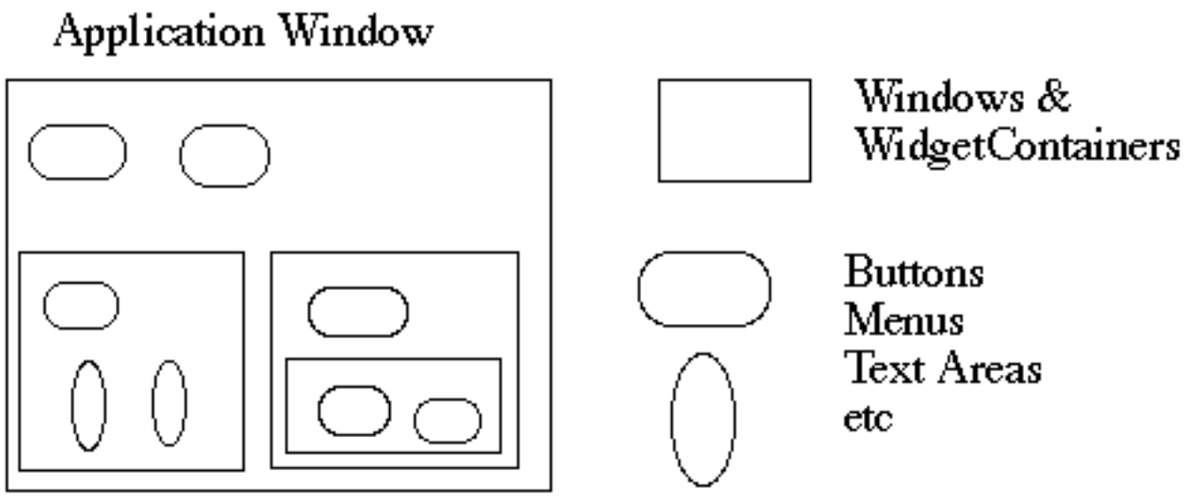
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2015
Doc 16 Adapter, Bridge, Composite
April 9, 2015

Composite

Composite Motivation



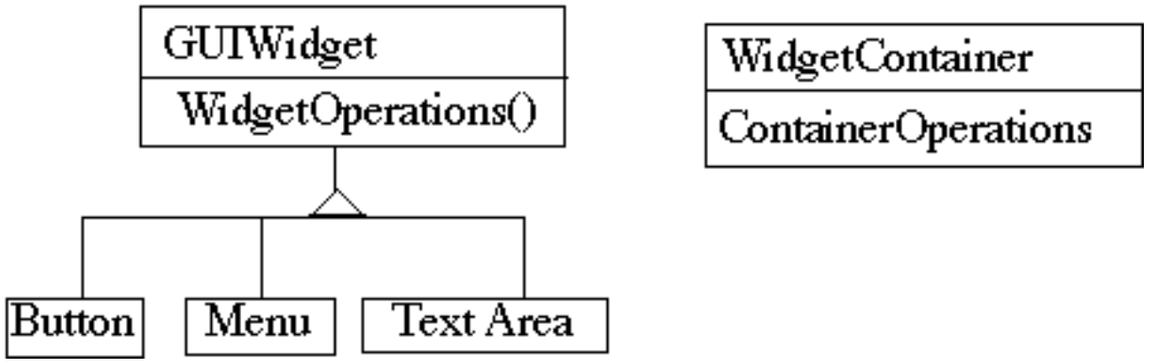
How does the window hold and deal with the different items it has to manage?

Widgets are different that WidgetContainers

Bad News

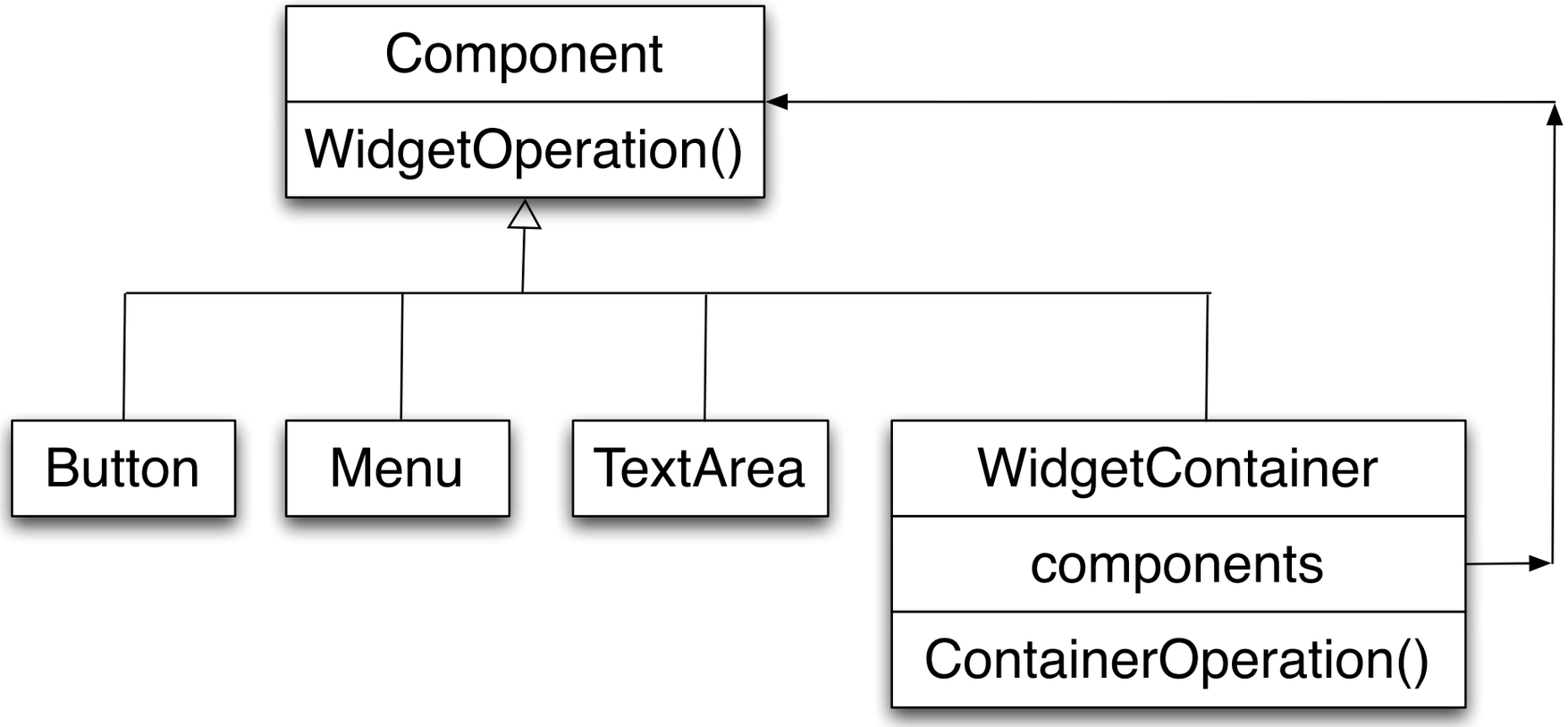
```
class Window {
        Buttons[] myButtons;
        Menus[] myMenus;
        TextAreas[] myTextAreas;
        WidgetContainer[] myContainers;

        public void update() {
                if ( myButtons != null )
                        for ( int k = 0; k < myButtons.length(); k++ )
                                myButtons[k].refresh();
                if ( myMenus != null )
                        for ( int k = 0; k < myMenus.length(); k++ )
                                myMenus[k].display();
                if ( myTextAreas != null )
                        for ( int k = 0; k < myButtons.length(); k++ )
                                myTextAreas[k].refresh();
                if ( myContainers != null )
                        for ( int k = 0; k < myContainers.length(); k++ )
                                myContainers[k].updateElements();
                etc.
        }
        public void fooOperation(){
                if (myButtons != null)
                etc.
```

An Improvement




```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update(){
        if ( myWidgets != null )
            for ( int k = 0; k < myWidgets.length(); k++ )
                myWidgets[k].update();
        if ( myContainers != null )
            for ( int k = 0; k < myContainers.length(); k++ )
                myContainers[k].updateElements();
        etc.
    }
}
```

Composite Pattern



Composite Pattern

Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

```
class WidgetContainer {
        Component[] myComponents;

        public void update() {
                if ( myComponents != null )
                        for ( int k = 0; k < myComponents.length(); k++ )
                                myComponents[k].update();
        }
}
```

Issue - WidgetContainer Operations

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

One out is to check the type of the object before using a WidgetContainer operation

Issue - Parent References

```
class WidgetContainer
    {
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for ( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }

    public add( Component aComponent ) {
        myComponents.append( aComponent );
        aComponent.setParent( this );
    }
}

class Button extends Component {
    private Component parent;
    public void setParent( Component myParent) {
        parent = myParent;
    }

    etc.
```

More Issues

Should Component implement a list of Components?

The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

Applicability

Use Composite pattern when you want

To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects

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

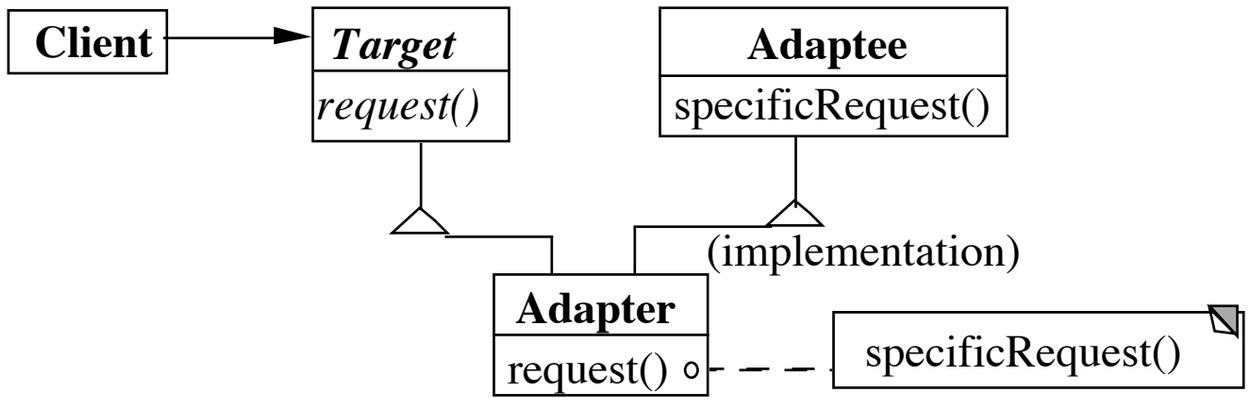
Display an AddressBook object in a JTable

JTables require objects of type TableModel

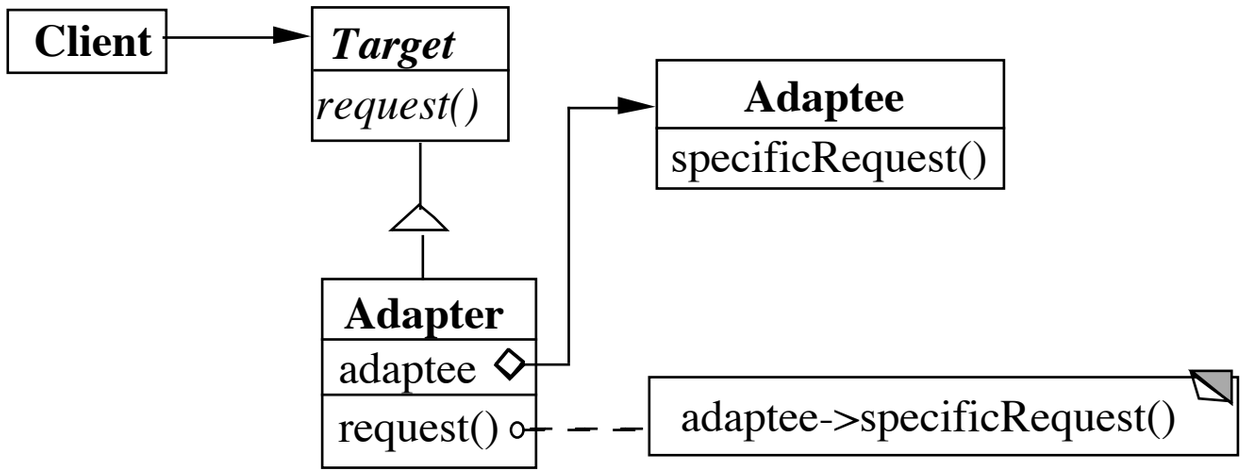
```
public class AddressBook{
    List personList;
    public int getSize(){...}
    public int addPerson(...){...}
    public Person getPerson(...){...}
    ...
}
```

```
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
```

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

```
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
```

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

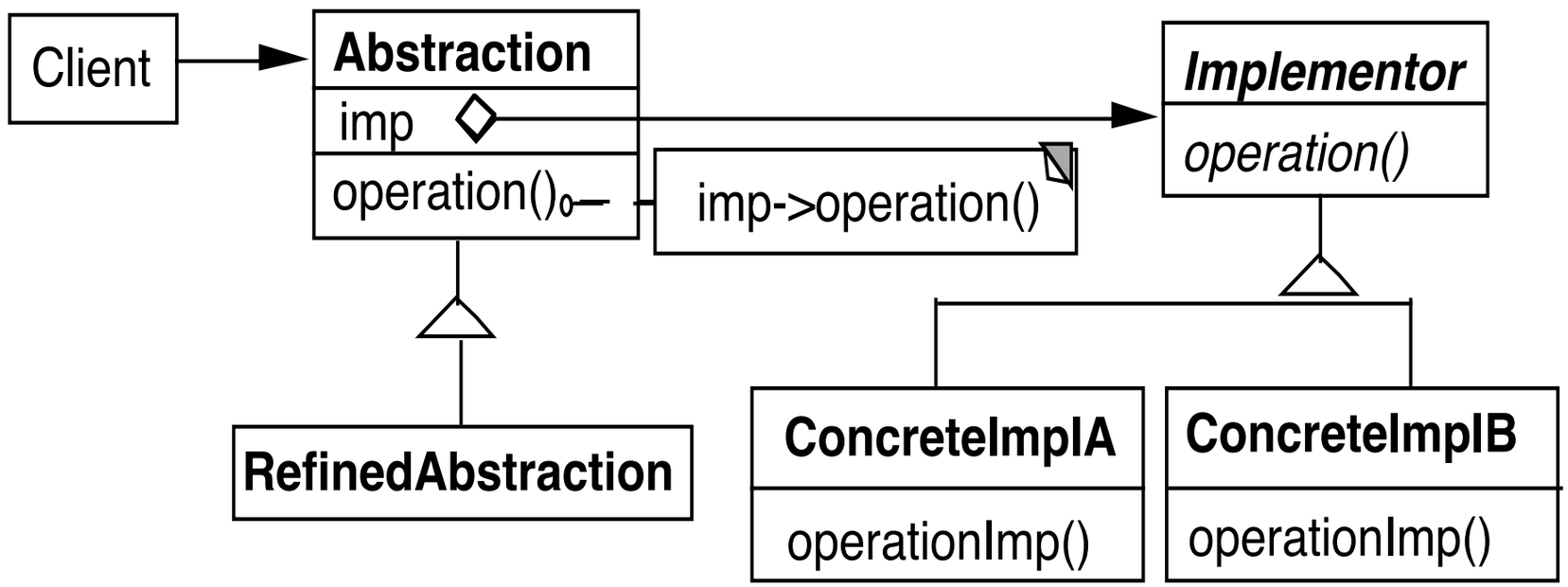
```
class MouseListenerAdapter
    def initialize(adaptee, clickLambda, enterLambda, exitLambda)
        @adaptee = adaptee
        @clickLambda = clickLambda
        @enterLambda = enterLambda
        @exitLambda = exitLambda
    end

    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()})
```

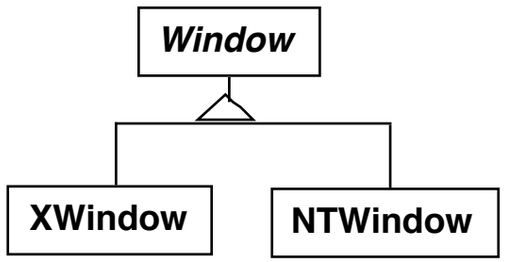
Bridge

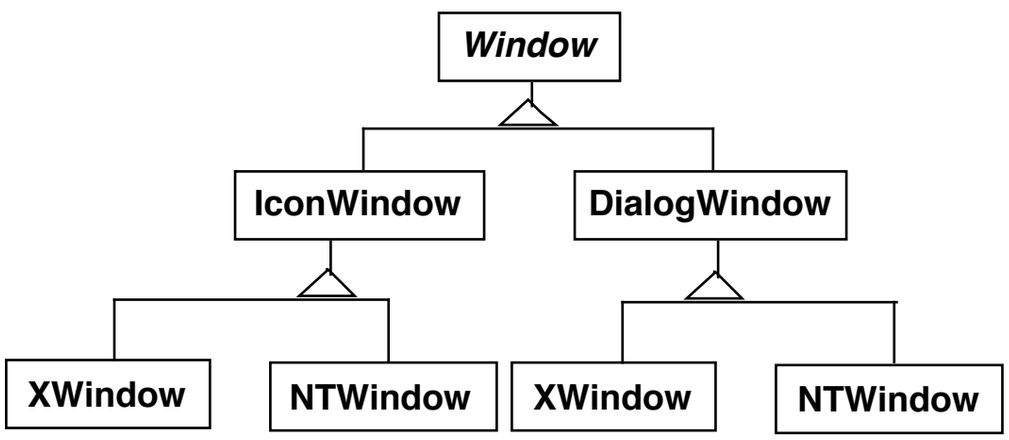
Bridge

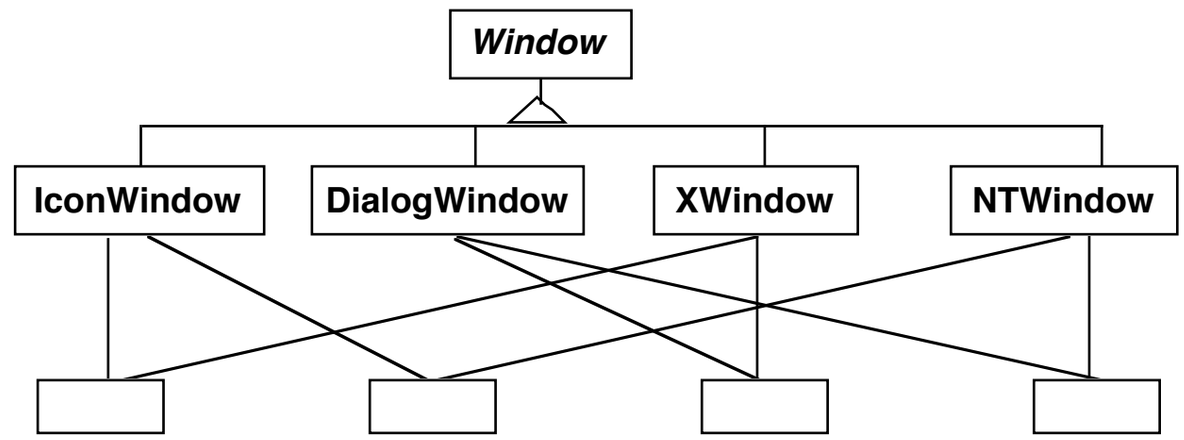
Decouple an abstraction from its implementation



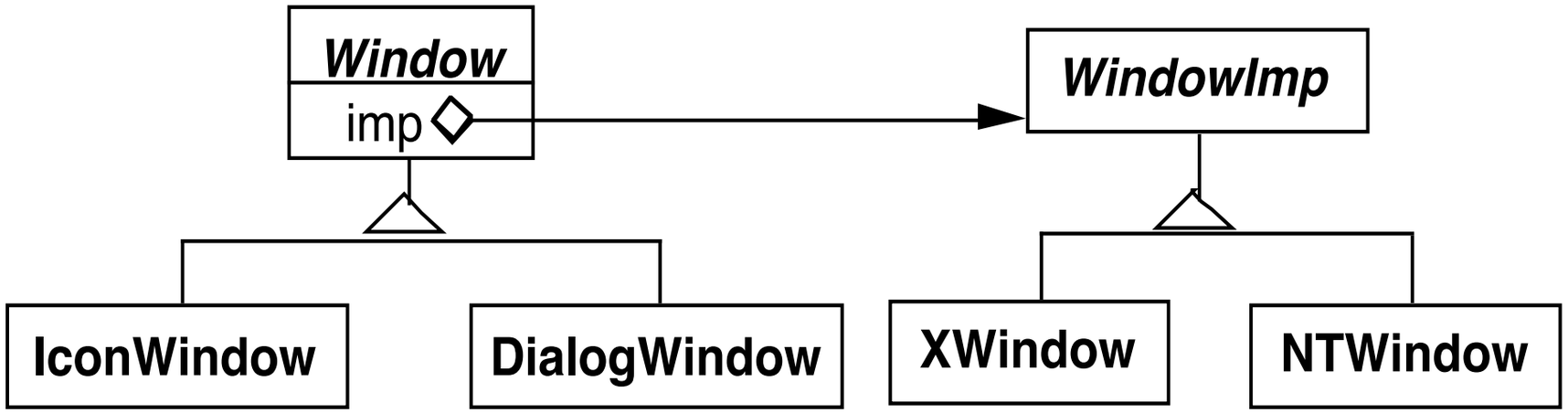
Windows


Window
XWindow
NTWindow

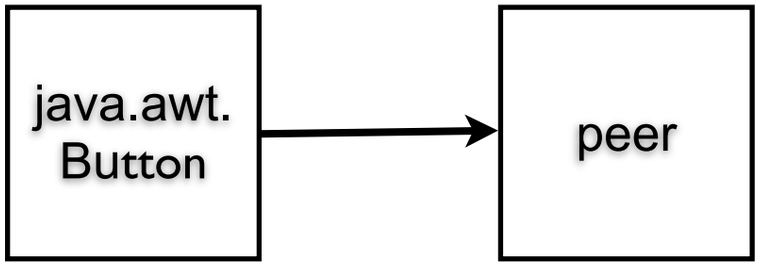

Window
IconWindow
DialogWindow
XWindow
NTWindow
XWindow
NTWindow


Window
IconWindow
DialogWindow
XWindow
NTWindow

Using the Bridge Pattern



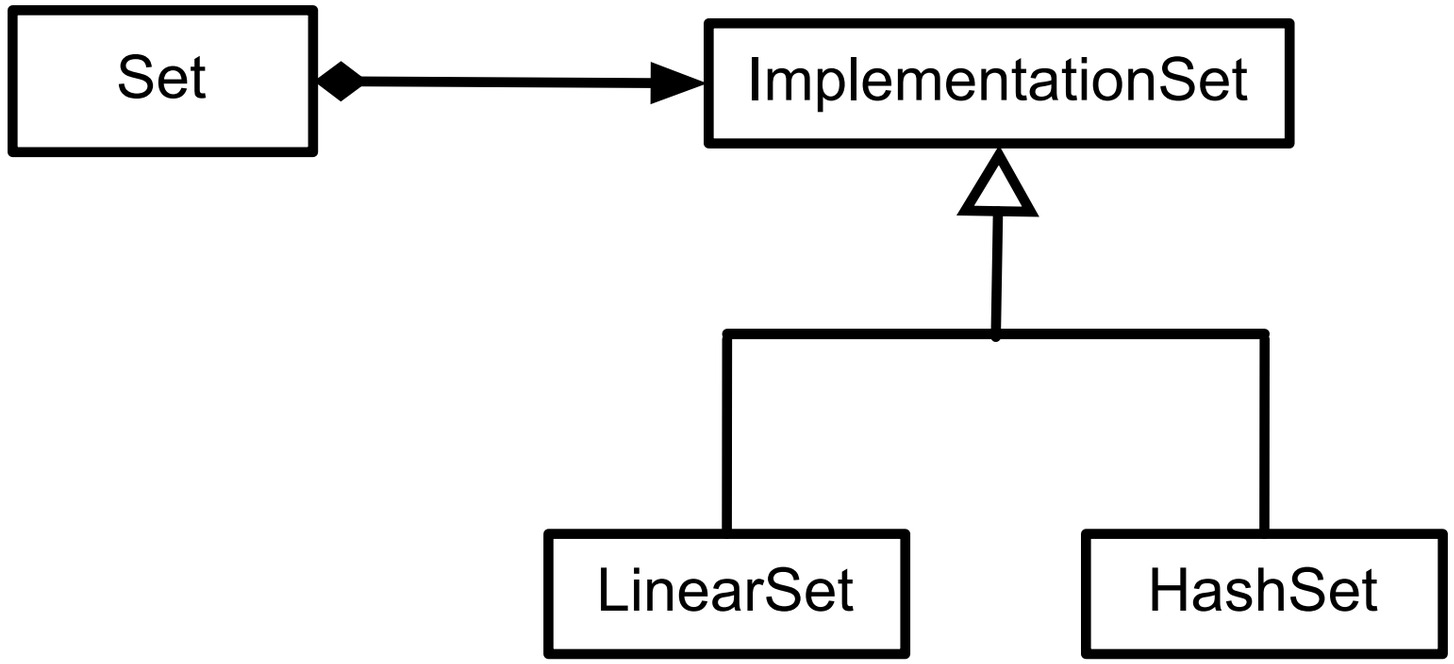
Peers in Java's AWT



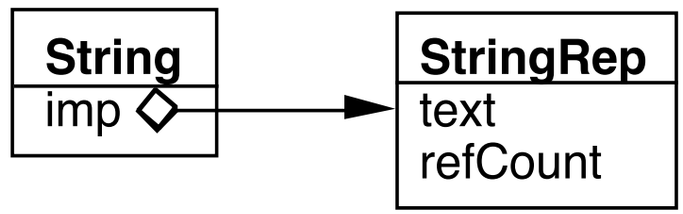
Peer = implementation

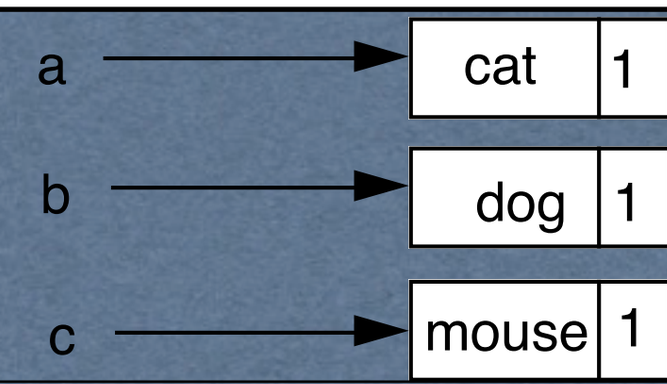
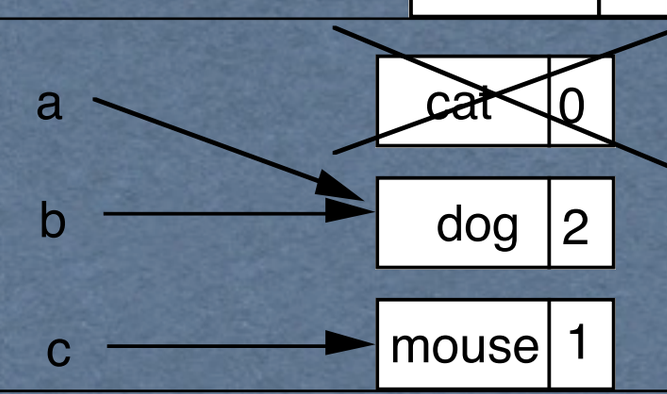
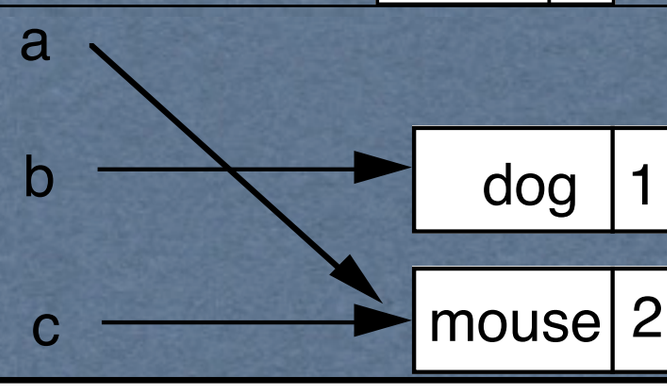
```
public synchronized void setCursor(Cursor cursor) {
    this.cursor = cursor;
    ComponentPeer peer = this.peer;
    if (peer != null) {
        peer.setCursor(cursor);
    }
}
```

IBM Smalltalk Collections



Smart Pointers in C++



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

Coplien's Implementation

```
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;
};
```

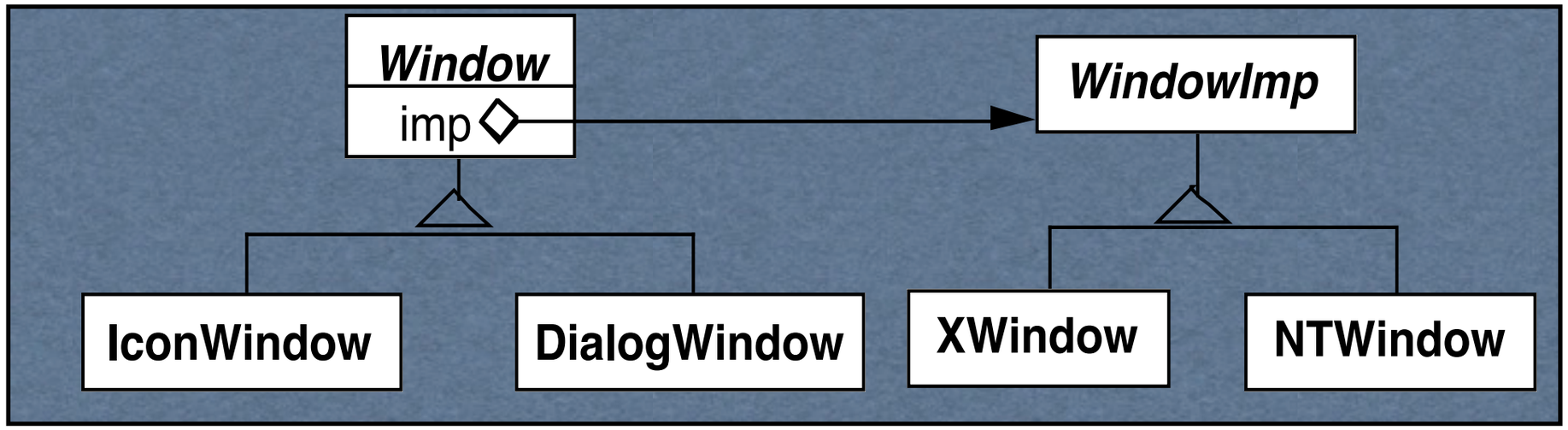
Why Use Bridge

Implementation selected at run-time

Implementation changed during run-time

Why Use Bridge

Abstraction & implementations are extensible by subclassing

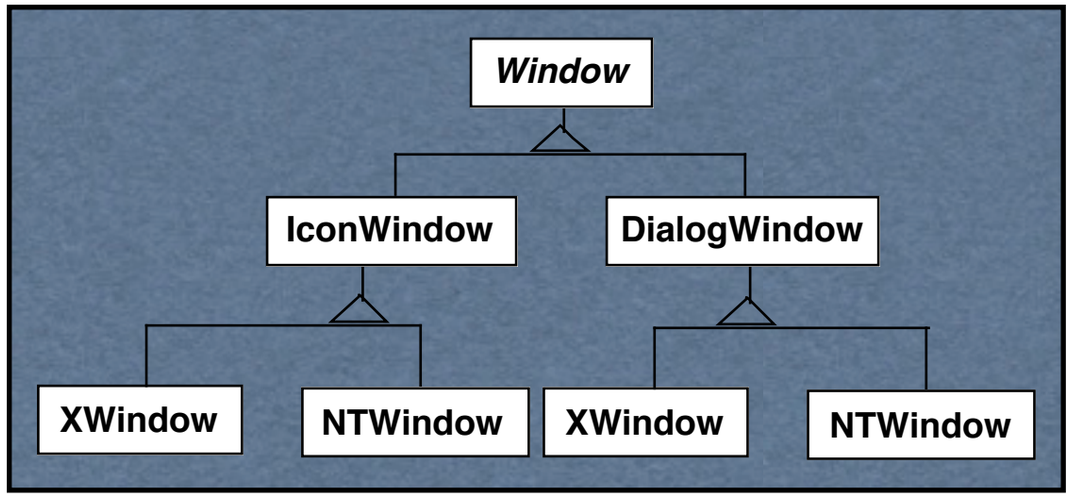


Why Use Bridge

When changes in the implementation should not require client code to be recompiled

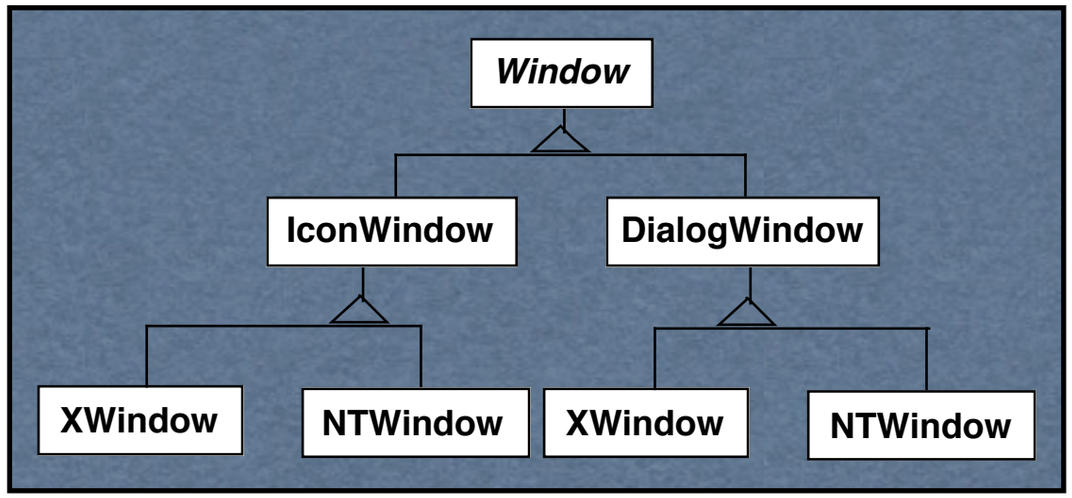
Why Use Bridge

Proliferation of classes



Why Use Bridge

Share implementation among multiple objects



Bridge verses Adapter