CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 4 Template Method & Null Object

Contents

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

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

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998,pp. 355-370

Refactoring: Improving the Design of Existing Code, Fowler, 1999, pp. 260-266

"Null Object", Woolf, in *Pattern Languages of Program Design 3*, Edited by Martin, Riehle, Buschmmann, Addison-Wesley, 1998, pp. 5-18

Reading

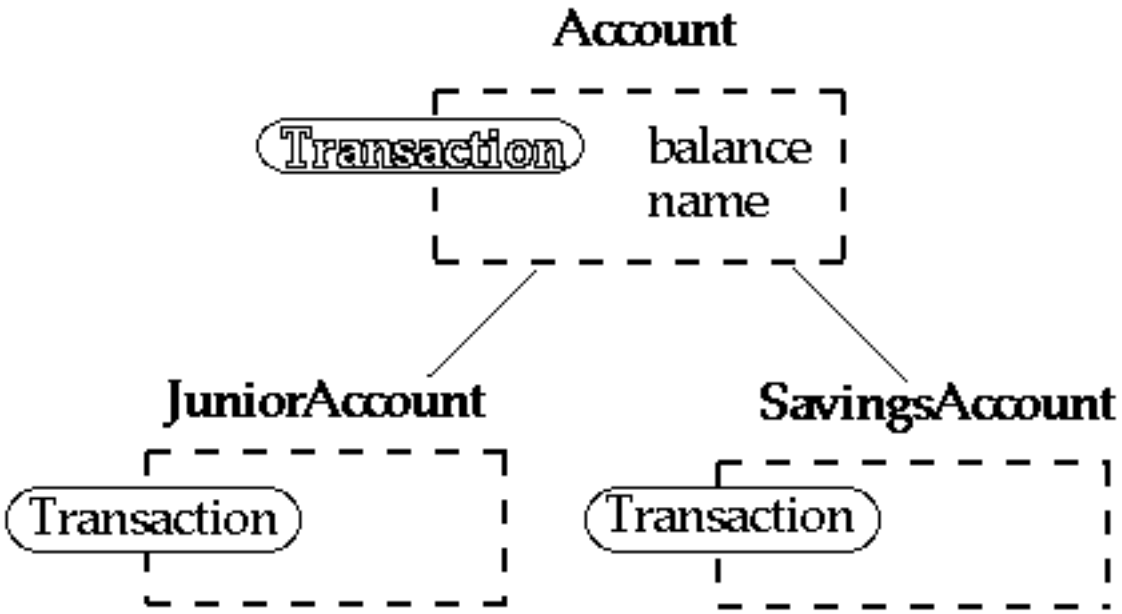
Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 325-330

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998,pp. 355-370

Template Method
Introduction
Polymorphism

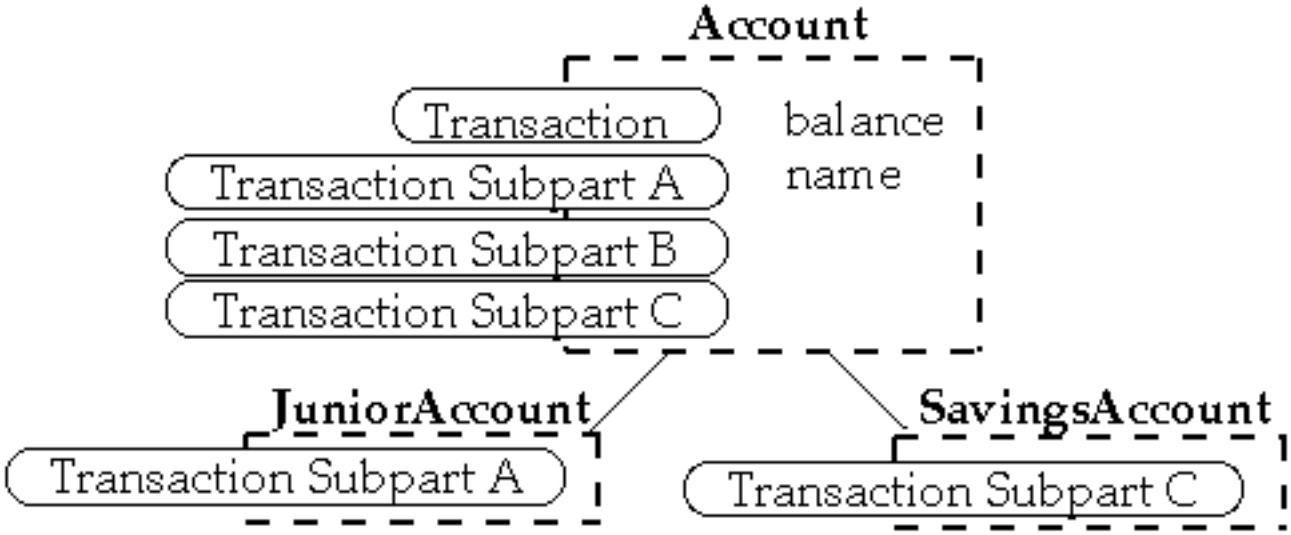
```
class Account {
   public:
      void virtual Transaction(float amount)
            { balance += amount;}
      Account(char* customerName, float InitialDeposit = 0);
   protected:
      char* name;
      float balance;
}

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);
}
```

Deferred Methods




```
class Account {
  public:
    void virtual Transaction() = 0;
}

class JuniorAccount : public Account {
  public
    void Transaction() { put code here}
}
```

Template Methods




```
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

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

  Document* doc = DoCreateDocument();

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

Smalltalk Examples
PrintString

```
Object>>printString
   | aStream |
   aStream := WriteStream on: (String new: 16).
   self printOn: aStream.
   ^aStream contents

Object>>printOn: aStream
   | title |
   title := self class printString.
   aStream nextPutAll:
         ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
   aStream nextPutAll: title
```

Object provides a default implementation of printOn:

Subclasses just override printOn:

Collections & Enumeration

Standard collection iterators
collect:, detect:, do:, inject:into:, reject:, select:

```
Collection>>collect: aBlock
    | newCollection |
    newCollection := self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ^newCollection

Collection>>do: aBlock
    self subclassResponsibility

Collection>>inject: thisValue into: binaryBlock
    | nextValue |
    nextValue := thisValue.
    self do: [:each | nextValue := binaryBlock value: nextValue value: each].
    ^nextValue

Collection>>reject: aBlock
    ^self select: [:element | (aBlock value: element) == false]

Collection>>select: aBlock
    | newCollection |
    newCollection := self species new.
    self do: [:each | (aBlock value: each) ifTrue: [newCollection add: each]].
    ^newCollection
```

Subclasses only have to implement:
species, do:, add:

Applicability

Template Method pattern should be used:

- To implement the invariant parts of an algorithm once.

  Subclasses implement behavior that can vary

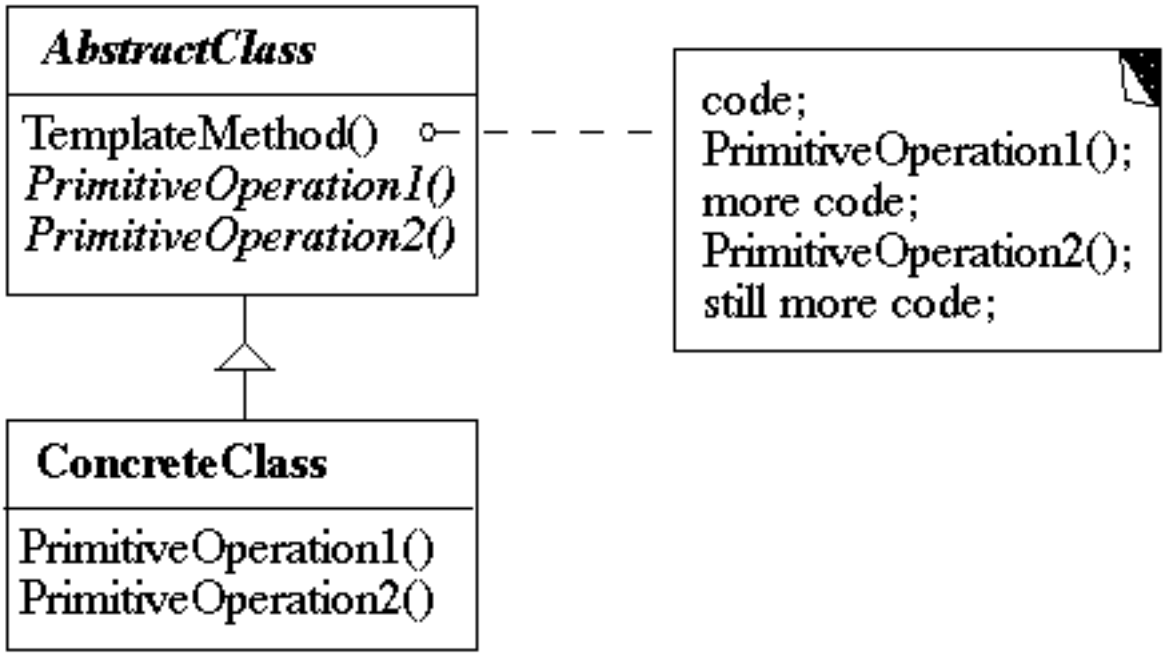
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication

To control subclass extensions

Template method defines hook operations

Subclasses can only extend these hook operations

Structure



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
- 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[1]

- 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

Constant Methods

Template method is common in lazy initialization[2]

```
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?

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

Now a subclass can just override defaultField()

[2] See http://www.eli.sdsu.edu/courses/spring01/cs683/notes/coding/coding.html#Heading19 or Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997 pp. 85-86

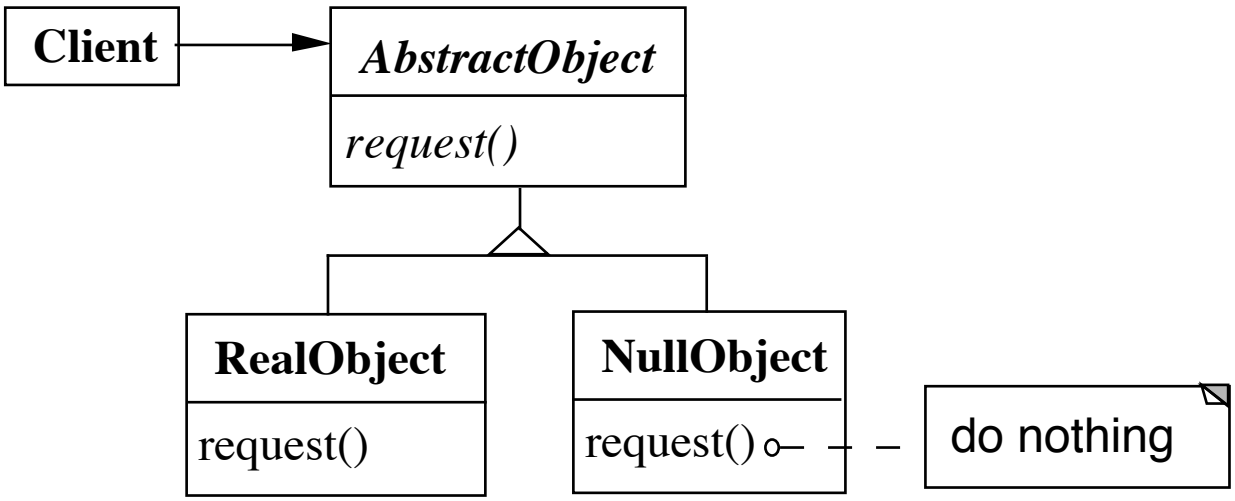
The same idea works in constructors

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

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

NullObject

Structure



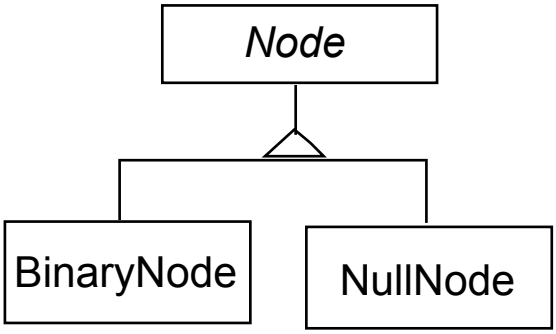
NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing

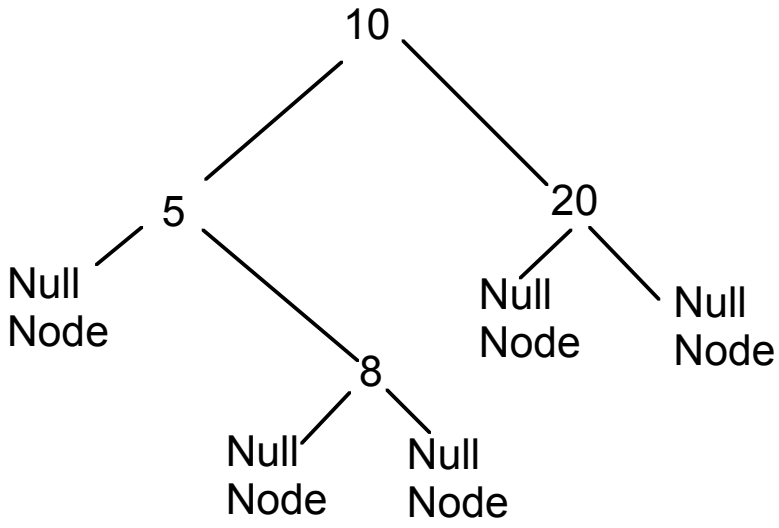
Binary Search Tree Example
Without Null Object

```
public class BinaryNode {
   Node left = new NullNode();
   Node right = new NullNode();
   int key;

   public boolean includes( int value ) {
      if (key == value)
         return true;
      else if ((value < key) & left == null) )
         return false;
      else if (value < key)
         return left.includes( value );
      else if (right == null)
         return false;
      else
         return right.includes(value);
   }
etc.
}
```

Binary Search Tree Example
Class Structure



Object Structure



Searching for a Key

```
public class BinaryNode extends Node {
   Node left = new NullNode();
   Node right = new NullNode();
   int key;

   public boolean includes( int value ) {
      if (key == value)
         return true;
      else if (value < key )
         return left.includes( value );
      else
         return right.includes(value);
   }
etc.
}

public class NullNode extends Node {
   public boolean includes( int value ) {
      return false;
   }
etc.
}
```

Comments on Example

- BinaryNode always has two subtrees

  No need check if left, right are null

- Since NullNode has no state just need one instance

  Use singleton pattern for the one instance

- Access to NullNode is usually restricted to BinaryNode

  Forces indicate that one may not want to use the Null Object pattern

  However, familiarity with trees makes it easy to explain the pattern

- Implementing an add method in NullNode

  Requires reference to parent or

  Use proxy

Refactoring
Introduce Null Object[3]

You have repeated checks for a null value

Replace the null value with a null object

Example

```
customer isNil
   ifTrue: [plan := BillingPlan basic]
   ifFalse: [plan := customer plan]
```

becomes:

- Create NullCustomer subclass of Customer with:

```
NullCustomer>>plan
   ^BillingPlan basic
```

- Make sure that each customer variable has either a real customer or a NullCustomer

Now the code is:

```
plan := customer plan
```

- Often one makes a Null Object a singleton

[3] Refactoring Text, pp. 260-266

Applicability

Use the Null Object pattern when:

- Some collaborator instances should do nothing
- You want clients to ignore the difference between a collaborator that does something and one that does nothing

  Client does not have to explicitly check for null or some other special value

- You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way

Use a variable containing null or some other special value instead of the Null Object pattern when:

- Very little code actually uses the variable directly
- The code that does use the variable is well encapsulated - at least in one class
- The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences
Advantages

- Uses polymorphic classes
- Simplifies client code
- Encapsulates do nothing behavior
- Makes do nothing behavior reusable

Disadvantages

- Forces encapsulation

  Makes it difficult to distribute or mix into the behavior of several collaborating objects

- May cause class explosion

- Forces uniformity

  Different clients may have different idea of what “do nothing” means

- Is non-mutable

  NullObject objects cannot transform themselves into a RealObject

  become: message in Smalltalk allows null objects to “transform” themselves into real objects

Implementation

- Too Many classes

  Eliminate one class by making NullObject a subclass of RealObject

- Multiple Do-nothing meanings

  If different clients expect do nothing to mean different things use Adapter pattern to provide different do-nothing behavior to NullObject

- Transformation to RealObject

  In some cases a message to NullObject should transform it to a real object

  Use the proxy pattern

Generalized Null Object Pattern

A generalized Null Object pattern based on Objective-C with an implementation in Smalltalk can be found at:

http://www.smalltalkchronicles.net/edition2-1/null_object_pattern.htm

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?