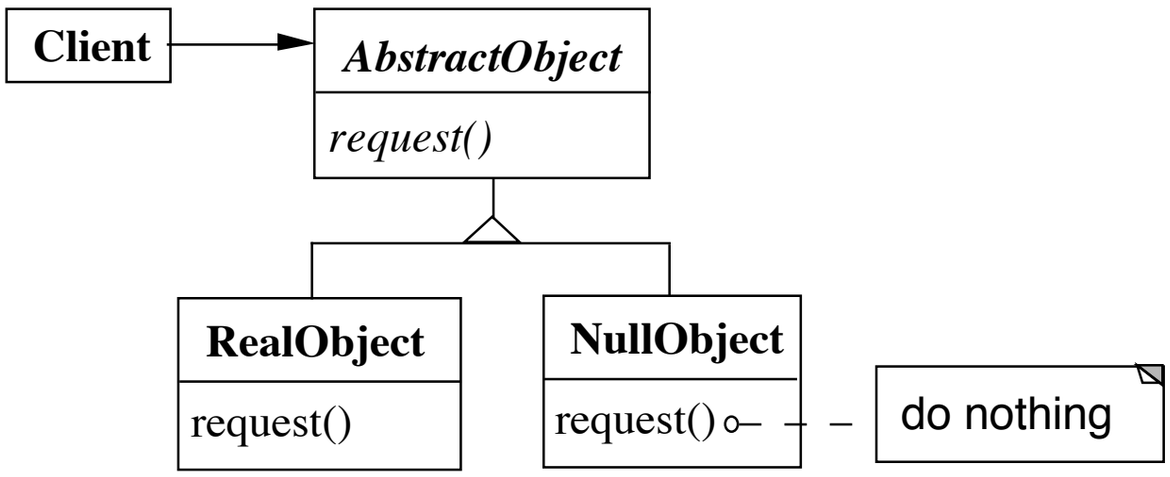
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2019

Doc 12 Null Object, Proxy, Bridge, Mockito
Oct 9, 2019

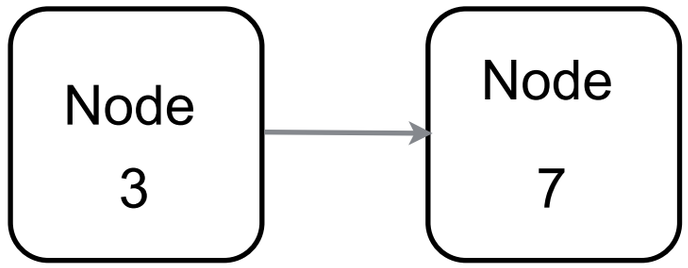
Null Object

Null Object



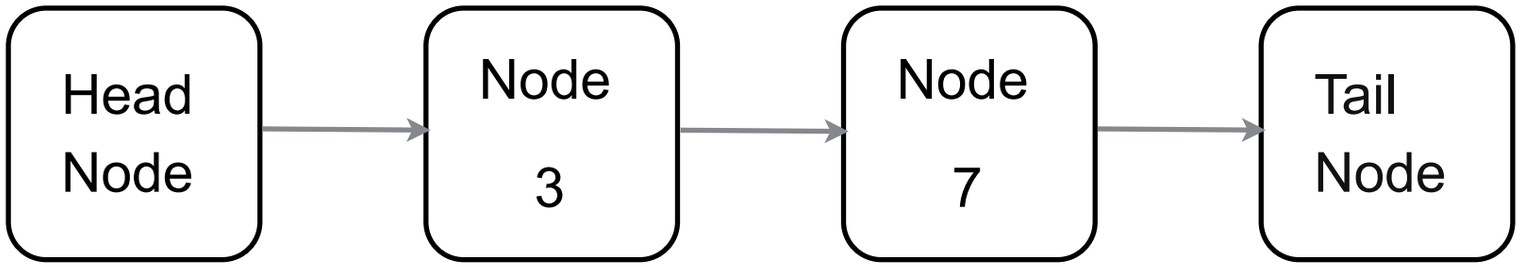
NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing




```
class LinkedList {
   Node head;

   public toString() {
      if (head == nil) {
         return “()”;
      }
      String listAsString = “(;
      Node current = head;
      while (current != null) {
         listAsString += current.value() + “, “;
         current = current.next;
      }
      listAsString = removetail(listsString, 2);
      return listAsString + “)”;
}
```




```
class LinkedList {
   Node head;

   public toString() {
      return head.toString();
   }
```

```
class HeadNode {
   public String toString() {
      return "(" + next.toString();
   }
}
```

```
class Node {
   public String toString() {
      return " " + element + next.toString();
   }
}
```

```
class TailNode {
   public String toString() {
      return " )";
   }
}
```

Smalltalk & Null Object

In Smalltalk

Everything is an object

Objects are instances of a class

Work is done by sending a message to an object

5 class SmallInteger

1.2 class Float

true class True

Smalltalk & Null Object

```
| a b |
a := 5.
b := 10.
(a < b) ifTrue: [ 'small'] ifFalse: ['larger'].
```

```
(a < b) ifTrue: [ 'small'] ifFalse: ['larger'].

 true ifTrue: [ 'small'] ifFalse: ['larger'].
```

```
True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock

        ^trueAlternativeBlock value
```

Smalltalk & Null Object

nil class UndefinedObject

We can add methods to the UndefinedObject class

Act as general Null Object

Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

- Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way

Applicability -When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable

Implementation

Too Many classes

Multiple Do-nothing meanings

Try Adapter pattern

Transformation to RealObject

Try Proxy pattern

Null (nil) as Null Object

Refactoring: Introduce Null Object

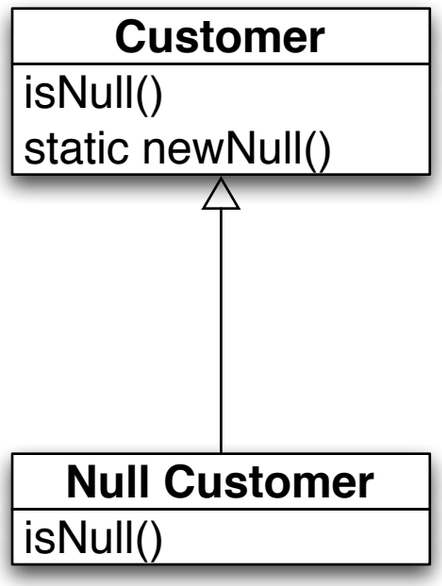
You have repeated checks for a null value

Replace the null value with a null object

```
if (customer == null)
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

Create Null Subclass




```
public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}
```

```
boolean isNull() { return true;}
```

Compile

Replace all nulls with null object

```
class SomeClassThatReturnCustomers {

   public Customer getCustomer() {
      if (_customer == null )
         return Customer.newNull();
      else
         return _customer;
      }
      etc.
}
```

Compile

Replace all null checks with isNull()

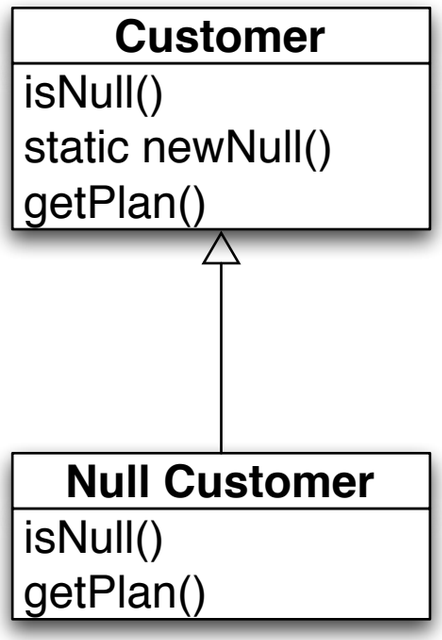
```
if (customer == null)
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

Compile and test

Find an operation clients invoke if not null
Add Operation to Null class

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```




```
class NullCustomer {
   public BillingPlan getPlan() {
      return BillingPlan.basic();
   }
```

Remove the Condition Check

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

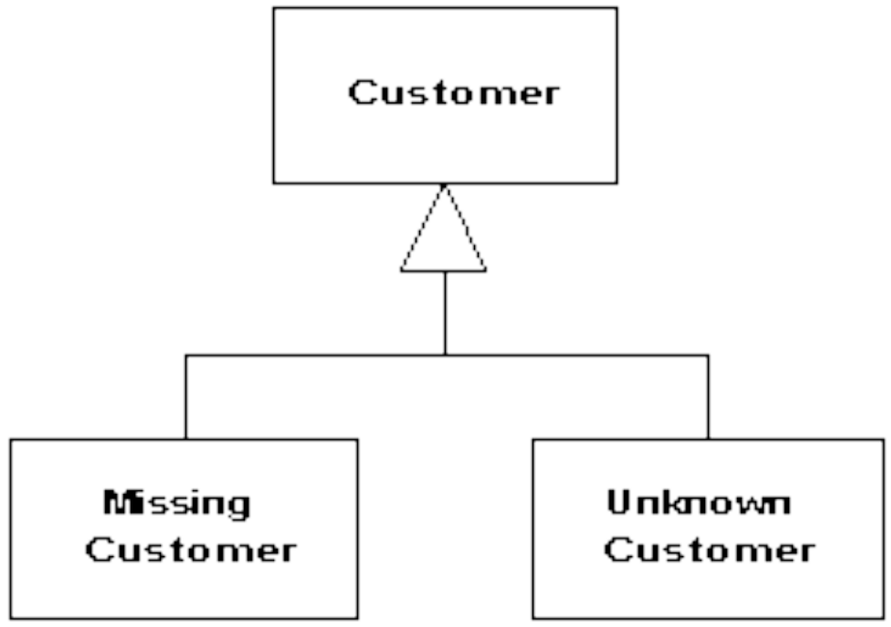
Compile & Test

Repeat last two slides for each operation
clients check if null

Special Case

Special Case

Represent special cases by a subclass



Use when multiple places that have same behavior

After conditional check for particular class instance

Or same behavior after a null check

Guards - Motivation

```
func calculateCost(price:Double,quantity:Int,taxRate:Double)->Double? {
    if price > 0 {
        if quantity > 0 {
            if taxRate >= 0 {
                return price * Double(quantity) * (1 + taxRate)
            } else {
                return nil
            }
        } else {
            return nil
        }
    } else {
        return nil
    }
}
```

Guard = if with just the else clause

```
func calculateCost2(price:Double,quantity:Int,taxRate:Double)->Double? {
    guard price > 0 else {
        return nil
    }

    guard quantity > 0 else {
        return nil
    }

    guard taxRate >= 0 else {
        return nil
    }

    return price * Double(quantity) * (1 + taxRate)
}
```

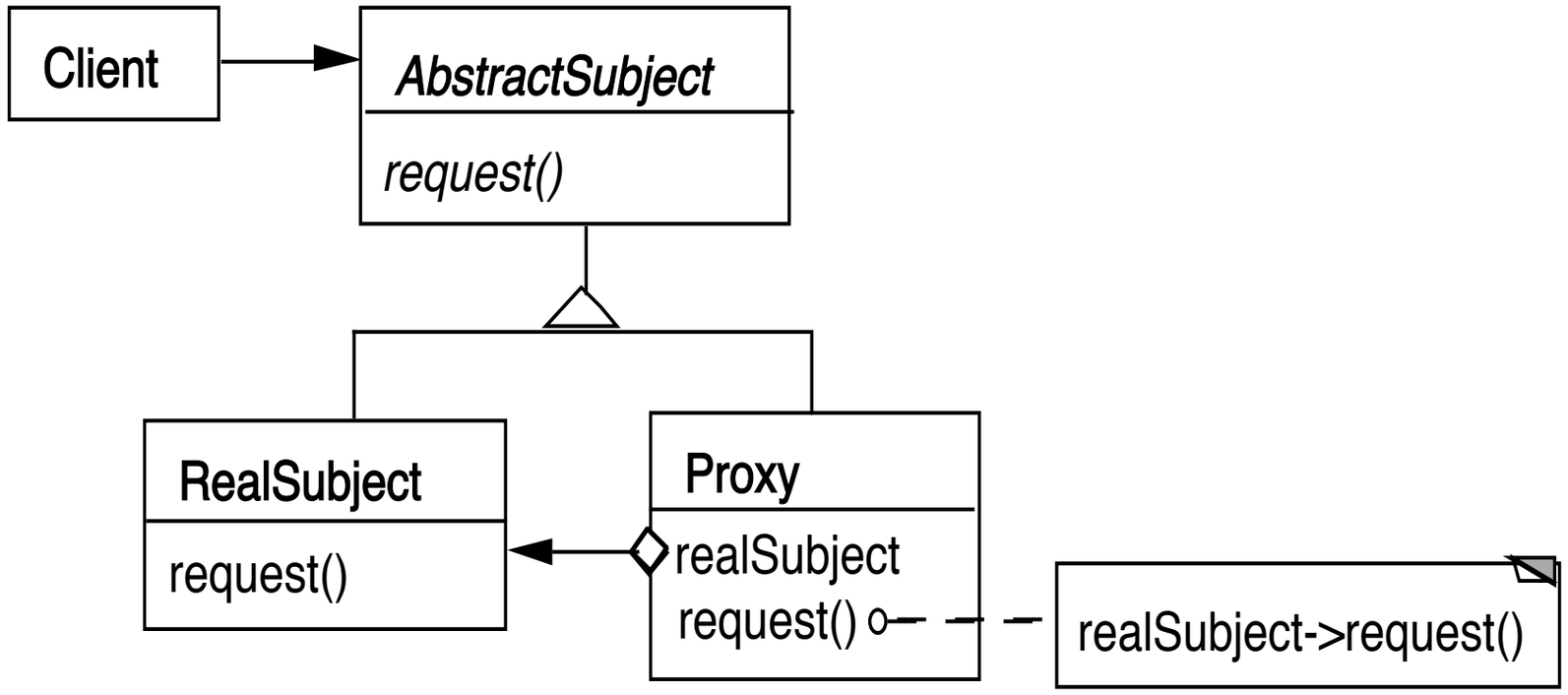
Shorter Version

```
func calculateCost3(price:Double,quantity:Int,taxRate:Double)->Double? {
    guard price > 0 && quantity > 0 && taxRate >= 0 else {
        return nil
    }
    return price * Double(quantity) * (1 + taxRate)
}
```

Proxy

Proxy (Surrogate)

a person authorized to act on behalf of another

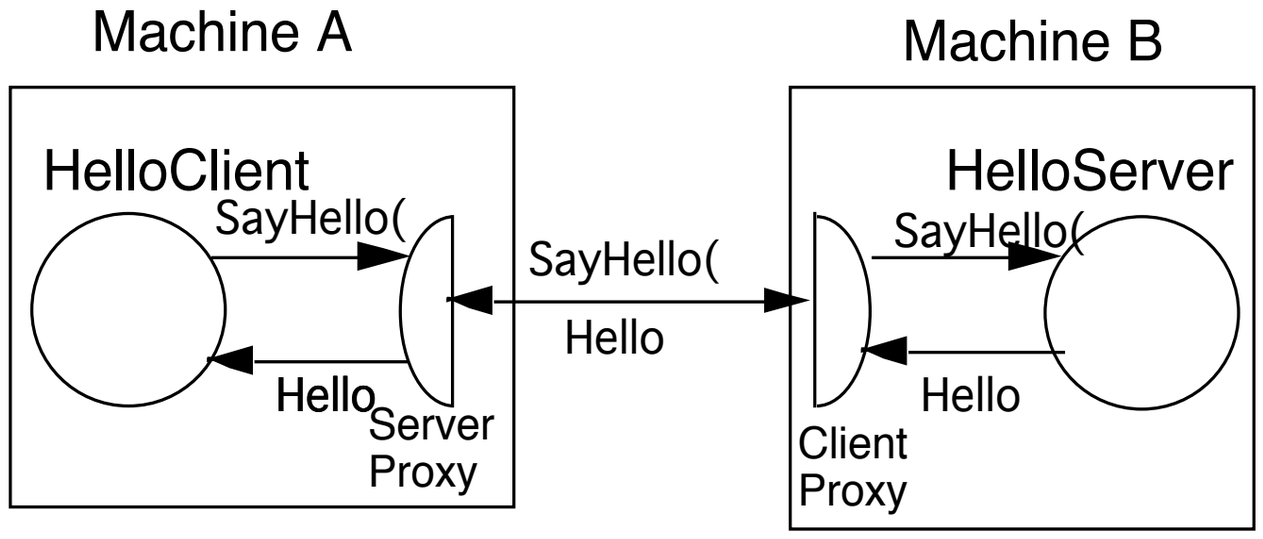



```
class Proxy {
   AbstractSubject realSubject;

   public Foo service(Bar x ) {
      return realSubject(x);
   }
```

Why do it?

Remote Proxy




```
String server = getHelloHostAddress( args);
Hello proxy = (Hello) Naming.lookup( server );
String message = proxy.sayHello();
System.out.println( message );
```

More General Proxy

```
class Proxy {
    AbstractSubject realSubject;

    public Foo service(Bar x ) {
        some preprocessing
        result = realSubject(x);
        some postprocessing
}
```

Virtual Proxy

Creates/accesses expensive objects on demand

O-R Mapping Layers

Java's Synchronized List

```
ArrayList notSafe = new ArrayList();
List threadSafe = Collections.synchronizedList(notSafe);
```

```
static class SynchronizedList {
    List list;
    public Object get(int index) {
synchronized(mutex) {return list.get(index);}
}
```

Java's Unmodifiable List

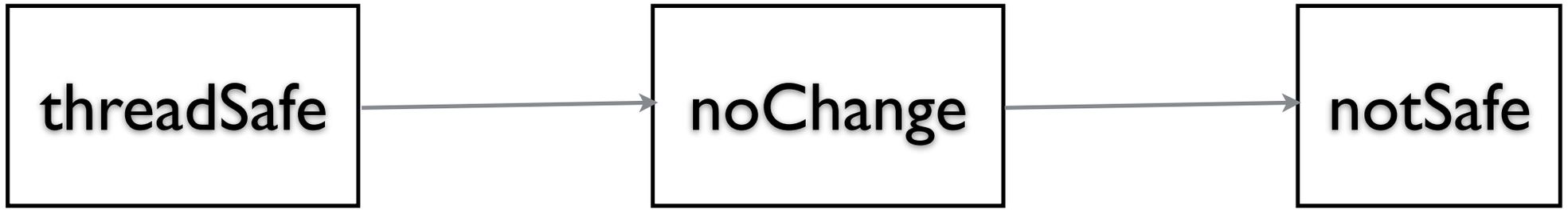
```
ArrayList notSafe = new ArrayList();
List noChange = Collections.unmodifiableList(notSafe);
```

```
static class UnmodifiableList {
    List list;
public Object get(int index) { return list.get(index);}

public Object set(int index, Object element) {
    throw new UnsupportedOperationException();
    }
```

Proxy or Decorator?

```
ArrayList notSafe = new ArrayList();
List noChange = Collections.unmodifiableList(notSafe);
List threadSafe = Collections.synchronizedList(noChange);
```



Proxy verses Decorator

"Decorators can have similar implementations as proxies"

Proxy controls access to an object

Decorator adds one or more responsibilities to an object

Mockito

Using Examples from

https://static.javadoc.io/org.mockito/mockito-core/2.23.0/org/mockito/Mockito.html

http://www.vogella.com/tutorials/Mockito/article.html

Mockito Examples - Class Used in Tests

```
public class Person {

    public int age() {
        return 4;
    }

    public String name() {
        return "Sam";
    }

    public String foo(String bar) {
        return "cat";
    }

    public void setAddress(String newAddress) {

    }
}
```

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class PersonTest {

    @Test
    public void testOneMethodCall()  {
    Person testPerson = mock(Person.class);

    when(testPerson.age()).thenReturn(43);

    assertEquals(testPerson.age(), 43);
}
```

Imports For All Examples

```
import org.junit.Rule;

import org.junit.jupiter.api.Test;
import org.junit.rules.ExpectedException;
import org.mockito.InOrder;

import java.util.List;
import java.util.Properties;
```

Multiple Calls

```
@Test
public void testMultipleMethodCall()  {

    Person testPerson = mock(Person.class);

    when(testPerson.age()).thenReturn(12)
                          .thenReturn(13);
    assertEquals(testPerson.age(), 12);
    assertEquals(testPerson.age(), 13);
    assertEquals(testPerson.age(), 13);
}
```

Throwing Exceptions

```
@Test
public void testExceptions() {
    Person testPerson = mock(Person.class);
    when(testPerson.age()).thenReturn(12)
            .thenThrow(new ArrayIndexOutOfBoundsException());

    assertEquals(12, testPerson.age());
    try {
        testPerson.age();
        fail( "Test fails" );
    } catch (ArrayIndexOutOfBoundsException expectedException) {
    }
}
```

Methods with Arguments

```
@Test
public void testReturnDependsonParameter() {
    Person testPerson = mock(Person.class);

    when(testPerson.foo("cat")).thenReturn("mouse");
    when(testPerson.foo("dog")).thenReturn("bone")
                                        .thenReturn("sleep");
    assertEquals("mouse", testPerson.foo("cat"));
    assertEquals("bone", testPerson.foo("dog"));
    assertEquals("sleep", testPerson.foo("dog"));
    assertEquals(null, testPerson.foo("rat"));
}
```

Order Of Calls

```
@Test
public void testOrder() {
    List singleMock = mock(List.class);

    singleMock.add("was added first");
    singleMock.add("was added second");

    InOrder inOrder = inOrder(singleMock);

    //Make sure that add is first called with "was added first, then with "was added second"
   inOrder.verify(singleMock).add("was added first");
   inOrder.verify(singleMock).add("was added second");

}
```

Order Of Calls - Show Failure

```
@Test
public void testOrder() {
    List singleMock = mock(List.class);

    singleMock.add("was added first");
    singleMock.add("was added second");

    //create an inOrder verifier for a single mock
    InOrder inOrder = inOrder(singleMock);

    // Test Fails
     inOrder.verify(singleMock).add("was added second");
     inOrder.verify(singleMock).add("was added first");

}
```

Order Of Calls - Multiple Objects

```
@Test
public void testOrderTwoObjects() {
    List firstMock = mock(List.class);
    List secondMock = mock(List.class);

    firstMock.add("was called first");
    secondMock.add("was called second");

    InOrder inOrder = inOrder(firstMock, secondMock);

    //following will make sure that firstMock was called before secondMock
    inOrder.verify(firstMock).add("was called first");
    inOrder.verify(secondMock).add("was called second");
}
```

Verify Times Method Called

```
@Test
 public void testAddressCall() {
     Person testPerson = mock(Person.class);
     testPerson.setAddress("A");
     testPerson.setAddress("B");
     testPerson.setAddress("C");
     testPerson.setAddress("D");
     testPerson.setAddress("C");
     verify(testPerson).setAddress("A");
     verify(testPerson).setAddress("B");

     verify(testPerson, times(2)).setAddress("C");
     verify(testPerson, never()).setAddress("Z");
     verify(testPerson, never()).age();
     verify(testPerson, atLeast(2)).setAddress("C");
     verify(testPerson, atMost(1)).setAddress("A");
}
```

Mocking Methods that Can Throw Exception

```
@Test
public void TestProperties() {
    Properties properties = new Properties();

    when(properties.get("showSize")).thenReturn("42");

    assertEquals("42", properties.get("showSize"));
}
```

get method can throw an exception

So above test crashes

Mocking Methods that Can Throw Exception

```
@Test
public void testProperties() {
    Properties properties = mock(Properties.class);

    doReturn("42").when(properties).get("showSize");

    assertEquals("42", properties.get("showSize"));
}
```

Mocking Methods that Can Throw Exception

```
@Test
public void testProperties() {
    Properties properties = mock(Properties.class);

    doReturn("42","24","12").when(properties).get("showSize");

    assertEquals("42", properties.get("showSize"));
    assertEquals("24", properties.get("showSize"));
    assertEquals("12", properties.get("showSize"));
}
```

doReturn Can Be Used All the Time

```
@Test
public void testReturnDependsonParameter() {
    Person testPerson = mock(Person.class);

    doReturn("mouse").when(testPerson).foo("cat");
    doReturn("bone","sleep").when(testPerson).foo("dog");

    assertEquals("mouse", testPerson.foo("cat"));
    assertEquals("bone", testPerson.foo("dog"));
    assertEquals("sleep", testPerson.foo("dog"));
    assertEquals(null, testPerson.foo("rat"));
}
```

doReturn vs when

```
doReturn("mouse").when(testPerson).foo("cat");
doReturn("bone","sleep").when(testPerson).foo("dog");
```

```
when(testPerson.foo("cat")).thenReturn("mouse");
when(testPerson.foo("dog")).thenReturn("bone")
                                   .thenReturn("sleep");
```

doReturn

- Handles methods that can throw exceptions

when

- Perhaps more natural English order

Mocking Methods that Can Throw Exception

```
@Test
public void TestWhenMethodCanThrowException() {
    Properties properties = new Properties();

    Properties spyProperties = spy(properties);

    doReturn("42").when(spyProperties).get("shoeSize");

    Object value = spyProperties.get("shoeSize");

    assertEquals("42", value);
}
```

mock vs spy

```
@Test
public void testMockVErsesSpy() {
    Person testPerson = mock(Person.class);
    assertEquals(null, testPerson.name());

    Person realPerson = new Person();
    Person spyPerson = spy(realPerson);
    assertEquals("Sam", spyPerson.name());
}
```

mock

- Does not call methods on real object

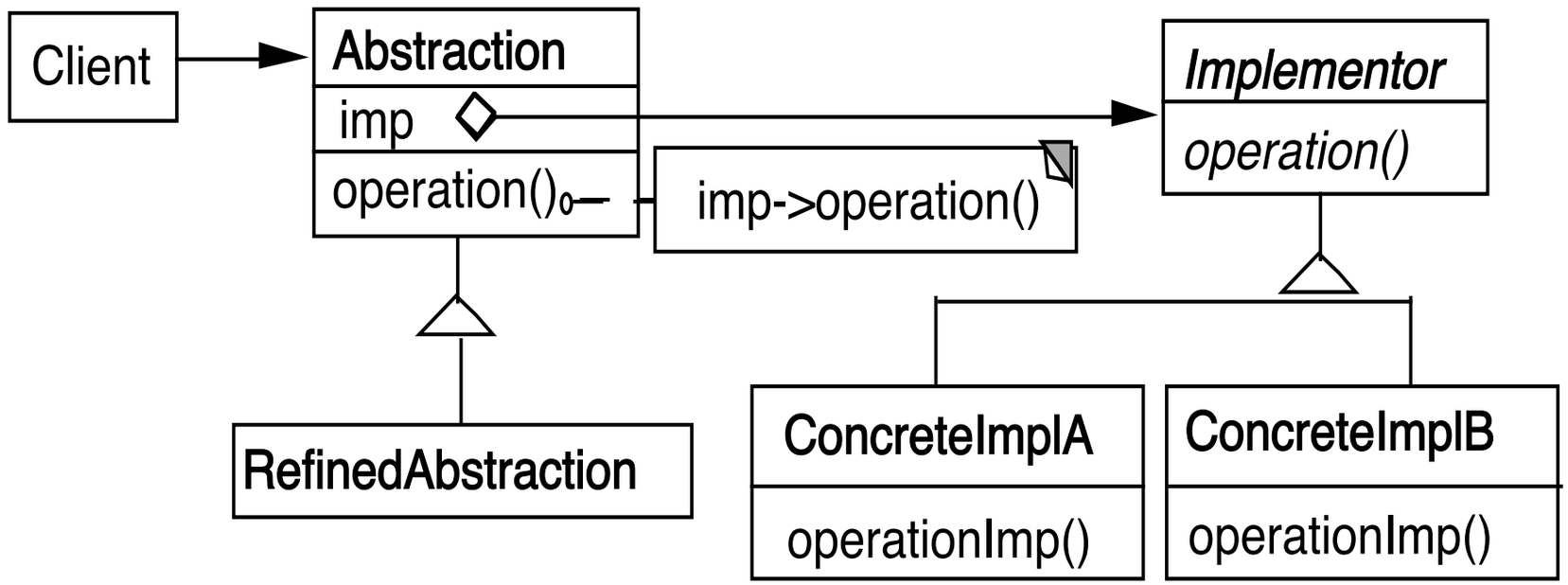
spy

- Calls methods on real object
- Allows to mock some methods on object

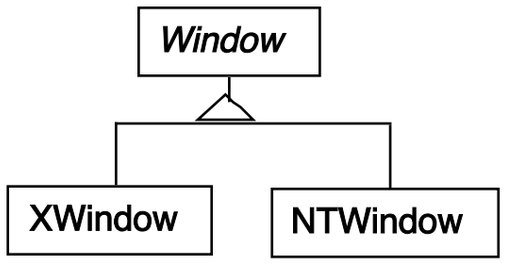
Bridge

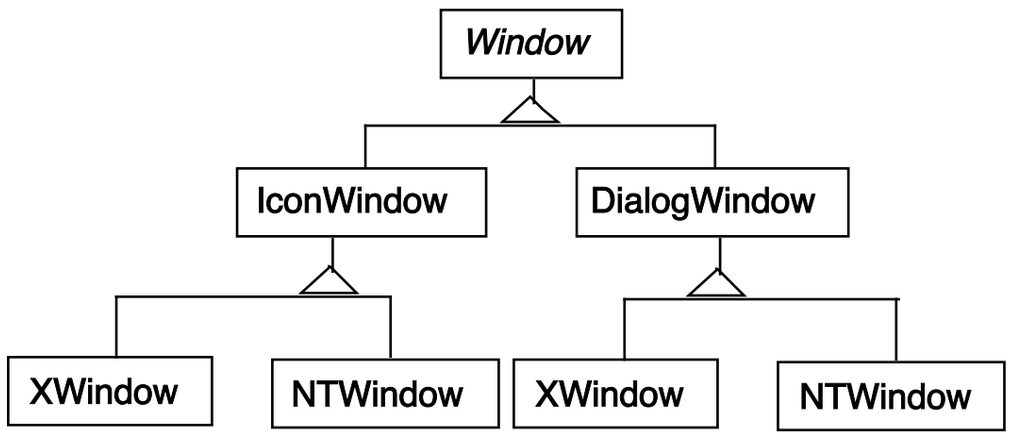
Bridge

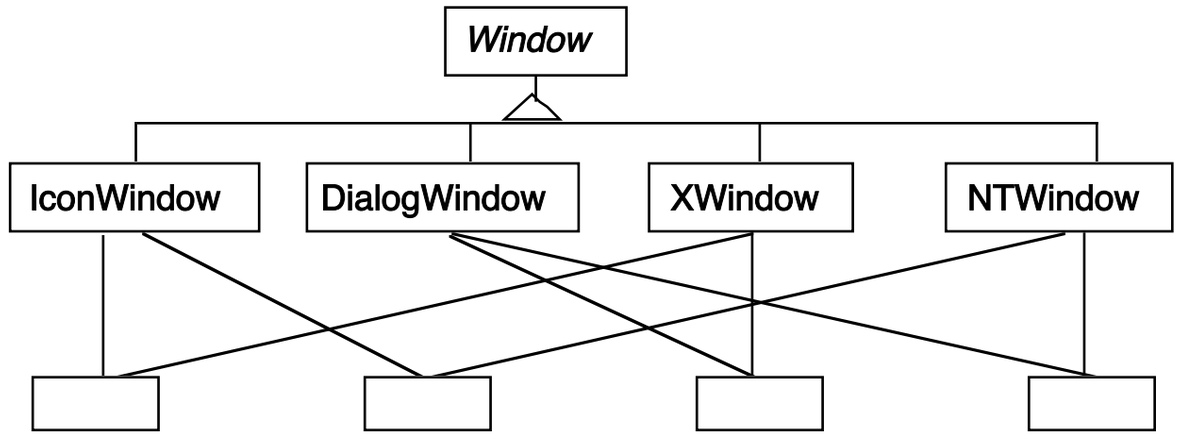
Decouple an abstraction from its implementation



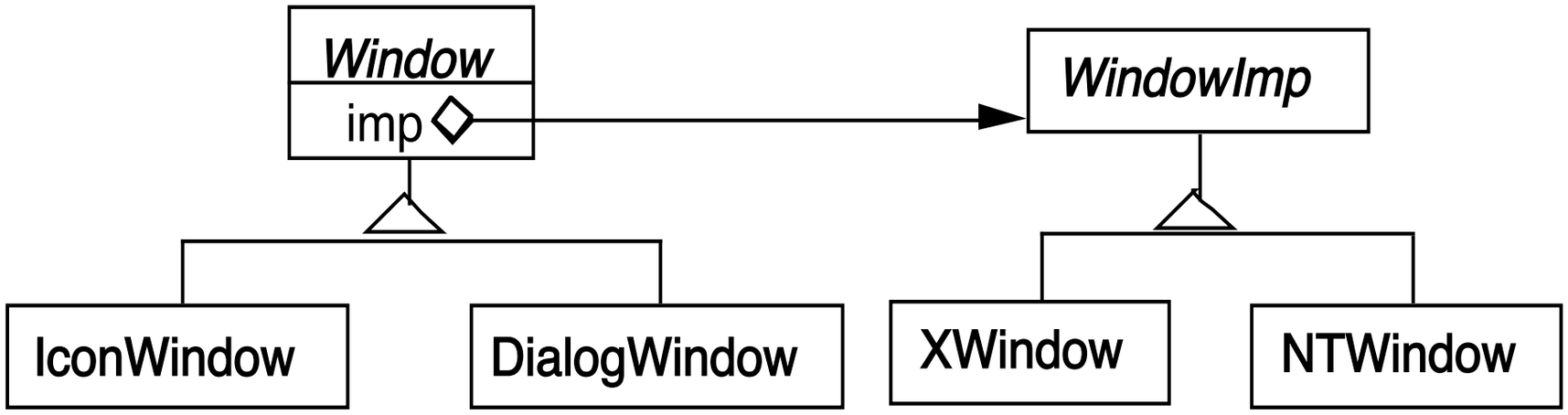
Windows


Window
XWindow
NTWindow


Window
IconWindow
DialogWindow
XWindow
NTWindow
XWindow
NTWindow


Window
IconWindow
DialogWindow
XWindow
NTWindow

Using the Bridge Pattern



Peers in Java's AWT

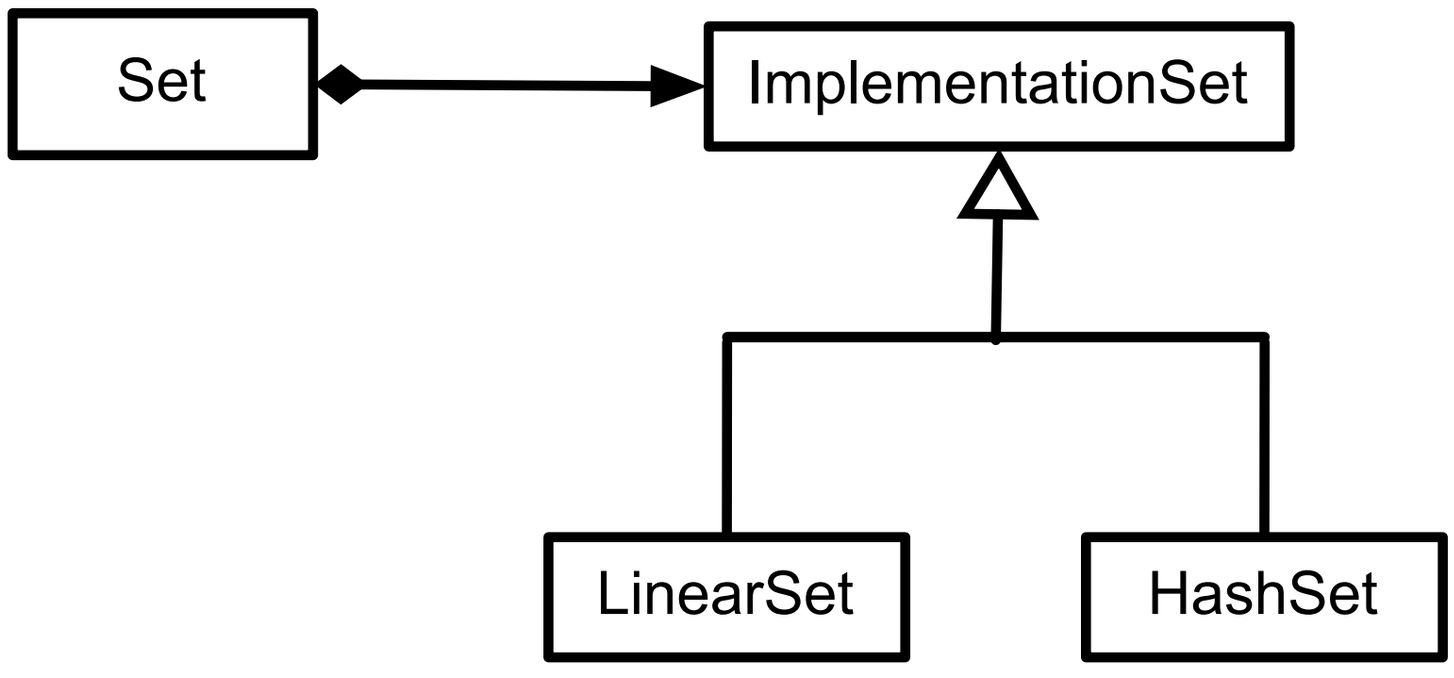
java.awt.
Button

peer

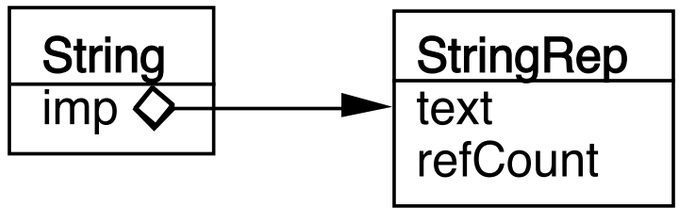
Peer = implementation

```
    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;
};
```

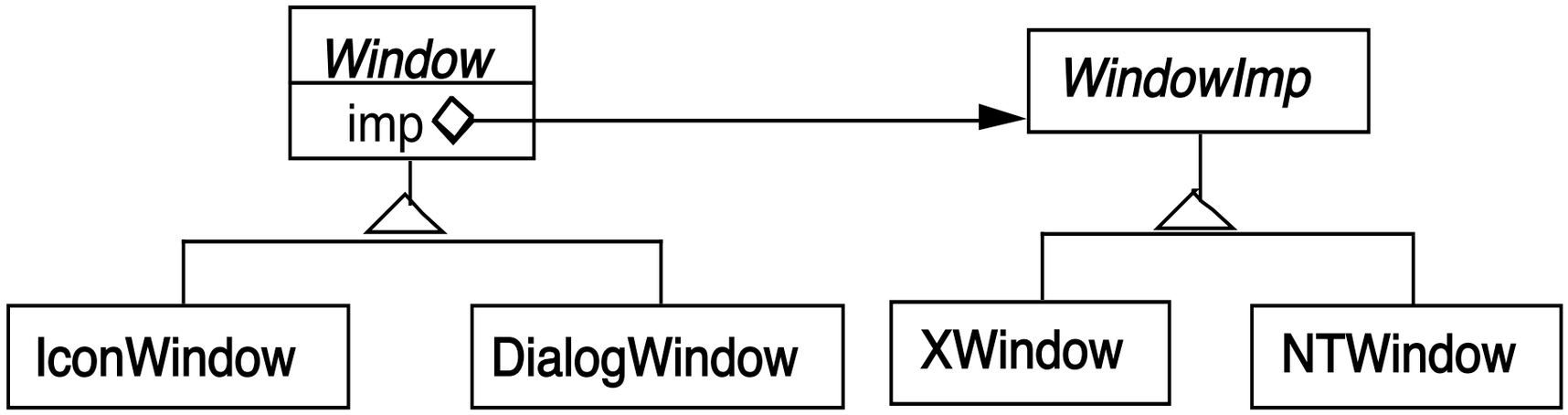
Why Use Bridge

Implementation selected at run-time

Implementation changed during run-time

Why Use Bridge

Abstraction & implementations are extensible by subclassing

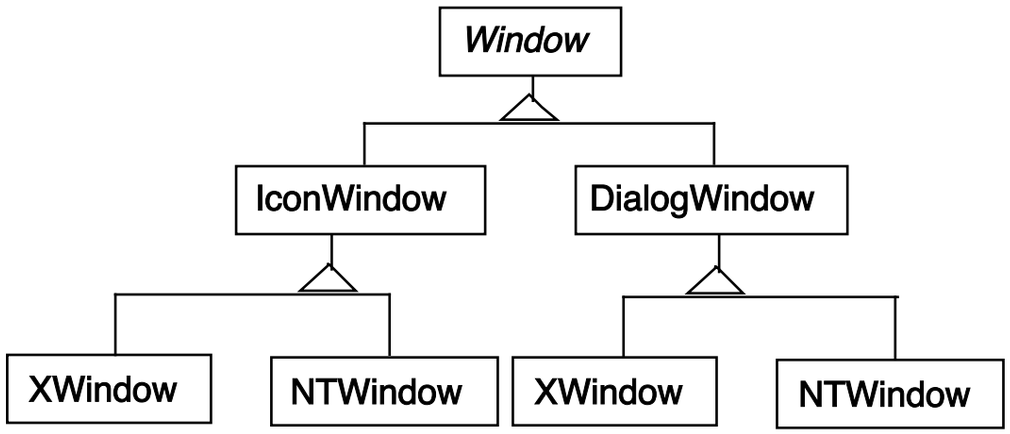


Why Use Bridge

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

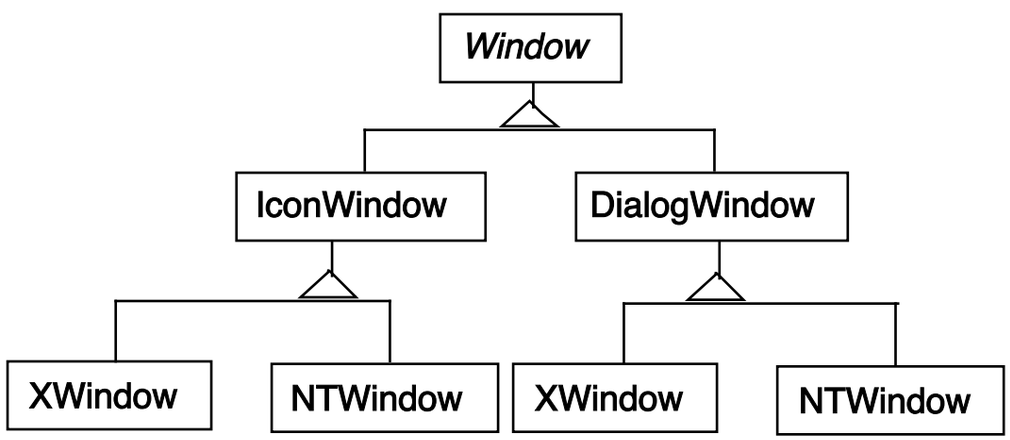
Why Use Bridge

Proliferation of classes



Why Use Bridge

Share implementation among multiple objects



Bridge verses Adapter