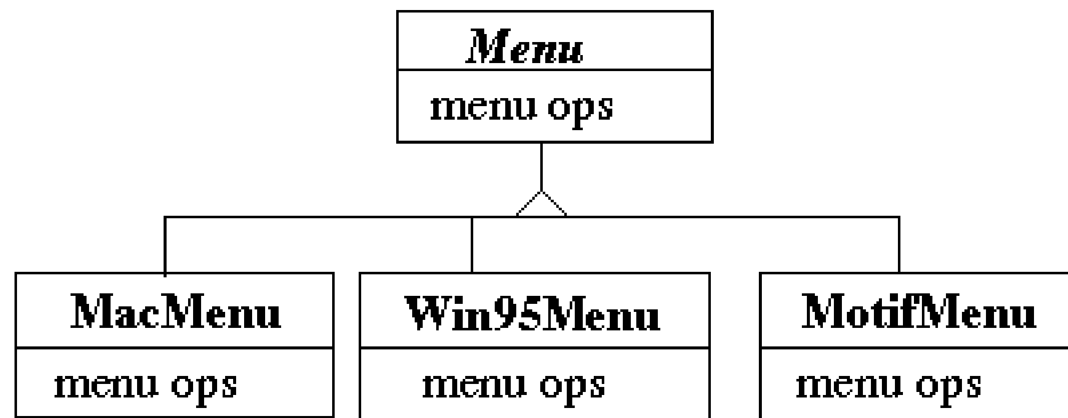
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 12 Abstract Factory, Singleton, Adapter
Oct 6, 2020

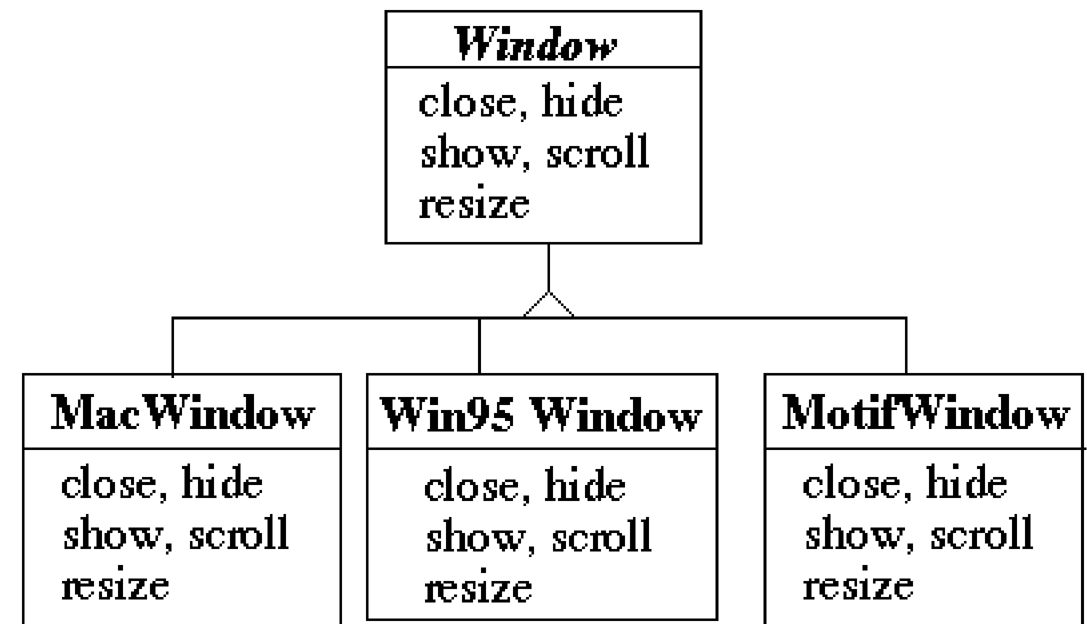
Copyright ©, All rights reserved. 2020 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.

Abstract Factory

Abstract Factory

Write a cross platform window toolkit





Bad Code Dependencies

```
public void installDisneyMenu()
   {
   Menu disney = new MacMenu();
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

Use Abstract Factory

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
    }
```

```
class MacWidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a mac window }

    public Menu createMenu()
        { code to create a mac Menu }

    public Button createButton()
        { code to create a mac button }
}

class Win95WidgetFactory extends WidgetFactory {
    public Window createWindow()
        { code to create a Win95 window }

    public Menu createMenu()
        { code to create a Win95 Menu }

    public Button createButton()
        { code to create a Win95 button }
}
```

Use one Factory per Application

```
public void installDisneyMenu(WidgetFactory myFactory)
   {
   Menu disney = myFactory.createMenu();
   disney.addItem( "Disney World" );
   disney.addItem( "Donald Duck" );
   disney.addItem( "Mickey Mouse" );
   disney.addGrayBar( );
   disney.addItem( "Minnie Mouse" );
   disney.addItem( "Pluto" );
   etc.
   }
```

Abstract Factory

Encapsulate a group of individual factories that have a common theme

Separates the details of implementation of a set of objects
from its general usage

How Do Abstract Factories create Things?

Use Subclass Factory Method

```
abstract class WidgetFactory
   {
   public Window createWindow();
   public Menu createMenu();
   public Button createButton();
   }

class MacWidgetFactory extends WidgetFactory
   {
   public Window createWindow()
      { return new MacWidow() }

   public Menu createMenu()
      { return new MacMenu() }

   public Button createButton()
      { return new MacButton()  }
   }
```

Use Widget Factory Method

```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }

   public Menu createMenu();
      { return menuFactory.createMenu() }

   public Button createButton()
      { return buttonFactory.createMenu() }
}
```

```
class MacWidgetFactory extends WidgetFactory {
   public MacWidgetFactory() {
      windowFactory = new MacWindow();
      menuFactory = new MacMenu();
      buttonFactory = new MacButton();
   }
}

class MacWindow extends Window {
   public Window createWindow() { blah }
   etc.
```

Why Widget Factory Method?

```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }

   public Window createWindow( Rectangle size)
      { return windowFactory.createWindow( size ) }

   public Window createWindow( Rectangle size, String title)
      { return windowFactory.createWindow( size, title) }

   public Window createFancyWindow()
      { return windowFactory.createFancyWindow() }

   public Window createPlainWindow()
      { return windowFactory.createPlainWindow() }
```

Multiple ways to create Widget

Use Prototype

```
class WidgetFactory{
    private Window windowPrototype;
    private Menu menuPrototype;
    private Button buttonPrototype;

    public WidgetFactory( Window windowPrototype,
                          Menu menuPrototype,
                          Button buttonPrototype)
        {
        this.windowPrototype = windowPrototype;
        this.menuPrototype = menuPrototype;
        this.buttonPrototype = buttonPrototype;
        }

    public Window createWindow()
        { return windowPrototype.createWindow() }

    public Window createWindow( Rectangle size)
        { return windowPrototype.createWindow( size ) }

    public Window                ()
        { return menuPrototype.createMenu() }
    etc.
```

How to prevent Cheating?

```
public void installDisneyMenu(WidgetFactory myFactory)
    {
    // We ship next week, I can't get the stupid generic Menu
    // to do the fancy Mac menu stuff
    // Windows version won't ship for 6 months
    // Will fix this later

    MacMenu disney = (MacMenu) myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addMacGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
    }
```

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

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;}
}
```

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;}
}
```

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;}
}
```

Java Templates & Singleton

Does not compile

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

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

When is a Singleton not a Singleton?

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

Adapter

Adapter

Convert interface of a class into another interface

Use adapter when

- You want to use an existing class but does not have interface on needs
- You want to create a reusable class that works with unrelated or unforseen classes

Address Book & JTable

Display an AddressBook object in a JTable

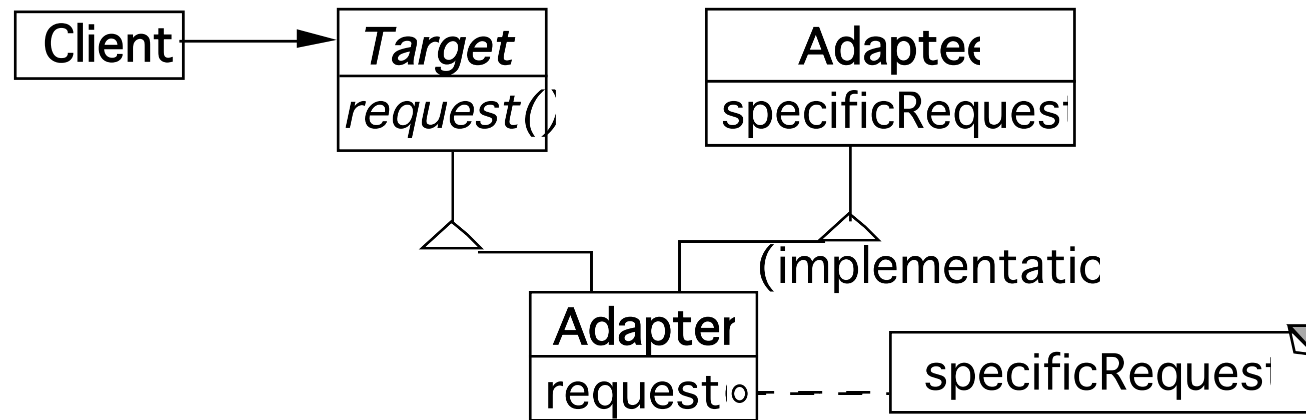
JTables require objects of type TableModel

```
public class AddressBook{
    List personList;
    public int getSize(){...}
    public int addPerson(...){...}
    public Person getPerson(...){...}
    ...
}
```

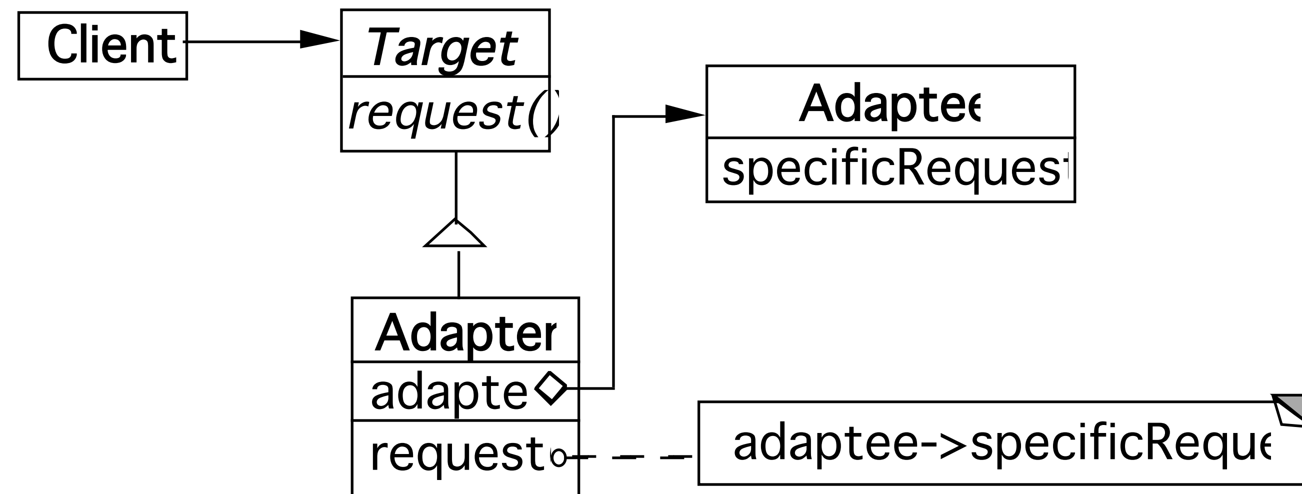
```
public class AddressBookTableAdapter implements TableModel
{
    AddressBook ab;
    public AddressBookTableAdapter( AddressBook ab ){
        this.ab = ab;
    }
    //TableModel impl
    public getRowCount(){
        ab.getSize();

    public Object getValueAt(int rowIndex, int columnIndex) {
        Person requested =
            ad.getPerson(convertRowToName(rowIndex);
    return requested.get(convert(columnIndex));
    }
```

Class Adapter



Object Adapter



Class Adapter Example

```
class OldSquarePeg {
    public: void squarePegOperation()  { do something }
}

class RoundPeg {
    public:  void virtual roundPegOperation = 0;
}

class PegAdapter: private OldSquarePeg,  public RoundPeg  {
    public:
        void virtual roundPegOperation()  {
            add some corners;
            squarePegOperation();
        }
}

void clientMethod() {
    RoundPeg* aPeg = new PegAdapter();
    aPeg->roundPegOperation();
}
```

Object Adapter

```
class OldSquarePeg{
    public: void squarePegOperation() { do something }
    }

class RoundPeg      {
    public: void virtual roundPegOperation = 0;
    }

class PegAdapter:  public RoundPeg       {
    private:
        OldSquarePeg* square;

    public:
        PegAdapter()  { square = new OldSquarePeg; }

        void virtual roundPegOperation()      {
            add some corners;
            square->squarePegOperation();
            }
    }
```

How Much Adapting does the Adapter do?

Two-way Adapters

```
class OldSquarePeg {
   public:
      void virtual squarePegOperation() { blah }
}

class RoundPeg {
   public:
      void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
   public:
      void virtual roundPegOperation() {
         add some corners;
         squarePegOperation();
      }
      void virtual squarePegOperation() {
         add some corners;
         roundPegOperation();
      }
}
```

Flasher and MouseListener

```
class Flasher
   def toggle()
      @flashing = !@flashing
   end

   def pause()
      #etc
   end

   def resume()
      #etc
   end
end
```

```
class MouseListener
   def mouseClicked(event)
   end

   def mouseEntered(event)
   end

   def mouseExited(event)
   end
end
```

Actions we want

mouse click toggles flasher
mouse enter pauses
mouse exits resumes

Flasher as MouseListener

```
class Flasher
   def toggle()
      @flashing = !@flashing
   end

   def pause()
      #etc
   end

   def resume()
      #etc
   end

   def mouseClicked(event)
      toggle()
   end

   def mouseEntered(event)
      pause()
   end

   def mouseExited(event)
      resume()
   end
end
```

Simple Adapter

```
class Flasher
   def toggle()
      @flashing = !@flashing
   end

   def pause()
      #etc
   end

   def resume()
      #etc
   end
end
```

```
class FlasherAdaptor
   def initialize(aFlasher)
      @flasher = aFlasher
   end

   def mouseClicked(event)
      @flasher.toggle()
   end

   def mouseEntered(event)
      @flasher.pause()
   end

   def mouseExited(event)
      @flasher.resume()
   end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
FlasherAdapter.new(yellowFlasher)
```

A Ruby Adapter - Fowardable

```
class Flasher
  def toggle()
    @flashing = !@flashing
  end

  def pause()
    #etc
  end

  def resume()
    #etc
  end
end
```

```
require 'forwardable'

class FlasherMouseListener
  extend Forwardable

  def initialize()
     @flasher = Flasher.new()
   end
  def_delegator(:@flasher, :toggle, :mouseClick)
  def_delegator(:@flasher, :pause, :mouseEnter)
  def_delegator(:@flasher, :resume, :mouseExit)

end


adaptor = FlasherMouseListener.new()
adaptor.mouseClick()
```

Parameterized Adapter

```
class MouseListenerAdapter
   def initialize(adaptee, clickMethod, enterMethod, exitMethod)
      @adaptee = adaptee
      @clickMethod = clickMethod
      @enterMethod = enterMethod
      @exitMethod = exitMethod
   end

   def mouseClicked(event)
      @adaptee.send(clickMethod)
   end

   def mouseEntered(event)
      @adaptee.send(clickMethod)
   end

   def mouseExited(event)
      @adaptee.send(clickMethod)
   end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
                         yellowFlasher,
                         :toggle,
                         :pause,
                         :resume)
```

Better Parameterized Adapter

```
class MouseListenerAdapter
   def initialize(adaptee, clickLambda, enterLambda, exitLambda)
      @adaptee = adaptee
      @clickLambda = clickLambda
      @enterLambda = enterLambda
      @exitLambda = exitLambda
   end

   def mouseClicked(event)
      @clickLambda.call(adaptee)
   end

   def mouseEntered(event)
       @enterLambda.call(adaptee)
   end

   def mouseExited(event)
       @exitLambda.call(adaptee)
   end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
        yellowFlasher,
        lambda {|flasher | flasher.toggle()},
        lambda {|flasher | flasher.pause()},
        lambda {|flasher | flasher.resume()})
```