UNIVERSITÀ
DEGLI STUDI
FIRENZE

INGEGNERIA DEL SOFTWARE

Composizione Abstract Factory + Observer

Sunto

Un piccolo progetto con lo scopo
di mostrare la composizione tra
due pattern da noi studiati.

Iacopo Erpichini

Sommario

Obbiettivo:

L'obbiettivo di questo elaborato è quello di unire le funzionalità di due design pattern.

Utilizzando il pattern abstract factory posso creare oggetti dinamicamente di vario tipo con varie astrazioni e con il pattern observer posso creare un subject concreto, a seconda del suo cambiamento di stato può notificare il suo cambiamento ad un altro oggetto, sarà nel mio caso specifico un observer che implementerà il pattern abstract factory per scegliere un oggetto da produrre in base allo stato del soggetto.

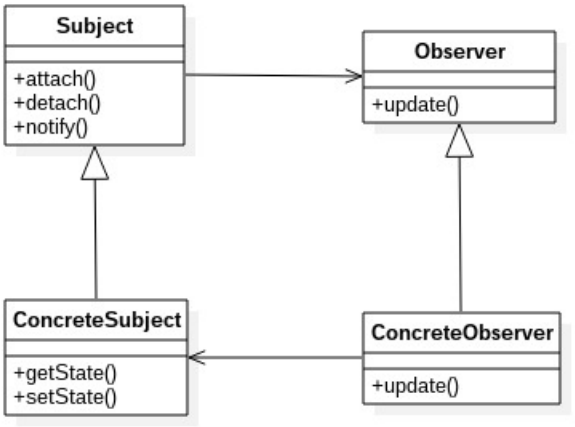
Design Pattern:

A seguito è illustrato il funzionamento dei due design pattern da me utilizzati.

Observer:

Il design pattern Observer è un pattern Comportamentale basato su oggetti. Definisce una dipendenza uno-a-molti tra oggetti cosi che quando un oggetto cambia stato, tutti i soggetti subordinati verranno avvertiti e aggiornati automaticamente. Consideriamo un oggetto che al suo interno contiene le coordinate di uno spazio e ogni sua variazione deve essere notificata ad un oggetto che effettua la sua rappresentazione grafica. Il subject deve essere indipendente dal numero e dal tipo dei sui osservatori. In altre parole, il subject non fa assunzioni sugli oggetti che dipendono da esso. In definitiva il subject rende disponibile delle operazioni che consentono ad un osservatore di dichiarare il proprio interesse per un cambiamento di stato.
Io ho utilizzato un observer di tipo pull.

Schema observer:



Partecipanti:

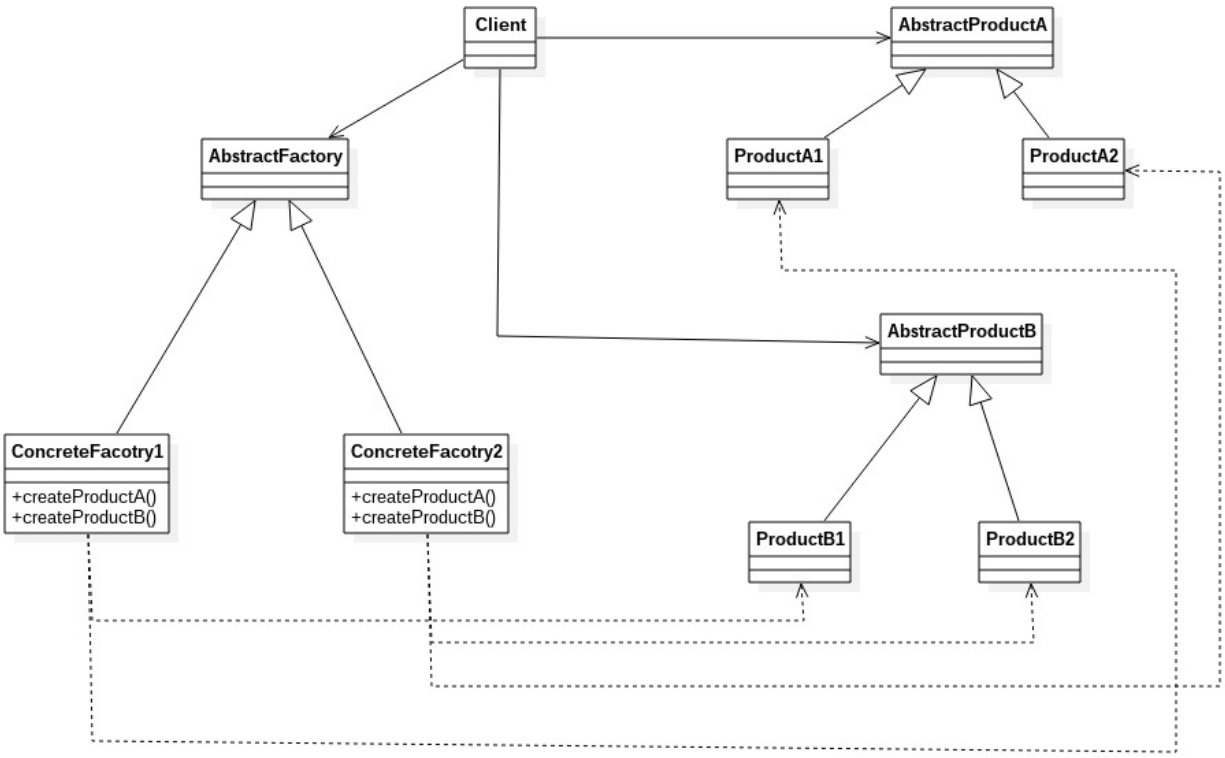
Questo pattern è composto dai seguenti partecipanti:
Subject: Conosce i sui osservatori e provvede a fornire un'interfaccia di accoppiamento e disaccoppiamento per gli oggetti osservati.
Observer: Definisce e aggiorna l'interfaccia per gli oggetti che dovranno essere notificati nel caso in cui il soggetto subisca una modifica.
ConcreteSubject: Invia una notifica ai sui osservatori quando il suo stato cambia.
ConcreteObserver: Mantiene un riferimento all'oggetto ConcreteSubject.

Abstract Factory:

L'Abstract Factory è un design pattern creazionale, fornisce un'interfaccia per creare famiglie di oggetti interconnessi senza dover specificare le loro classi concrete. Si consideri un'applicazione la cui interfaccia grafica fornisca all'utilizzatore aspetti diversi a cui sono associati comportamenti

diversi di vari widget.
Affinchè l'applicazione possa fornire diversi look-and-feel occorre che i widget
non siano codificati per il particolare aspetto, infatti istanziare delle classi specifiche per ogni
aspetto complicherebbe il successivo cambio di interfaccia.

Schema Abstract Factory:



Partecipanti:

Questo pattern è composto dai seguenti partecipanti:
AbstractFactory: Interfaccia di esposizione di operazioni realizzate dai prodotti concreti.
ConcreteFactory: Implementa delle operazioni per la creazione degli oggetti dei prodotti.
AbstractProduct: Interfaccia di esposizione delle operazioni dei prodotti concreti.
ConcreteProduct: Implementazione delle operazioni dei prodotti concreti.
Client: Invocazione delle interfacce per la creazione dei prodotti.

Progetto:

Descrizione:

Il mio progetto si basa sulla creazione di due factory che producono automobili in particolare una fabbrica produce automobili senza optional (standard) mentre l'altra fabbrica produce automobili con optional.

I prodotti che vengono realizzati da ciascuna fabbrica sono due tipologie di auto, (potevano essere molte di più ma ai fini della realizzazione del mio esempio due sono sufficienti) le auto prodotte sono di tue tipi: berline oppure suv.

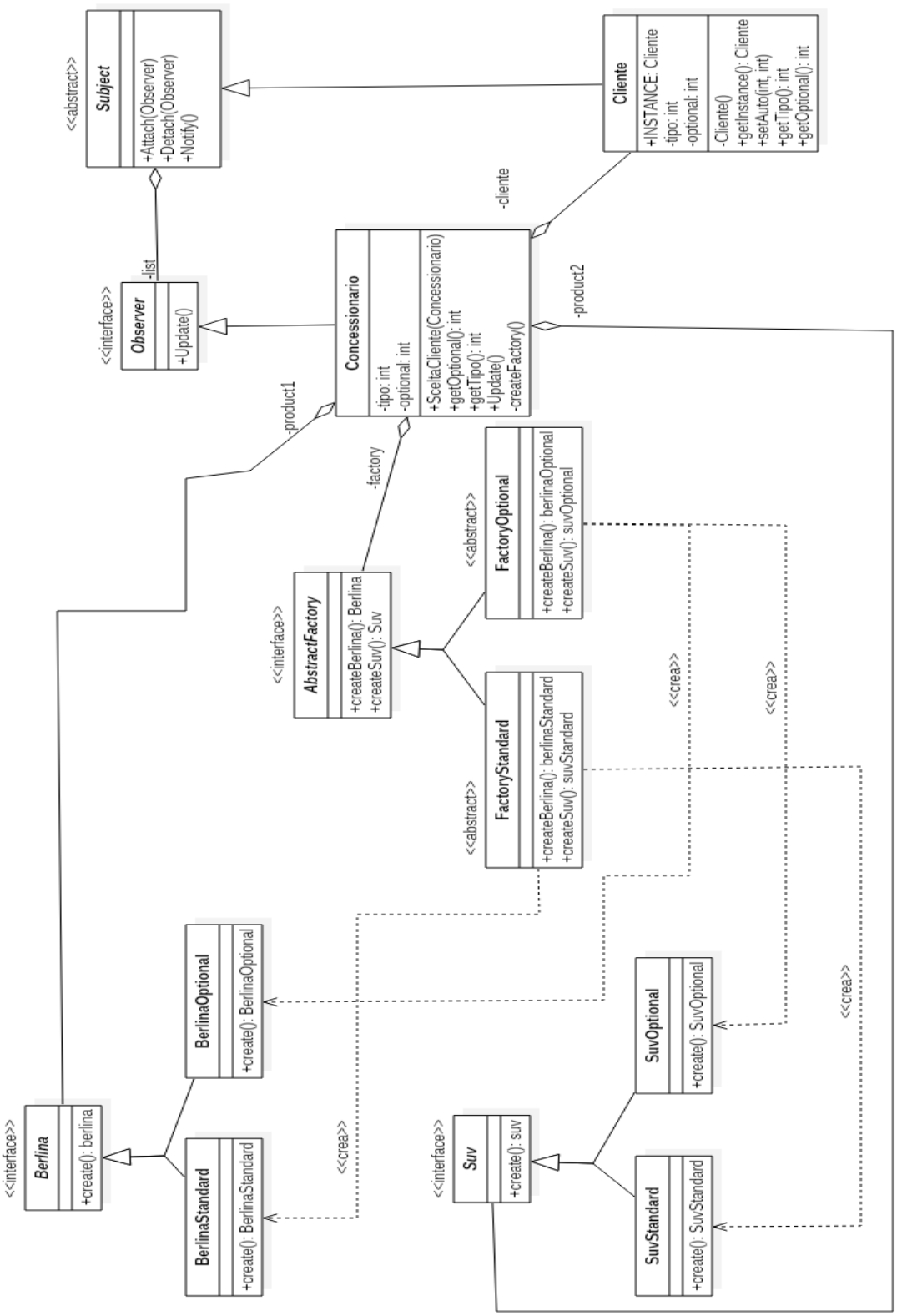
Immagino che ci sia un concessionario che deve fornire a un cliente una scelta tra due tipologie di macchina e tra la possibilità di avere la macchina con o senza optional.

Il soggetto sarà quindi il cliente che avrà due stati che determinano la scelta dell' auto e dell' optional mentre il concessionario avrà il ruolo di osservatore.

Una volta viste le richieste del cliente ovvero il soggetto l'osservatore si occuperà di creare la factory adatta per soddisfare la necessità di prodotto richiesta che a sua volta creerà il prodotto finale.

Uml:

A seguito è riportato l' uml del mio progetto.



Codice:

Qui riporto tutte le classi del mio progetto a eccezione di berlina, berlinaStandard e berlinaOptional in quanto sono uguali a suv cambia solo il messaggio che si verrà a stampare.

Classe Cliente:

```
public class Cliente extends Subject{
    private int tipo;
    private int optional;
    public static Cliente INSTANCE = null;

    private Cliente(){}

    public static Cliente getIstance(){
        if(INSTANCE == null){
            INSTANCE = new Cliente();
        }
        return INSTANCE;
    }

    public void setAuto(int tipo, int optional) {
        this.tipo = tipo;
        this.optional = optional;
        Notify();
    }

    public int getTipo() {
        return tipo;
    }

    public int getOptional() {
        return optional;
    }

}
```

Classe Observer:

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

Classe Subject:

```
public abstract class Subject {
    ArrayList<Observer> list = new ArrayList<>();
    public void Attach(Observer o){
        list.add(o);
    }
    public void Detach(Observer o){
        list.remove(o);
        Notify();
    }

    public void Notify() {
        for(int i = 0 ; i < list.size();i++){
            list.get(i).update();
        }
    }
}
```

Classe Abstract Factory:

```
public interface AbstractFactory {
    public Berlina createBerlina();
    public Suv createSuv();
}
```

Classe FactoryOptional:

```
public class FactoryOptional implements AbstractFactory{

    @Override
    public Berlina createBerlina() {
        return new BerlinaOptional();
    }

    @Override
    public Suv createSuv() {
        return new SuvOptional();
    }

}
```

Classe FactoryStandard:

```
public class FactoryStandard implements AbstractFactory{

    @Override
    public Berlina createBerlina() {
        return new BerlinaStandard();
    }

    @Override
    public Suv createSuv() {
        return new SuvStandard();
    }

}
```

Classe Concessionario:

```
public class Concessionario implements Observer{

    private int tipo;
    private int optional;
    private Cliente c;
    private AbstractFactory factory;
    private Berlina berlina;
    private Suv suv;

    public Concessionario(Cliente c){
        this.c = c;
        this.c.Attach(this);
    }
    @Override
    public void update() {
        this.tipo = c.getTipo();
        this.optional = c.getOptional();
        createFactory();
    }

    private void createFactory(){
        if(optional == 0){
            factory = new FactoryStandard();
        }else if (optional == 1){
            factory = new FactoryOptional();
        }
        if(tipo == 0){
            berlina = factory.createBerlina();
            berlina.create();
        }else if (tipo == 1){
            suv = factory.createSuv();
            suv.create();
        }
    }
    public int getTipo() {
        return tipo;
    }
    public int getOptional() {
        return optional;
    }

}
```

Classe Suv:

```
public interface Suv {
    Suv create();
}
```

Classe SuvOptional:

```
public class SuvOptional implements Suv {

    @Override
    public Suv create() {
        System.out.println("Creazione suv full optional");
        return this;
    }

}
```

Classe SuvStandard:

```
public class SuvStandard implements Suv {

    @Override
    public Suv create() {
        System.out.println("Creazione suv standard");
        return this;
    }

}
```

A seguito riporto anche un Test del mio programma:

```
public class Progetto {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Cliente c = Cliente.getIstance();
        Concessionario sc = new Concessionario(c);
        c.setAuto(0, 0);
        c.setAuto(0, 1);
        c.setAuto(1, 0);
        c.setAuto(1, 1);

    }

}
```

Il risultato del seguente test è:

```
run:
Creazione berlina standard
Creazione berlina full optional
Creazione suv standard
Creazione suv full optional
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sequence Diagram:

Riporto anche un diagramma delle sequenze del mio test omettendo tutti i setAuto tranne il primo per motivi di ridondanza.