Template method

Comportamentale

Prof. A. M. Calvagna

GOF PATTERN CATALOG

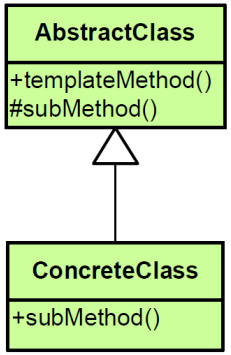
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory(87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Motivazione: application frameworks

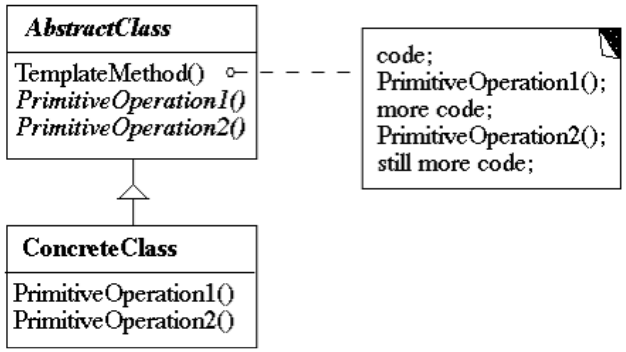
```
abstract Class MyApplication {
...
        void OpenDocument (String  name ) {
                if (!CanNotOpenDocument (name)) return;
                Document doc = DoCreateDocument();

                if (doc) {
                        docs->AddDocument( doc);
                        AboutToOpenDocument( doc);
                        Doc->Open();
                        Doc->DoRead();
                }
        }
...

}
```


Template method

- Intento:
  - Permettere la codifica dell'algoritmo di una certa operazione, delegando l'implementazione di alcuni suoi passi alle sottoclassi
  - Attraverso le sottoclassi posso cambiare implementazione a certi passi di un algorimo senza però cambiarne la struttura complessiva



Struttura



Polimorfismo

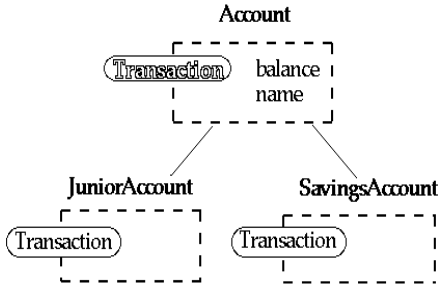
```
class Account {
        String name;
        float balance;
        Account(String customerName, float InitialDeposit ) {
                name = customerName;
                balance= InitialDeposit;
        };
        public void Transaction(float amount){ balance += amount;}
}
                class JuniorAccount  extends Account {
                        public void Transaction(float amount) {// put code here}
                }

                class SavingsAccount extends Account {
                        public void Transaction(float amount) {// put code here}
                }
```


Dal lato del Client…

```
Account createNewAccount(){
        // code to query customer and determine what type of
        // account to create
};

void main(...) {
        Account customer;
        customer = createNewAccount();
        customer->Transaction(amount);
}
```

l'implementazione di Transaction in Account è sostituita da quella della sottoclasse scelta: ho usato solo il polimorfismo

Metodi «differiti»

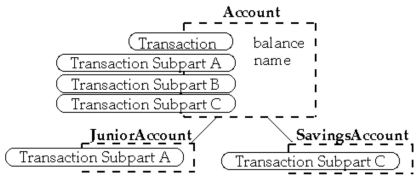



```
abstract class Account {
        public abstract void Transaction();
}


class JuniorAccount  extends Account {
        public void Transaction() { //put code
here}
}


class SavingsAccount  extends Account {
        public void Transaction() { //put code
here}
}
```

Template method

```
class Account {
    public void TransactionSubpartA(){};
    public void TransactionSubpartB(){};
    public void TransactionSubpartC(){};

     Public void Transaction(float amount) {
          TransactionSubpartA();
          TransactionSubpartB();
          TransactionSubpartC();
          // EvenMoreCode;
}
          ...nel client....
          Account customer;
          customer = createNewAccount();
          customer->Transaction(amount);
```

```
class JuniorAccount extends Account {
        public void TransactionSubpartA() { //
code};
}

class SavingsAccount extends Account {
        public void TransactionSubpartC(){//
code};
}
```



In Java (>8)

- Default methods nelle interfacce

```
public interface algoritmTemplate {

    void partA();
    void partB();
    void partC();

    default void algorithm(){
        partA();
        System.out.print("invariable part");
        partB();
        partC();
    }
}
```



```
class MyWayAlgorithm implements algTemplate {

   //mandatory
    public void partA(){
         System.out.print("my partA\n");
     }

     @Override
     public void partB() {
          System.out.print("my partB\n");
     }
     @Override
     public void partC() {
         System.out.print("my partC\n");
     }

    public static void main (String[] args){
            new PartA().algorithm();
    }
}
```


Applicabilità

- Applica questo pattern….
  - Per implementare le parti invarianti di una algoritmo una sola volta per tutte:
    - Le sottoclassi implementano le parti variabili
  - Per fattorizzare in una sola classe dei comportamenti comuni a varie sottoclassi ed eliminare la loro duplicazione
  - Per avere controllo di cosa sia modificabile in una classe attraverso la derivazione:
    - Passi dei metodi template uniche parti estendibili dalle sottoclassi

Esempio di utilizzo in Java: paint() in AWT

public void paint(Graphics g) definito in *java.awt.Component*

Il metodo Java paint è una operazione primitiva che devo implementare ma viene chiamata da un metodo della sua classe padre, mai direttamente da me *(callback function)*

```
class HelloApplication extends Frame {
    public void paint( Graphics display ){
        int startX = 30;
        int startY = 40;
        display.drawString( "Hello World", startX, startY );
    }
}
```

Si inverte la struttura di controllo: È la classe padre che si ritrova ad eseguire metodi della sottoclasse

paint() in swing

- javax.swing.JComponent implementa il metodo paint dividendolo in tre metodi separati, invocati nell'ordine seguente:
  - protected void paintComponent(Graphics g)
  - protected void paintBorder(Graphics g)
  - protected void paintChildren(Graphics g)

– Metto il mio codice in paintComponent() e lo invoca paint():

```
public void paintComponent(Graphics g) {
    g.drawString("This is my custom Panel!",10,20);
    redSquare.paintSquare(g);
}
```


Refactoring per Template method

- 1) scrivo tutto il codice in un unico grande metodo che diverrà il template
- 2) Lo divido in passi successivi usando ad es. i commenti
- 3) Incapsulo ogni passo in un metodo separato
- 4) Riscrivo il template invocando i metodi estratti
- Ripeto da 1) su ognuno dei metodi estratti, finchè:
  - Tutti i metodi primitivi (ridefinibili) hanno la stessa granularità (dichiararli **protected**)
  - Tutte le parti costanti sono incapsulate nei rispettivi metodi (dichiararli **private**)
- dichiaro il template method come **final** affinché sia inalterabile dalle sottoclassi

Caso particolare: sostituzione di valori costanti

- Applicazione molto comune del template
- Non hanno decisioni: tornano sempre lo stesso valore
- Supponiamo di avere una *lazy inizialization*:

```
public class Foo {
    private Bar field;
    public final Bar getField() {
        if (field == null) field = new Bar(10);
        return field;
    }
}
```

- Potrebbe anche essere un metodo costruttore
- Come cambiare il valore (oggetto) di default a cui è inizializzato il campo field, senza esporlo (a qualsiasi valore) come parametro?

Sostituzione di valori costanti

- Applico il template pattern al metodo:

```
public final Bar getField({
        if (field == null) field = defaultField();
        return field;
}

protected Bar defaultField() {
        return new Bar(10);
}

public clientCode() {
        field := getField();
}
```

- Il cambio risulta trasparente al codice dei client
- Ora le sottoclassi possono cambiare il valore/oggetto restituito
- Opera come un factory method, ma con l'intento di rendere alterabile funzionalità stessa, il comportamento eseguito (e osservato nei client), non solo la sua implementazione