

11-05-2023

Si levano le **VARIABILI LOCALI** a un **METODO** e al loro posto si mette un **METODO** che **contiene** quel **VALORE**
VANTAGGI:
Pr. Modularità:
Spezza l'ALGORITMO in METODI
(Mille su variabili Assegnate Singola Volta)

2° Tecnica sostituisci Temp con Query

Le Temp sono **variabili temporanee**, quindi non sono attributi. Si eliminano le **variabili locali a un metodo** e al loro posto si può mettere un **metodo che tiene il valore che potrebbe tenere la variabile (Query)**.

```
//Temp
double prezzoBase = quantita * prezzo;

//diventa Query
private double prezzoBase(){
    return quantita*prezzo;
}
```

I **VANTAGGI** nell'uso di questa tecnica sono:

- Quando si ha una variabile locale (temp) *si può usare solo all'interno del proprio scope*, quindi all'interno di un unico blocco, mentre *adesso lo scope aumenta* e migliora la possibilità di usare tale variabile e la **modularità**.
- Ci si libera delle variabili locali che producono codice lungo e quindi si **spezzetta l'algoritmo in più metodi**
- E' utile applicare questa tecnica *prima di Estrai Metodo*
- Se la variabile è **assegnata solo una volta** allora si può pensare di applicare questa tecnica. Se, invece, le **assegnazioni sono diverse** e varie, allora bisogna riflettere sul da farsi.

Per **verificare** che la variabile è **assegnata solo una volta** basta provare a **compilare** assegnando la variabile temp come `final`.

La parte destra dell'assegnazione va dentro il corpo del metodo e nei punti dove la variabile viene usata si mette il nome del metodo creato.

```
private double quantita, prezzo;
public double getPrezzo1() {
    double prezzoBase = quantita * prezzo;
    double sconto;

    if (prezzoBase > 1000) sconto = 0.95;
    else sconto = 0.98;

    return prezzoBase * sconto;
}

//diventa
private double quantita, prezzo;
public double getPrezzo2(){
    return prezzoBase() * sconto();
}

private double prezzoBase() {
    return quantita * prezzo;
}
```

```

}

private double sconto() {
    if (prezzoBase() > 1000) return 0.95;
    return 0.98;
}

```

→ *VARIABLE TEMPORANEA* Assegnata più di una volta. Non è Assegnata in un loop o Accumulare Valori.

3° Tecnica dividi variabile Temp

Una variabile temporanea è **assegnata più di una volta**. **NON** è assegnata in un *loop* o usata per *accumulare valori* ma viene **sempre sovrascritta**.

- La variabile temporanea ha un *nome non significativo*
- Se la variabile viene divisa si **rende il codice più comprensibile**
- **Non ci si preoccupa delle prestazioni** (*memoria* in uso) perchè, quando si esce dal metodo, le variabili create vengono distrutte perchè escono dal proprio *scope*
- Le parti divise della Temp diventano `final`

```

double temp = 2 * (height + width); //prima assegnazione
System.out.println(temp);
temp = height * width; //seconda assegnazione
System.out.println(temp);

//diventa
final double perim = 2 * (height + width);
System.out.println(perim);
final double area = height * width;
System.out.println(area);

```

- Se *l'assegnazione avviene molte volte ma si tratta di **accumulazione***, allora la tecnica non si applica
- *Se l'assegnazione avviene molte volte e **Temp ha più responsabilità***, allora la tecnica si applica.
- Si prova assegnando `final` a Temp: se **ci sono altre assegnazioni**, si verifica che **non sia usata come accumulatore**, allora si applica la tecnica. Il nome della seconda assegnazione (*e anche della prima*) si cambia e tutte le future occorrenze presenti nel codice; altrimenti si lascia così e com'è.
- La **rinominazione** può essere **facilitata rinominando la prima assegnazione e compilando il codice**. Il compilatore segnerà un errore del tipo "*Variabile non dichiarata*" ed è proprio lì che bisognerà dare *dichiarare la seconda variabile*.

Esempio di utilizzo:

```
private double primaryForce, secondaryForce, mass, delay;

public double getDistanceTravelled1(int time) {
    double result;
    double acc = primaryForce / mass; // prima assegnazione
    int primaryTime = (int) Math.min(time, delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondT = (int) (time - delay);
    if (secondT > 0) {
        double primaryVel = acc * delay;
        acc = (primaryForce + secondaryForce) / mass; // seconda assegnazione
        result += primaryVel * secondT + 0.5 * acc * secondT * secondT;
    }
    return result;
}
```

Diventa

```
public double getDistanceTravelled2(int time) {
    double result;
    final double primAcc = primaryForce / mass;
    int primaryTime = (int) Math.min(time, delay);
    result = 0.5 * primAcc * primaryTime * primaryTime;
    int secondT = (int) (time - delay);
    if (secondT > 0) {
        double primaryVel = primAcc * delay;
        final double secondAcc = (primaryForce + secondaryForce) / mass;
        result += primaryVel * secondT + 0.5 * secondAcc * secondT * secondT;
    }
    return result;
}
```

17

Prof. Tramontana - Marzo 2

1) Classi con UNA SOLA RESPONSABILITÀ, vi sono dati sparsi NELLE CLASSI

Design Pattern Observer ⇒ COMPORTAMENTALE

STABILISCE UNA DIPENDENZA 1 a Molti tra gli oggetti in modo che QUANDO

Intento ⇒

un oggetto cambia stato, tutti gli oggetti dipendenti da esso vengono notificati e aggiornati AUTOMATICAMENTE.

(Questo permette una forte separazione tra il SUBJECT e gli OBSERVER che devono rispondere a tali livelli.)

- Le classi devono avere una sola responsabilità e possono contenere molti oggetti e le interazioni possono diventare complicate.
- Si hanno dati sparsi fra le classi e tali dati devono rimanere consistenti fra loro. Il dato su cui deve lavorare una classe viene appena aggiornato da un'altra classe.
- Quindi **il dato aggiornato in una prima classe** deve essere **aggiornato nella seconda classe** così che la seconda classe possa **usarlo in modo consistente**.
- Un dato viene aggiornato da una classe e deve essere fatto avere da tutte le altre classi.
- La propagazione dell'aggiornamento del dato deve avvenire in maniera automatica in modo da rendere complicate le dipendenze fra oggetti.

Esempio: Si ha una tabella di dati che viene aggiornata e il dato che si inserisce in una cella deve aggiornare automaticamente tutte le altre caselle che usano tale dato.

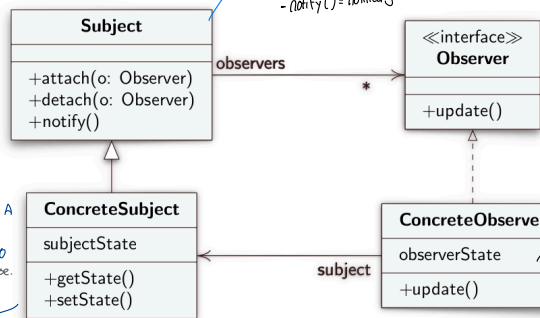
In questo design pattern vengono descritti:

- OBSERVER**: osservatore dei dati. Qui ci vanno gli algoritmi che usano i dati senza aggiornarli.
- SUBJECT**: tiene i dati osservati, quindi ha tanti Observer. Qui ci vanno gli algoritmi che tengono e aggiornano il dato. Tale aggiornamento deve essere visto dagli Observer. Non deve conoscere quante/quali sono gli osservatori e il suo lavoro deve essere il più semplice possibile.

Soluzione

Diagramma delle classi

TIENE I DATI ED È PRONTA A DARLI A CHI GLI SERVE.
SA COME AGGIORNARE IL DATO GRAZIE A `notify()` della superclasse.



CLASSE NON ASTRATTA HA UNA LISTA DI OSSERVATORI:
- contiene gli algoritmi che AGGIORNANO AUTOMATICAMENTE i DATI
- `attach(o: Observer)` = aggiunge osservatori.
- `detach(o: Observer)` = rimuove degli osservatori.
- `notify()` = notifica gli osservatori e quindi chiama i metodi (`update()`) per aggiornarli scorrendo la LISTA.

SAVANO FARE DUESE COSE IN PIU' CONOSCE il ConcreteObject e ne può usare i METODI.
// Ci sono due MODALITÀ per il PATTERN.

PUSH = I dati arrivano con la chiamata `update()`

PULL = il ConcreteSubject avvisa con `notify()` ad `update()` ma non gli passa i DATI (Quando gli serve il DATO il ConcreteObserver gli chiede il DATO a ConcreteSubject e fa una PULL del dato tramite getState().)

ConcreteSubject è sottoclasse di **Subject**:

- Tiene i dati ed è pronta a darli aggiornati a chi ha bisogno di conoscerli
- Lo stato `subjectState` può variare e non per forza si deve sovrascrivere
- Sa come aggiornare il dato mediante `notify()` della superclasse

Subject è una classe non astratta (**ESAME**) ha una lista di osservatori e:

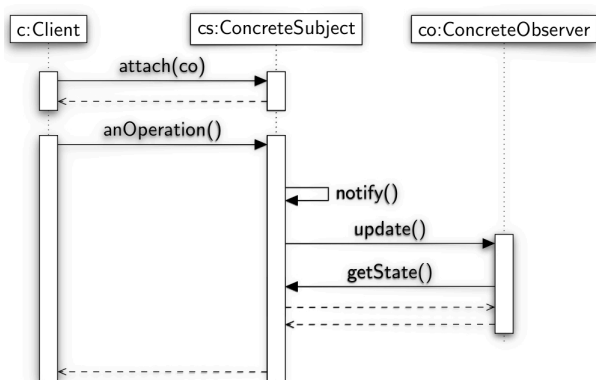
- tiene gli algoritmi che aggiornano automaticamente i dati e notifica *Observer*
- `attach(o: Observer)` -> aggiunge degli osservatori
- `detach(o: Observer)` -> rimuove degli osservatore
- `notify()` -> notifica tutti gli osservatori e quindi chiama metodi (`update()`) sui vari osservatori utilizzando (scorrendo) la lista di osservatori che ha al suo interno

Ci sono *ConcreteObserver* che sanno fare **determinate** cose differenti dagli altri mentre **observer** è un'interfaccia (**ESAME**).

Il *ConcreteObserver* conosce il *ConcreteSubject* e ne può usare i metodi.

Quindi ci sono **2 modalità** per questo Design Pattern:

- I dati arrivano con la chiamata ad `update()` e la modalità è detta **PUSH** perchè Subject spinge i dati verso il ricevente Observer
- L'altra modalità è **PULL**: il CS avvisa con `notify()` ad `update()` ma non gli passa i dati. Quando gli serve il dato, CO chiede il dato a CS e fa un'operazione di pull del dato mediante il metodo `getState()`



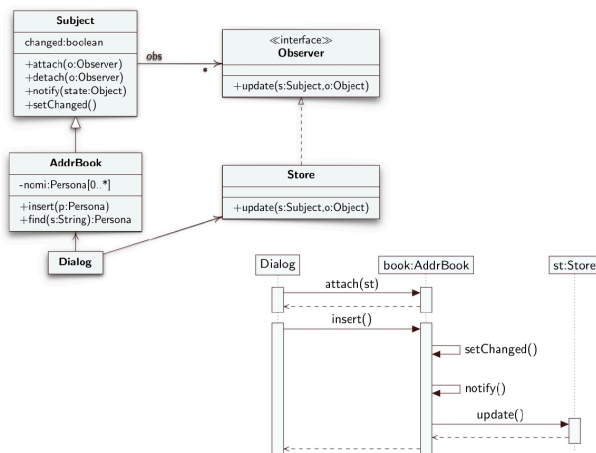
Prima e Dopo l'Uso di Observer



PRIMA: in A devo mettere le chiamate di tutte le altre classi passando il proprio stato. Le classi B, C, D costringono A a tenere al suo interno dei riferimenti a B, C, D e le 3 istanze non hanno nulla in comune

DOPO: si ha una semplificazione delle relazioni fra le class. A non conosce le altre classi (con Observer) ma solo Subject di cui è sottoclasse. Subject conosce **solo l'interfaccia Observer** e quindi non conosce le sottoclassi B, C, D. A runtime riuscirà a invocare i vari metodi update() delle sottoclassi mediante l'interfaccia

Esempio



- `find()` non modifica i dati e quindi non avrà al suo interno una chiamata a `notify()`
- `changed()` con `setChanged()` serve a Subject per farsi dire dal CS se lo stato è cambiato e, solo quando si vorrà, si notificheranno gli altri.
- Da `insert()` si chiama `notify(nomi)` -> `nomi` passato a `notify()` diventa `Object` -> `update(Object)` e la classe `Store` converte in `List<Persona>`
- `update()` deve avere come primo parametro `Subject` e il **secondo è facoltativo** (in base alla modalità scelta).
 - Gli viene passato `Subject` così che, nel CO si può usare `getState()` se previsto per farsi dare lo stato (quindi *PULL*).
 - `Subject` si mette *obbligatoriamente* perchè gli `Observer` potrebbero osservare più CS (quindi **più istanze di sottoclassi di Subject**) e quindi si può risalire a chi ha fatto l'aggiornamento.
 - In questo esempio è inutile visto che c'è solo un CS ed è una predisposizione

```
public class Subject {
    private List<Observer> obs = new ArrayList<>();
    private boolean changed = false;

    public void notify(Object state) { //state è Object perchè deve essere molto
        //generale perchè non si conosce lo stato del CS (AddrBook)
        if (!changed) return;
        for (Observer o : obs) o.update(this, state);
        changed = false;
    }

    public void setChanged() {
        changed = true;
    }

    public void attach(Observer o) {
        obs.add(o);
    }

    public void detach(Observer o) {
        obs.remove(o);
    }
}
```

```

}

public class AddrBook extends Subject {
    private List<Persona> nomi = new ArrayList<>();

    public void insert(Persona p) {
        if (nomi.contains(p)) return;
        nomi.add(p);
        setChanged(); // la prossima notifica avverrà
        notify(nomi); // notifica i ConcreteObserver
    }

    public Persona find(String cognome) {
        for (Persona p : nomi)
            if (p.getCognome().equals(cognome)) return p;
        System.out.println("AddrBook.find: NOT found");
        return null;
    }
}

public interface Observer { //modalità push visto che viene passato lo stato a update()
    public void update(Subject s, Object o);
}

public class Store implements Observer {
    @Override
    public void update(Subject s, Object o) {
        List<Persona> l = (List<Persona>) o;
        String nom;
        try (FileWriter f = new FileWriter("nomi.txt")) {
            for (Persona p : l) {
                nom = p.getNome() + "\t" + p.getCognome() + "\t" +
p.getTelefono();
                f.write(nom + "\n");
            }
        } catch (IOException e) { }
    }
}

public class Dialog {
    private static final AddrBook book = new AddrBook();
    private static final Store st = new Store();
    private static final Persona p1 = new Persona("Oliver", "Stone", "012345", "NY");

    public static void main(String[] args) {
        book.attach(st);
        book.insert(p1);
    }
}

```

MAGGIORE FLESSIBILITÀ e FACILITÀ DI MANUTENZIONE del codice.

Conseguenze Osservatori possono essere Aggiunti o Rimossi dinamicamente senza dover modificare il codice del Soggetto.
 Il Soggetto non vede gli Observer, solo la loro interfaccia comune perciò si HA BASSO ACCOPPIAMENTO TRA I COMPONENTI DEL SISTEMA.

- C'è una completa indipendenza fra *ConcreteSubject* e *Subject*
- Codici più semplici da riusare e modificare
- La notifica avviene in automatico a tutte le varie istanze di *ConcreteObserver*

- Se `update()` avviene **troppo spesso**, si può modificare la **tempistica di esecuzione**
- Si possono avere **molti Subject e pochi Observer**: i Subject devono avere una lista di Observer e quindi viene spesso duplicata. Si può avere un'unica tabella che fa i dovuti riferimenti
- Quando si chiama `update()` si deve conoscere il CS e per questo si passa un riferimento al Subject
- Subject chiama `notify()` dopo un cambiamento, oppure *aspetta un certo numero di cambiamenti*, in modo da evitare continue notifiche agli Observer
- Il CO potrebbe modificare lo stato del CS e visto che deve essere sempre **consistente** allora può usare `setState()` per farlo passando come **parametro** lo stato aggiornato. Il CS decide se prendere tale parametro e **modifica lo stato**. Con `notify()`, in seguito, *aggiorna tutti gli altri Observer (compreso chi gliel'ha passato)*
- Se il CO ha un attributo CS può invocare `setState()` e `getState()` in qualsiasi momento e **se si volesse eliminare l'istanza di CS**, essa non può essere deallocata perchè c'è almeno un CO che ne tiene un riferimento.
 - Se nessun oggetto viene puntato da qualcuno, viene automaticamente eliminata da Java.
 - Si deve eliminare da CO il riferimento di Subject e quindi **aggiungere un metodo che lo faccia diventare NULL** e così l'istanza di CS può essere deallocata in maniera automatica
- Il CS tiene uno stato partizionabile in molte altre cose. `update()` potrebbe non passare lo stato (quindi usando poi `pull` per farsi dare lo stato) oppure si potrebbe usare: CO, quando usa `attach()` (?) passa anche come parametro lo stato che gli interessa.