CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 4 Design Pattern Intro

Contents

Template Method..........................................................................................2
Introduction .................................................................................................2
Template Method- The Pattern .................................................................5
Intent ...........................................................................................................5
Motivation..................................................................................................5
Applicability ..............................................................................................6
Structure ...................................................................................................7
Consequences ...........................................................................................8
Implementation ........................................................................................10
Implementing a Template Method .........................................................11
Sample Code ...........................................................................................12
Exercises ....................................................................................................14

References

http://c2.com/cgi/wiki?TemplateMethodPattern WikiWiki comments on
the Template Method

http://wiki.cs.uiuc.edu/PatternStories/TemplateMethodPattern Stories
about the Template Method

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf,
Addison Wesley, 1998, pp. 355-369

Design Patterns: Elements of Reusable Object-Oriented Software,
Gamma, Helm, Johnson, Vlissides, 1995, pp. 325-330

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San
Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.
Template Method
Introduction
Polymorphism

class Account {
public:
    void virtual Transaction(float amount)
    { balance += amount; }
    Account(char* customerName, float InitialDeposit = 0);
protected:
    char* name;
    float balance;
};

class JuniorAccount : public Account {
    public: void Transaction(float amount) { // put code here}
};

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

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

main() {
    Account* customer;
    customer = createNewAccount();
    customer->Transaction(amount);
}
class Account {
    public:
        void virtual Transaction() = 0;
}

class JuniorAccount : public Account {
    public
        void Transaction() { put code here}
}
class Account {
    public:
        void Transaction(float amount);
        void virtual TransactionSubpartA();
        void virtual TransactionSubpartB();
        void virtual TransactionSubpartC();
    }

void Account::Transaction(float amount) {
    TransactionSubpartA(); TransactionSubpartB();
    TransactionSubpartC(); // EvenMoreCode;
}

class JuniorAccount : public Account {
    public: void virtual TransactionSubpartA(); }

class SavingsAccount : public Account {
    public: void virtual TransactionSubpartC(); }

Account* customer;
customer = createNewAccount();
customer->Transaction(amount);
**Template Method- The Pattern**

**Intent**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure

**Motivation**

An application framework with Application and Document classes

Abstract Application class defines the algorithm for opening and reading a document

```cpp
void Application::OpenDocument (const char* name ) {  
    if (!CanNotOpenDocument (name)) {
        return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument( doc);
        AboutToOpenDocument( doc);
        Doc->Open();
        Doc->DoRead();
    }
}
```
Applicability

Template Method pattern should be used:

- To implement the invariant parts of an algorithm once.
  
  Subclasses implement behavior that can vary

- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication

To control subclass extensions

  Template method defines hook operations

  Subclasses can only extend these hook operations
Structure

**AbstractClass**
- TemplateMethod()
  - PrimitiveOperation1()
  - PrimitiveOperation2()

**ConcreteClass**
- PrimitiveOperation1()
- PrimitiveOperation2()

```java
code;
PrimitiveOperation1();
more code;
PrimitiveOperation2();
still more code;
```

Participants

- **AbstractClass**
  Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm
  Implements a template method defining the skeleton of an algorithm

- **ConcreteClass**
  Implements the primitive operations
  Different subclasses can implement algorithm details differently
Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

   Parent class calls subclass methods

   Java's paint method is a primitive operation called by a parent method

   Beginning Java programs don't understand how the following works:

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

Template methods tend to call:
• Concrete operations
• Concrete AbstractClass operations
• Primitive operations - must be overridden
• Factory methods
• Hook operations

Methods called in Template method and have default implementation in AbstractClass

Provide default behavior that subclasses can extend

Smalltalk's printOn: aStream is a hook operation

It is important to denote which methods
• Must overridden
• Can be overridden
• Can not be overridden
Implementation

Using C++ access control

  Primitive operations can be made protected so can only be called by subclasses

  Template methods should not be overridden - make nonvirtual

Minimize primitive operations

Naming conventions

  Some frameworks indicate primitive methods with special prefixes

  MacApp use the prefix "Do"
Implementing a Template Method

- Simple implementation
  Implement all of the code in one method
  The large method you get will become the template method

- Break into steps
  Use comments to break the method into logical steps
  One comment per step

- Make step methods
  Implement separate method for each of the steps

- Call the step methods
  Rewrite the template method to call the step methods

- Repeat above steps
  Repeat the above steps on each of the step methods
  Continue until:
    - All steps in each method are at the same level of generality
    - All constants are factored into their own methods

---

1 See Design Patterns Smalltalk Companion pp. 363-364. Also see Reusability Through Self-Encapsulation, Ken Auer, Pattern Languages of Programming Design, 1995, pp. 505-516
Sample Code

Template method is common in lazy initialization

```java
public class Foo {
    Bar field;

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

What happens when subclass needs to change the default field value?

```java
public Bar getField() {
    if (field == null)
        field = defaultField();
    return field;
}
protected Bar defaultField() {
    return new Bar(10);
}
```

Now a subclass can just override defaultField()
The same idea works in constructors

    public Foo() {
        field := defaultField();
    }

Now a subclass can change the default value of a field by overriding the default value method for that field
Exercises

1. Find the template method in the Java class hierarchy of Frame that calls the paint(Graphics display) method.

3. Find other examples of the template method in Java or Smalltalk.

4. When I did problem one, my IDE did not help much. How useful was your IDE/tools? Does this mean imply that the use of the template method should be a function of tools available in a language?

5. Much of the presentation in this document follows very closely to the presentation in Design Patterns: Elements of Reusable Object-Oriented Software. This seems like a waste of lecture time (and perhaps a violation of copyright laws). How would you suggest covering patterns in class?

6. In Doc 4 slide 20 exercise 1 you were asked to select a project to examine. Find possible uses of the template method in the project. How do the sample uses effect the project?

7. In your project or other assignments this semester, use the method described in the slide "Implementing a Template Method" to implement a template method.