InterBit
Training & Consulting Ltd.

Template Pattern

Sun microsystems | AUTHORIZED MySQL EDUCATION CENTER | solaris | FORTINET | symantec.

Chapter Content

- Template Pattern Overview
- AbstractList
- Template victory check
- Constructors
- Discussion
- Exercise

Template Pattern Overview

- Allows base-class to define the general outline of an algorithm, while leaving some of the implementation-details to subclasses.
- Relies on calling abstract methods.
- Quite common.
  - Saves code duplication.
  - Allows one to design the algorithm once, and use sub-classes for fine-tuning.

AbstractList

```
public abstract class AbstractList {

    public abstract ListIterator listIterator(); // Template Method

    public int indexOf(Object obj) {
        for(ListIterator iter=listIterator(); iter.hasNext();) {
            if(obj.equals(iter.next()) return iter.previousIndex();
        }
        return -1;
    }
    ...
}
public class MyList extends AbstractList {
    private Object[] data;
    private int nextIndex;
    public MyList(int size){
        data=new Object[size];
    }
    public ListIterator listIterator(){
        return new MyIterator();
    }
```

AbstractList (cont.)

```
     public boolean add(Object obj){
      if(nextIndex == data.length)  return false;
      data[nextIndex++]=obj;
      return true;
     }
     ...

     // inner class representing iterator
     class MyIterator implements ListIterator {
      private int ind =0;
      public Object next(){
          return data[ind++];  // TODO: check array bounds
      }
      public int previousIndex(){
          return ind-1;
      }
      public boolean hasNext(){
          return ind<data.length;
      }
      ...
     }
}
```

Template victory check

- Re-writing our Tic-Tac-Toe game, using template method rather than strategy:

```
abstract class TicTacToeGame {
    JFrame gameFrame;
    char[] board;
    Player[] players;
    ...
    abstract boolean isVictorious(Player p); // Template Method
    void playerOccupiedSqure(Player p, int row, int col){
        . . . // Graphically mark the move on frame
        if (isVictorious(p))
            . . . // Stop game, announcing victory
    }
}
```

Game manager relies on template method isVictorisous() implementation in sub-classes, not knowing or caring what it is.

Template victory check (cont.)

```
class DefaultGame extends TicTacToeGame {
    boolean isVictorious(Player p) {
        . . . // look for rows / columns / diagonals
    }
}

class NoDiagGame extends TicTacToeGame {
    boolean isVictorious(Player p) {
        . . . // look for rows / columns, but no diagonals
    }
}
```

Subclasses provide different implementations for isVictorisous()

Constructors

- What's wrong with a template constructor?

```
abstract class Game {
    protected Game() {
        LoadGameData(); // call to template method
        loadPlayersData(); ...
    }
    protected abstract void loadGameData(); // define template method
}
class GraphicGame extends Game {
    private File imgDir; // dir where image files are located
    public GraphicGame(File imaDir) {
        // super();
        this.imgDir = imgDir;
    }
    protected void loadGameData() { // template method implementation
        ...  // load images from imgDir
    }
```

Problems and implementation

Concrete base class

- It is not necessary to have the superclass as a abstract class. It can be a concrete class containing a method (template method) and some default functionality
- In this case the primitive methods can not be abstract and this is a flaw because it is not so clear which methods have to be overridden and which not
- A concrete base class should be used only when ***customizations hooks*** are implemented.

Problems and implementation

Template method can not be overridden

- The template method implemented by the base class should not be overridden
- The specific programming language modifiers should be used to ensure this.
  - e.g. final

Problems and implementation

Customization Hooks

- A particular case of the template method pattern is represented by the **hooks**
- The hooks are generally empty methods that are called in superclass (and does nothing because are empty), but can be implemented in subclasses.
- Customization Hooks can be considered a particular case of the template method as well as a totally different mechanism.

Problems and implementation

Customization Hooks

- Usually a subclass can have a method extended by overriding id and calling the parent method explicitly
- Code in a sub class:

```
class Subclass extends Superclass {
    …
   @Override
    public void doSomething() {
        // some customization code to extend functionality
        super.doSomething ();
        // some customization code to extend functionality
    }
}
```

Problems and implementation

Customization Hooks

- Instead of overriding - some hook methods can be added. Only the hooks should be implemented in sub-classes:

```
public class Superclass {

    protected void preSomethingHook(){};
    protected void postSomethingHook(){};

    protected void doSomething() {
        preSomethingHook();
        // something implementation
        postSomethingHook();
    }
}

public class Subclass extends Superclass {
    protected void preSomethingHook(){
        // customization code
    }
    protected void postSomethingHook(){
        // customization code
    }
}
```

Problems and implementation

Naming Conventions

- In order to identify the primitive methods it's better to use a specific naming convention.
- For example the prefix “do” can be used for primitive methods.
- In a similar way the customizations hooks can have prefixes like “pre” and “post”.

Problems and implementation

When methods should be abstract?

- When there is a method in the base class that should contain some default code, but on the other hand must be extended in the subclasses
- it should be split in two: one abstract method and one concrete.
- We can not rely on the fact that the subclasses will override the method and developers will remember it:

```
void doSomething() {
    super.doSomething(); // this is forgetable
}
```

Template Method Summary

- Template method is using an inverted controls structure, sometimes referred as “the Hollywood principle”
- From the superclass point of view: “**Don't call us, we'll call you**”
- This refers to the fact that instead of calling the methods from base class in the subclasses, the methods from subclass are called in the template method from superclass.

Template Method Summary

- Due to the above fact a special care should be paid to the access modifiers:
- The template method should be implemented only in the base class, and the primitive method should be implemented in the subclasses.
- A particular case of the template method is represented by the customization hooks