CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2002
Doc 16 Bridge & Abstract Factory

Contents

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995 pp. 87-106, 151-162

Advanced C++: Programming Styles and Idioms, James Coplien, 1992, pp. 31-62, 58-72

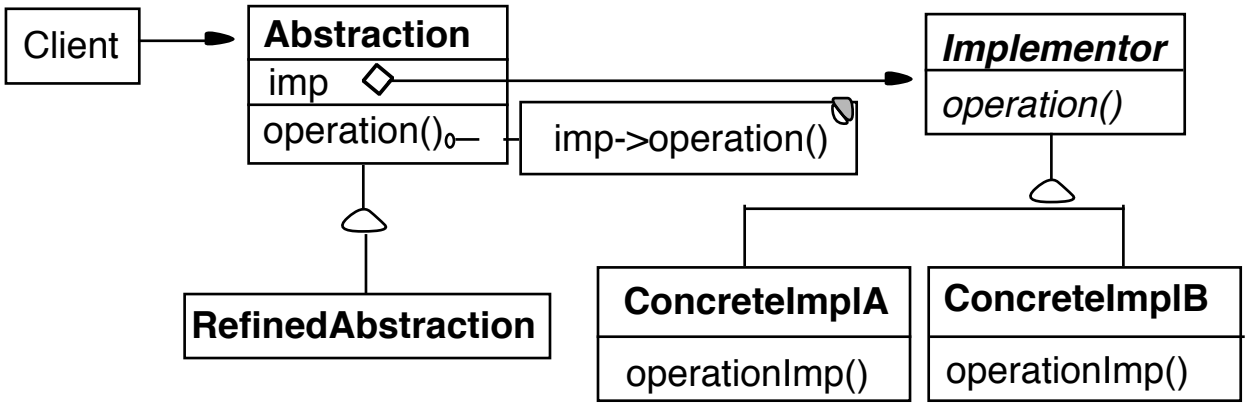
Bridge

Decouple the abstraction from its implementation

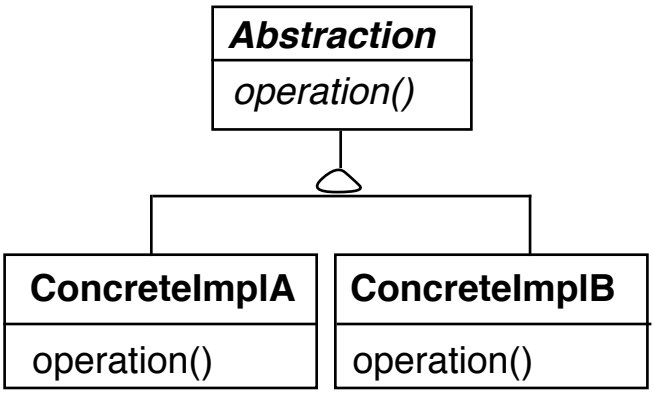
This allows the implementation to vary from its abstraction

The abstraction defines and implements the interface

All operations in the abstraction call method(s) its implementation object



What is Wrong with Using an Interface?



Make Abstraction a pure abstract class or Java interface

In client code:

```
Abstraction widget = new ConcreteImplA();
widget.operation();
```

This will separate the abstraction from the implementation

We can vary the implementation!

Applicability

Use the Bridge pattern when

- You want to avoid a permanent binding between an abstraction and its implementation
- Both the abstractions and their implementations should be independently extensible by subclassing
- Changes in the implementation of an abstraction should have no impact on the clients; that is, their code should not have to be recompiled
- You want to hide the implementation of an abstraction completely from clients (users)
- You want to share an implementation among multiple objects (reference counting), and this fact should be hidden from the client

Binding between abstraction & implementation

In the Bridge pattern:

- An abstraction can use different implementations
- An implementation can be used in different abstraction

Hide implementation from clients

Using just an interface the client can cheat!

```
Abstraction widget = new ConcreteImplA();
widget.operation();
((ConcreteImplA) widget).concreteOperation();
```

In the Bridge pattern the client code can not access the implementation

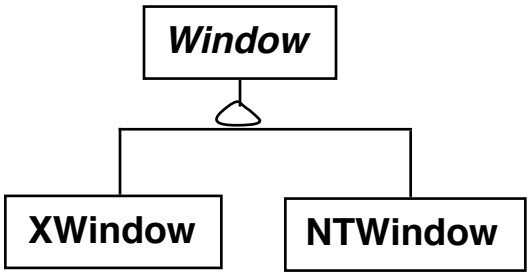
Java AWT uses Bridge to prevent programmer from accessing platform specific implementations of interface widgets, etc.

Peer = implementation

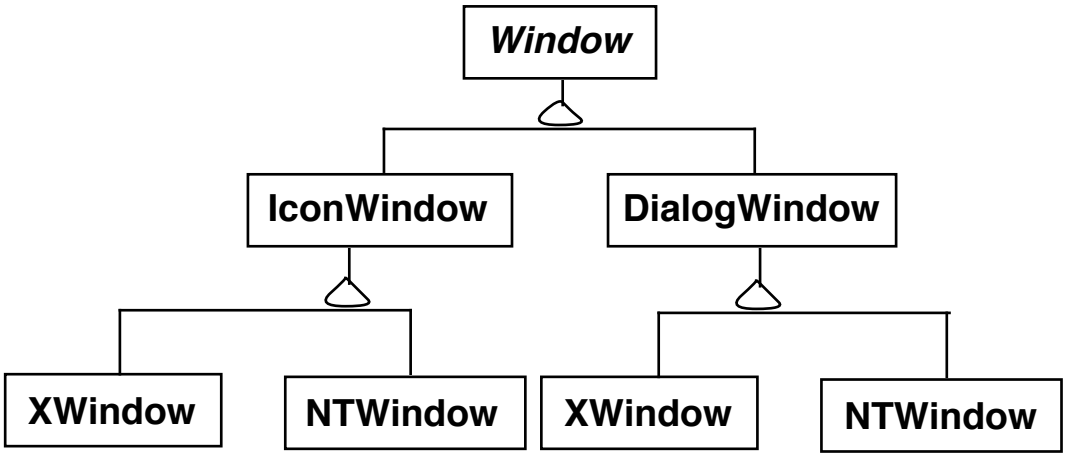
```
public synchronized void setCursor(Cursor cursor) {
   this.cursor = cursor;
   ComponentPeer peer = this.peer;
   if (peer != null) {
      peer.setCursor(cursor);
   }
}
```

Abstractions & Imps independently subclassable

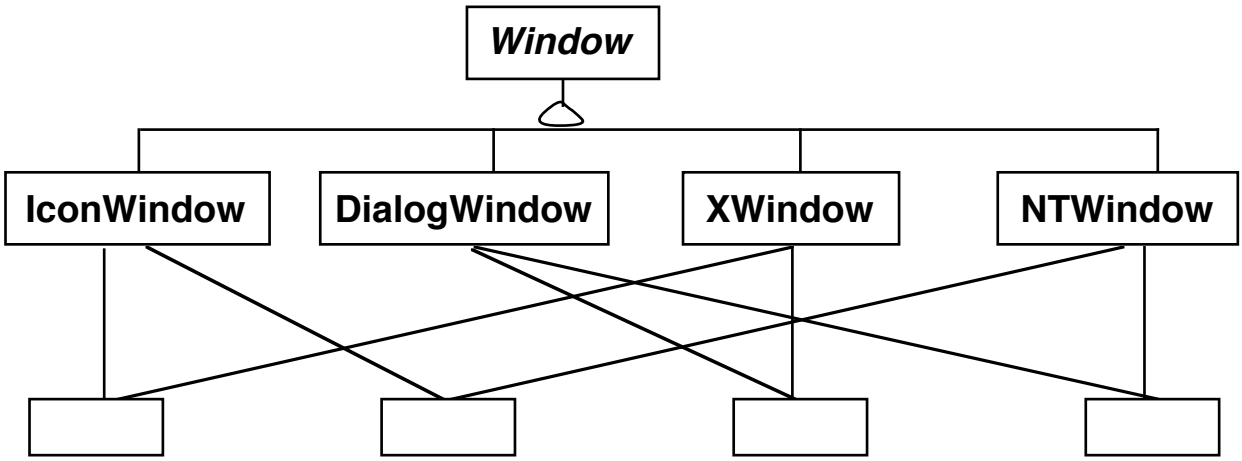
Start with Widow interface and two implementations:



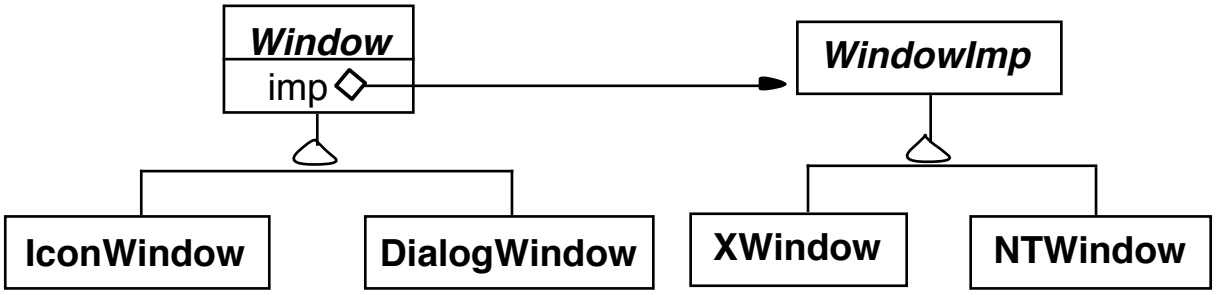
Now what do we do if we need some more types of windows: say IconWindow and DialogWindow?



Or using multiple inheritance



The Bridge pattern provides a cleaner solution



IconWindow and DialogWindow will add functionality to or modify existing functionality of Window

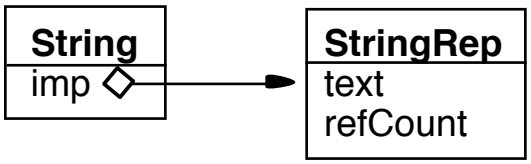
Methods in IconWindow and DialogWindow need to use the implementation methods to provide the new/modified functionality

This means that the WindowImp interface must provide the base functionality for window implementation

This does not mean that WindowImp interface must explicitly provide an iconifyWindow method

Share an implementation among multiple objects

Example use is creating smart pointers in C++

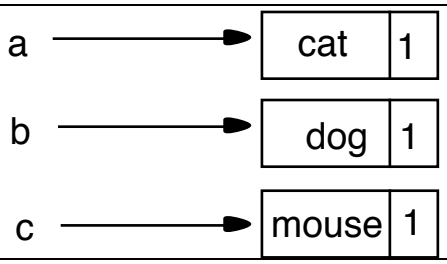
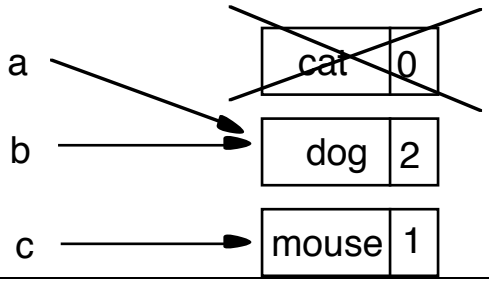
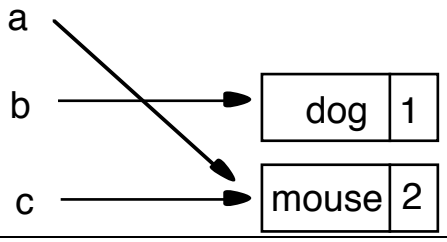


String contains a StringRep object

StringRep holds the text and reference count

String passes actual string operations to StringRep object

String handles pointer operations and deleting StringRep object when reference count reaches zero

String a(“cat”); String b(“dog”); String c(“mouse”);	
a = b;	
a = c;	

C++ Implementation from Coplien

```
class StringRep {
  friend String;

  private:
    char *text;
    int refCount;

    StringRep(){ *(text = new char[1] = '\0'; }

    StringRep( const StringRep& s )     {
      ::strcpy( text = new char[::strlen(s.text) + 1, s.text);
    }

    StringRep( const char *s) {
      ::strcpy( text = new char[::strlen(s) + 1, s);
    }

    StringRep( char** const *r)     {
      text = *r;
      *r = 0;
      refCount = 1;;
    }

    ~StringRep() { delete[] text; }

    int length() const      { return ::strlen( text ); }

    void print() const      { ::printf("%s\n", text ); }
  }
```

```
class String   {
   friend StringRep

   public:
      String operator+(const String& add) const {
         return *imp + add;
      }

      StringRep* operator->() const        { return imp; }

      String()  { (imp = new StringRep()) -> refCount = 1;        }

      String(const char* charStr)    {
         (imp = new StringRep(charStr)) -> refCount = 1;
      }

      String operater=( const String& q) {
         (imp->refCount)--;
         if (imp->refCount <= 0 &&
             imp != q.imp )
            delete imp;

         imp = q.imp;
         (imp->refCount)++;
         return *this;
      }

      ~String()    {
         (imp->refCount)--;
         if (imp->refCount <= 0 ) delete imp;
      }

   private:
      String(char** r)  {imp = new StringRep(r);}
      StringRep *imp;
};
```

Using Counter Pointer Classes

```
int main() {
   String a( “abcd”);
   String b( “efgh”);

   printf( “a is “);
   a->print();

   printf( “b is “);
   b->print();

   printf( “length of b is %d\n“, b-<length() );

   printf( “ a + b “);
   (a+b)->print();
}
```

Abstract Factory

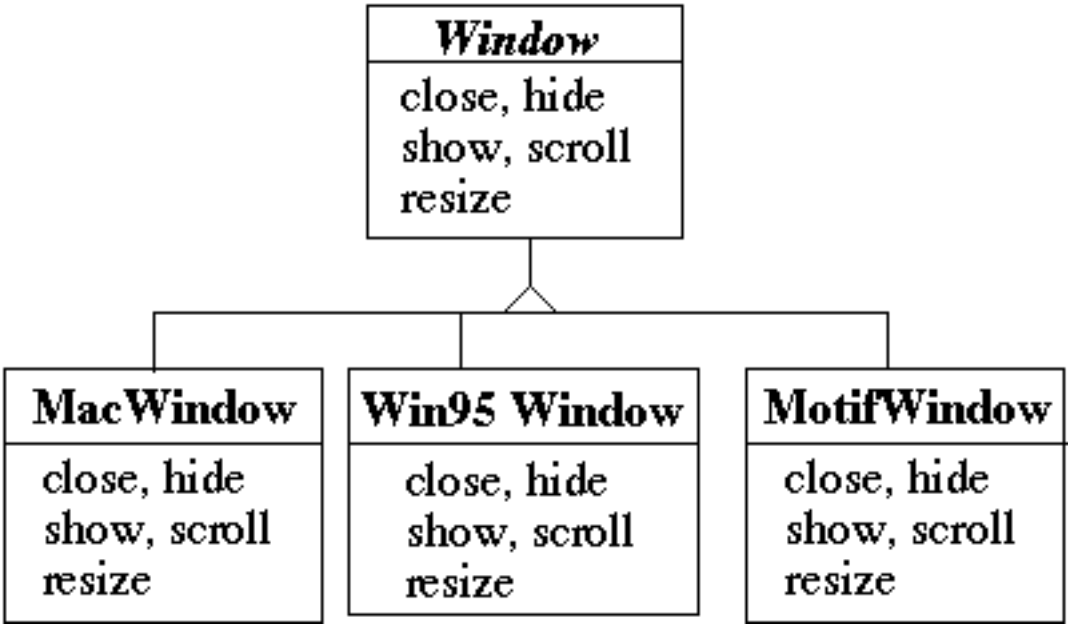
Task - Write a cross platform window toolkit

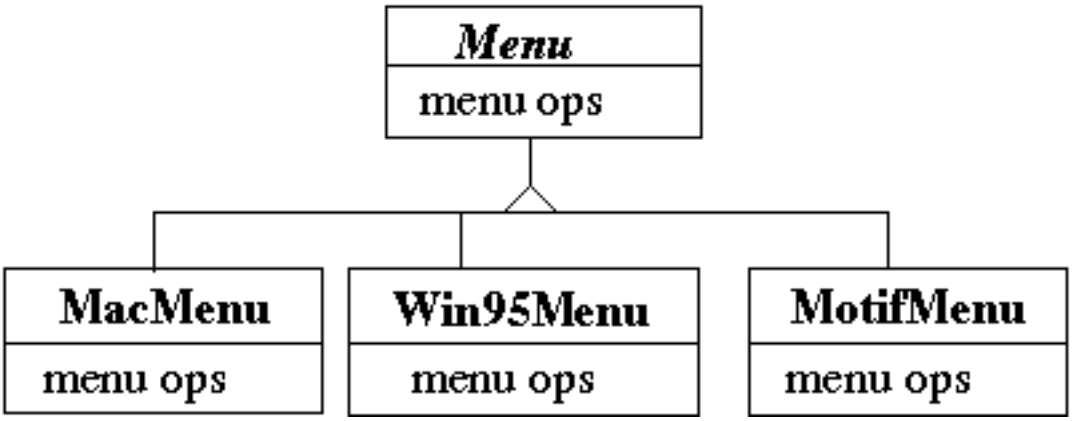
GUI interfaces to run on
- Mac, PC & Unix
- Use the look and feel of each platform

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?