CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 20 Abstract Factory and Builder
Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 87-106

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 31-62

Abstract Factory

Task - Write a cross platform window toolkit

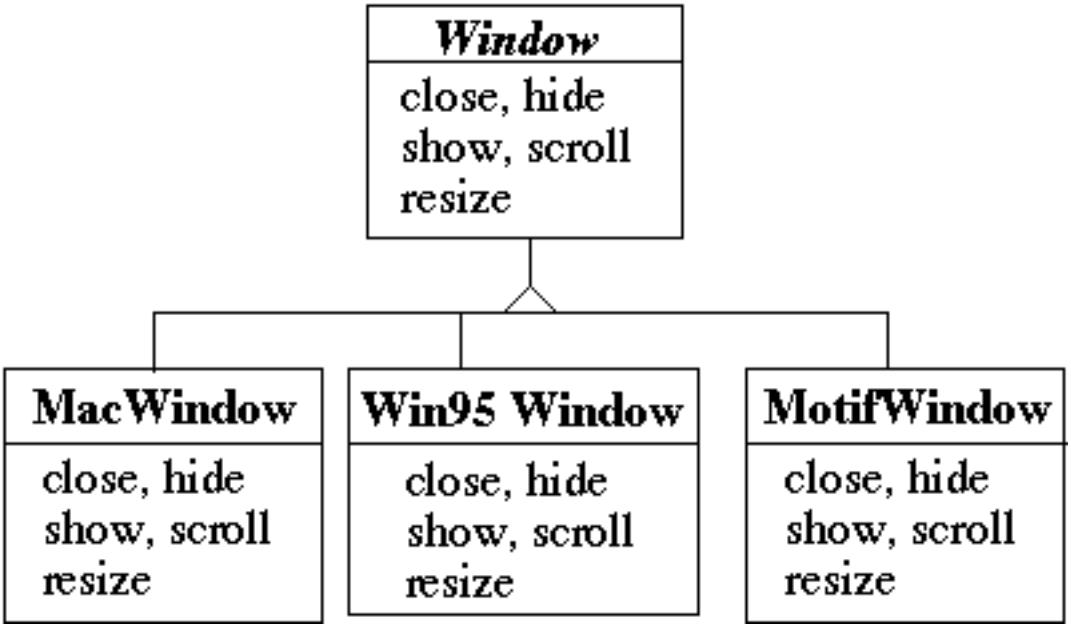
GUI interfaces to run on

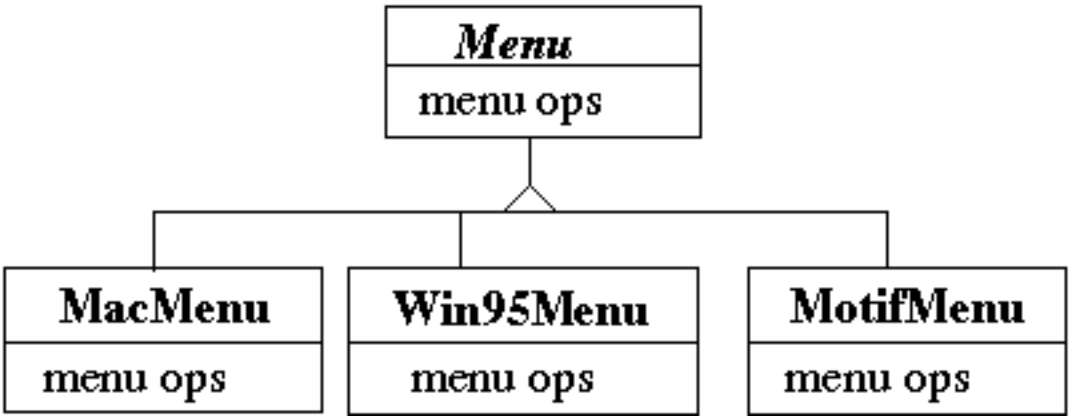
- Mac, PC & Unix
- Use the look and feel of each platform

We will look at widgets: Windows, Menu's and Buttons

Create

- An interface (or abstract class) for each widget
- A concrete class for each platform:





This allows the application to write to the widget interface

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

How to create the widget so

- We get the correct interface widgets
- Minimize places in code that know the platform

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 }
   }
```

Now to get code that works for all platforms we get:

```
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.
   }
```

We just need to make sure that the application for each platform creates the proper factory

How Do Factories create Widgets?
Method 1) My 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()  }
   }
```

How Do Factories create Widgets?
Method 2) Their 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.
```

Method 2) Their Factory Method
When does this make Sense?

There might be more than one way to create a widget

```
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() }
```

Using factory method allows abstract class to do all the different ways to create a window.

Subclasses just provide the objects windowFactory, menuFactory, buttonFactory, etc.

How Do Factories create Widgets?
Method 2.5) Subclass returns Class

```
abstract class WidgetFactory {

   public Window createWindow()
      { return windowClass().newInstance() }

   public Menu createMenu();
      { return menuClass().newInstance()  }

   public Button createButton()
      {  return buttoneClass().newInstance() }

   public Class windowClass();
   public Class menuClass();
   public Class buttonClass();
}

class MacWidgetFactory extends WidgetFactory {
   public Class windowClass()
      { return MacWindow.class; }

   public Class menuClass()
      { return MacMenu.class; }

   public Class buttonClass()
      { return MacButton.class; }
}
```

Smalltalk practice
Parent class normally does more complex stuff

How Do Factories create Widgets?
Method 3) 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 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() }

  etc.
```

There is no need for subclasses of WidgetFactory.

Applicability

Use when

- A system should be independent of how its products are created, composed and represented
- A system should be configured with one of multiple families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementation

Consequences

- It isolates concrete classes
- It makes exchanging product families easy
- It promotes consistency among products
- Supporting new kinds of products is difficult

Implementation

- Factories as singletons
- Defining extensible factories

Problem: Cheating Application Code

```
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.
   }
```

How to avoid this problem?

Builder
Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations

Applicability

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
- The construction process must allow different representations for the object that's constructed

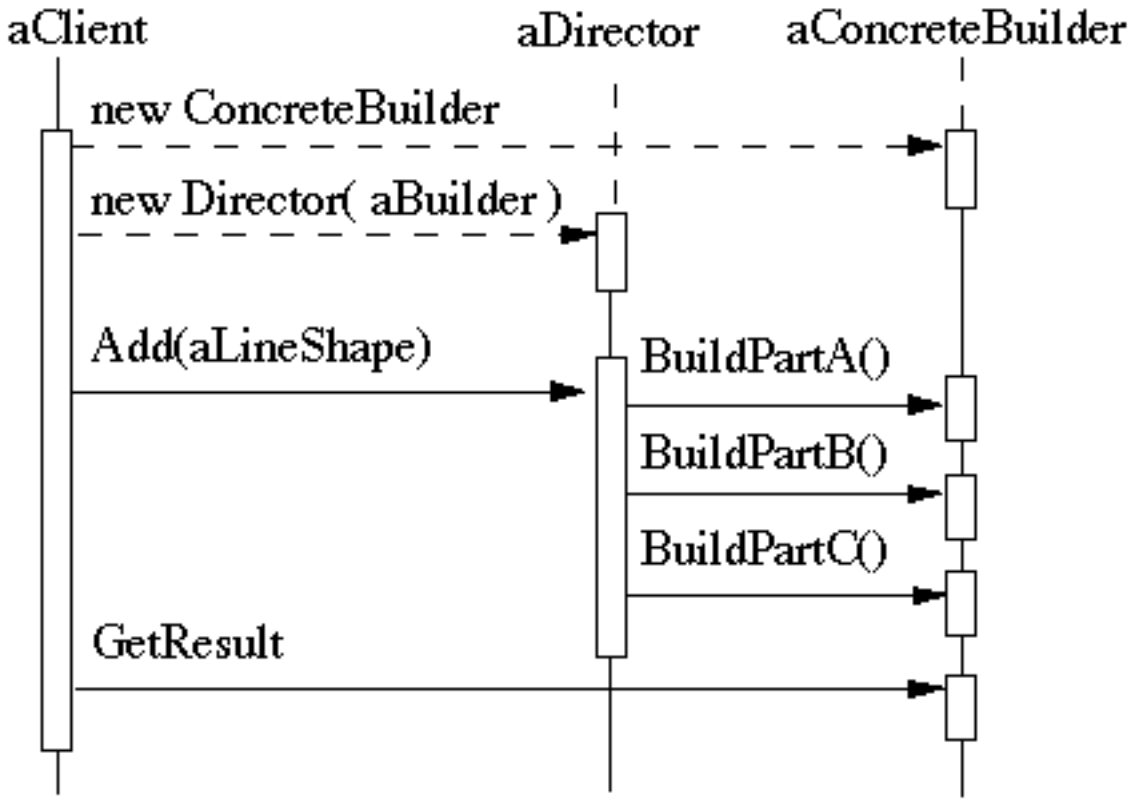
Collaborations

The client creates the Director object and configures it with the desired Builder object

Director notifies the builder whenever a part of the product should be built

Builder handles requests from the director and adds parts to the product

The client retrieves the product from the builder



Example - RTF Converter

A word processing document has complex structure:

words	paragraphs
fonts - times, courier, etc.	fonts sizes
fonts styles - bold, italic, etc.	subscripts, superscripts, etc
headings	style sheets
page numbers	page breaks
images	equations
tables	

RTF (Rich Text Format) is one of several standard formats for converting documents between applications and hardware platforms

The problem is to write a program to convert RTF to other document formats:

TeX, html, ASCII, etc.

RTF for a Page

{\header\pard\plain\s1\s2\tqr\tx8640\f3\fs24\f2\fs24\f2\fs24 March 9, 1998\tab Doc 14 Builder and Product Trader slide # \chpgn\par}\fs36\f14\f2\fs24\f2\fs24\f14\fs36{\b Builder}
\par{\b Intent}
\par\ql
\par Separate the construction of a complex object from its representation so that the same construction process can create different representations
\par
\par\qc{\b Applicability}
\par\ql
\par Use the Builder pattern when
\par
\par\li720\ri0\'a5 the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
\par\li0
\par\li720\ri0\'a5 the construction process must allow different representations for the object that's constructed
\par\li0

Outline of Solution Using Builder

```
class RTF_Reader
  {
  TextConverter builder;
  String RTF_Text;

  public RTF_Reader( TextConverter aBuilder,
                     String RTFtoConvert )
    {
    builder = aBuilder;
    RTF_Text = RTFtoConvert;
    }

  public void parseRTF()
    {
    RTFTokenizer rtf = new RTFTokenizer( RTF_Text );

    while ( rtf.hasMoreTokens() )
      {
      RTFToken next = rtf.nextToken();

      switch ( next.type() )
        {
        case CHAR:     builder.character( next.char() ); break;
        case FONT:     builder.font( next.font() ); break;
        case PARA:     builder.newParagraph( ); break;
        etc.
        }
      }
    }
  }
```

More Outline

```
abstract class TextConverter
   {
   public void character( char nextChar ) {  }
   public void font( Font newFont ){ }
   public void newParagraph() {}
   }

class ASCII_Converter extends TextConverter
   {
   StringBuffer convertedText = new StringBuffer();

   public void character( char nextChar )
      {
      convertedText.append( nextChar );
      }

   public String getText()
      {
      return convertedText.toString();
      }
   }

class TeX_Converter extends TextConverter
   {
   public void character( char nextChar )       { blah  }
   public void font( Font newFont )        { blah }
   public void newParagraph()                   { blah }
   public TeX getText()          { return the correct thing }
   }
```

Main Program

```
main()
   {
   ASCII_Converter simplerText = new ASCII_Converter();
   String rtfText;

   // read a file of rtf into rtfText

   RTF_Reader myReader =
      new RTF_Reader( simplerText, rtfText );

   myReader.parseRTF();

   String myProduct = simplerText.getText();
   }
```

Consequences

- It lets you vary a product's internal representation
- It isolates code for construction and representation
- It gives you finer control over the construction process

Implementation

- Assembly and construction interface

  Builder may have to pass parts back to director, who will then pass them back to builder

- Why no abstract classes for products
- Empty methods as default in Builder