# CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2014
Doc 15 Assignment 2, Template Method, Singleton
April 8, 2014

# References


Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 127-134

When is a Singleton not a Singleton, Joshua Fox, January 2001, http://java.sun.com/developer/technicalArticles/Programming/singletons/

http://en.wikipedia.org/wiki/Singleton_pattern

The "Double-Checked Locking is Broken" Declaration, http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

Use your Singletons wisely, http://www.ibm.com/developerworks/webservices/library/co-single.html

Why Singletons are Evil, http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx

Why Singletons Are Controversial, http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial

Photographs used with permission from www.istockphoto.com

# Assignment 2

# Getters, Setters

How many times do does a method

Extract data out of an object

Put data into existing object

Where the object is not "this"

# Iterator

```
Iterator test = heap.iterator();

a = test.next();

b = test.next();
```

What values should a and b have

# Iterator & Types

In Java all iterators are the same type

```
Iterator elements = aCollection.iterator();
```

Don't create your own type

```
public class HeapIterator {

}
```

# NullNode

Why does it have fields:

leftChild
rightChild
value

# Null Object

Is it just replacing

```
if (node == null)
```

with

```
if (node.isNull())
```

# Strategy

```
public class Heap extends AbstractCollection<T> {

    private Strategy strategy;
```

# Strategy

```
public class MaxHeapStrategy<T> implements HeapStrategy<T> {
    public void add(Node<T> node, T value) {
        if (node.isNull()) {
            node.add(value);
            return;
        }
        if (node.compareTo(value) <= 0 )
            value = node.swapNodeValueWith(value);
        }
        if (node.rightHeight() < node.leftHeight() )
            add(node.right(),value);
        } else {
            add(node.left(), value);
        }
    }
```

# See the Difference?

```
public class MinHeapStrategy<T> implements HeapStrategy<T> {
    public void add(Node<T> node, T value) {
        if (node.isNull()) {
            node.add(value);
            return;
        }
        if (node.compareTo(value) >= 0 )
            value = node.swapNodeValueWith(value);
        }
        if (node.rightHeight() < node.leftHeight() )
            add(node.right(),value);
        } else {
            add(node.left(), value);
        }
    }
```

# Do it Once

Avoid repeating logic

# Move common code to parent

```
public class AbstractHeapStrategy<T> implements HeapStrategy<T> {
    public void add(Node<T> node, T value) {
        if (node.isNull()) {
            node.add(value);
            return;
        }
        if (valueHasPriority(node, value) )
            value = node.swapNodeValueWith(value);
        }
        if (node.rightHeight() < node.leftHeight() )
            add(node.right(),value);
        } else {
            add(node.left(), value);
        }

    public abstract boolean valueHasPriority(Node<T> node, T value);
}
```

# Duplicate Code Reduced

```
public class MinHeapStrategy<T> implements AbstractHeapStrategy<T> {
    public boolean valueHasPriority(Node<T> node, T value) {
        return node.compareTo(value) >= 0;
    }
}
```

```
public class MaxHeapStrategy<T> implements AbstractHeapStrategy<T> {
    public boolean valueHasPriority(Node<T> node, T value) {
        return node.compareTo(value) <= 0;
    }
}
```

# Strategy

Some had Strategies that were a page or two per strategy

Strategies were identical except

"<" was replaced with ">"

So make "<", ">" the strategies

# Dilbert Design Philosophy

Never do any work that you can get someone else to do for you

# Adding a value - Heap

```
public class Heap<T> extends AbstractCollection<T> {
    private Node<T> root;
    private Comparator<T> priority;

    public boolean add(T element) {
        if (root == null) {
            root = new HeapNode<T>(element, null);
            return true;
        }
        return root.add(element, priority);
    }
```

# Adding a value - HeapNode

```
public class HeapNode<T> implements Node<T> {
    private E value;
    private Node<T> left;
    private Node<T> right;

    public boolean add(T element, Comparator<T> priority) {
        T lowPriorityElement = element;
        if (priority.compare(value,element) <0 ) {
            lowPriorityElement = value;
            value = element;
        }
        if (right.height() < left.height() )
            return right.add(lowPriorityElement, priority);
        else
            return left.add(lowPriorityElement, priority);
    }
}
```

# Adding a value - NullNode

```
public class NullNode<T> implements Node<T> {
    private Node<T> parent;

    public boolean add(T element, Comparator<T> priority) {
        return parent.addNode( new HeapNode<T>(element);
    }
}
```

# Adding a value - HeapNode

```
public class HeapNode<T> implements Node<T> {

    public boolean addNode(Node<T> newChild) {
        if (right.height() < left.height() )
            right = newChild;
        else
            left = newChild;
        return true;
    }
}
```

# The Strategy

```
// Small values have high priority
public class MinHeap implements Comparator<String> {

    // Return 1 if a has higher priority (smaller) than b
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
}
```

# Getters, Setters

How many times do does a method

Extract data out of an object

Put data into existing object

Where the object is not "this"

# Decorator & Types

Decorator & RealSubject need to be the same type

Need to be able to declare a variable that can hold either

# Why?

```
interface A {
      public String toString();
      public Object[] toArray();
      public int size();
      etc.
}

interface B extends A {
      public String toString();
      public Object[] toArray();
      public int size();
}
```

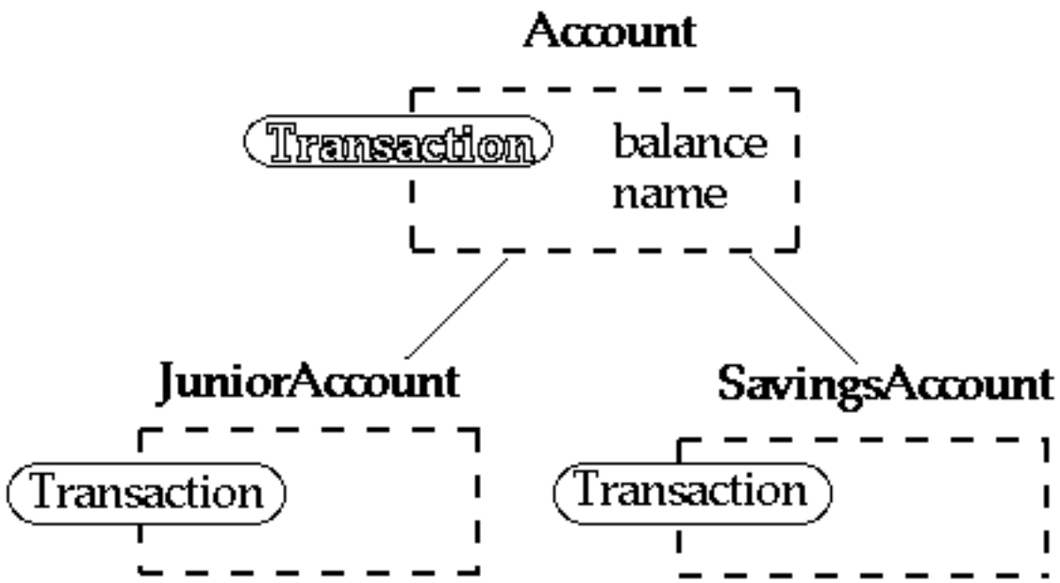
# Template Method

# 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) {//code here}
}

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

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

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

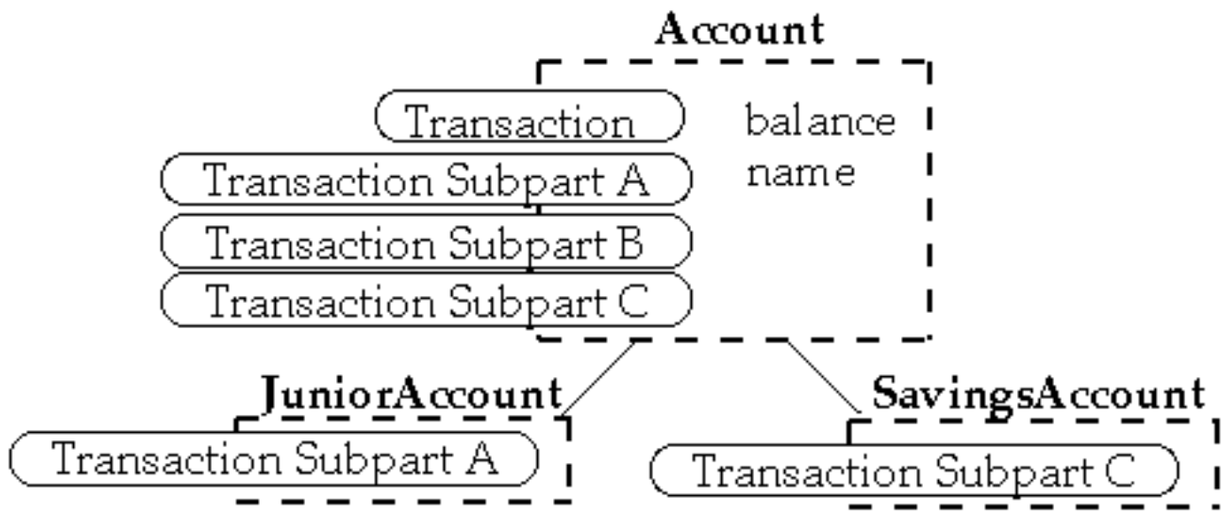
# Deferred Methods




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

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

# Template Method

```
class Account {
        public:
                void Transaction(float amount);
        protected:
                void virtual TransactionSubpartA();
                void virtual TransactionSubpartB();
                void virtual TransactionSubpartC();
}
```




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

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

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

Account*  customer;
customer = createNewAccount();
customer->Transaction(amount);
```

# 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

# Java Example

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

# Ruby LinkedList Example

```
class LinkedList
  include Enumerable

  def [](index)
    Code not shown
  end

  def size
    Code not shown
  end

  def each
     Code not shown
  end

  def push(object)
   Code note shown
  end

end
```

```
def testSelect
  list = LinkedList.new
  list.push(3)
  list.push(2)
  list.push(1)

  a = list.select {|x| x.even?}
  assert(a == [2])
end
```

Where does list.select come from?

# Methods defined in Enumerable

| | | | |
|---|---|---|---|
| all? | any? | collect | detect |
| each_cons | each_slice | each_with_index | entries |
| enum_cons | enum_slice | enum_with_index | find |
| find_all | grep | include? | inject |
| map | max | member? | min |
| partition | reject | select | sort |
| sort_by | to_a | to_set | zip |

All use "each"

Implement "each" and the above will work

# java.util.AbstractCollection

Subclass AbstractCollection

Implement

- iterator
- size
- add

Get

- addAll
- clear
- contains
- containsAll
- isEmpty
- remove
- removeAll
- retainAll
- size
- toArray
- toString

# Consequences

This is the most commonly used of the 23 GoF patterns

Important in class libraries

Inverted control structure

Parent class calls subclass methods

# Consequences

Inverted control structure

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 (abstract) operations
- Factory methods
- Hook operations

Provide default behavior that subclasses can extend

It is important to denote which methods

- Must overridden
- Can be overridden
- Can not be overridden

# Refactoring to 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 methods 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

Design Patterns Smalltalk Companion pp. 363-364.

# Singleton

# Warning

Simplest pattern

- But has subtle issues particularly in Java

Most controversial pattern

# Intent

Ensure a class only has one instance

Provide global point of access to single instance

# Singleton

```
public class Counter {
      private int count = 0;
      private static Counter instance;
      private Counter() { }

      public static Counter instance() {
            if (instance == null)
                  instance = new Counter();
            return instance;
      }

      public int increase()  {return ++count;}
}
```

One instance

Global access





This version does not work correctly all the time. See later slides

# Ruby Singleton

```
class Counter
    private_class_method :new
    @@instance = nil

    def Counter.instance
        @@instance = new unless @@instance
        @@instance
    end

    def increase
        @count = 0 unless @count
        @count = @count + 1
        @count
    end
end
```

```
require 'singleton'

class Counter
   include Singleton

    def increase
        @count = 0 unless @count
        @count = @count + 1
        @count
    end
end
```

# Some Uses

Java Security Manager

Logging a Server

Null Object

# Globals are Evil

# Why Singletons Are Controversial(Evil)

Singletons provide global access point for some service

Hidden dependencies

Is there a different design that does not need singletons

- Pass a reference

# Why Singletons Are Controversial(Evil)

Singletons allow you to limit creation of objects of a class

Should that be the responsibility of the class?

Class should do one thing

Use factory or builder to limit the creation

# Why Singletons Are Controversial(Evil)

Singletons tightly couple you to the exact type of the singleton object

No polymorphism

Hard to subclass

# Why Singletons Are Controversial(Evil)

Singletons carry state with them that last as long as the program lasts

Persistent state makes testing hard and error prone

# Why Singletons Are Controversial(Evil)

A Singleton today is a multiple tomorrow

Singleton pattern makes it hard to change to allow multiple objects

# Why Singletons Are Controversial(Evil)

In Java Singletons are a lie

More on this later

# Singleton Implementation

# Why Not Use This?

```
public class Counter {
        private static int count = 0;

        public static int increase()  {return ++count;}
}
```

# Why Not Use This?

```
public class Counter {
        private int count = 0;
        private Counter() { }

        public static Counter instance = new Counter();

        public int increase()  {return ++count;}
}
```





Very subtle the error here

# Two Useful Features

Lazy

Only created when needed

Thread safe

# Recommended Implementation

```
public class Counter {
      private int count = 0;
      private Counter() { }

      private static class SingletonHolder {
        private final static Counter INSTANCE = new Counter();
    }

      public static Counter instance() {
              return SingletonHolder.INSTANCE;
      }

      public int increase()  {return ++count;}
}
```





Implementation due to Bill Pugh, found in Wikipedia, http://en.wikipedia.org/wiki/Singleton_pattern. It is thread-safe, lazy and works with all know versions of Java

# Correct but not Lazy

```
public class Counter {
        private int count = 0;
        protected Counter() { }

        private final static Counter INSTANCE = new
Counter();

        public static Counter instance() {
                return INSTANCE;
        }

        public int increase()  {return ++count;}
}
```

# Lazy, Thread safe with Overhead

```
public class Counter {
        private int count = 0;
        private static Counter instance;
        private Counter() { }

        public static synchronized Counter instance() {
                if (instance == null)
                        instance = new Counter();
                return instance();
        }

        public int increase()  {return ++count;}
}
```

# Double-Checked Locking does not work

```
public class Counter {
        private int count = 0;
        private static Counter instance;
        private Counter() { }

        public static Counter instance() {
                if (instance == null)
              synchronize(this) {
                    if (instance == null)
                                instance = new Counter();
              }
                 return instance();
        }

        public int increase()  {return ++count;}
}
```





See http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

# Java Templates & Singleton

Does not compile

```
public class TemplateSingleton<Type> {
    Type foo;

    public static TemplateSingleton<Type> instance =
        new TemplateSingleton<Type>();
}
```

# When is a Single Cell not a Single Cell?

# When Java Garbage Collects Classes

Singleton class can be garbage collected
Singleton loses any value it had

Solution

Turn off garbage collection of classes  (-Xnoclassgc)

Make sure there is always a reference to the class/instance

# When Multiple Java Class Loaders are Used

When loaded by two different class loaders there will be two versions of the class

Some servlet engines use different class loader for each servlet

Using custom class loaders can cause this

# Purposely Reloading a Java Class

Servlet engines can force a class to be reloaded

# Serialize and Deserialize Singleton Object

Serialize the singleton
Deserialize the singleton
You now have two copies

One way to serialize a Java object is using ObjectOutputStream

Ruby Marshal.dump() will not marshal a singleton