CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2002
Doc 6 Singleton & Template Method
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. 127-134, 325-330

Pattern-Oriented Software Architecture: A System of Patterns (POSA 1), Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996,

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998,pp. 91-101, 355-370

Singleton
Intent

Insure a class only has one instance, and provide a global point of access to it

Motivation

There are times when a class can only have one instance

Applicability

Use the Singleton pattern when

- There must be only one instance of a class, and it must be accessible to clients from a well-known access point
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Examples of Using a Singleton

Java Security manager

All parts of a program must access the same security manager

Once set a security manager cannot be changed in a program

Logging the activity of a server

All parts of the server should use the same instance of the logging system

The server should not be able to change the instance of the logging system was it has been set

Null Object

If Null object does not have state, only need one instance

Implementation
Java

```
// Only one object of this class can be created
class Singleton
   {
   private static Singleton _instance = null;

   private Singleton()      { fill in the blank }

   public static Singleton getInstance()
      {
      if (  _instance == null )
          _instance = new Singleton();
      return _instance;
      }
   public void otherOperations() { blank; }
   }

class Program
   {
   public void aMethod()
      {
      X = Singleton.getInstance();
      }
   }
```

Java Singletons, Classes, Garbage Collection

Classes can be garbage collected in Java

Only happens when there are
- No references to instances of the class
- No references to the class

If a singleton's state is modified and its class is garbage collected, its modified state is lost

To avoid having singletons garbage collected:

- Disable class garbage collection with -Xnoclassgc flag

- Insure singleton or class always has a reference

  Store singleton or class in system property

Implementation
C++

```
// Only one object of this class can be created
class Singleton
   {
   private:
      static Singleton* _instance;
      void otherOperations();

   protected:
      Singleton();

   public:
      static Singleton* getInstance();
   }


Singleton*  Singleton::_instance = 0;

Singleton* Singleton::getInstance()
   {
   if (  _instance == 0 )
       _instance = new Singleton;
   return _instance;
   }
```

Implementation
Smalltalk

```
Smalltalk.CS635 defineClass: #SingletonExample
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: ''
   classInstanceVariableNames: 'uniqueInstance '
   imports: ''
   category: 'Lecture notes'!


CS635.SingletonExample class methodsFor: 'instance creation'

current
   uniqueInstance isNil ifTrue:[uniqueInstance := super new].
   ^uniqueInstance

new
   self error: 'Use current to get an instance of Class: ' , self name
```

One could also use a private constant shared variable to store the unique instance

Overriding new in Smalltalk

Since can control what new returns one might be tempted to use:

```
new
   uniqueInstance isNil ifTrue:[uniqueInstance := super new].
   ^uniqueInstance
```

This can be misleading; user might think they are getting different objects when calling new

Do we have two different windows below or not?

```
| left right |
left := SingleWindow new.
Right := SingleWindow new.
left position: 100@ 100.
right position: 500@100.
```

Naming the Access Method

GOF uses: **instance**()

POSA 1 uses: **getInstance**()

Smalltalk uses **default** and **current**

Selecting names is one of the more difficult problems in object-oriented analysis. No name is perfect[1]

[1] Fowler pp. 9, Alpert pp. 98

Singletons and Static

If one needs only one instance of a class why not just implement all methods as static?

- Classes do not inherit Object's protocol
- Hard to modify design if need more that one instance
- Builds bad habits in beginners

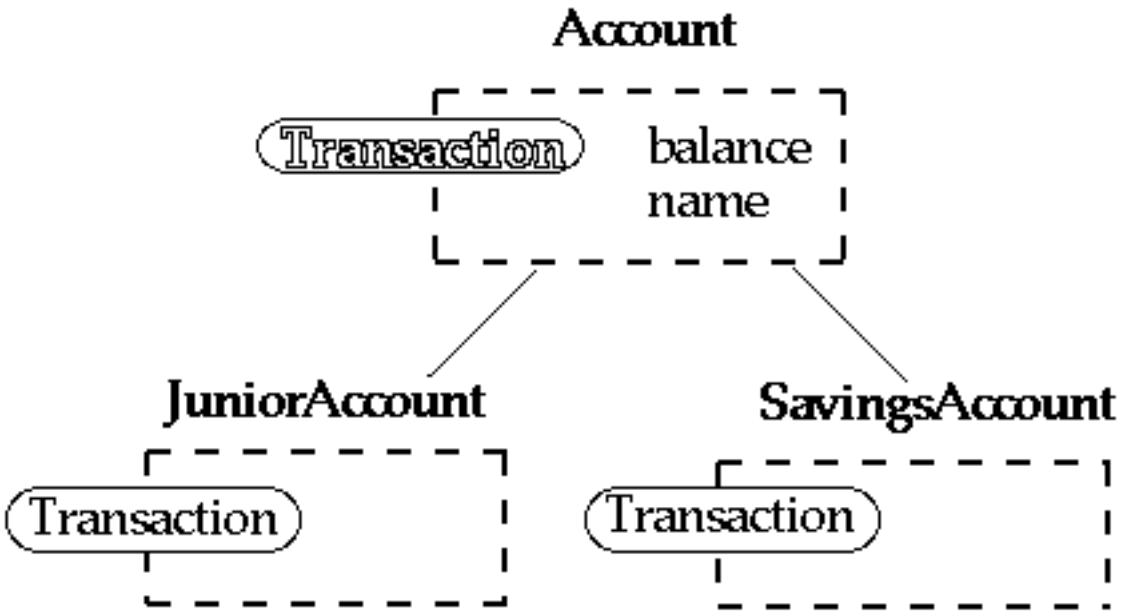
Consequences

- Controlled access to sole instance
- Reduced name space
- Permits subclassing
- Permits a variable number of instances
- More flexible than class operations
- Leads to improper use of globals

Template Method
Introduction
Polymorphism

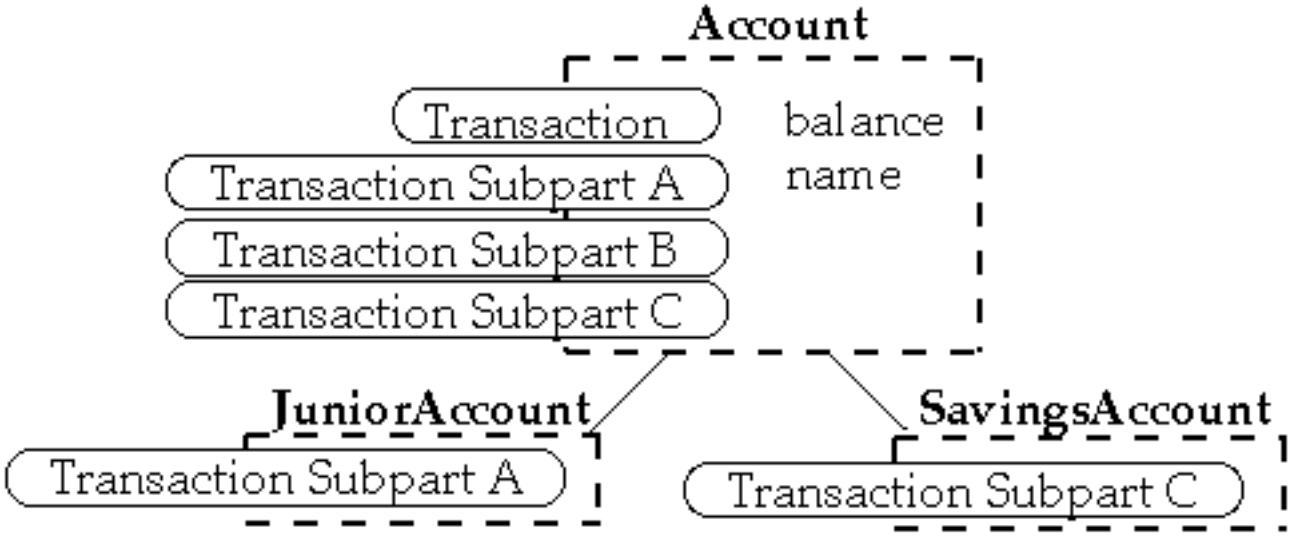
```
class Account {
   public:
      void virtual Transaction(float amount)
            { balance += amount;}
      Account(char* customerName, float InitialDeposit = 0);
   protected:
      char* name;
      float balance;
}

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

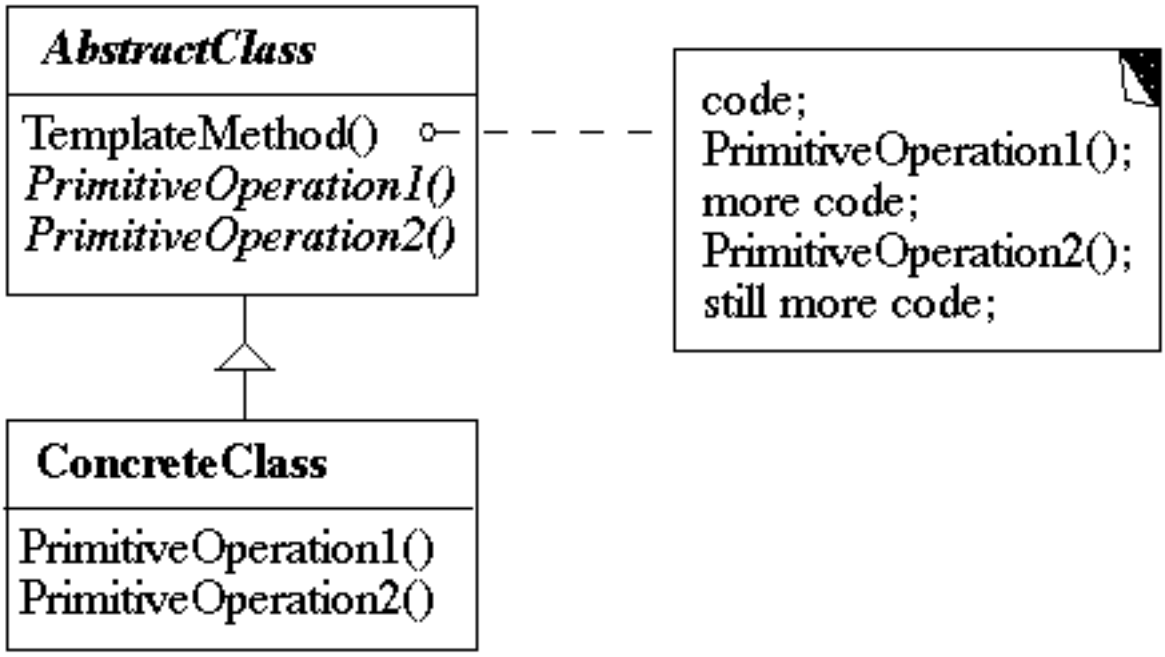
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication

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[2]

- 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

[2] 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[3]

```
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()

[3] 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

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?