CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 13 Bridge & Builder

Contents

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995 pp. 151-162, 97-106

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

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addison Wesley, 1998, pp. 121-136, 47-62

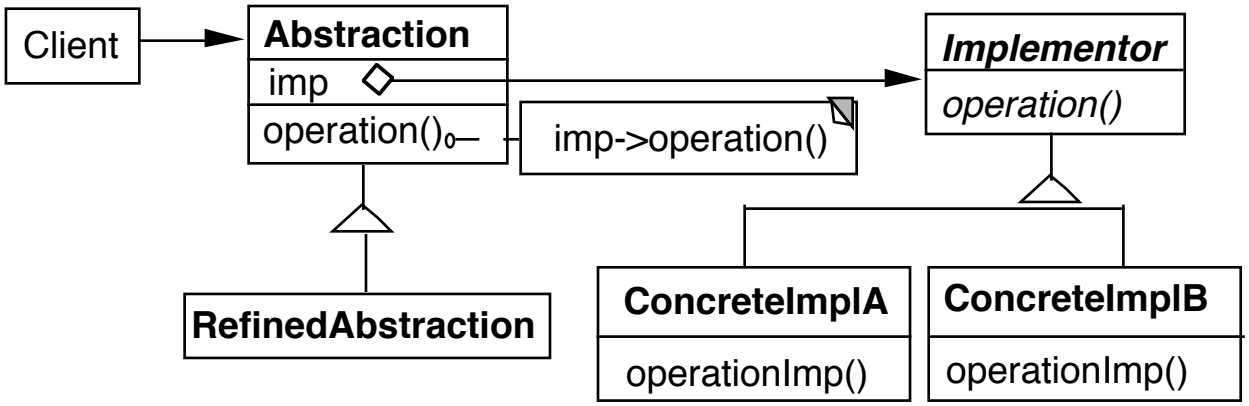
Bridge

Decouple the abstraction from its implementation

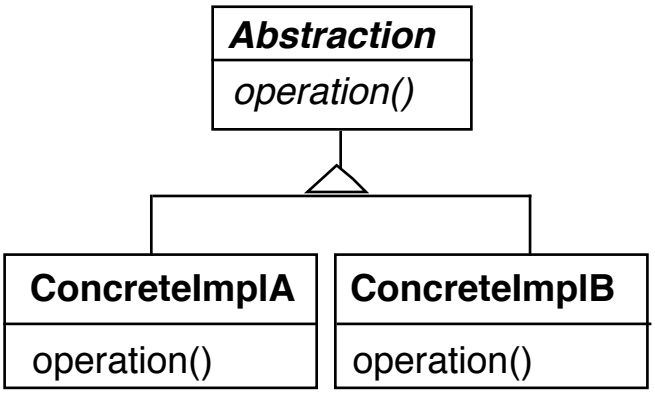
This allows the implementation to vary from its abstraction

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

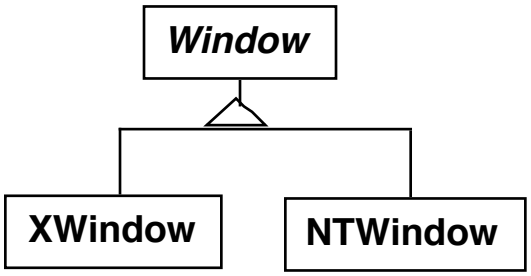
Java AWT uses Bridge to prevent programmer from accessing platform specific implementations of interface widgets, etc.

Peer = implementation

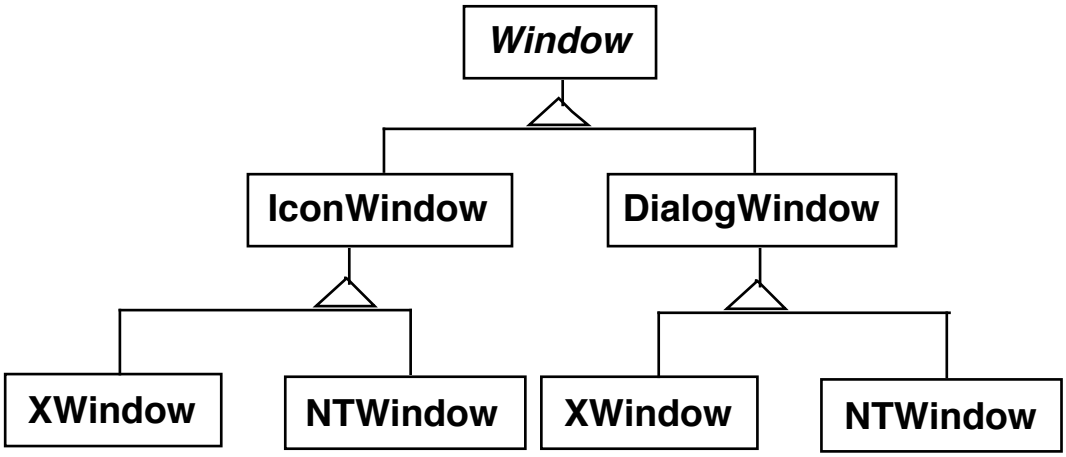
```
public synchronized void setCursor(Cursor cursor) {
   this.cursor = cursor;
   ComponentPeer peer = this.peer;
   if (peer != null) {
      peer.setCursor(cursor);
   }
}
```

Abstractions & Imps independently subclassable

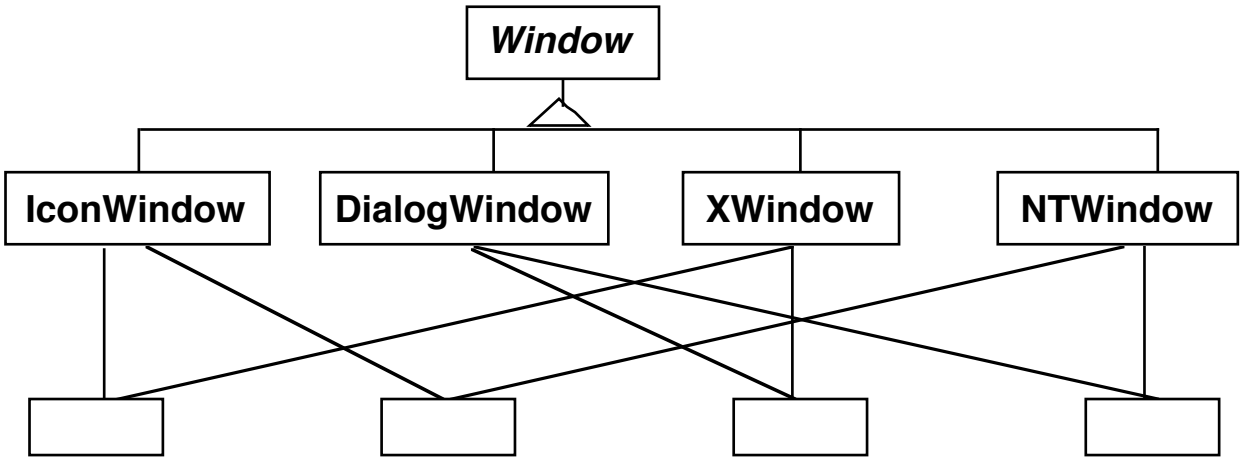
Start with Widow interface and two implementations:



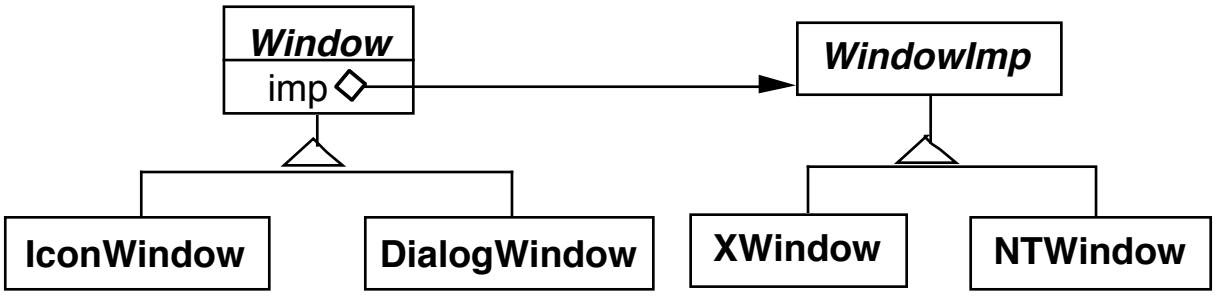
Now what do we do if we need some more types of windows: say IconWindow and DialogWindow?



Or using multiple inheritance



The Bridge pattern provides a cleaner solution



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

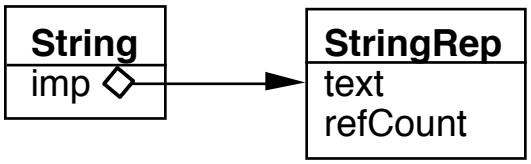
Methods in IconWindow and DialogWindow need to use the implementation methods to provide the new/modified functionality

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++

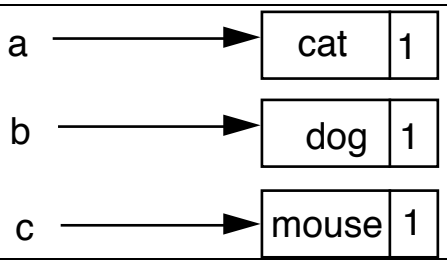
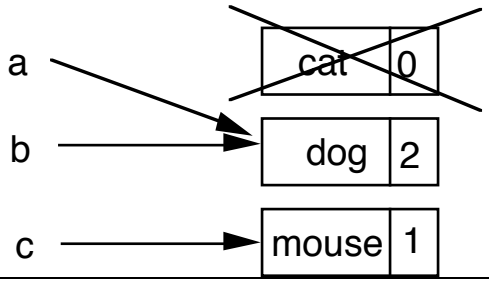
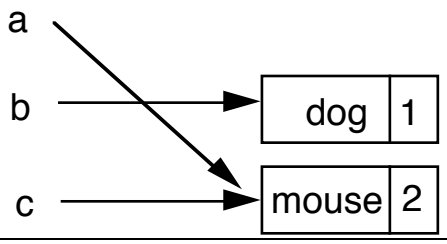


String contains a StringRep object

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();
}
```

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

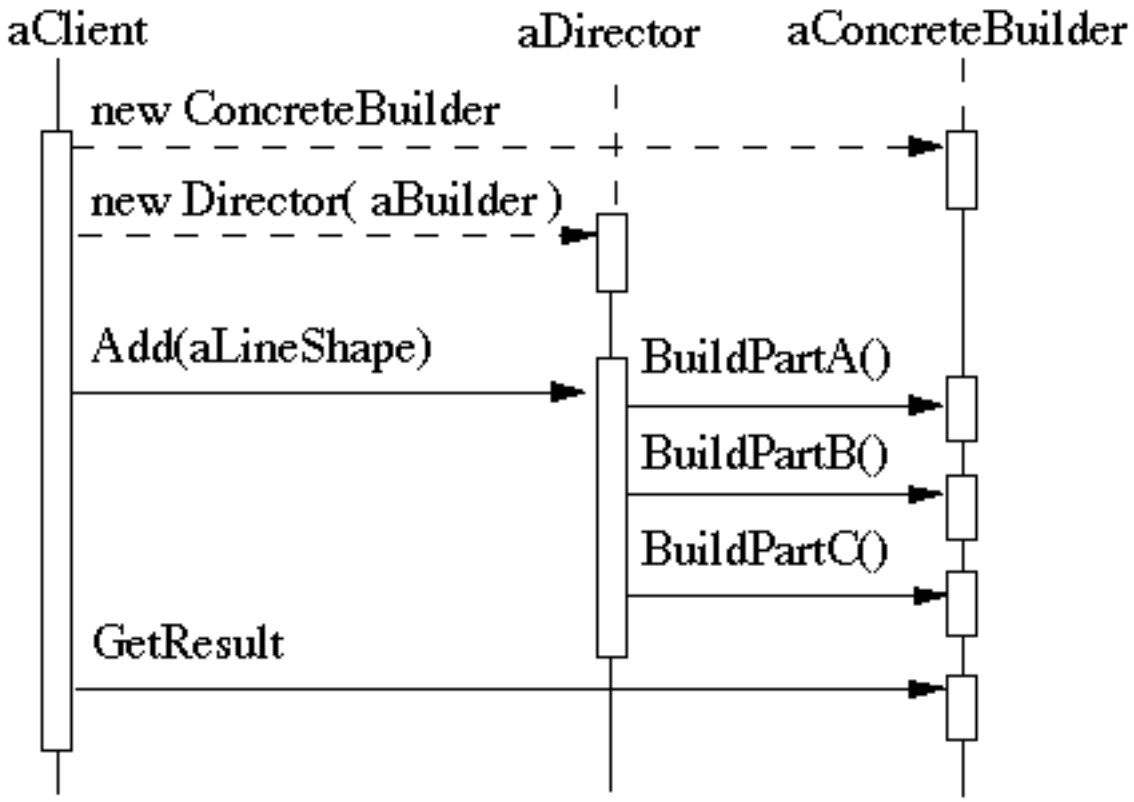
Collaborations

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 – XML Parser

Director
  XML Parser

Abstract Builder Class
  XML.SAXDriver (Smalltalk)
  org.xml.sax.helpers.DefaultHandler (Java)
  DefaultHandler (C++)

Concrete Builder Class
  Your subclass of the abstract builder

Client
    Your code that uses the tree built

Java Example

```
public static void main(String argv[])
  {
  SAXDriverExample handler = new SAXDriverExample();

  // Use the default (non-validating) parser
  SAXParserFactory factory = SAXParserFactory.newInstance();
  try
    {
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse( new File("sample"), handler );
    }
  catch (Throwable t)
    {
    t.printStackTrace();
    }
  System.out.println( handler.root());
  }
```

Smalltalk Example

```
| builder exampleDispatcher |

builder := SAXDriverExample new.
exampleDispatcher := SAXDispatcher new contentHandler: builder.
XMLParser
   processDocumentInFilename: 'page'
   beforeScanDo:
      [:parser |
      parser
         saxDriver:(exampleDispatcher);
         validate: true].
builder root.
```

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