CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2018
Doc 17 Mock Objects, Bridge
Nov 8, 2018

Copyright ©, All rights reserved. 2017 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.

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);
    }
```

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

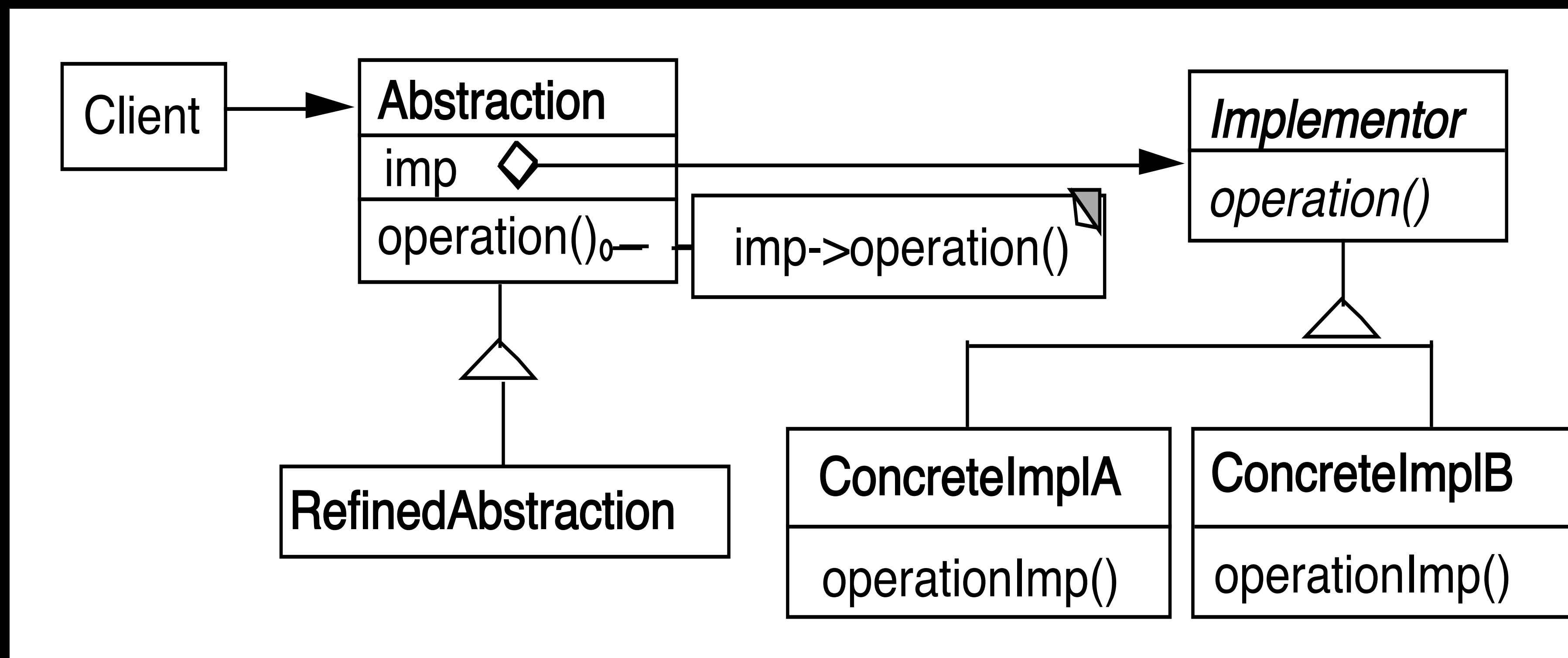
Calls methods on real object

Allows to mock some methods on object

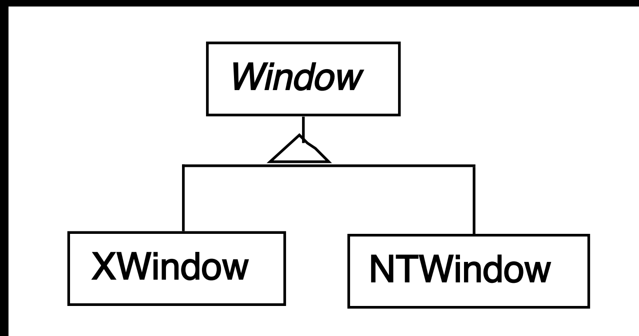
Bridge

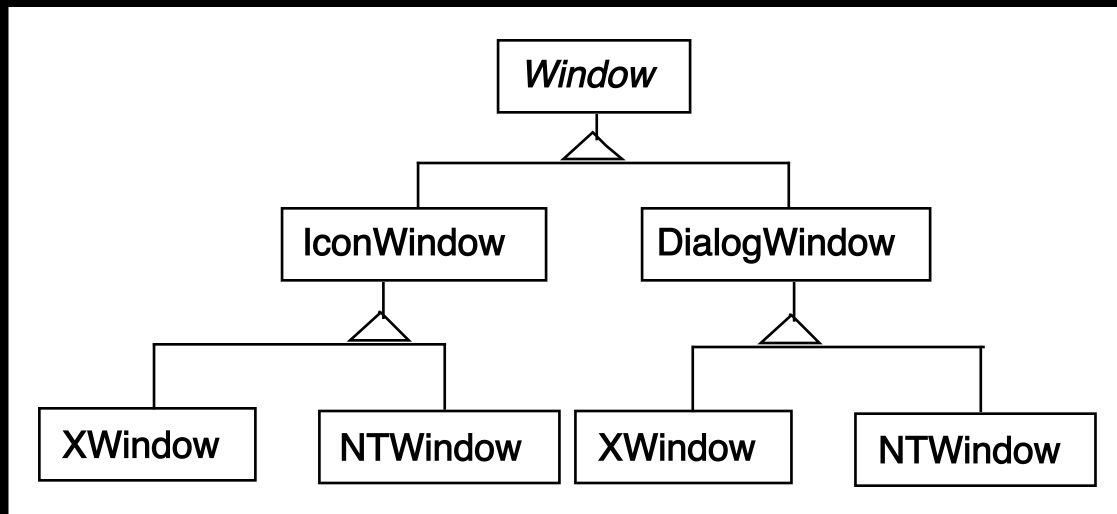
Bridge

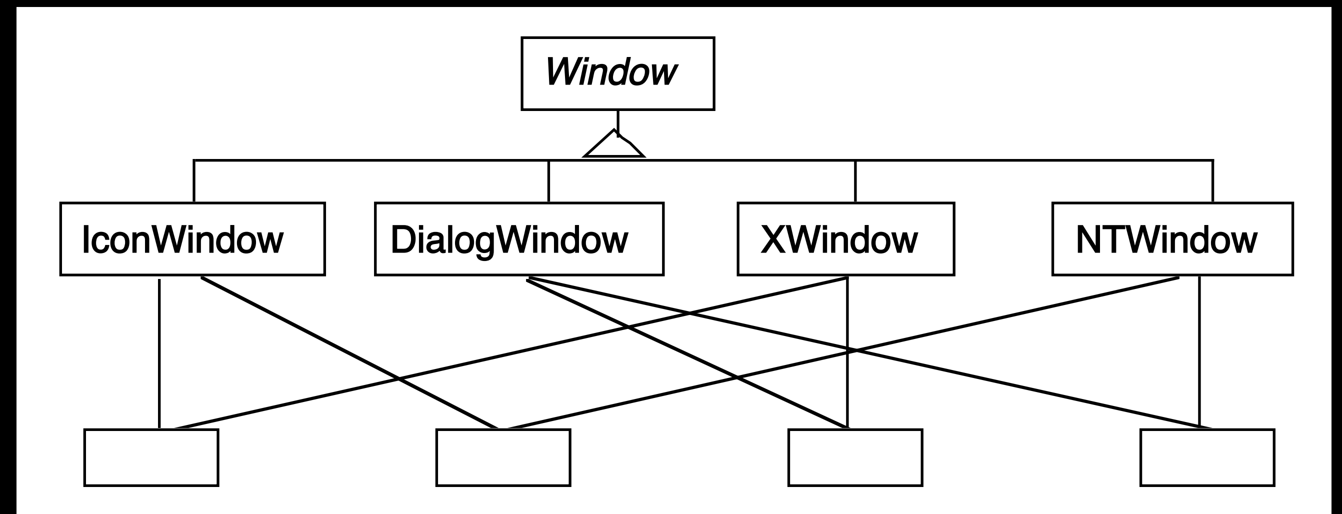
Decouple an abstraction from its implementation



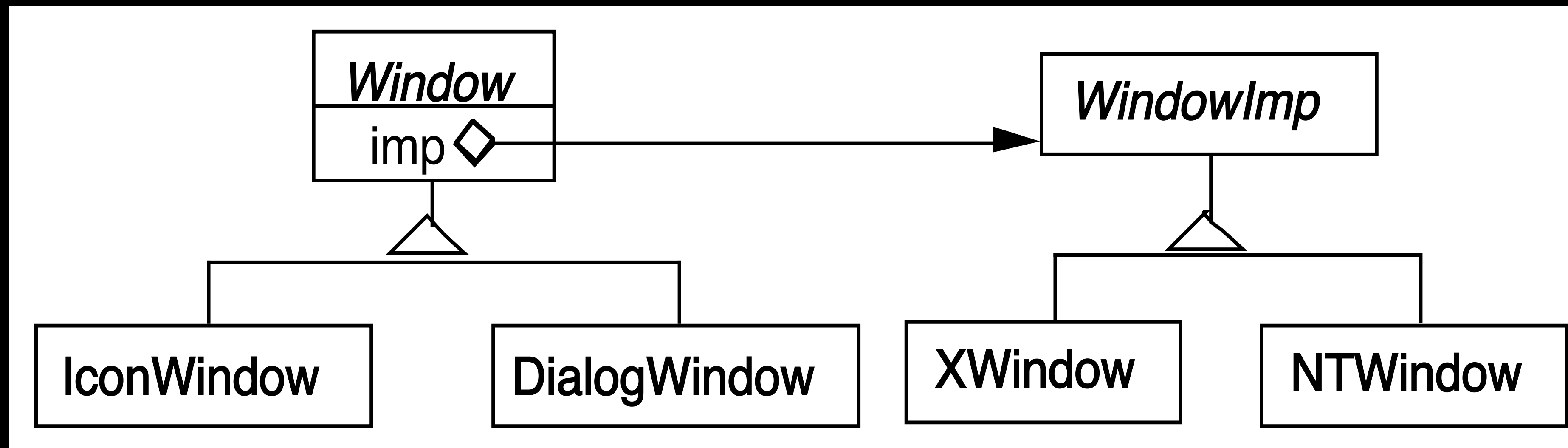
Windows



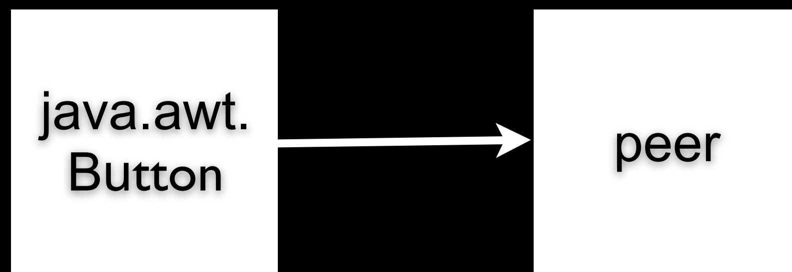




Using the Bridge Pattern



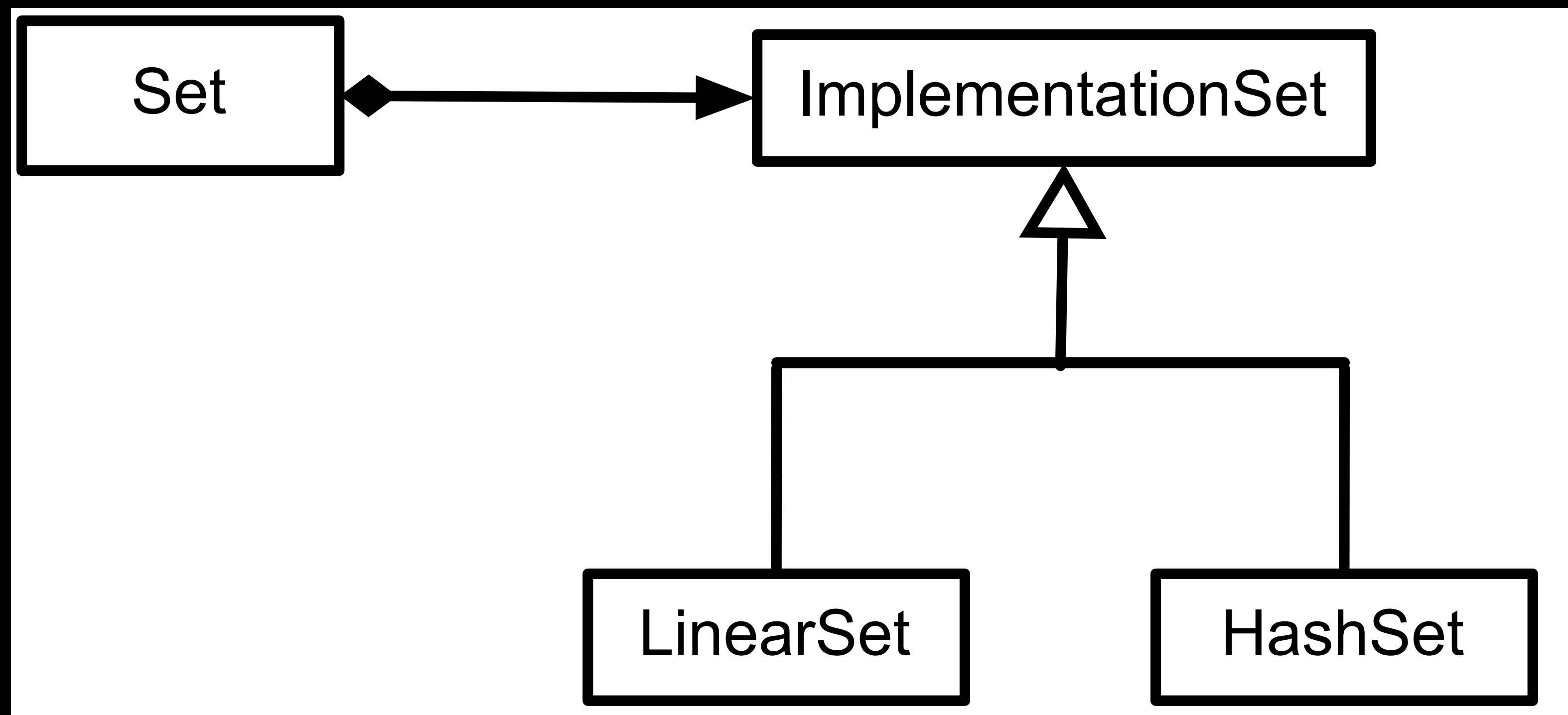
Peers in Java's AWT



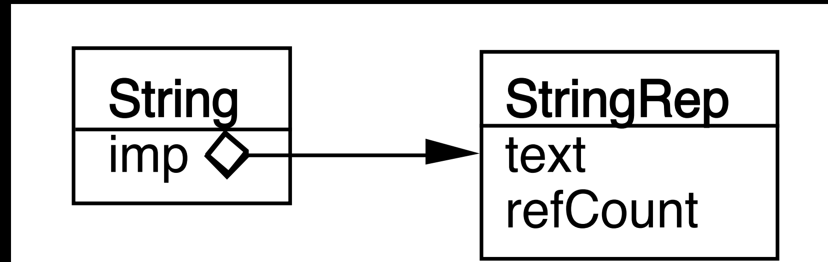
Peer = implementation

```
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”);	

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;
};
```

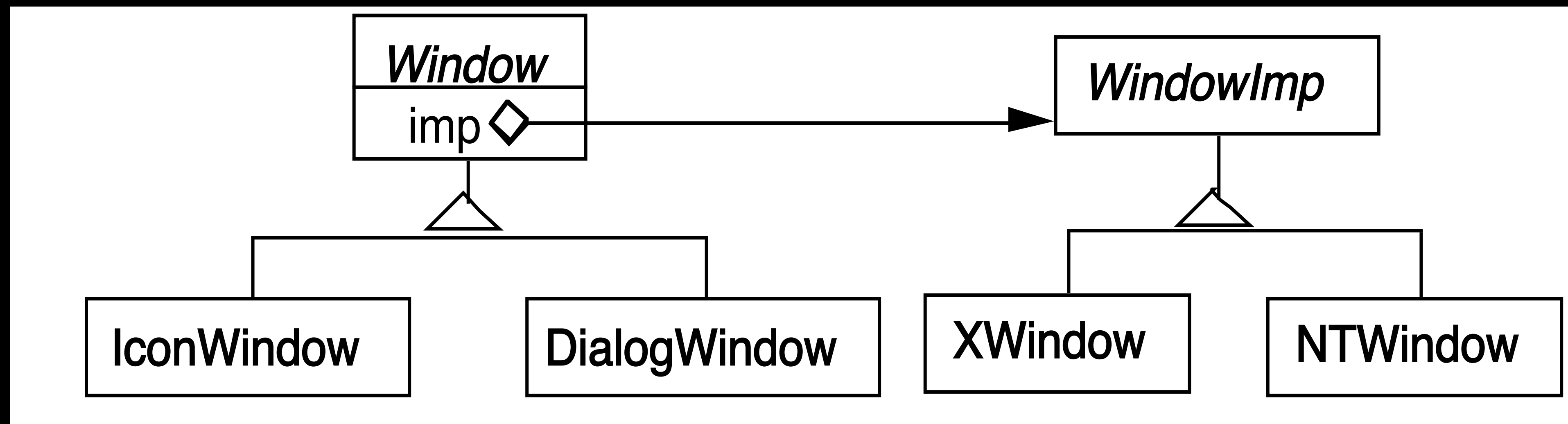
Why Use Bridge

Implementation selected at run-time

Implementation changed during run-time

Why Use Bridge

Abstraction & implementations are extensible by subclassing

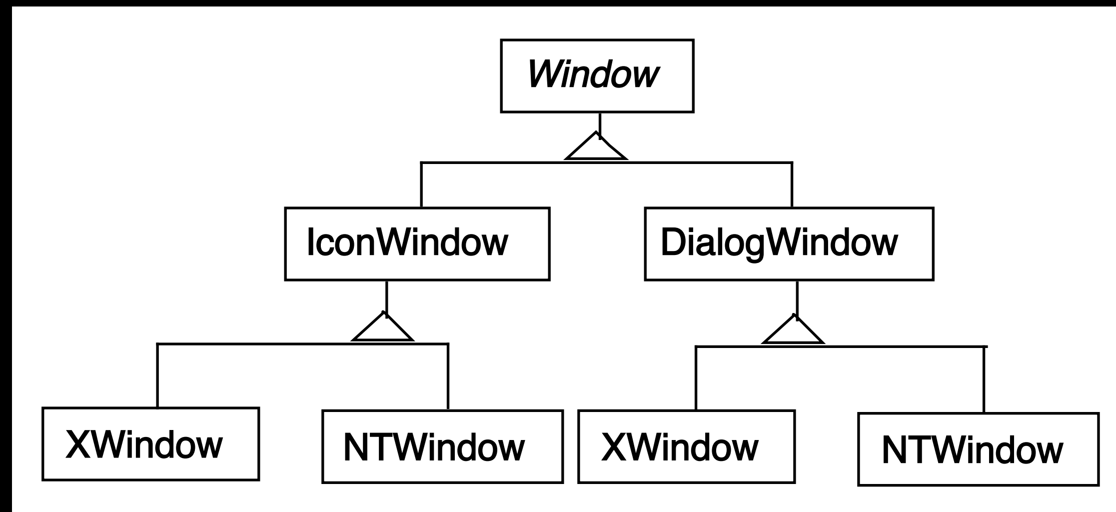


Why Use Bridge

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

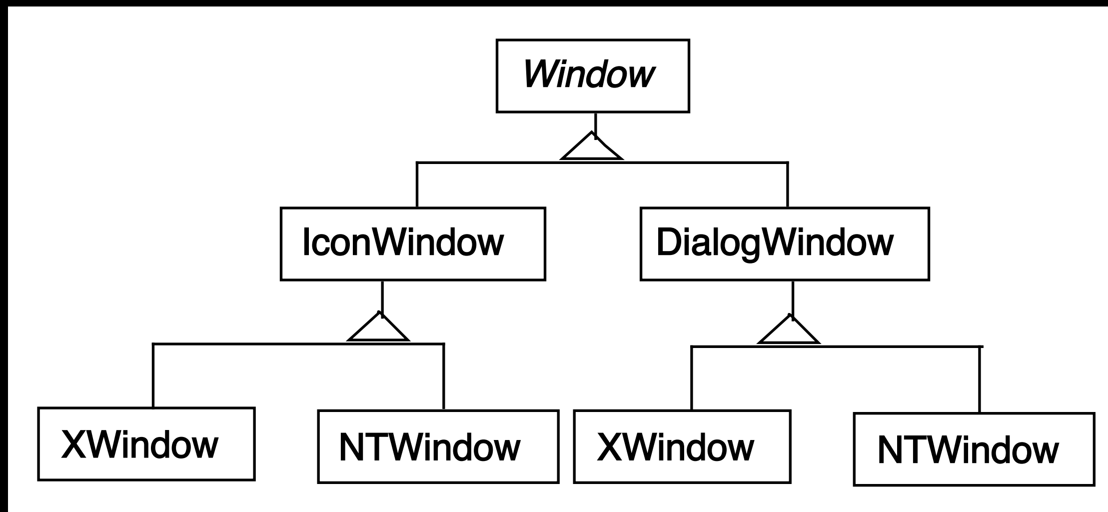
Why Use Bridge

Proliferation of classes



Why Use Bridge

Share implementation among multiple objects



Bridge verses Adapter