Just to warm up (15 minutes)

1 - Define a class Student, Person, PostgradStudent and PregradStudent

2 - Define a way to compare Person

3 - Show me that you understand what *this* and *super* are

Dealing with objects
Part III

Alexandre Bergel
abergel@dcc.uchile.cl
09/08/2012

Goal of this lecture

Understanding some of the *design rules* that govern *inheritance*

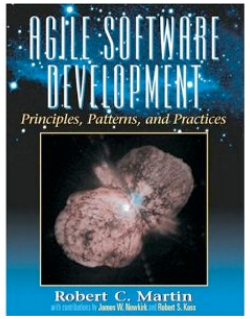
See a *bit* of *theory*

See practical problems of class inheritance

Recommended Texts

Agile Software Development, Principles, Patterns, and Practices

Robert C. Martin “Uncle Bob”, 2002



Outline

1. Liskov principle
    1. theory
    2. concrete applications
2. Inheritance examples
3. Example1: Swing and AWT
4. Example2: The Smalltalk collection class hierarchy

Outline

1. **Liskov principle**
    1. **theory**
    2. **concrete applications**
2. Inheritance examples
3. Example1: Swing and AWT
4. Example2: The Smalltalk collection class hierarchy

Liskov substitution principle

Initially introduced in 1987 by Barbara Liskov

Formulated in 1994 with Jeannette Wing as follows:

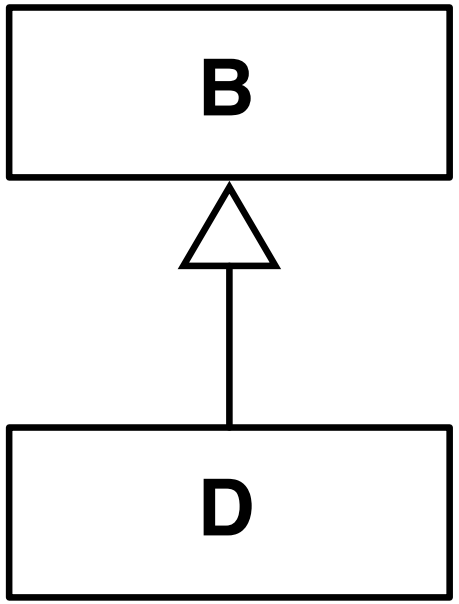
Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

Liskov principle vulgarized

Subtypes must be substitutable for their base types

Liskov principle vulgarized

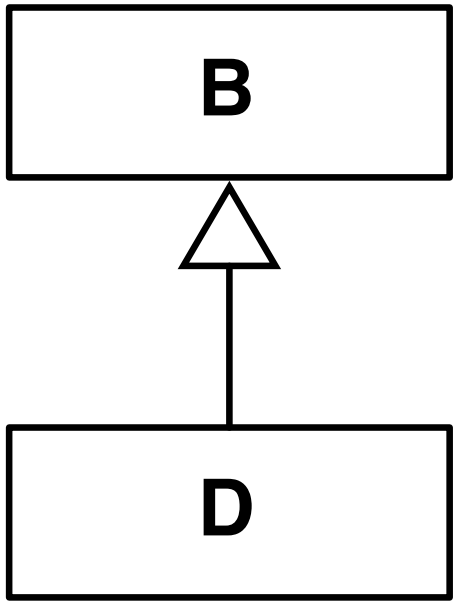
```
void f (B object) {
   ...
}
```



Liskov principle vulgarized

```
void f (B object) {
   ...
}
```

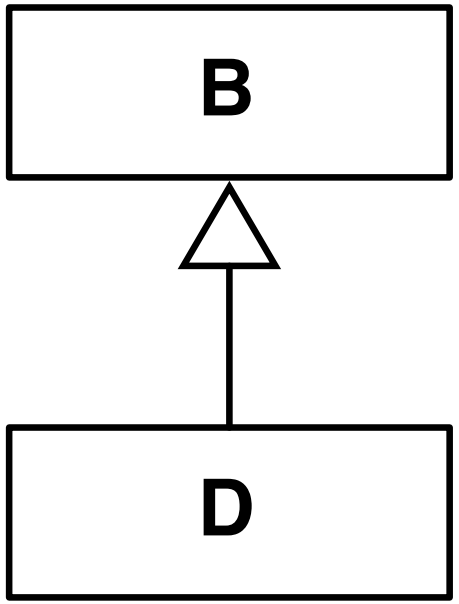
if `f(new B())` behaves correctly, `f(new D())` has to correctly behave as well



Fragile class

```
void f (B object) {
   ...
}
```

if `f(new B())` behaves correctly and `f(new D())` not, then we say that `D` is fragile in the presence of `f`



Some practical illustrations

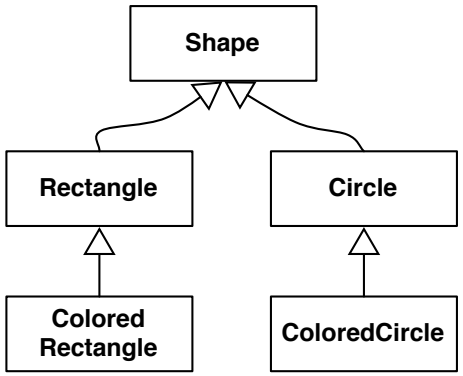
Procedural coding style

Object initialization

Access privileges cannot be weakened

Procedural coding style

```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        switch (shapes[i].kind()) {
        case Shape.RECTANGLE:          // a class constant
            sum += shapes[i].rectangleArea();
            break;
        case Shape.CIRCLE:
            sum += shapes[i].circleArea();
            break;
        }
    }
    return sum;
}
```

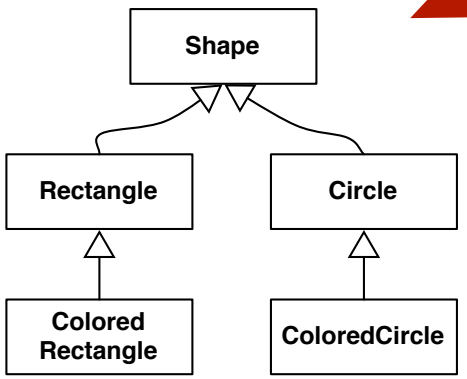


???

Procedural coding style

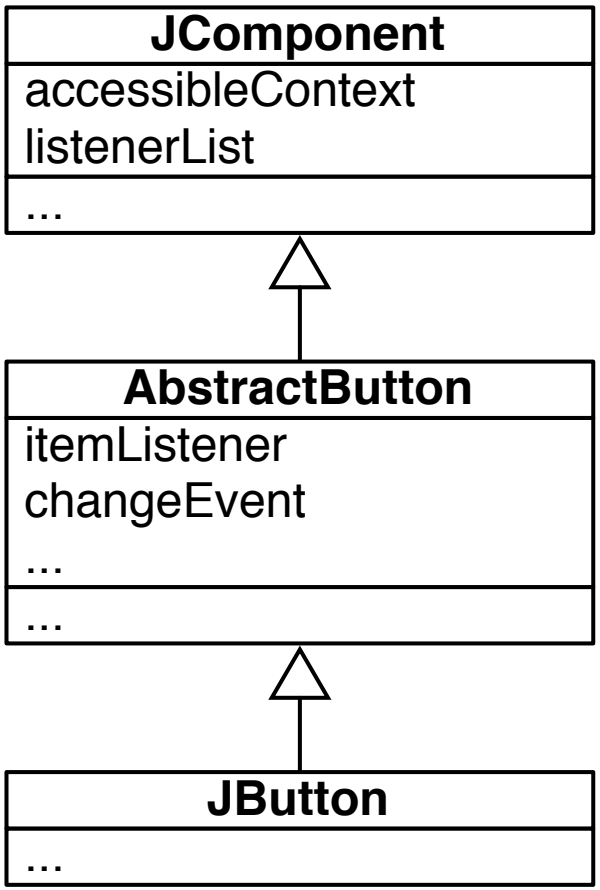
```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        switch (shapes[i].kind()) {
        case Shape.RECTANGLE:          // a class constant
            sum += shapes[i].rectangleArea();
            break;
        case Shape.CIRCLE:
            sum += shapes[i].circleArea();
            break;
        }
    }
    return sum;
}
```

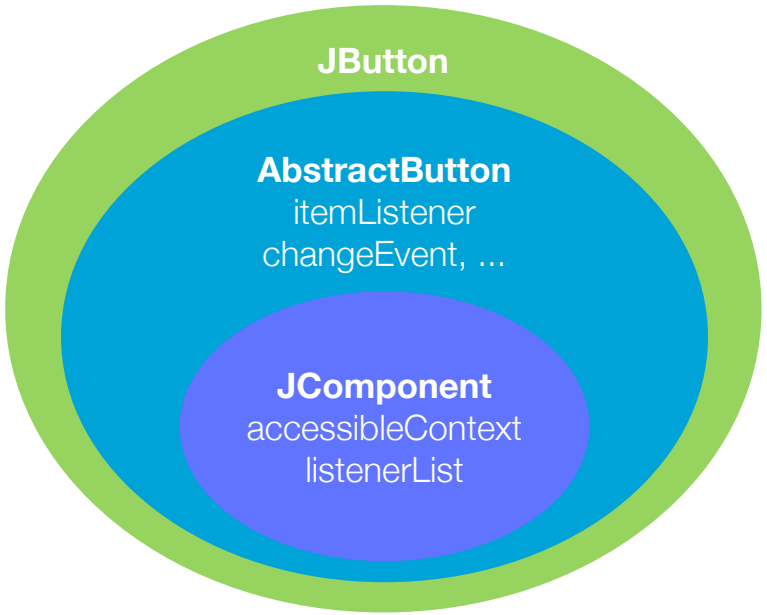
Simple violation of the Liskov principle



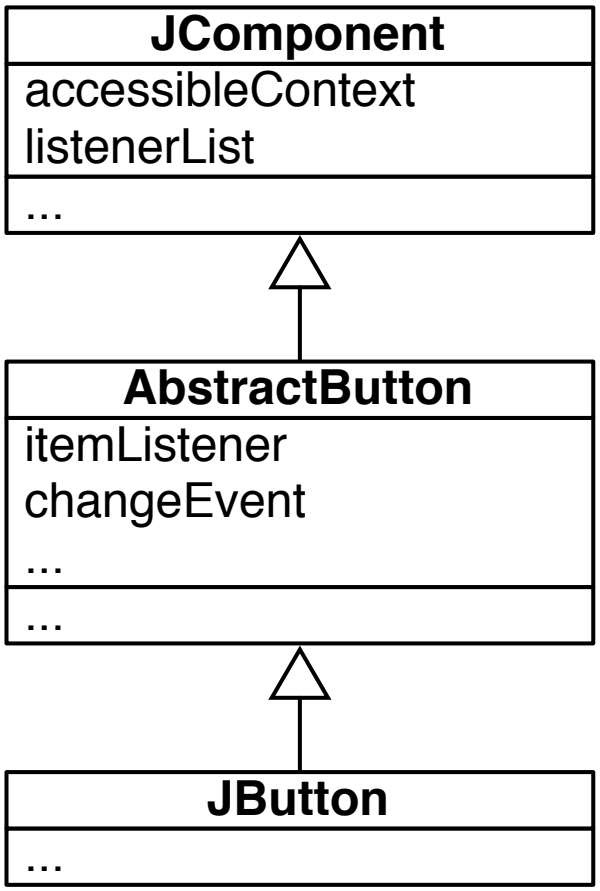
???

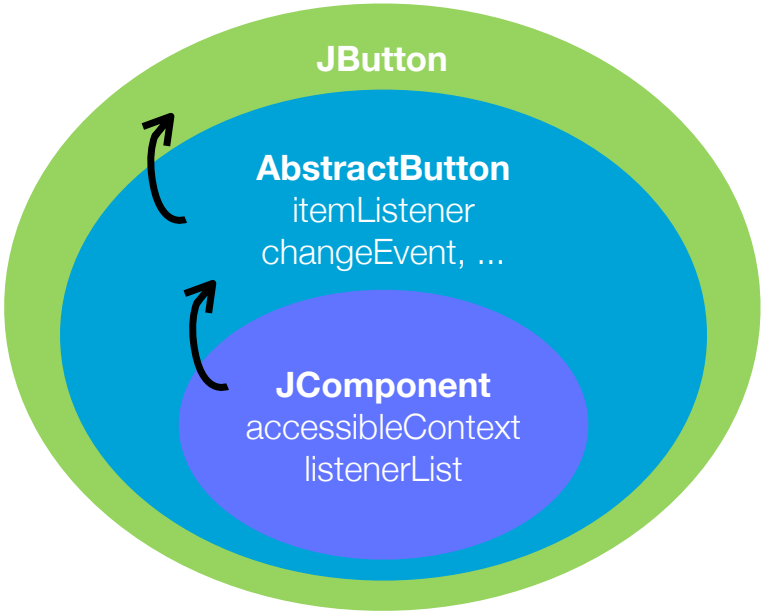
Object initialization





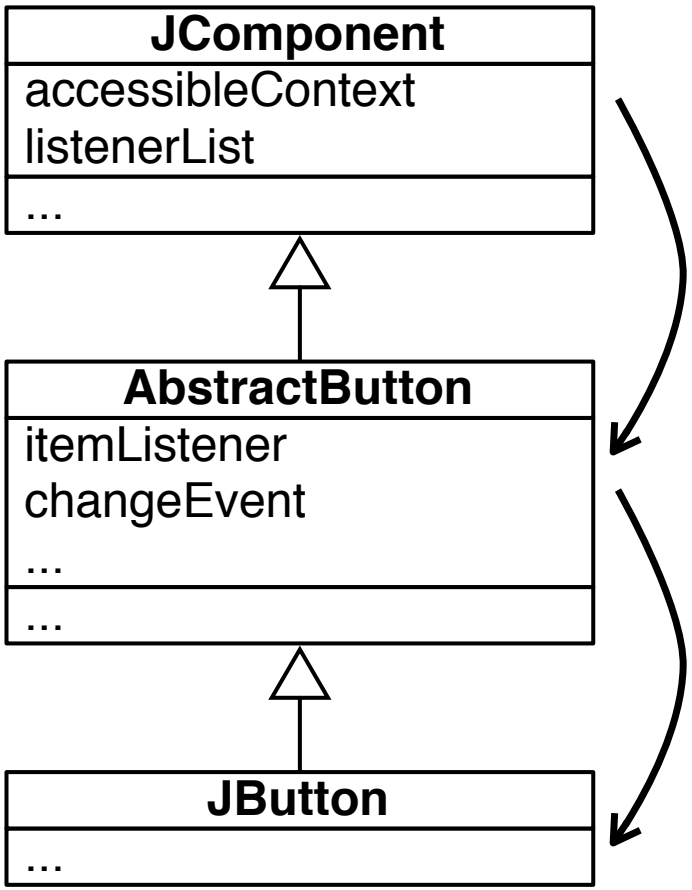
Object initialization

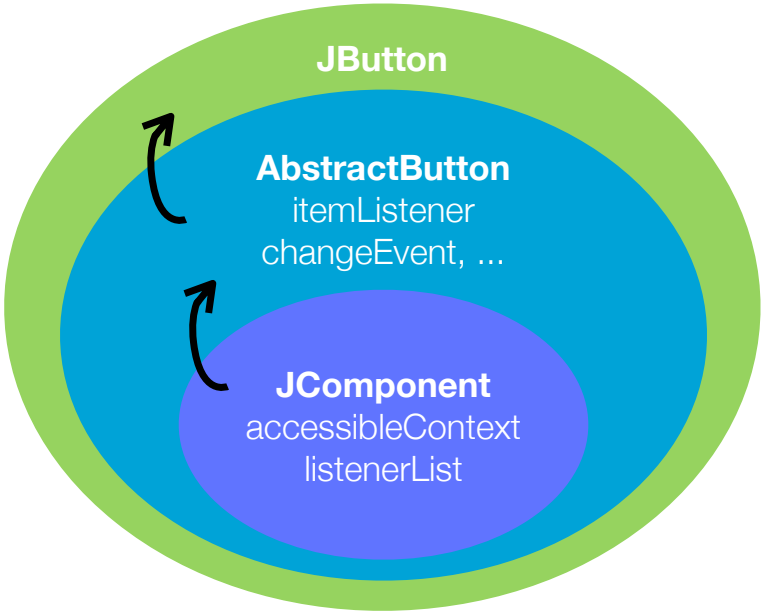




→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Object initialization





→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Access privileges apply to class definition
and class members (e.g., field, method, inner class)

More on http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Access privileges can only be widened

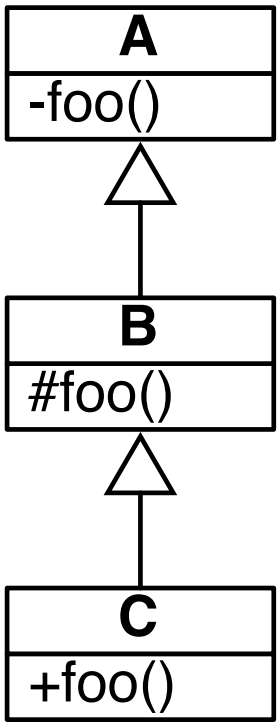
```
class A {
    private void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    public void foo () {

    }
}
```



Would it be okay to have this?

```
class A {
    public void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    private void foo () {

    }
}
```

Access privileges can only be widened

A protected method may be overridden as public

A private method cannot be overridden in Java

- a private method is statically bound
- a message send toward a private method is not looked up along the class hierarchy

Tips on Choosing an Access Level

If other programmers use your class, you want to ensure that errors from misuse cannot happen.

Access levels can help you do this

Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.

Avoid public fields except for constants. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

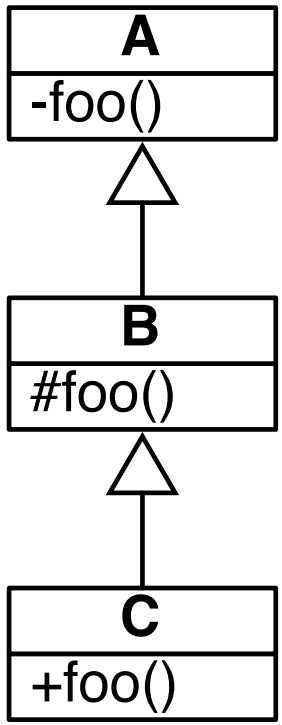
http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Private methods cannot be overridden

```
class A {
    private void foo () {
    }
}

class B extends A {
    protected void foo () {
       super.foo();
    }
}

class C extends B {
    public void foo () {
       super.foo();
    }
}
```

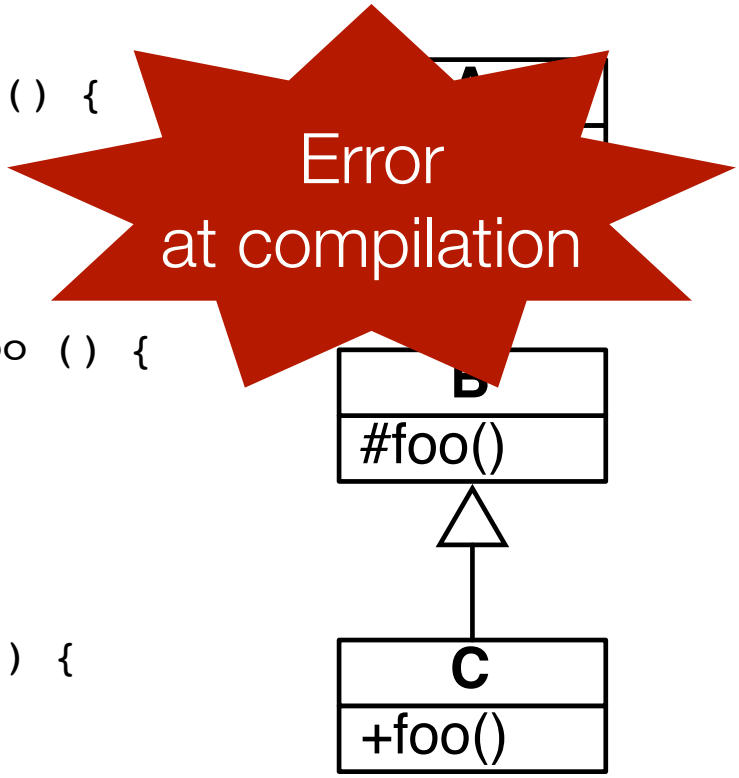


Private methods cannot be overridden

```
class A {
    private void foo () {
    }
}

class B extends A {
    protected void foo () {
       super.foo();
    }
}

class C extends B {
    public void foo () {
       super.foo();
    }
}
```



Remember that private methods are statically bound!

```
class A {
    public String callFoo (){
      return this.foo();
    }
    private String foo () {
      return “A”;
    }
}

class B extends A {
    protected String foo () {
       return “B”;
    }
}

class C extends B {
    public String foo () {
      return “C”;
    }
}
```

`new C().callFoo()` returns ???

Remember that private methods are statically bound!

```
class A {
    public String callFoo (){
      return this.foo();
    }
    private String foo () {
      return “A”;
    }
}

class B extends A {
    protected String foo () {
       return “B”;
    }
}

class C extends B {
    public String foo () {
      return “C”;
    }
}
```

`new C().callFoo()` returns “A”

What happens if A.foo is turned as public?

Outline

1. Liskov principle
    1. theory
    2. concrete applications
2. **Inheritance examples**
3. Example1: Swing and AWT
4. Example2: The Smalltalk collection class hierarchy

Virtual Classes
A powerful mechanism in object-oriented programming

Ole Lehrmann Madsen
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
Tlf.: +45 6 12 71 88 - E-mail: olmadsen@daimi.dk

Birger Møller-Pedersen
Norwegian Computing Center
P.O. Box 114, Blindern, N-0314 Oslo 3, Norway
Tlf.: +47 2 45 35 00 - E-mail: birger@nr.uninett.no

Abstract

The notions of class, subclass and virtual procedure are fairly well understood and recognized as some of the key concepts in object-oriented programming. The possibility of modifying a virtual procedure in a subclass is a powerful technique for specialising the general properties of the superclass.

In most object-oriented languages, the attributes of an object may be references to objects and (virtual) procedures. In Simula and BETA it is also possible to have class attributes. The power of class attributes has not yet been widely recognized. In BETA a class may also have *virtual class attributes*. This makes it possible to defer part of the specification of a class attribute to a subclass. In this sense virtual classes are analogous to virtual procedures. Virtual classes are mainly interesting within strongly typed languages where they provide a mechanism for defining general parameterized classes such as set, vector and list. In this sense they provide an alternative to generics.

Although the notion of virtual class originates from BETA, it is presented as a general language mechanism.

Keywords: languages, virtual procedure, virtual class, strong typing, parameterized class, generics, BETA, Simula, Eiffel, C++, Smalltalk

1 Introduction

The notions of class and subclass are some of the key language concepts associated with object-oriented programming. Classes support the classification of objects with the same properties, and subclassing supports the specialisation of the general properties. A class defines a set of attributes associated with each instance of the class. An attribute may be either an object reference (or just reference for short) or a procedure.

In a subclass it is possible to specialize the general properties defined in the superclass. This can be done by adding references and/or procedures. However, it is also possible to modify the procedures defined in the superclass. Modification can take place in different ways. In Simula 67 [4] a procedure attribute may be declared virtual. A virtual procedure may then be redefined in a subclass. A non-virtual procedure cannot be redefined[1]. This is essentially the same scheme adapted by C++ [16] and Eiffel [13]. In Smalltalk [6] any procedure is virtual in the sense that it can be redefined in a subclass, and even the parameters of a procedure may be redefined.

In BETA [8] a virtual procedure cannot be redefined in a subclass, but it may be further defined by an *extended* definition. The extended procedure is a "subprocedure" (in the same way as for subclass) of the procedure defined in the superclass. This implies that the actions of a virtual procedure definition are automatically combined with the actions of the extended procedure in a subclass. This is the case for all levels of subclasses that further defines a virtual procedure. In Smalltalk and C++ it is the responsibility of the programmer to combine a redefined virtual procedure with the corresponding virtual procedure of the superclass. This is of course more flexible, since the programmer can ignore the procedure in the superclass. However, it is also a potential source of error since the programmer may forget to execute the virtual procedure from the superclass.

Using the terminology from [18] a class in BETA

[1] In Simula a subclass may declare a *new* procedure with the same name as a procedure defined in a superclass. This does not have the effect of a redefinition as in Smalltalk.

@OOPSLA'89

```
Window: class Stream
   (# UpperLeft,LowerRight: @ Point;
      Label: ^ Text;
      Move: proc (# ... #);
      Display: virtual proc (# ... #);
   #)
```

Figure 2: Example of class declaration

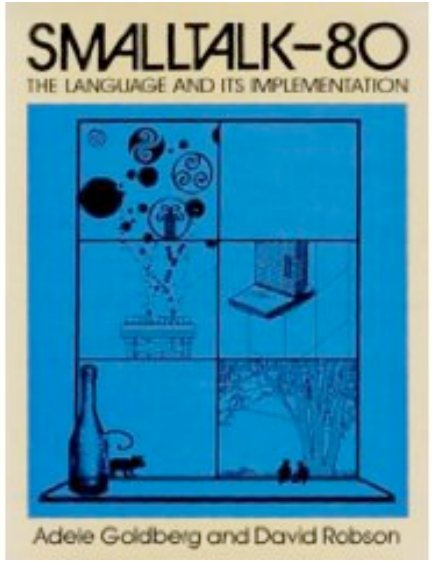
"In Figure 2 an example of a class is given. Class Window is described as a subclass of class Stream. ..."

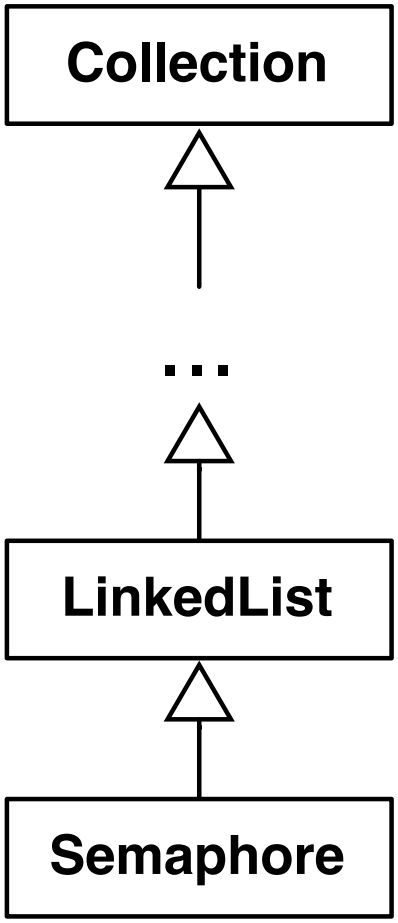
```
Window: class Stream
   (# UpperLeft,LowerRight: @ Point;
      Label: ^ Text;
      Move: proc (# ... #);
      Display: virtual proc (# ... #);
   #)
```

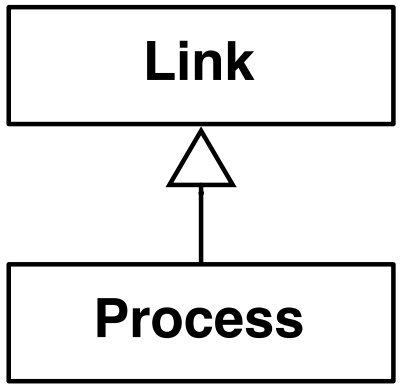
Figure 2: Example of class declaration

"In Figure 2 an example of a class is given. Class Window is described as a subclass of class Stream. ..."

Do you think a window can be considered as a stream?





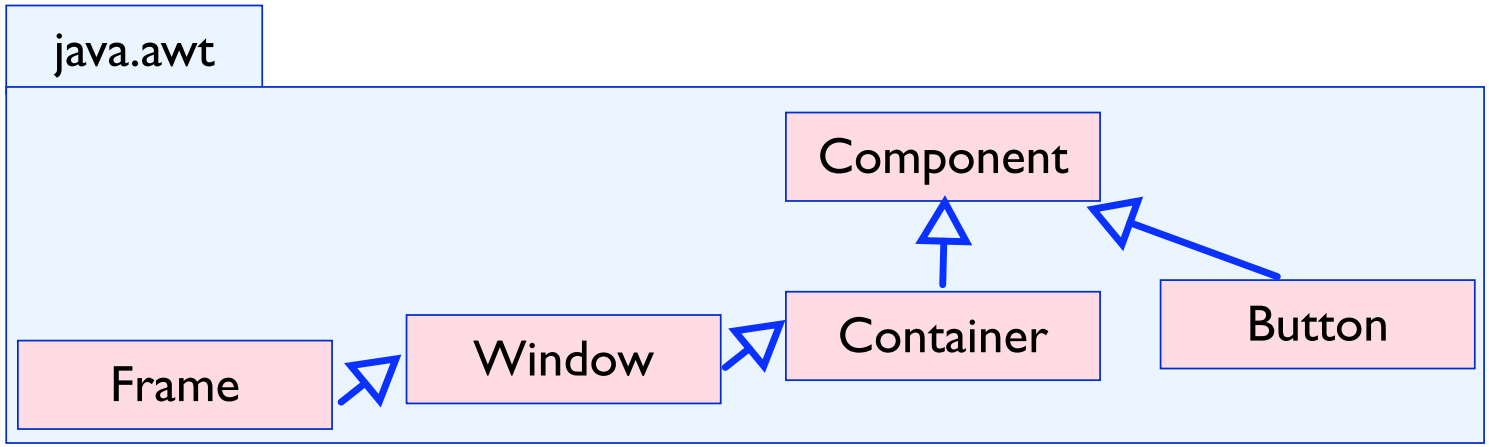


Probably a semaphore can be seen as a collection, but is it worth subclassing LinkedList in that case?

Outline

1. Liskov principle
    1. theory
    2. concrete applications
2. Inheritance examples
3. **Example1: Swing and AWT**
4. Example2: The Smalltalk collection class hierarchy

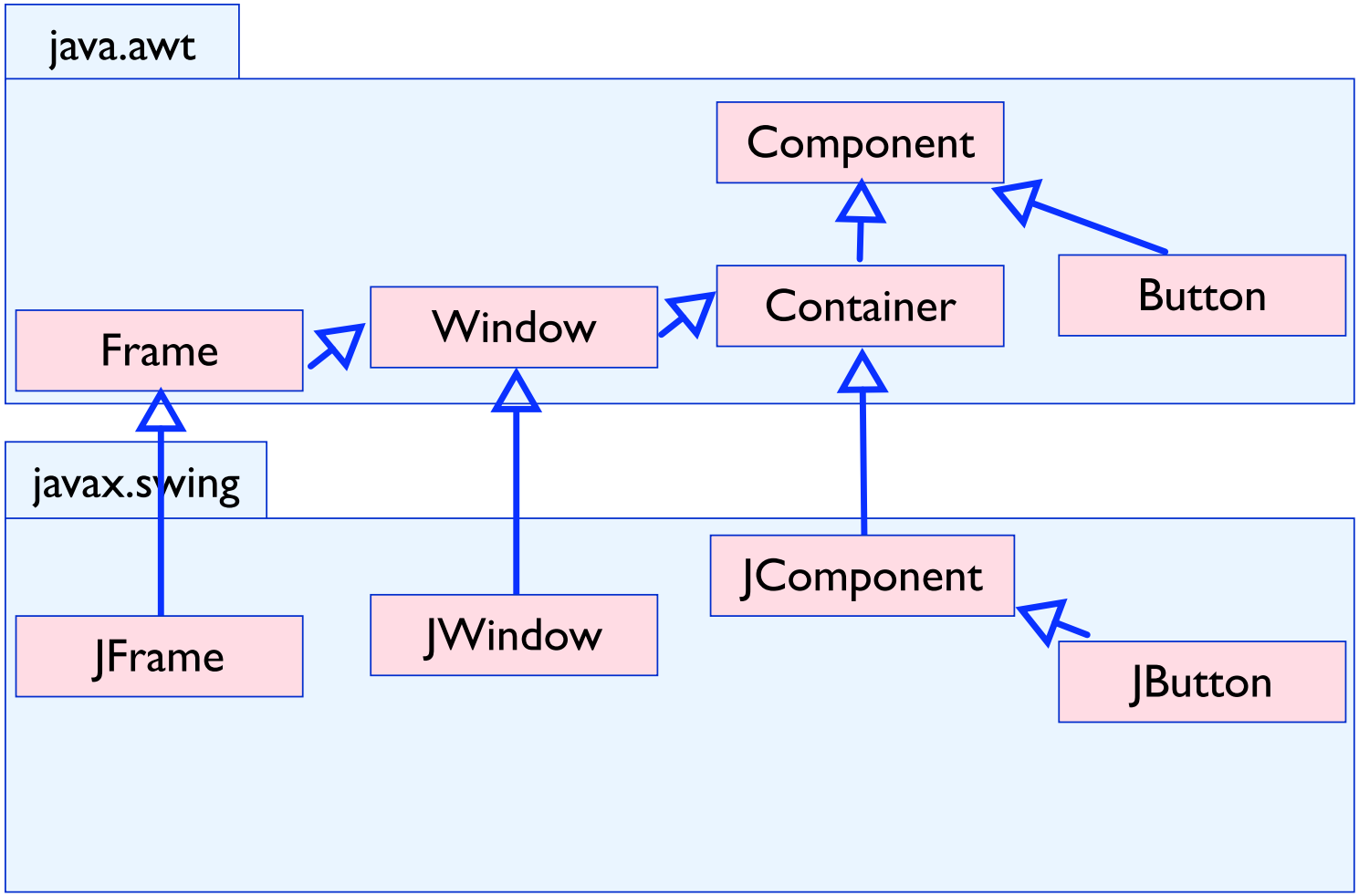
Presentation of AWT



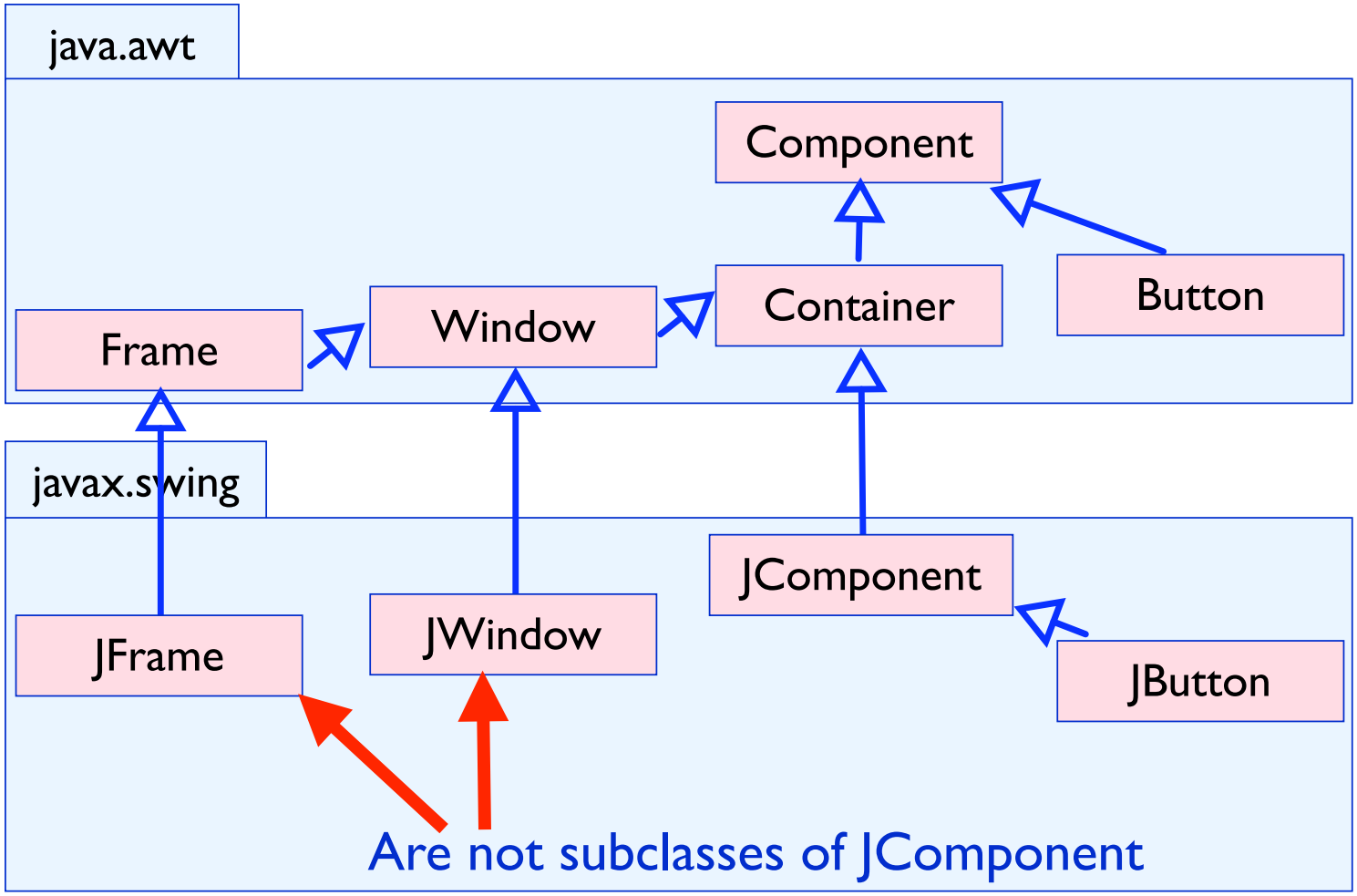
In the AWT framework:

- Widgets are components (i.e., inherit from Component)
- A frame is a window (Frame is a subclass of Window)

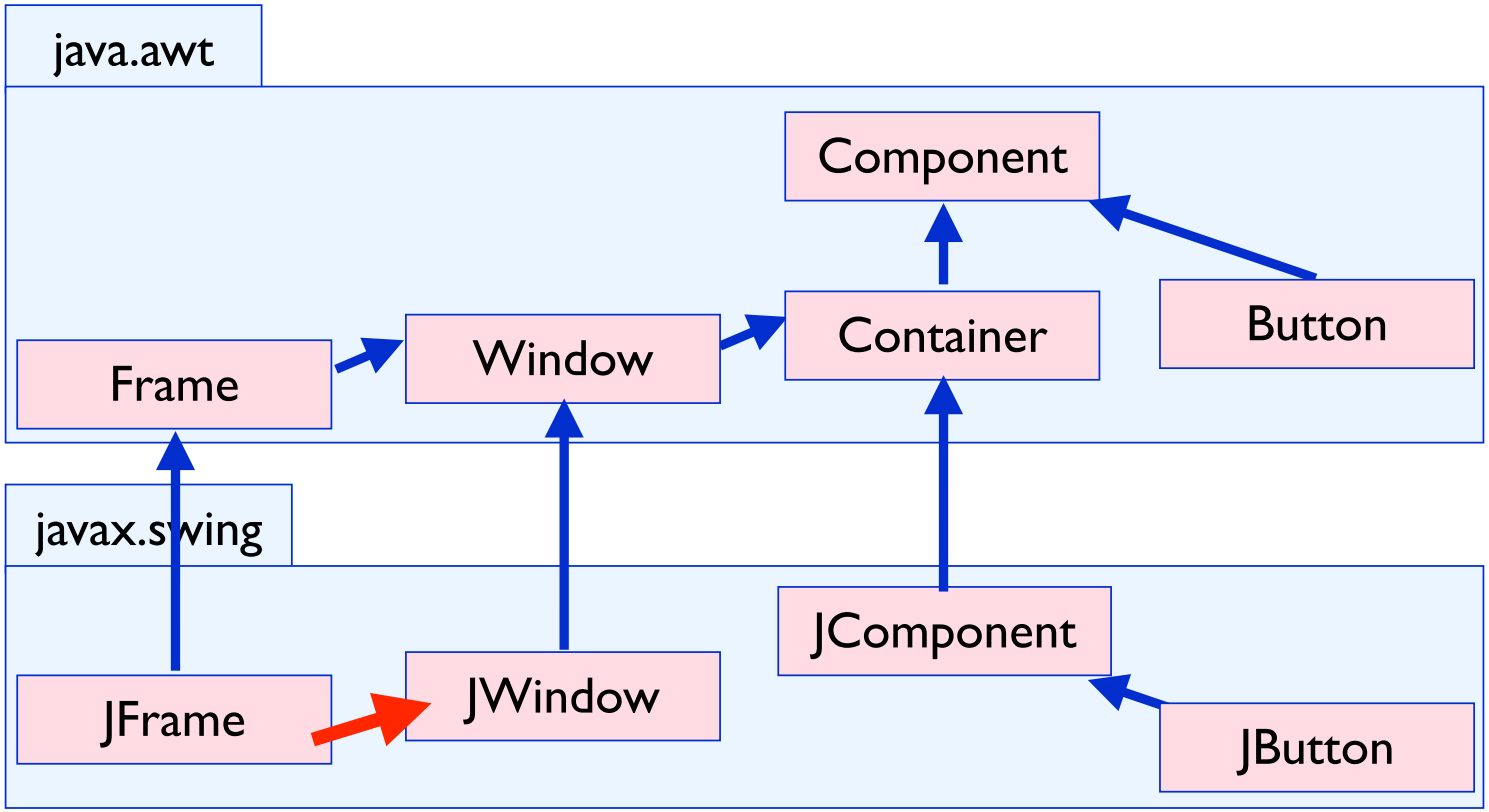
Swing at the top of AWT



Problem #1: Brocken Inheritance

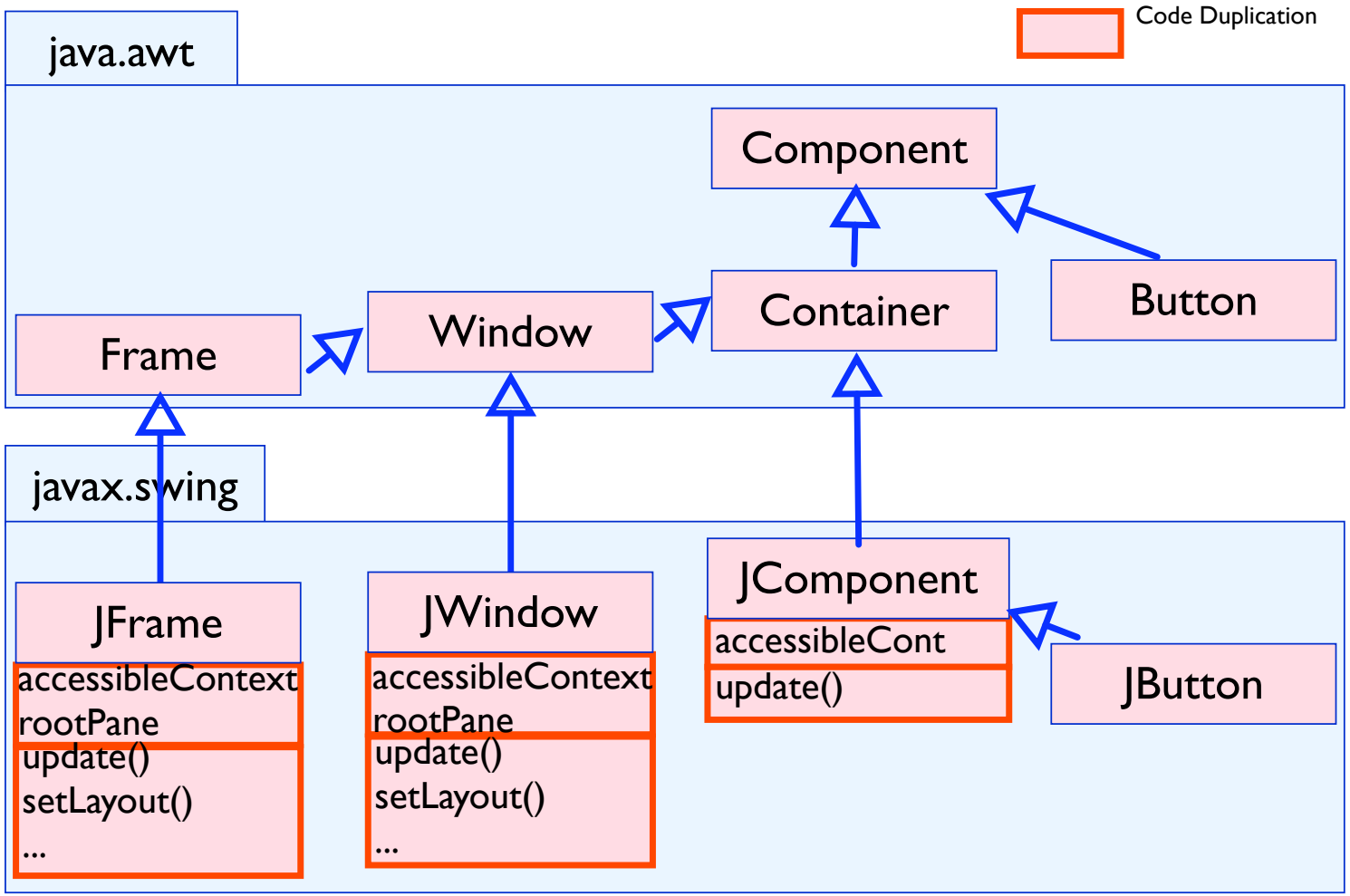


Problem #1: Brocken Inheritance



Missing inheritance link between JFrame and JWindow

Problem #2: Code Duplication



Problem #3: Explicit Type Checks and Casts

```
public class Container extends Component {
  Component components[] = new Component [0];
  public Component add (Component comp) {...}
}

public class JComponent extends Container {
  public void paintChildren (Graphics g) {
    for (; i>=0 ; i--) {
      Component comp = getComponent (i);
      isJComponent = (comp instanceof JComponent);
      ...
      ((JComponent) comp).getBounds();
    }
  }}
```

Supporting Unanticipated Changes

AWT couldn't *be enhanced* without risk of *breaking* existing code

Swing is, therefore, *built on the top* of AWT using *subclassing*

As a result, *Swing is a big mess internally*!

Why do we care to have a messy Swing ?

Swing appeared in 1998, and *has not evolved since*!

Swing is too heavy to be ported to PDA, cellphones, ...

SWT is *becoming* a *new standard*.

Either a system evolves, or it is dead. [Lehmans74]

Outline

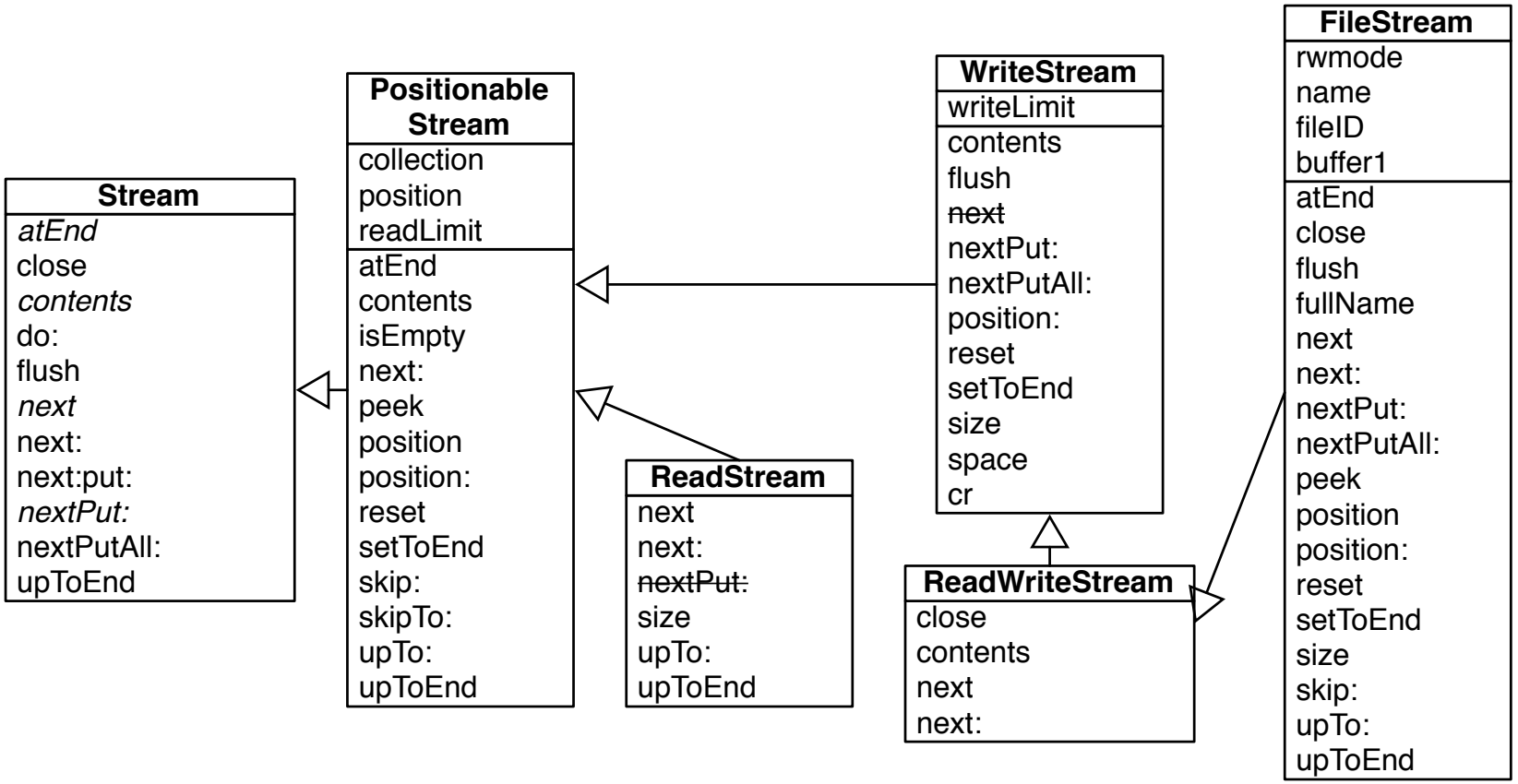
1. Liskov principle
    1. theory
    2. concrete applications
2. Inheritance examples
3. Example1: Swing and AWT
4. **Example2: The Smalltalk collection class hierarchy**

the Stream framework in Squeak

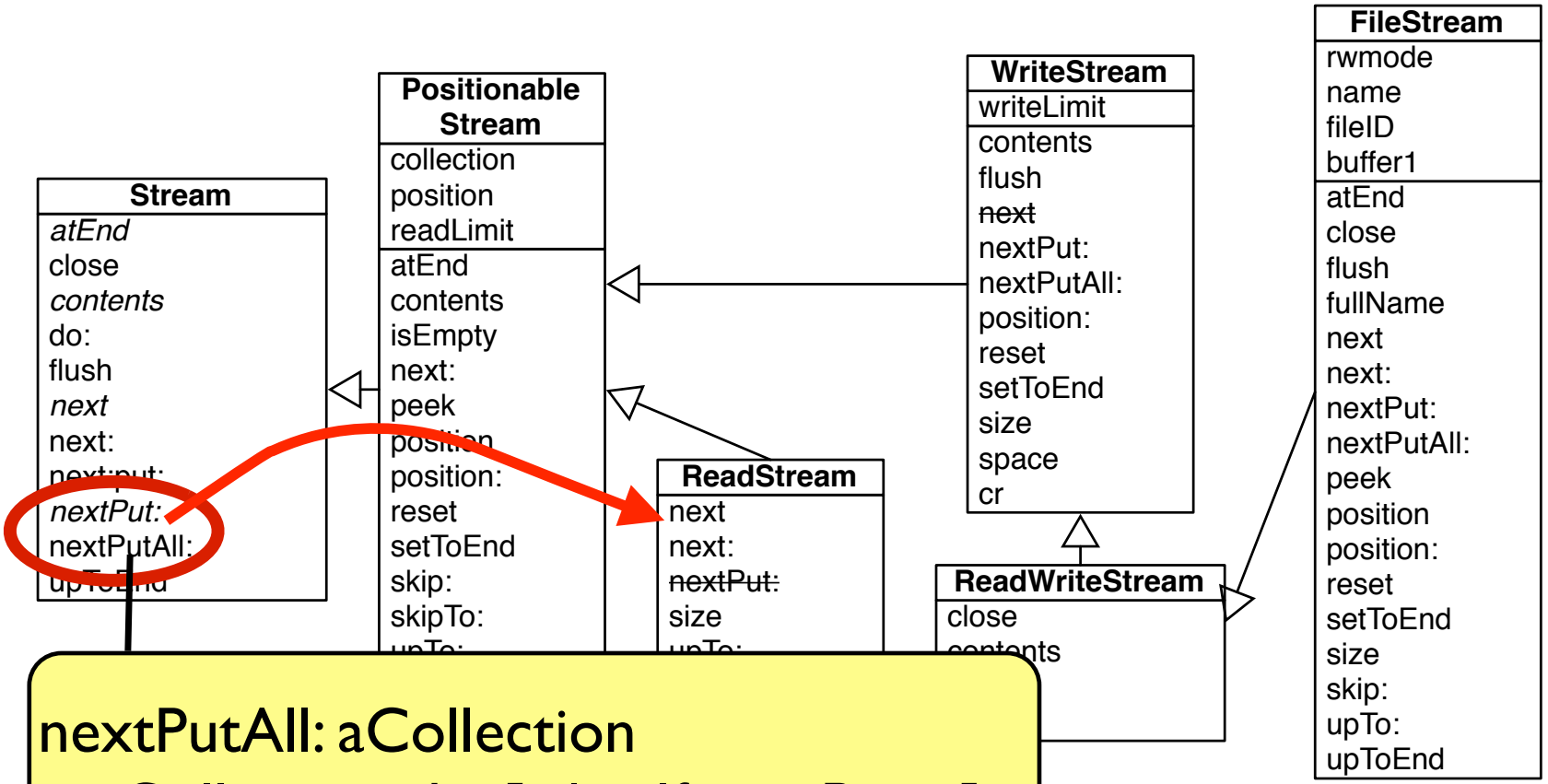
Example of a library that has been in use for almost 20 years

Contains many flaws in its design

the Stream framework in Squeak



Methods too high




```
nextPutAll: aCollection
    aCollection do: [:v| self nextPut: v].
    ^ aCollection
```

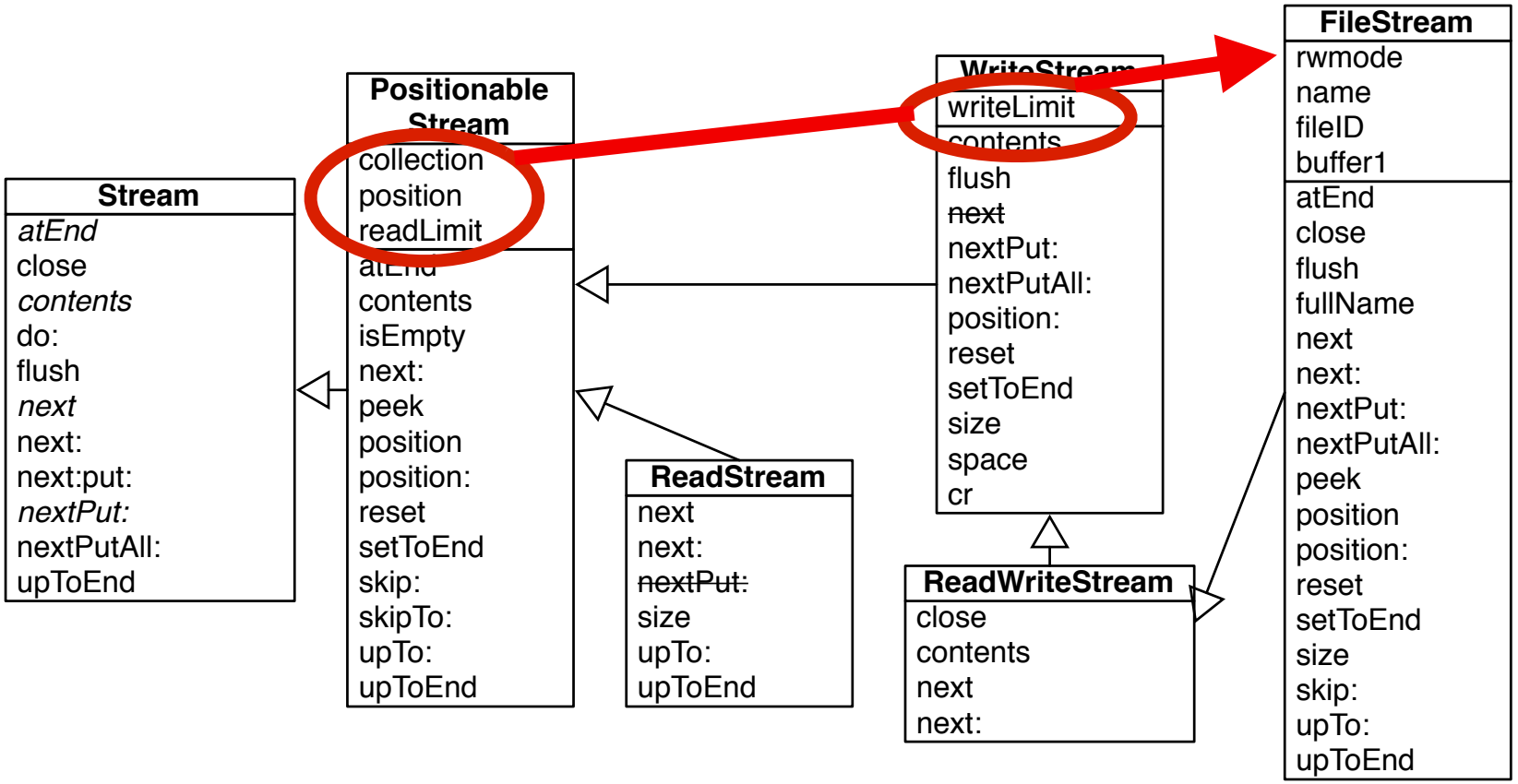
Methods too high

The nextPut: method defined in Stream allows for element addition

The ReadStream class is read-only

It therefore needs to “cancel” this method by redefining it and throwing an exception

Unused state

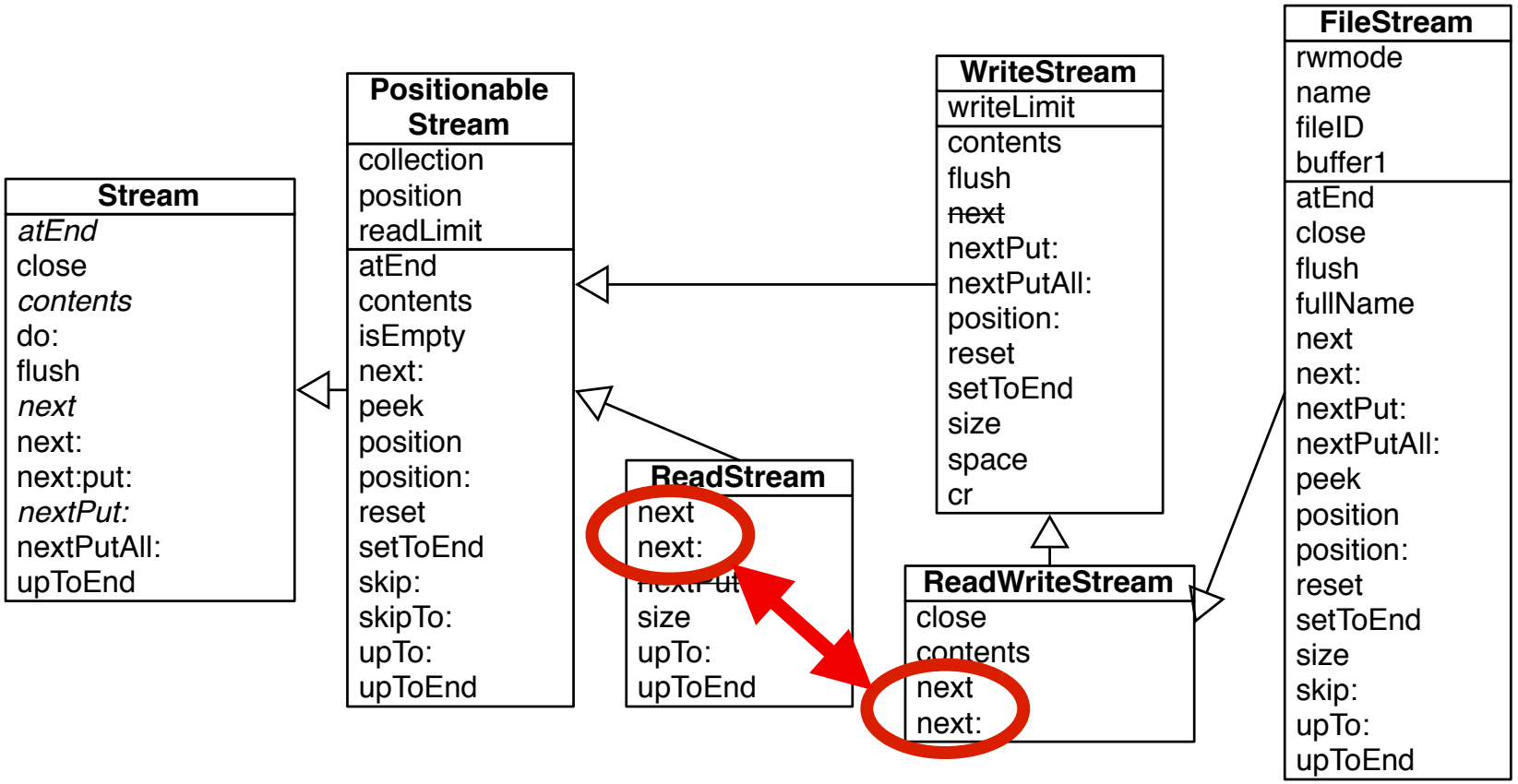


Unused state

State defined in the super classes are becoming irrelevant in subclasses

FileStream does not use inherited variables

Multiple Inheritance Simulation



Multiple Inheritance Simulation

Methods are duplicated among different class hierarchies

Class schizophrenia?

Too many responsibilities for classes

- object factories
- group methods when subclassing

Class schizophrenia?

Too many responsibilities for classes

- object factories => *need for completeness*
- group methods when subclassing => *need to incorporate incomplete fragments*

What you should know!

What is the Liskov principle?

How the Liskov principle affects the design of a programming language

Why a good class hierarchy is not easy to obtain and requires experience

Defining a subclass should be driven by the IS-A relation

Can you answer to these questions?

Why a class Window should not be defined as a subclass of Stream?

What makes class inheritance so difficult to use?

Is there a definitive answer on what a good class hierarchy is?

License

http://creativecommons.org/licenses/by-sa/2.5