CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 17 Memento, Bridge & Facade
Contents

Memento....................2
  Structure....................2
  Applicability.................... 3
  Consequences/ Implementation....................7
  Iterators & Mementos....................11
Bridge.................... 12
  Applicability.................... 14
    Binding between abstraction & implementation....................15
    Hide implementation from clients....................16
    Abstractions & Imps independently subclassable....................17
    Share an implementation among multiple objects....................21
Façade....................25

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 283-292, 151-163, 185 -194

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 297-304, 121-136, 179-188

Copyright ©, All rights reserved. 2005 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

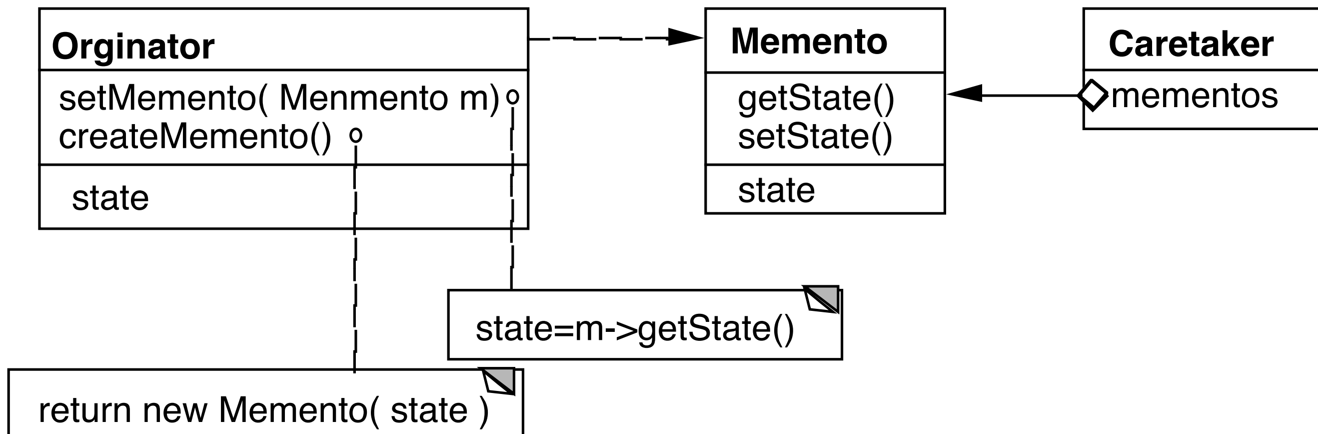
Memento

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

Motivation

Allow undo, rollbacks, etc.

Structure



Only originator:

- Can access Memento’s get/set state methods
- Create Memento

Applicability

Use when you:

- Need to save all or part of the state of an object and
- Do not wish to expose the saved state to the outside world

An Example

```
package Examples;
class Memento
   {
   private Hashtable savedState = new Hashtable();

   protected Memento() {}; //Give some protection

   protected void setState( String stateName, Object stateValue )
      {
      savedState.put( stateName, stateValue );
      }

   protected Object getState( String stateName)
      {
      return savedState.get( stateName);
      }

   protected Object getState(String stateName, Object defaultValue )
      {
      if ( savedState.containsKey( stateName ) )
         return savedState.get( stateName);
      else
         return defaultValue;
      }
   }
```

A Class whose state is saved

```
package Examples;
class ComplexObject
   {
   private String name;
   private int someData;
   private Vector objectAsState = new Vector();

   public Memento createMemento()
      {
      Memento currentState = new Memento();
      currentState.setState( "name", name );
      currentState.setState( "someData", new Integer(someData) );
      currentState.setState( "objectAsState", objectAsState.clone() );
      return currentState;
      }

   public void restoreState( Memento oldState)
      {
      name = (String) oldState.getState( "name", name );
      objectAsState = (Vector) oldState.getState( "objectAsState" );
      Integer data = (Integer) oldState.getState( "someData");
      someData = data.intValue();
      }
```

```
    // Show a way to do incremental saves
    public Memento setName( String aName )
        {
        Memento deltaState = saveAState( "name", name);
        name = aName;
        return deltaState;
        }


    public void setSomeData( int value )
        {
        someData = value;
        }

    private Memento saveAState(String stateName, Object
stateValue)
        {
        Memento currentState = new Memento();
        currentState.setState( stateName, stateValue );
        return currentState;
        }
    }
```

Consequences/ Implementation
Simplifies Originator

You may be tempted to let the originator manage its state history

This adds to the complexity of the Originator

- How to store state history and for how long?

Using Mementos might be expensive

Copying state takes time and space

If this takes too much time/space pattern may not be appropriate

Preserve encapsulation boundaries

Give Memento two interfaces: wide and narrow

Let originator have access to all set/get/state of Memento

Let others only hold Mementos and destroy them

Defining Narrow and Wide Interfaces C++

Make Memento's interface private

Make Originator a friend of the Memento

```
Class Memento {
public:
   virtual ~Memento();
private:
   friend class Originator;
   Memento();
   void setState(State*);
   State* GetState();
   …
```

Java[1]

Use private nested/inner class to hide memento's interface

```
class ComplexObject {
   private String name;
   private int someData;

   public Memento createMemento() {
      return new Memento();
   }

   public void  restoreState( Memento oldState) {
      oldState.restoreStateTo( this );
   }

   public class Memento {
      private String savedName;
      private int savedSomeData;

      private Memento() {
         savedName = name;
         savedSomeData = someData;
      }

      private void restoreStateTo(ComplexObject target) {
         target.name = savedName;
         target.someData = savedSomeData;
      }
   }
}
```

[1] RestoreStateTo does not access the fields of the outer object in case one wants to restore the state to a different ComplexObject object. One may wish to use an nested class to avoid tangling the memento to the outer object

Using Clone to Save State

One can wrap a clone of the Originator in a Memento or

Just return the clone as a type with no methods

```
interface Memento extends Cloneable { }
```

```
class ComplexObject implements Memento {
   private String name;
   private int someData;

   public Memento createMemento() {
      Memento myState = null;
      try {
         myState =  (Memento) this.clone();
      }
      catch (CloneNotSupportedException notReachable) {
      }
      return myState;
   }

   public void  restoreState( Memento savedState) {
      ComplexObject myNewState = (ComplexObject)savedState;
      name = myNewState.name;
      someData = myNewState.someData;
   }
}
```

Iterators & Mementos

Using a Memento we can allow multiple concurrent iterations

```
class IteratorState {
   int currentPosition = 0;

   protected IteratorState() {}

   protected int getPosition() {      return currentPosition;     }

   protected void advancePosition() { currentPosition++; }
   }

class Vector {
   protected Object elementData[];
   protected int elementCount;

   public IteratorState newIteration() { return new IteratorState(); }

   public boolean hasMoreElements(IteratorState aState) {
      return aState.getPosition() < elementCount;
   }

   public Object nextElement( IteratorState aState ) {
      if (hasMoreElements( aState ) ) {
         int currentPosition = aState.getPosition();
         aState.advancePosition();
         return elementData[currentPosition];
         }
   throw new NoSuchElementException("VectorIterator");
   }
   …
```

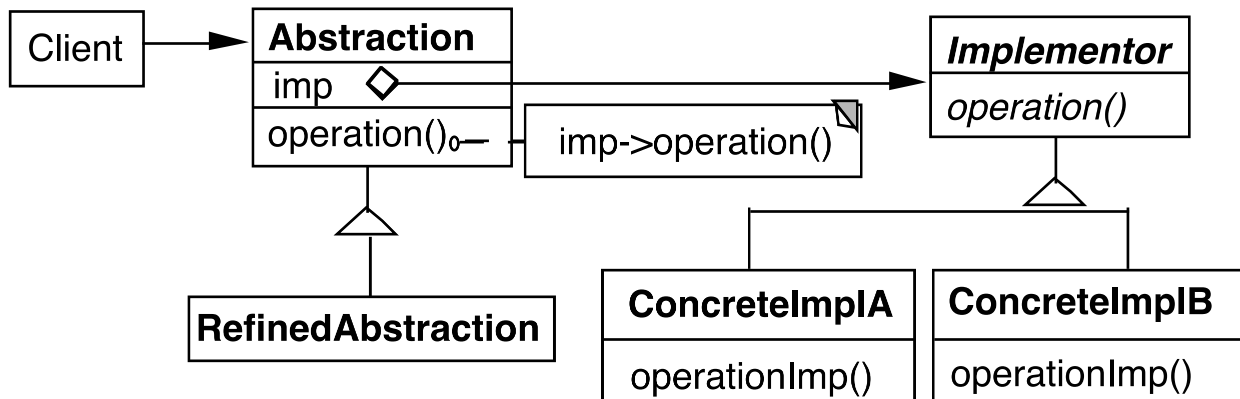
Bridge

Decouple the abstraction from its implementation

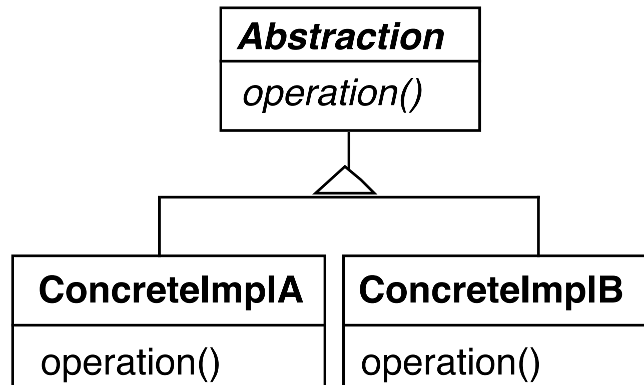
This allows the implementation to vary from its abstraction

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

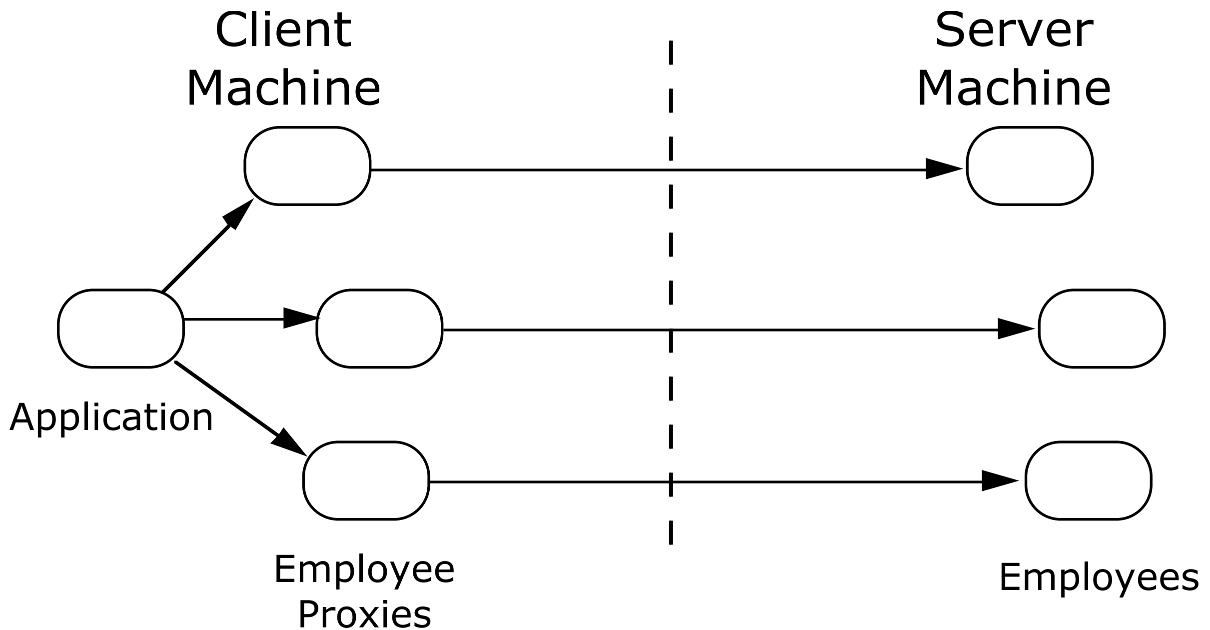
Java AWT uses Bridge to prevent programmer from accessing platform specific implementations of interface widgets, etc.

Peer = implementation

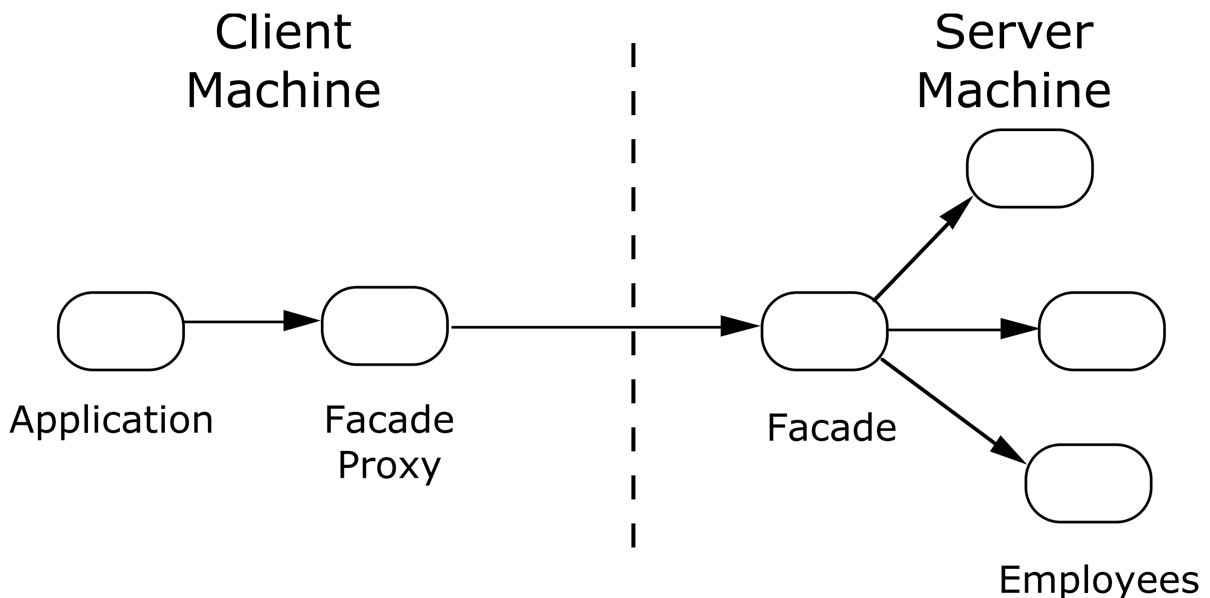
```
public synchronized void setCursor(Cursor cursor) {
   this.cursor = cursor;
   ComponentPeer peer = this.peer;
   if (peer != null) {
      peer.setCursor(cursor);
   }
}
```

Abstractions & Imps independently subclassable

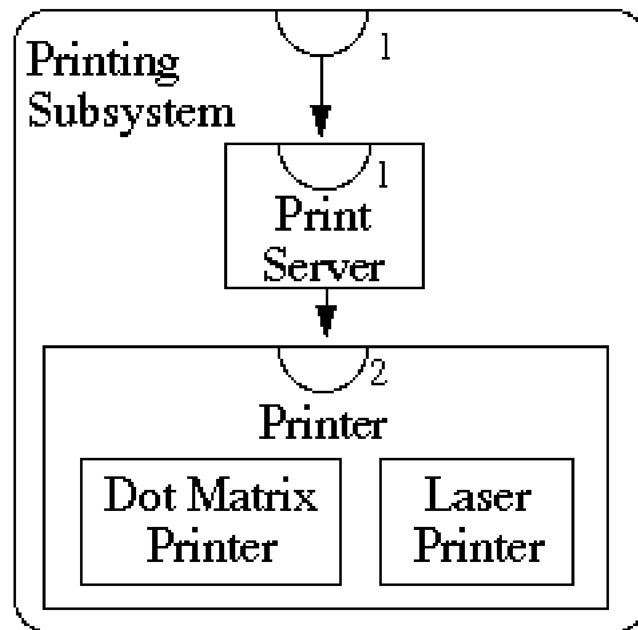
Start with Widow interface and two implementations:



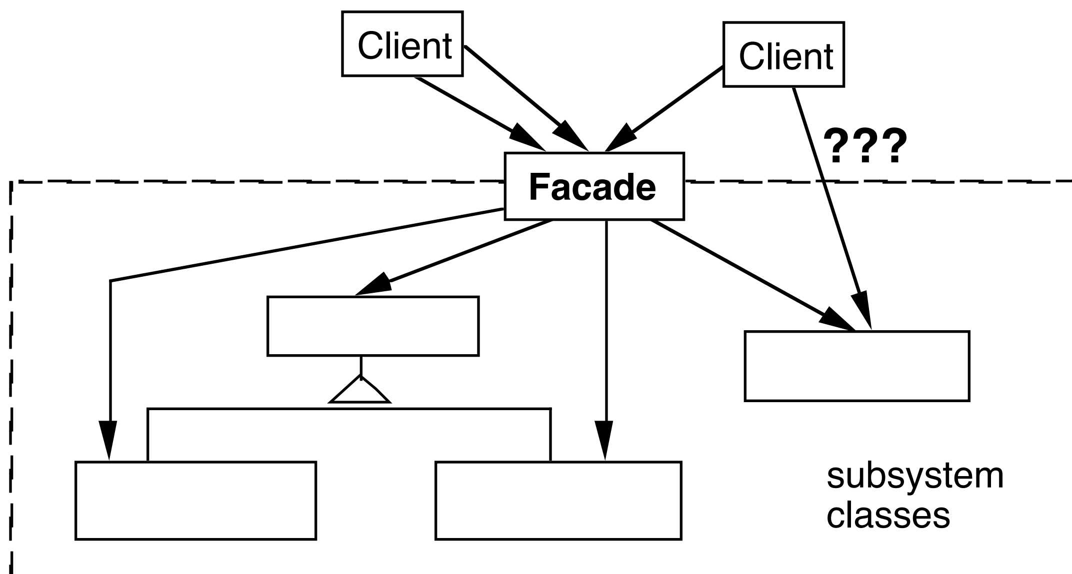
Now what do we do if we need some more types of windows: say IconWindow and DialogWindow?



Or using multiple inheritance



The Bridge pattern provides a cleaner solution



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

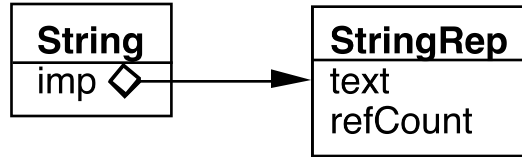
Methods in IconWindow and DialogWindow need to use the implementation methods to provide the new/modified functionality

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++

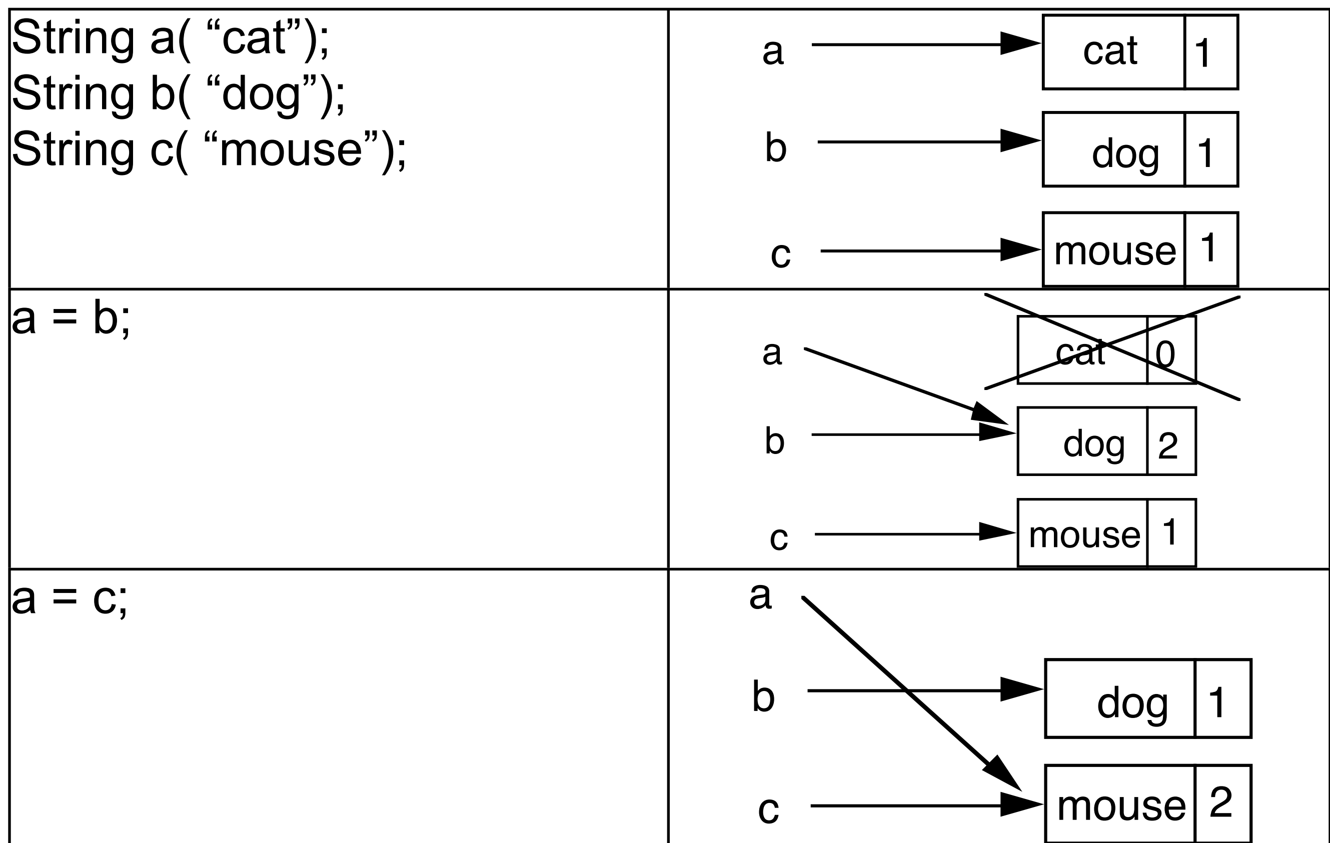


String contains a StringRep object

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



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();
}
```

Façade

Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

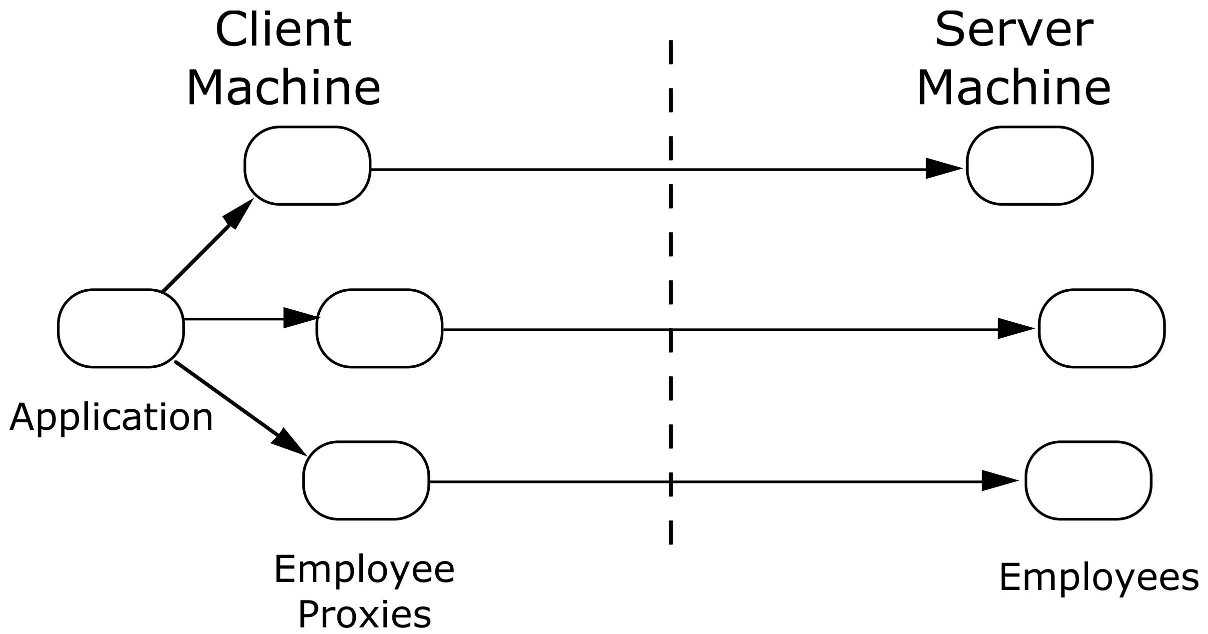
Programmers only use Compiler, which uses the other classes

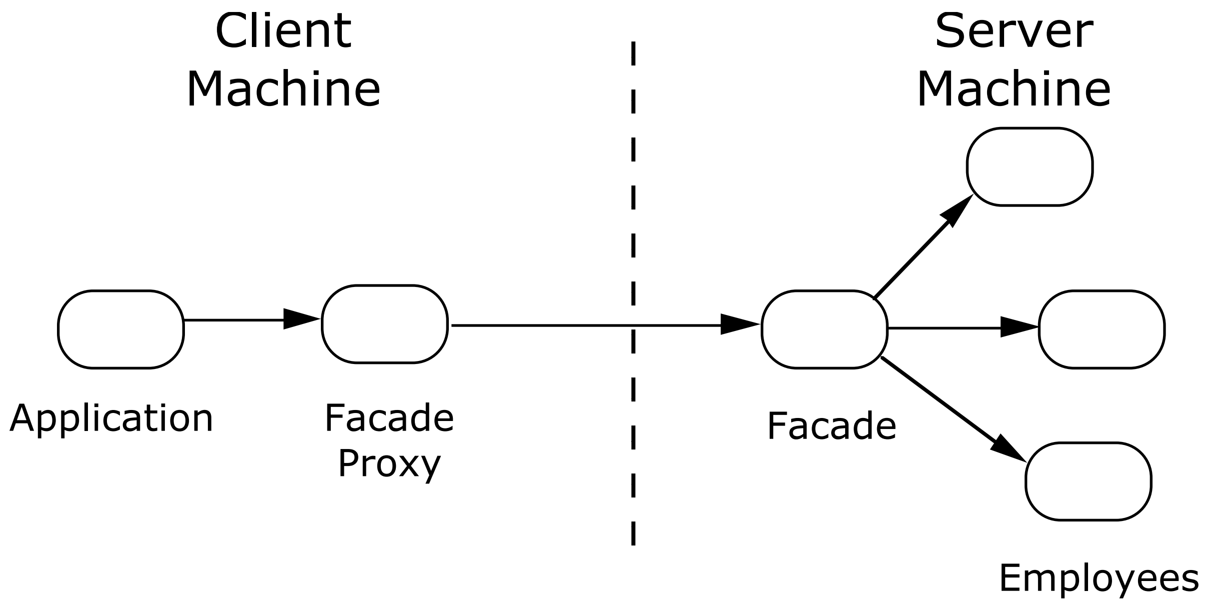
```
Compiler evaluate: '100 factorial'
```

```
| method compiler |
method := 'reset
   "Resets the counter to zero"
   count := 0.'.
```

```
compiler := Compiler new.
compiler
   parse:method
   in: Counter
   notifying: nil
```

Distributed Object Systems





Subsystems

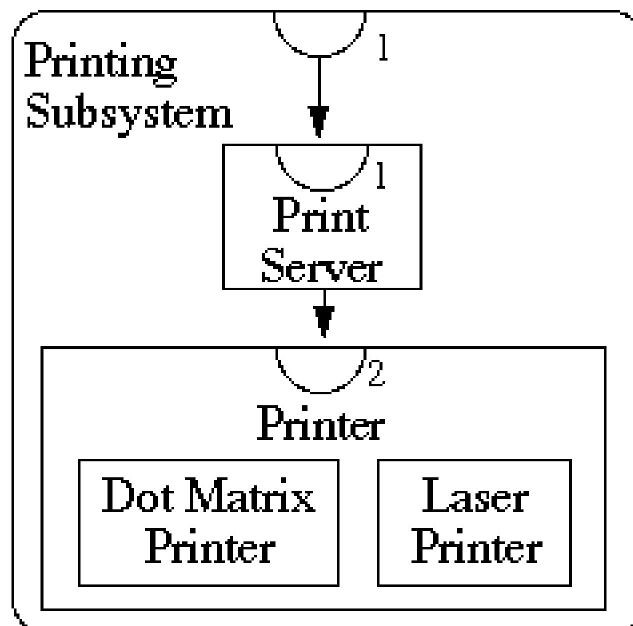
Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts

There is no conceptual difference between the responsibilities of a class and a subsystem of classes

The difference between a class and subsystem of classes is a matter of scale

A subsystem should be a good abstraction

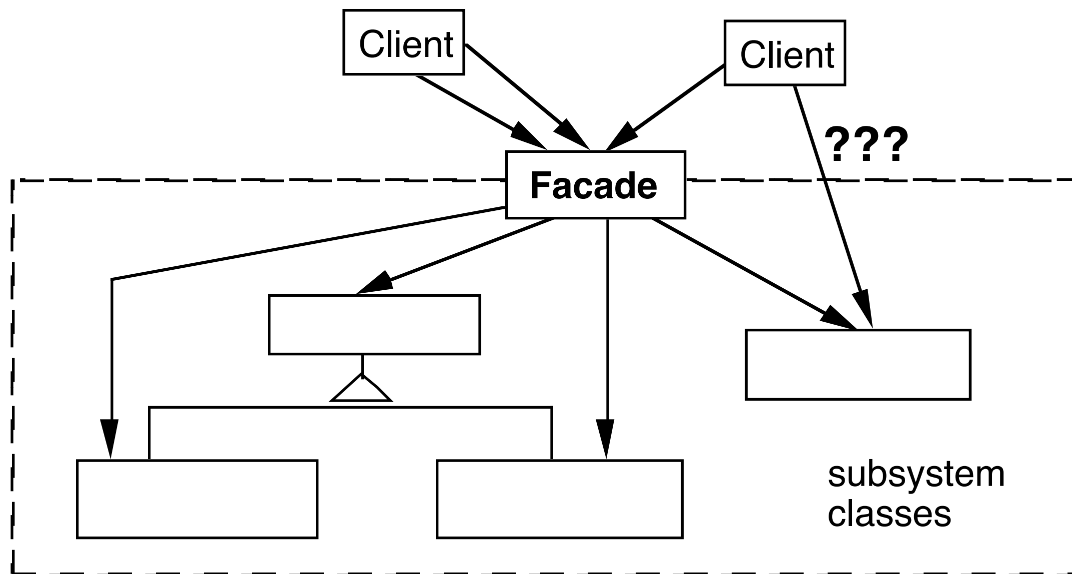
There should be as little communication between different subsystems as possible



The Facade Pattern - Basic Idea

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem