzionalità aggiutive quando si accede all'oggetto

Proxy

- Struttura e Codice di esempio:




```
public interface Graphic {...}

public class Image implements Gaphic {
  Image(FileInputStream fis) {...}; ...
}

public class ImageProxy implements Graphic {
  String fileName;
  public void Draw(at) { Image i = new Image(fis); i.Draw(at); }
}
```

Proxy

- Conseguenze:
  - introduce un livello di indirezione nell'accesso ad un oggetto. In generale operazioni molto complesse possono portare a degrado.
  - disaccoppiano cliente e oggetto stesso ottimizzando l'uso delle risorse rimandando solo al momento necessario l'esecuzione di operazioni potenzialmente costose.
- Partecipanti:
  - ***Proxy***: mantiene un riferimento che consente al proxy di accedere all'oggetto rappresentato. Proxy deve implementare la stessa interfaccia del referenziato al fine di nascondere la propria presenza al cliente
  - ***Subject***: definisce l'interfaccia comune per l'oggetto referenziato ed il proxy. Il proxy può sempre essere usato dove è richiesto l'oggetto referenziato.
  - ***RealSubject***: caratterizza l'oggetto referenziato dal proxy

Proxy

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: ....

Caratterizzazioni importanti che comunque nel nostro studio non verranno approfondite.

Pattern Comportamentali

I pattern comportamentali hanno a che fare con gli algoritmi e gli assegnamenti di responsabilità tra gli oggetti. Caratterizzano sistemi di controllo complessi.

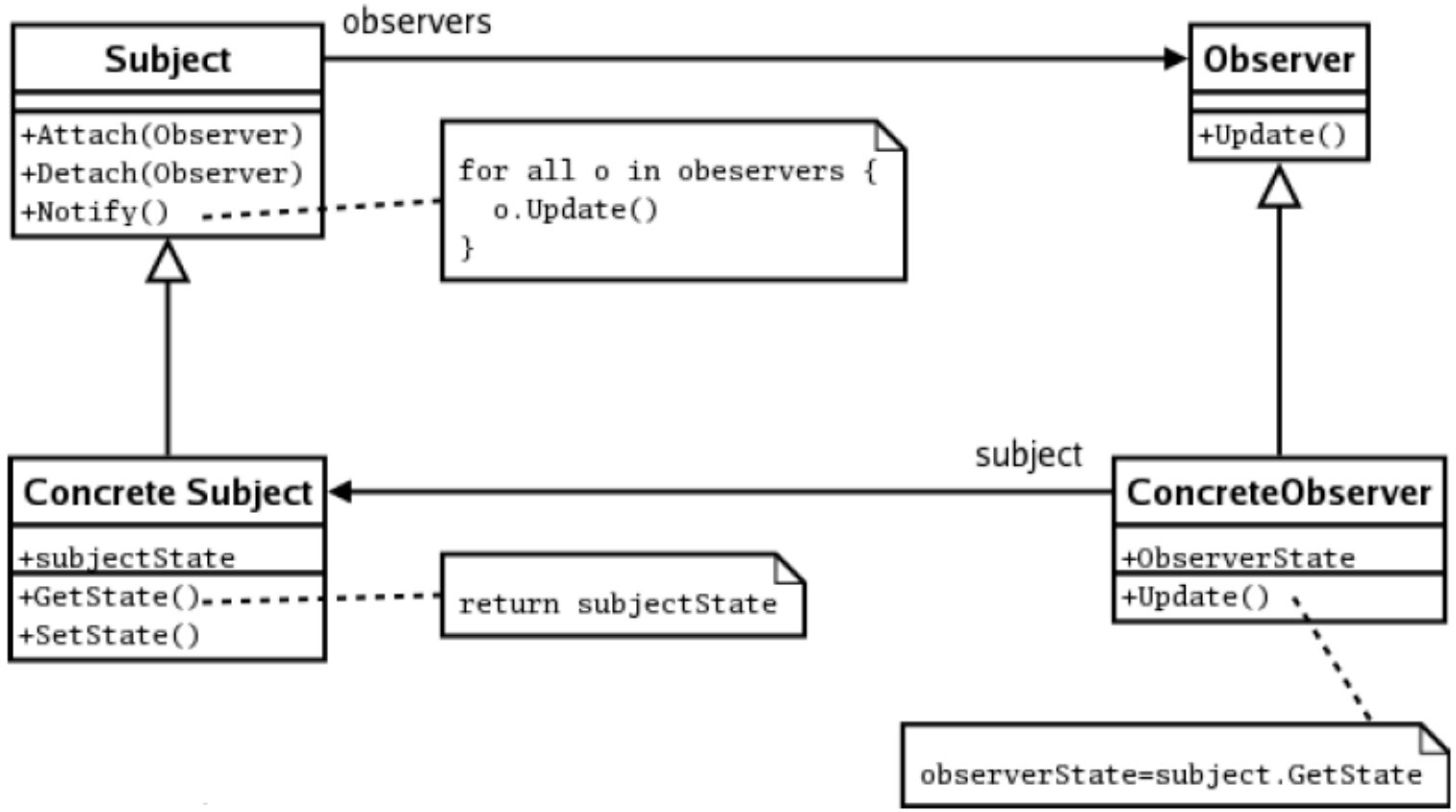
L'unico pattern strutturale che vedremo è quello delloi Observer

Observer

- Scopo: definisce una relazione uno-a-molti tra oggetti cosicché quando l'oggetto (uno) cambia stato tutti gli altri oggetti (molti) coinvolti vengano informati del cambiamento
- Motivazione: c'è bisogno di mantenere consistenza tra gli oggetti correlati
- Applicabilità: quando una modifica ad un oggetto richiede modifiche ad altri oggetti e non è noto a priori quali saranno tali oggetti

Observer

- Struttura:



Observer

- Conseguenze:
  - Accoppiamento astratto e minimale tra subject e observer: `subject` conoscono lista di `observer`
  - Supporto per comunicazioni broadcast: fornisce meccanismi per implementare comunicazione di un l'evento è da notificare a tutti coloro abbiano manifestato interesse ad un evento. Aggiunta e rimozione di osservatori estremamente semplice.
  - Aggiornamenti inattesi: modifiche apportate da un osservatore potrebbero scatenare sequenza costosa di invocazioni senza che questo possa essere facilmente stabilito a priori.
- Partecipanti:
  - ***Subject***: conosce i suoi "osservatori" e fornisce meccanismi per il collegamento e lo scollegamento degli osservatori
  - ***Observer***: fornisce un'interfaccia per la notifica
  - ***ConcreteSubject***: contiene lo stato ed invia le notifiche
  - ***ConcreteObserver***: gestisce il riferimento al soggetto e mantiene l'informazione sullo stato dell'oggetto osservato.

Observer

- Implementazione: ...
- Utilizzi noti: ...
- Pattern correlati: ....

Caratterizzazioni importanti che comunque nel nostro studio non verranno approfondite.