CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018
Doc 14 Factory Method, Abstract Factory, Effective Java
Oct 30, 2018

Factory Method

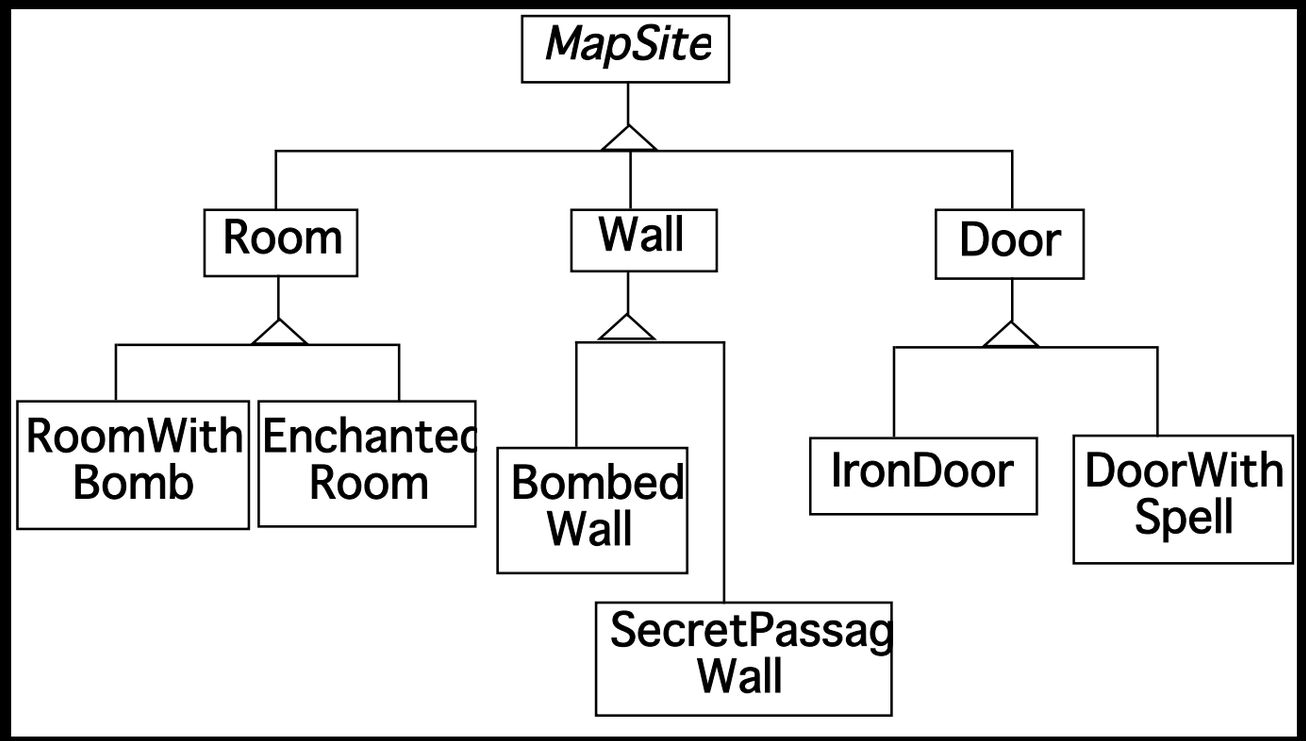
Factory Method

A template method for creating objects

```
public class Example {
   protected Bar bar() { return new Bar(); }

    public void foo() {
      blah
      Bar soap = bar();
      blah;
   }
```

Maze Game Example



Maze Game Example

```
class MazeGame{
   public Maze makeMaze() { return new Maze(); }
   public Room makeRoom(int n ) { return new Room( n ); }
   public Wall makeWall() { return new Wall(); }
   public Door makeDoor() { return new Door(); }

   public Maze CreateMaze(){
      Maze aMaze = makeMaze();
      Room r1 = makeRoom( 1 );
      Room r2 = makeRoom( 2 );
      Door theDoor = makeDoor( r1, r2);

      aMaze.addRoom( r1 );
      aMaze.addRoom( r2 );
      etc

      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {

   public Room makeRoom(int n )  {
      return new RoomWithABomb( n );
   }

   public Wall makeWall() {
      return new BombedWall();
   }
}
```

Don't repeat your self

```
public class LinkedList extends Collection {
  public OrderedLinkedList() {
    this(defaultOrder());
  }

  public LinkedList(Order listOrder ) {
    this(listOrder, new OrderedCollection());
  }

public LinkedList(Collection items) {
  this(defaultOrder(), items);
}

protected Order defaultOrder() {
  return new RandomOrder();
}

public LinkedList(Order listOrder, Collection items) {
  blah
```

Implementation Variation

```
class Hershey {

   public Candy makeChocolateStuff( CandyType id ) {
      if ( id == MarsBars ) return new MarsBars();
      if ( id == M&Ms ) return new M&Ms();
      if ( id == SpecialRich ) return new SpecialRich();

      return new PureChocolate();
   }

class GenericBrand extends Hershey {
   public Candy makeChocolateStuff( CandyType id ) {
      if ( id == M&Ms ) return new Flupps();
      if ( id == Milk ) return new MilkChocolate();
      return super.makeChocolateStuff(id);
   }
}
```

Using C++ Templates

```
template <class ChocolateType>
class Hershey
   {
   public:
      virtual Candy* makeChocolateStuff( );
   }

template <class ChocolateType>
Candy*
Hershey<ChocolateType>::makeChocolateStuff( )
   {
   return new ChocolateType;
   }

Hershey<SpecialRich> theBest;
```

Smalltalk Variant

Return the class, caller creates an object

```
chocolateStuff
      ^SpecialRich
```

```
some code
candy := (self chocolateStuff) new
mode code
```

Use Factory Method When

A class can't anticipate the class of objects it must create

A class wants its subclasses to specify the objects it creates

You want to localize the knowledge of which help classes is used in a class

But when is this?

CS 580 Example - Testing a Server

```
public class SDTwitterServer {
   public void run(int port) throws IOException {
      ServerSocket input = new ServerSocket( port );

      while (true) {
         Socket client = input.accept();
         processRequest(
            client.getInputStream(),
            client.getOutputStream());
         client.close();
      }
   }

   void processRequest(InputStream in,OutputStream out) {
      do a bunch of stuff
   }

   etc.
```

Using Factory Method

```
public class SDTwitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = this.serverSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    ServerSocket serverSocket( int port) {
        return new ServerSocket(port);
    }

    etc.
```

TestServer

```
public class TestServer extends SDTwitterServer {
   MockServerSocket testSocket;

   ServerSocket serverSocket( int port) {
      return testSocket;
   }
}
```

Other than using a different type of socket it performs the operations as the parent class

```
public class Tests extends Testcase {
   public void testLogin() {
      TestServer server = new TestServer();
      server.testSocket = new MockServerSocket("client command to login");
      server.run();
      assertTrue(server.testSocket.serverResponse() = "the correct response here");
   }
}
```

MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection
- Does not use network
- Contains fixed requests
- Records server responses

Dependency Injection

```
public class SDTwitterServer {
    ServerSocket input;
    public SDTwitterServer(ServerSocket input) {
          this.input = input;
    }

    public void run(int port) throws IOException {
        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
}
```

Dependency Injection

"One object (or static method) supplies the dependencies of another object"
Wikipedia

Constructor injection
Setter injection
Interface injection

Effective Java

Effective Java

Book by Joshua Bloch

First Edition 2001
Second Edition 2008

Item 1. Consider Static Factory methods

Consider using static Factory methods instead of constructors

Java String class

```
public static String valueOf(boolean b) {
        return b ? "true" : "false";
    }

public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}
```

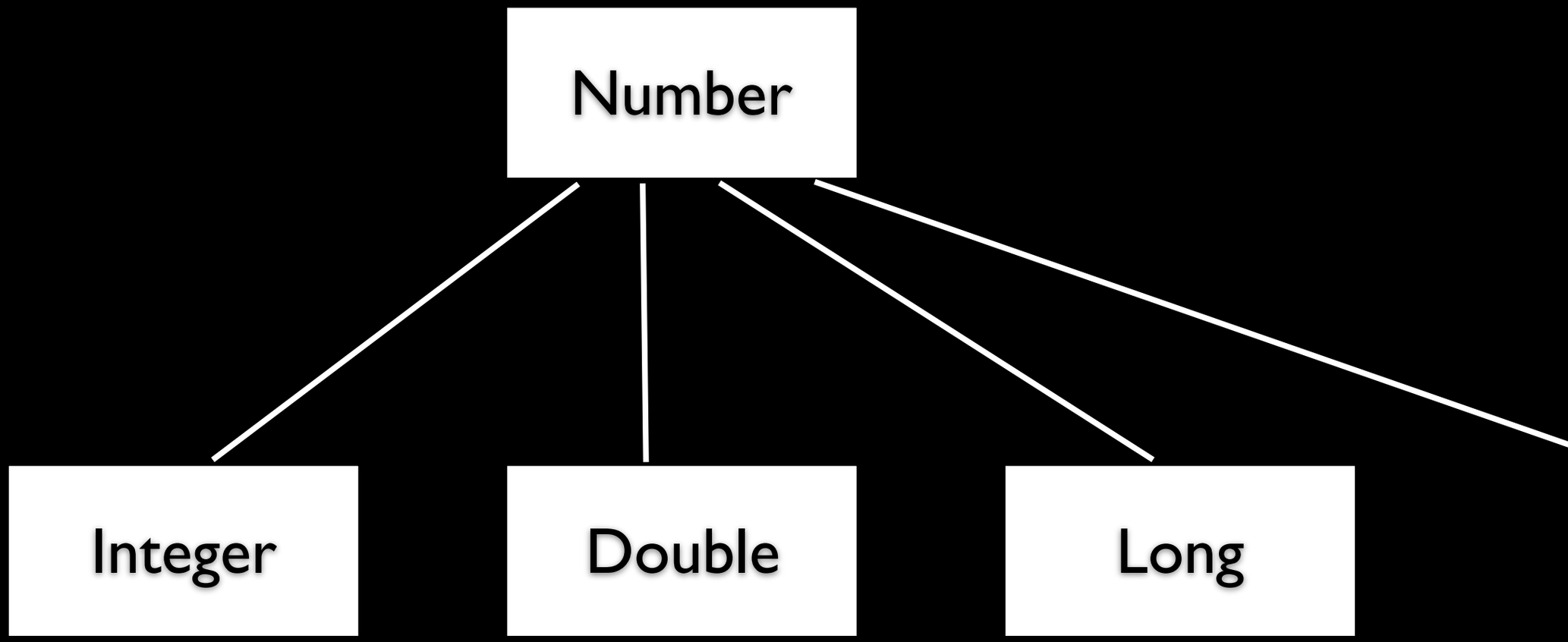
Advantages of Static Factory methods

They have names

Don’t need to create a new object each time

They can return an Object of any type

Java Boxing of Primitives




```
Integer x = new Integer( 5);
Boolean y = new Boolean( true);
```

Objective C Boxing of Primitives

Uses static factory methods in Number

```
Number x = Number.value(5);
Number y = Number.value(true);
```

Programmers only need to know Number class

Class Cluster

Smalltalk

No constructors

Just static factory methods

Item 12 Minimize accessibility

Rule of thumb

Make each class or member as inaccessible as possible

Item 13 Favor Immutability

Immutable objects are simple

Immutable objects are thread-safe

Immutable objects can be shared freely

Immutable objects are good building blocks for other objects

Item 13 Favor Immutability

Don’t provide any methods that modify the object (setters)

Ensure that no methods may be overridden

Make all fields final

Make all fields private

Ensure exclusive use to any mutable components

Make defensive copies of data provided/given to client

Item 24 Make Defensive Copies when Needed

```
public final class Period {
  private final Date start;
  private final Date end;

  public Period(Date start, Date End) {
    if (start.compareTo(end) > 0 )
      throw new IllegalArgumentException(start + " is after " + end);
    this.start = start;
    this.end = end;
  }

  public Date start() {
    return start;
  }
```

Item 24 Make Defensive Copies when Needed

```
public final class Period {
  private final Date start;
  private final Date end;

  public Period(Date start, Date End) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
    if (this.start.compareTo(this.end) > 0 )
      throw new IllegalArgumentException(start + " is after " + end);
  }

  public Date start() {
    return start.clone();
  }
```

Item 14 Favor Composition over Inheritance

Inheritance breaks encapsulation

Safe to use inheritance when

Superclass and subclass in same package

When superclass is designed for inheritance

Item 16 Prefer Interfaces to Abstract Classes

Existing classes can be modified to implement a new interface

Interfaces are ideal for defining mixins

Interfaces allow construction of nonhierarchical frameworks

Provide skeletal implementation class to go with nontrivial interface

Item 30 Know and use the Libraries

Item 32 Avoid strings if other types are better

String compoundKey = name + “#” + i.next();

What happens if “#” is in name?

Create CompoundKey class

Item 34 Refer to objects by their Interfaces

Your code will be more flexible

List subscribers = new Vector();

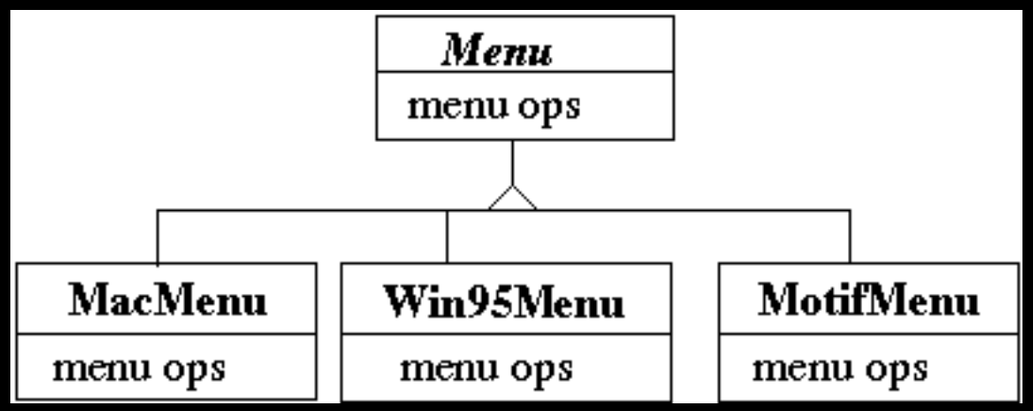
Vector subscribers = new Vector();

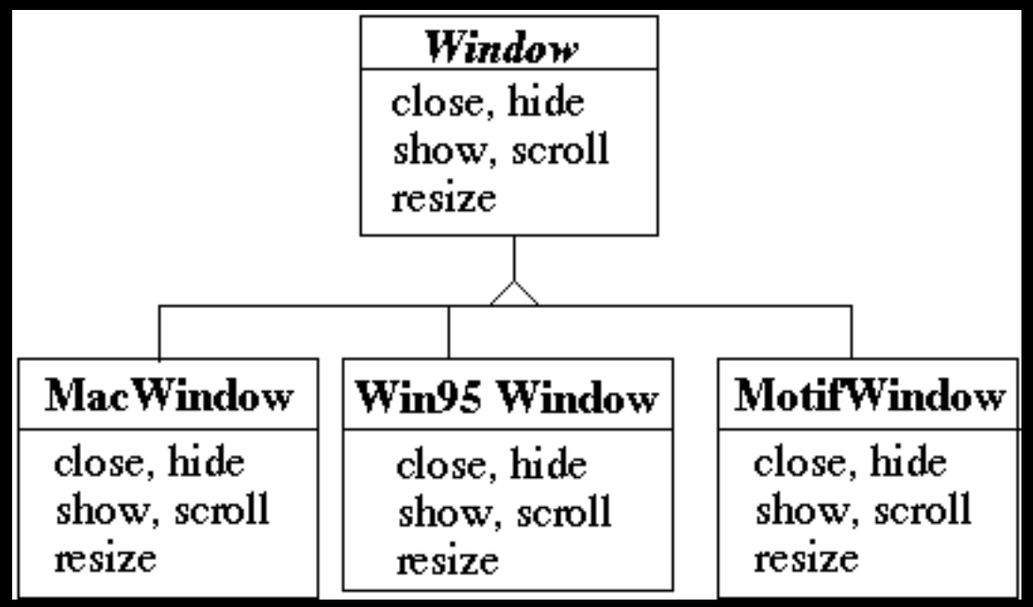
If no interface exists then ok to refer to object via class

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 createMenu()
    { 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.
    }
```