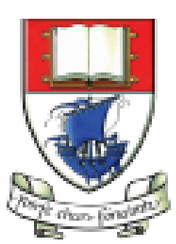
Liskov Substitution Principle

Produced by:

Eamonn de Leastar (edeleastar@wit.ie)

Dr. Siobhán Drohan (sdrohan@wit.ie)



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

SOLID Class Design Principles – Module Scope

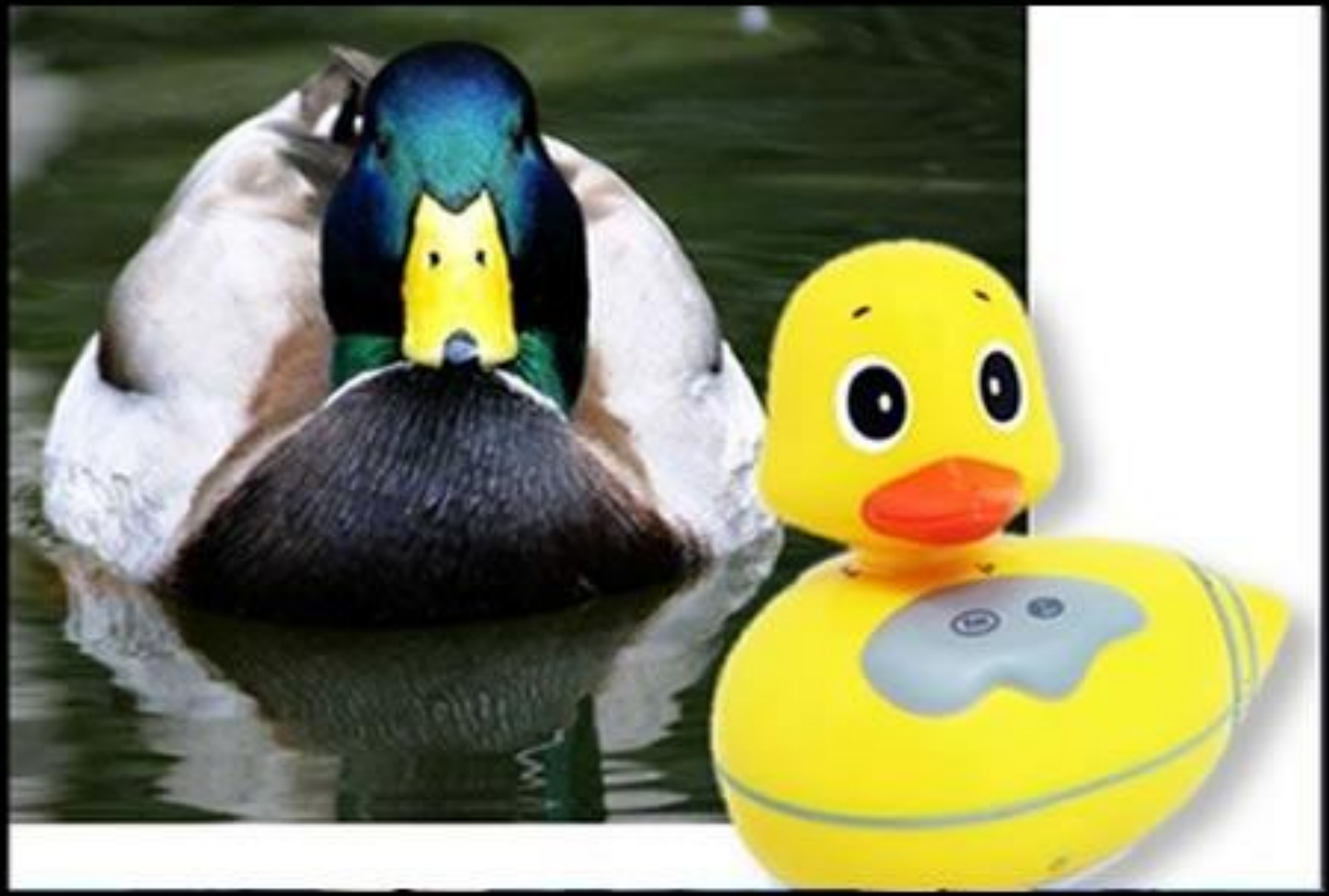
S Single Responsibility Principle (SRP). Classes should have one, and only one, reason to change. Keep your classes small and single-purposed.

O Open-Closed Principle (OCP). Design classes to be open for extension but closed for modification; you should be able to extend a class without modifying it. Minimize the need to make changes to existing classes.

L Liskov Substitution Principle (LSP). Subtypes should be substitutable for their base types. From a client's perspective, override methods shouldn't break functionality.

I Interface Segregation Principle (ISP). Clients should not be forced to depend on methods they don't use. Split a larger interface into a number of smaller interfaces.

D Dependency Inversion Principle (DIP). High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

Barbara Liskov

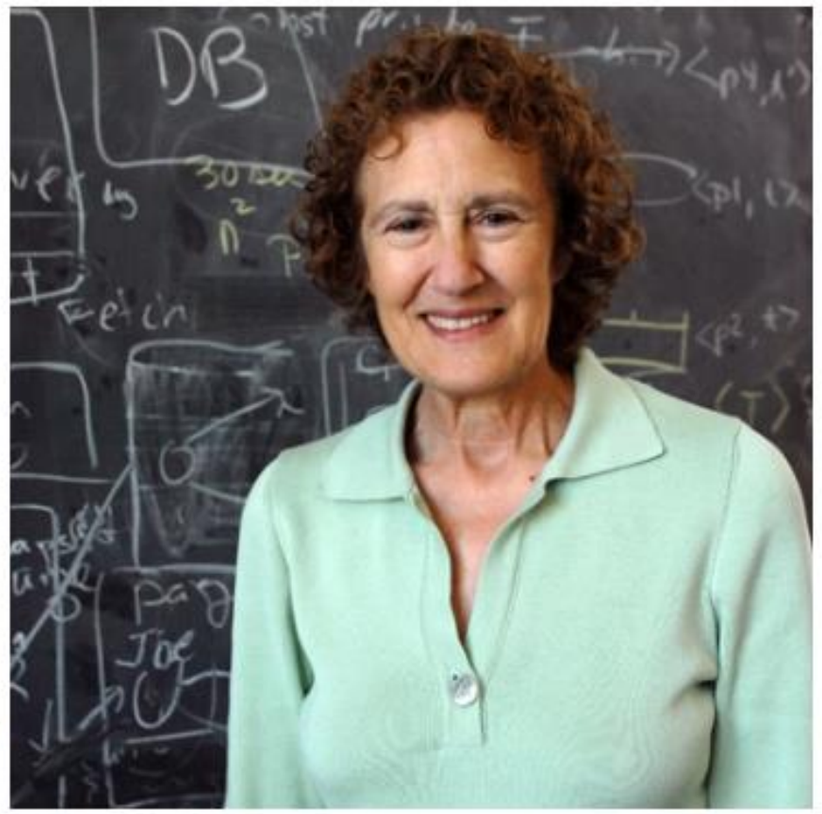
Barbara Liskov wins Turing Award

ACM cites 'foundational innovations' in programming language design

March 10, 2009

Share

Institute Professor Barbara Liskov has won the Association for Computing Machinery's A.M. Turing Award, one of the highest honors in science and engineering, for her pioneering work in the design of computer programming languages. Liskov's achievements underpin virtually every modern computing-related convenience in people's daily lives.



Barbara Liskov
Photo / Donna Coveney

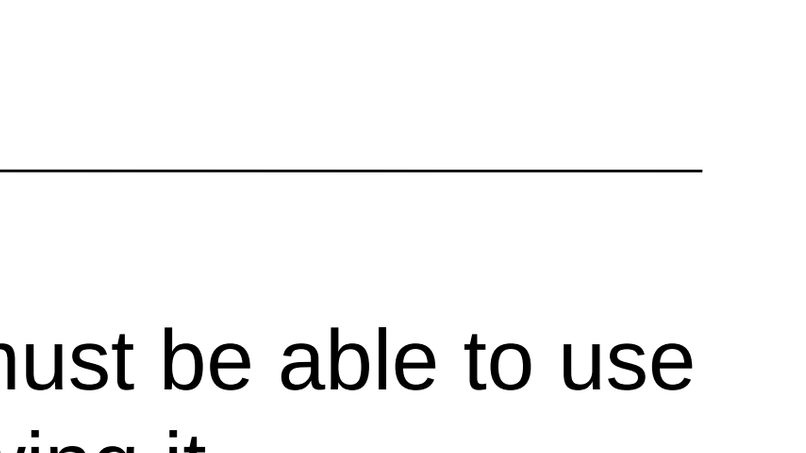
Liskov, the first U.S. woman to earn a PhD from a computer science department, was recognized for helping make software more reliable, consistent and resistant to errors and hacking. She is only the second woman to receive the honor, which carries a $250,000 purse and is often described as the "Nobel Prize in computing."

LSP

- Methods that refer to base classes must be able to use objects of derived types without knowing it.

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

Barbara Liskov, “Data Abstraction and Hierarchy,” *SIGPLAN Notices,* 23,5 (May, 1988).



LSP: Simple Violation (and fix)

Simple Violation of LSP

references a base type Shape

```
void drawShapes (Shape shape)
{
   if (shape instanceof Square)
   {
      drawSquare ((Square)shape);
   }
   else if (shape instance of Circle)
   {
      drawCircle ((Circle) shape);
   }
}
```

violates LSP because it must know of every derived type of Shape.

- drawShapes must be modified whenever new derivatives of Shape are presented.
- What other SOLID principle does it violate?

Adhering to LSP

```
class Shape
{
  void draw()
  {//…}
}

class Circle extends Shape
{
  private double itsRadius;
  private Point  itsCenter;
  public void draw()
  { //… }
}

class Square extends Shape
{
  private double itsSide;
  private Point itsTopLeft;
  public void draw()
  { //… }
}
```

```
void drawShape (Shape s)
{
  s.draw();
}
```

- drawShape now adheres to LSP

LSP: Semantic Violation

LSP

An object inheriting from
a base class, interface,
or other abstraction
*must be **semantically** substitutable*
for the original abstraction.

http://www.codemag.com/article/1001061

Rectangle

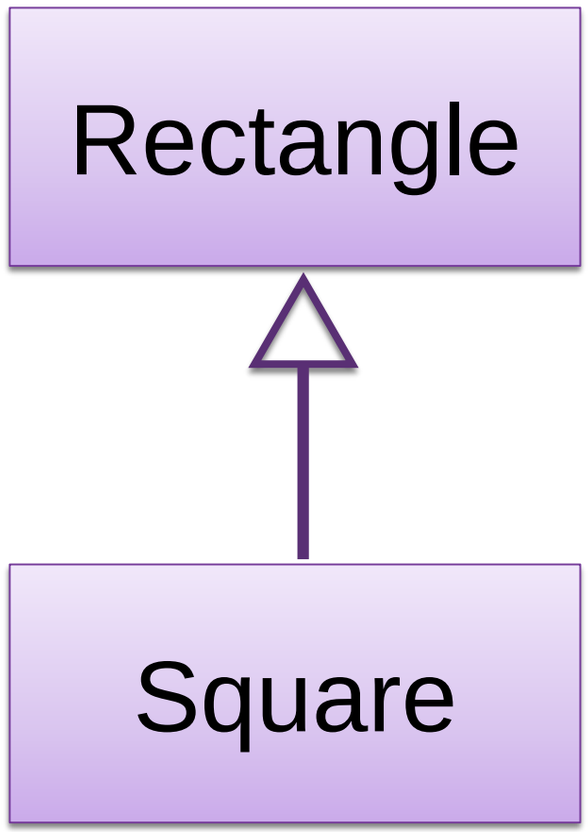
```
class Rectangle
{
  private int width;
  private int height;

  public void setWidth (int width)
  {...}
  public void setHeight (int height)
  {...}
  public int getWidth ()
  {...}
  public int getHeight ()
  {...}
}
```

Rectangle

- Rectangle class is released for general use.

Square

- Introduce Square as a subclass of Rectangle.

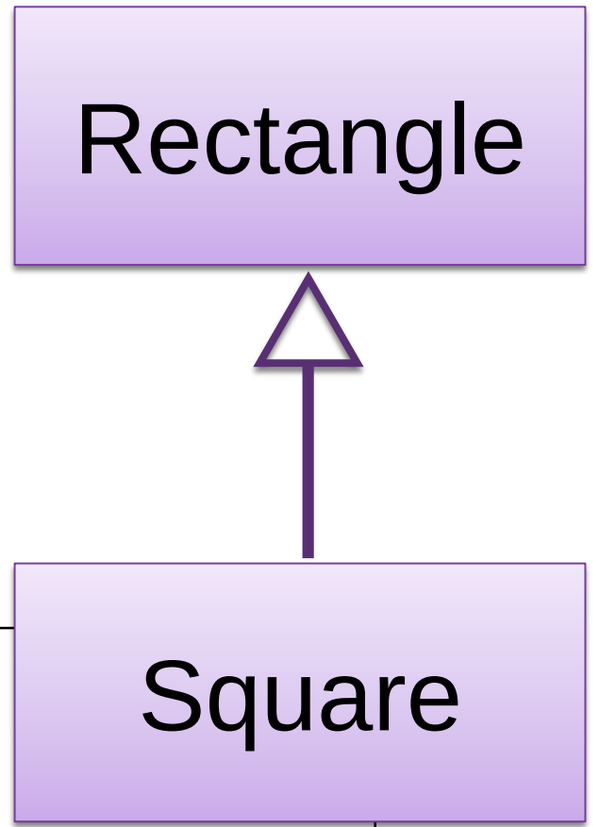


- Inheritance “is a” relationship:
  - A Square is a rectangle.
  - However, there is a subtle difference…it’s width and height are equal:
    - Square only needs one dimension but both are inherited.

Square

- For a Square, both setWidth() and setHeight() should not vary independently.

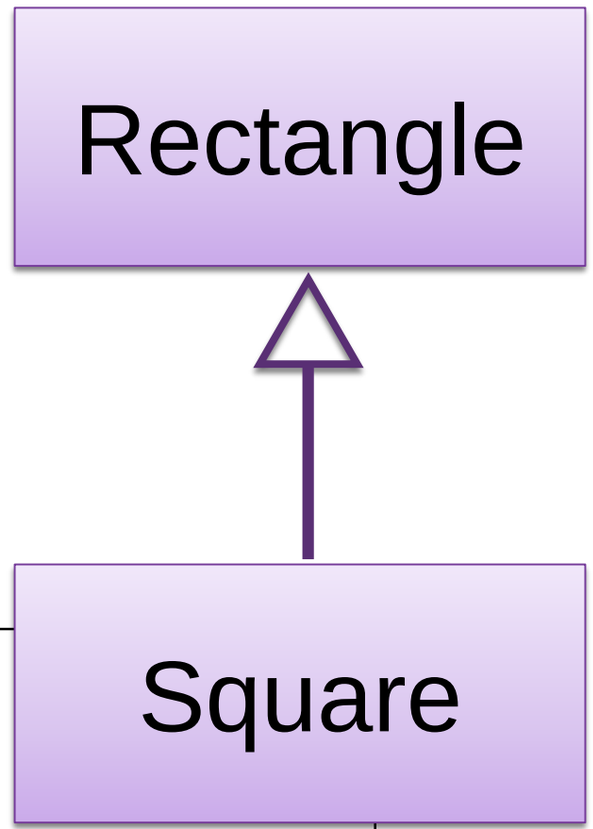
- Client could easily call one and not the other – thus compromising the Square.




```
class Rectangle
{
  private int width;
  private int height;

  public void setWidth (int width)
  {...}
  public void setHeight (int height)
  {...}
  public int getWidth ()
  {...}
  public int getHeight ()
  {...}
}
```

Square

- Potential solution:
  - implement setWidth() and setHeight() in Square class.
- Each of these methods should then make sure both width & height are adjusted.

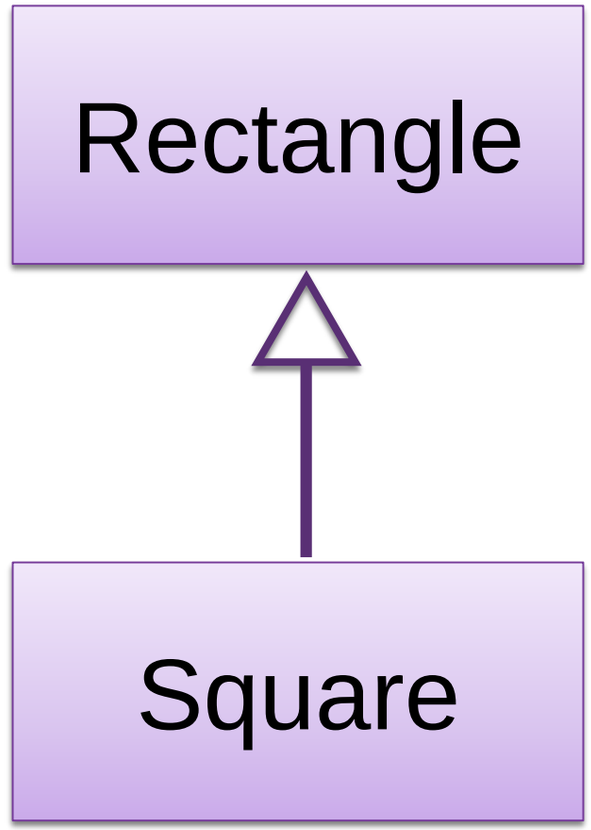



```
class Rectangle
{
  private int width;
  private int height;

  public void setWidth (int width)
  {...}
  public void setHeight (int height)
  {...}
  public int getWidth ()
  {...}
  public int getHeight ()
  {...}
}
```

Square

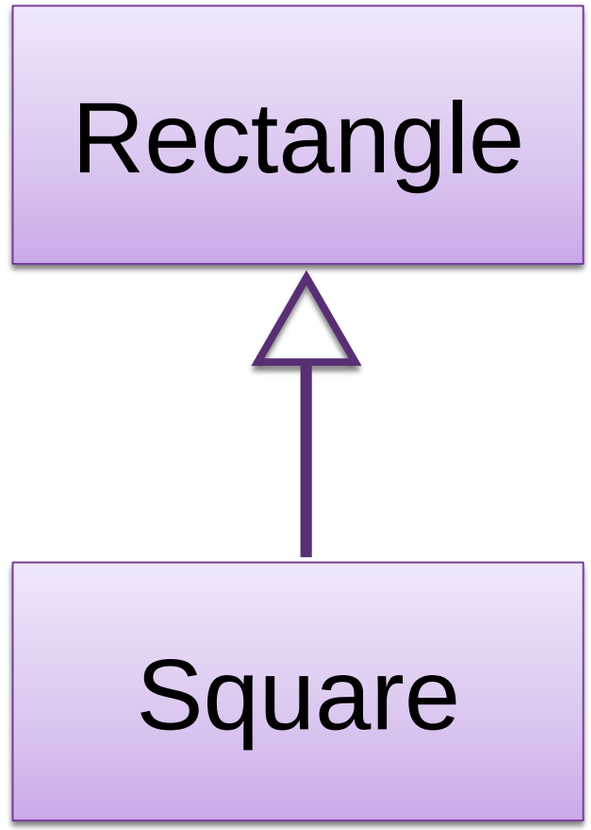
Potential solution implementation:

```
class Square extends Rectangle
{
  public void setWidth (int width)
  {
    super.setWidth(width);
    super.setHeight(width);
  }
  public void setHeight (int height)
  {
    super.setWidth(height);
    super.setHeight(height);
  }
}
```



Polymorphism

```
void f (Rectangle r)
{
  r.setWidth(5);
}
```



- Polymorphism ensures, if the f() method:
  - is passed a Rectangle, then its width will be adjusted.
  - is passed a Square, then both height and width will be changed
- Assume model is consistent & correct.
- However….

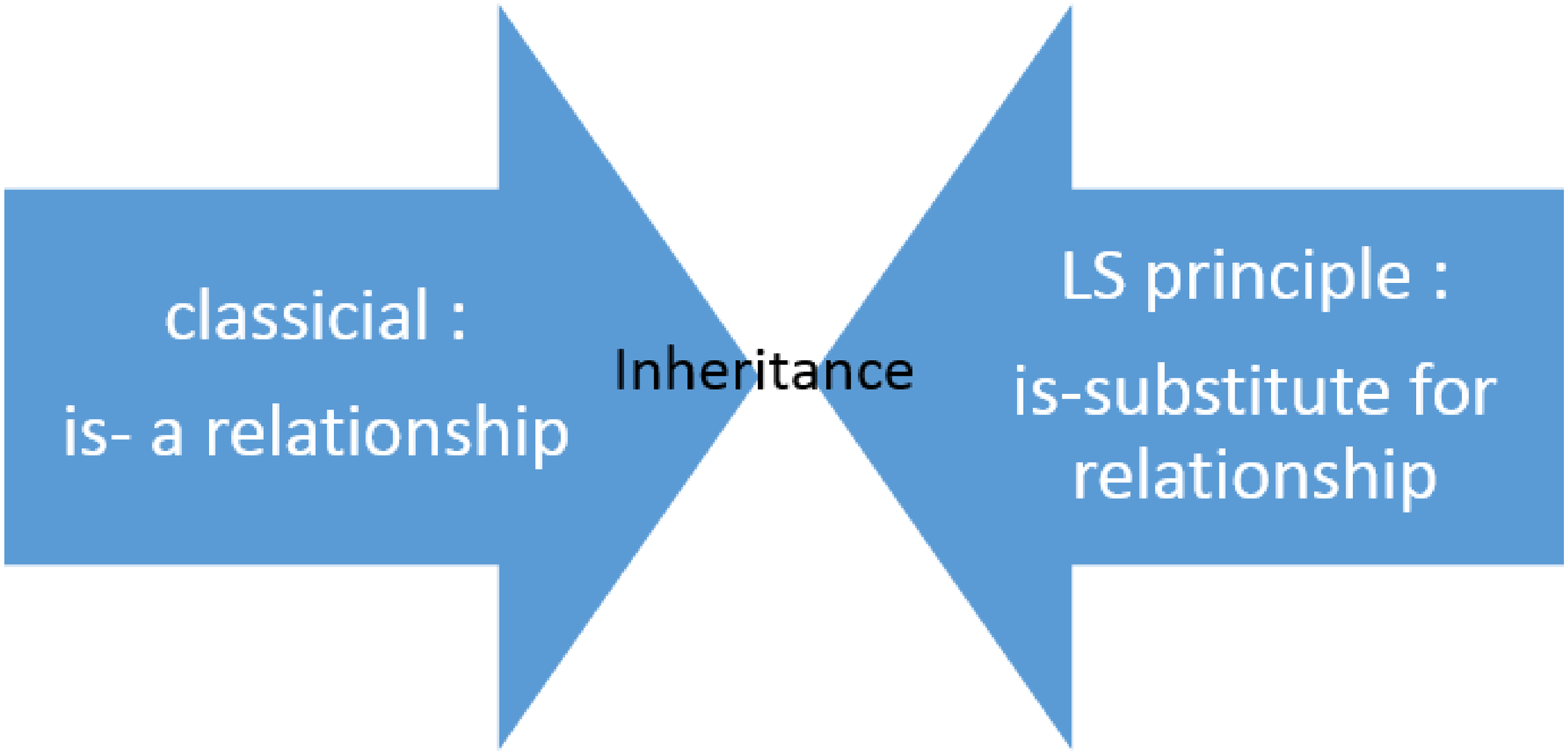
More Subtle Problem

```
void g (Rectangle r)
{
  r.setWidth(5);
  r.setHeight(4);
  assert (r.getWidth() * r.getHeight()) == 20;
}
```

- If r is a Rectangle instance…
  - g() methods works as expected
- If r is a Square instance…
  - g() assertion fails
- g() assumes that width and height of a Rectangle can be varied independently.
- Substitution of a Square violates this assumption.
- Square violates LSP.

LSP: Semantic Violation

Square != Rectangle



LSP: Subtypes should be substitutable for their base types.

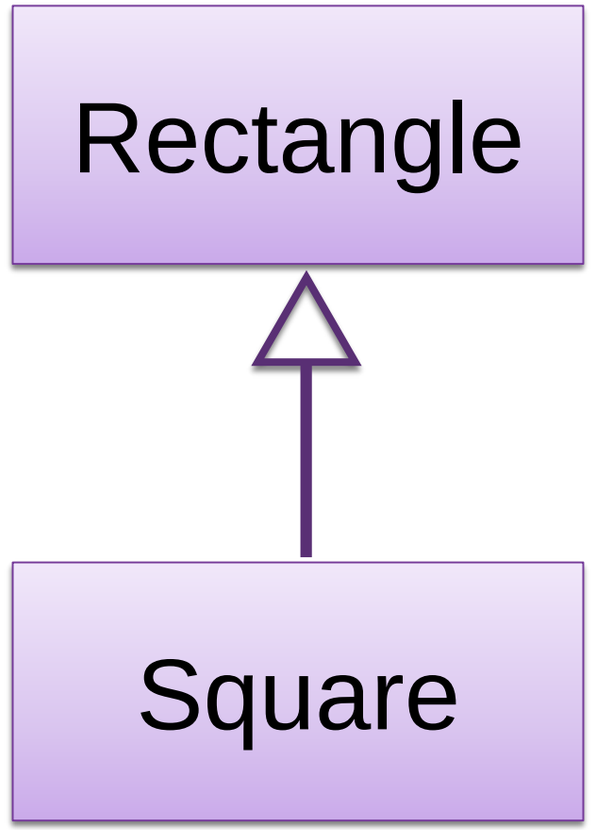
Validating the Model

- A model, viewed in isolation, cannot be meaningfully validated.

- The validity of a model can only be expressed in terms of its clients:

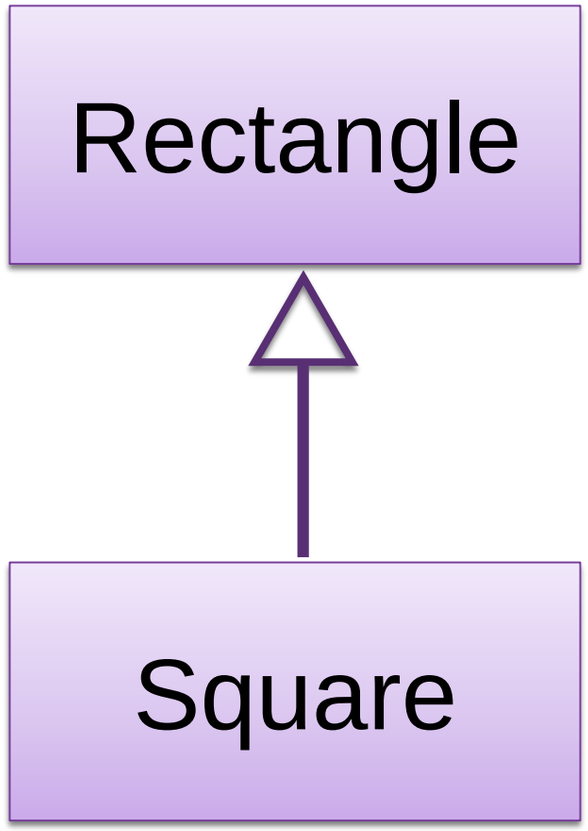
- Examining the final version of the Square and Rectangle classes in isolation, we found that they were self consistent and valid.

- When we examined from the viewpoint of g() (which made reasonable assumptions) the model broke down.



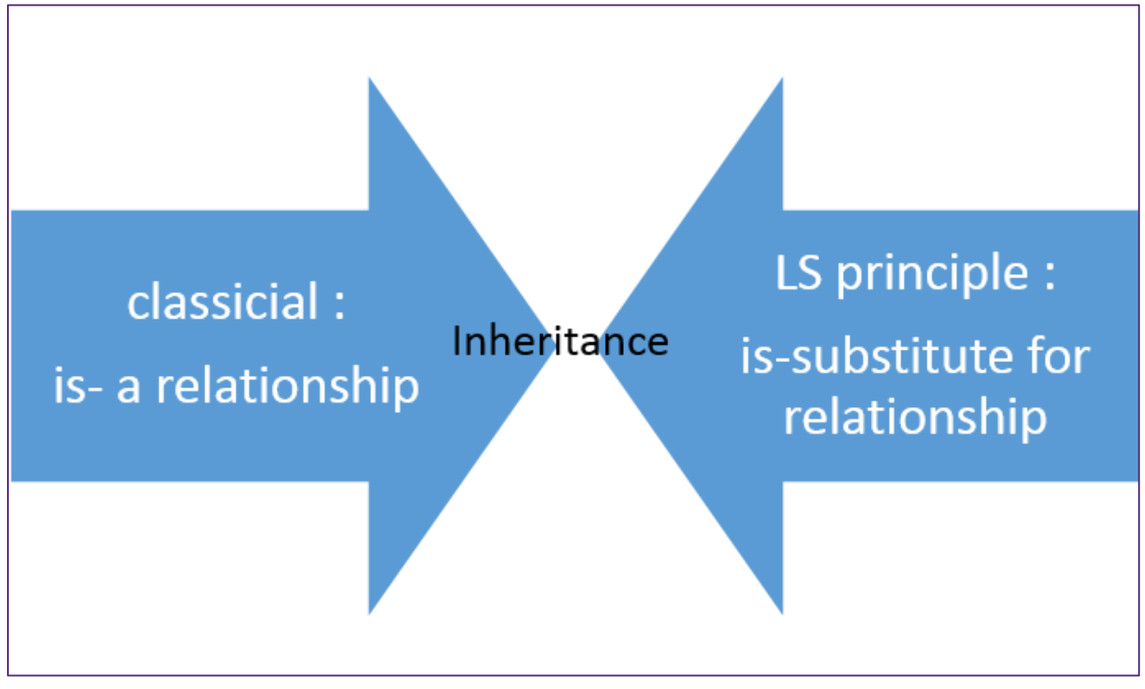
Validating the Model

- When considering whether a design is appropriate or not, it must be examined in terms of the ***reasonable assumptions*** that will be made by the users of that design.



Behavioural Problems

- A square might be a rectangle, but a ***Square object*** is *not* a ***Rectangle object***.
  - the *behaviour* of a Square object is <u>not</u> consistent with the *behaviour* of a Rectangle object.
- The LSP makes clear that inheritance relationship pertains to *behaviour* that clients depend upon.



Square

!=

Rectangle

With LSP…

“is-a” really means

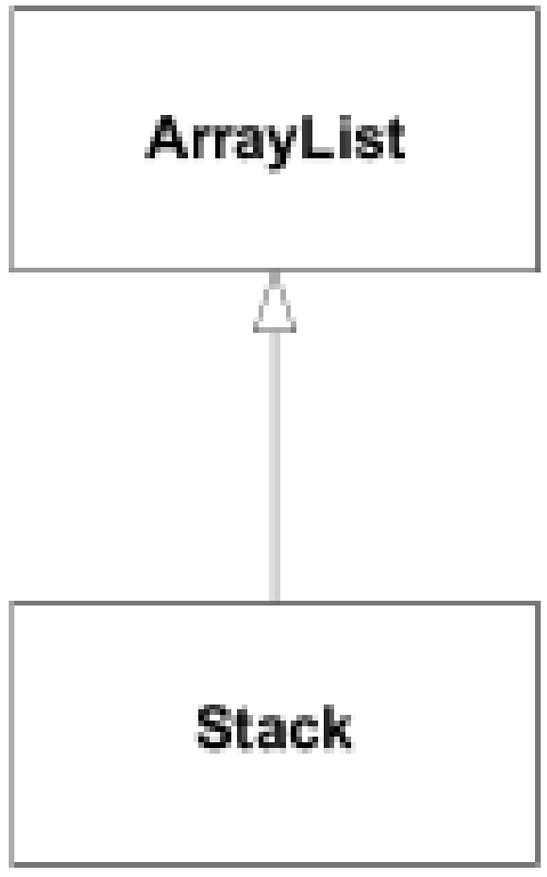
“behaves exactly like”

Fragile Base Class Problem

A common problem with Implementation Inheritance

Fragile Base Class Problem

```
class Stack extends ArrayList
{
  private int stack_pointer = 0;

  public void push( Object article )
  {
    add( stack_pointer++, article );
  }

  public Object pop()
  {
    return remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < articles.length; ++i )
      push( articles[i] );
  }
}
```



Fragile Base Class Problem

```
class Stack extends ArrayList
{
  private int stack_pointer = 0;

  public void push( Object article )
  {
    add( stack_pointer++, article );
  }

  public Object pop()
  {
    return remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < articles.length; ++i )
      push( articles[i] );
  }
}
```

Uses the ArrayList's clear() method to pop everything off the stack

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

Fragile Base Class Problem

```
class Stack extends ArrayList
{
  private int stack_pointer = 0;

  public void push( Object article )
  {
    add( stack_pointer++, article );
  }

  public Object pop()
  {
    return remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < articles.length; ++i )
      push( articles[i] );
  }
}
```

Code successfully executes, but since the base class doesn't know anything about the stack pointer, the Stack object is now in an undefined state.

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

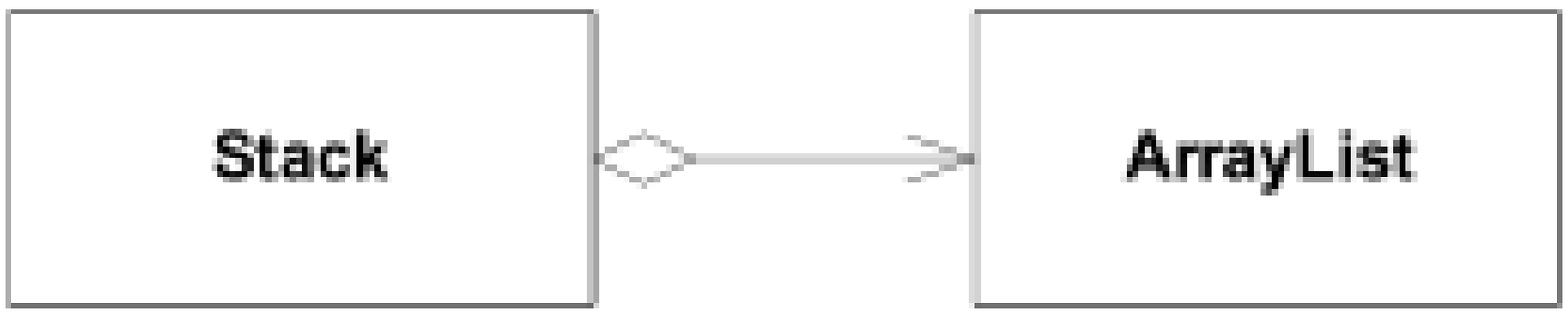
Fragile Base Class Problem

```
class Stack extends ArrayList
{
  private int stack_pointer = 0;

  public void push( Object article )
  {
    add( stack_pointer++, article );
  }

  public Object pop()
  {
    return remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < articles.length; ++i )
      push( articles[i] );
  }
}
```

The next call to push() puts the new item at index 2 (the stack_pointer's current value), so the stack effectively has three elements on it—the bottom two are garbage.

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

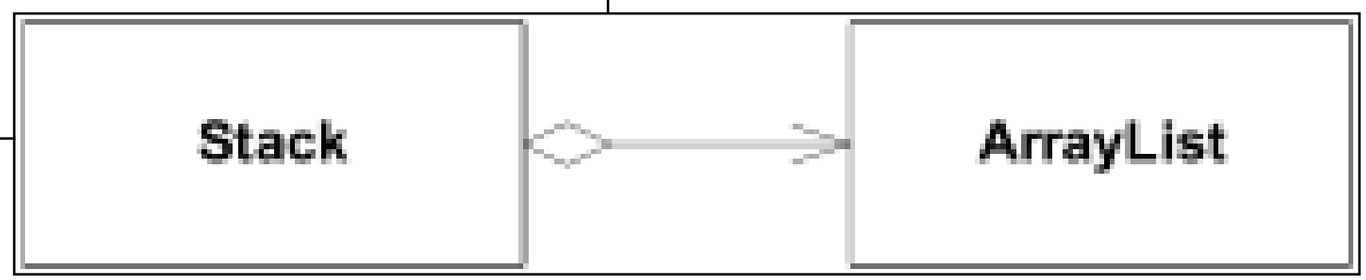
Use Composition instead of Inheritance



Inheritance is an “is-a” relationship.
Composition is a “has-a” relationship.

Composed Solution

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```



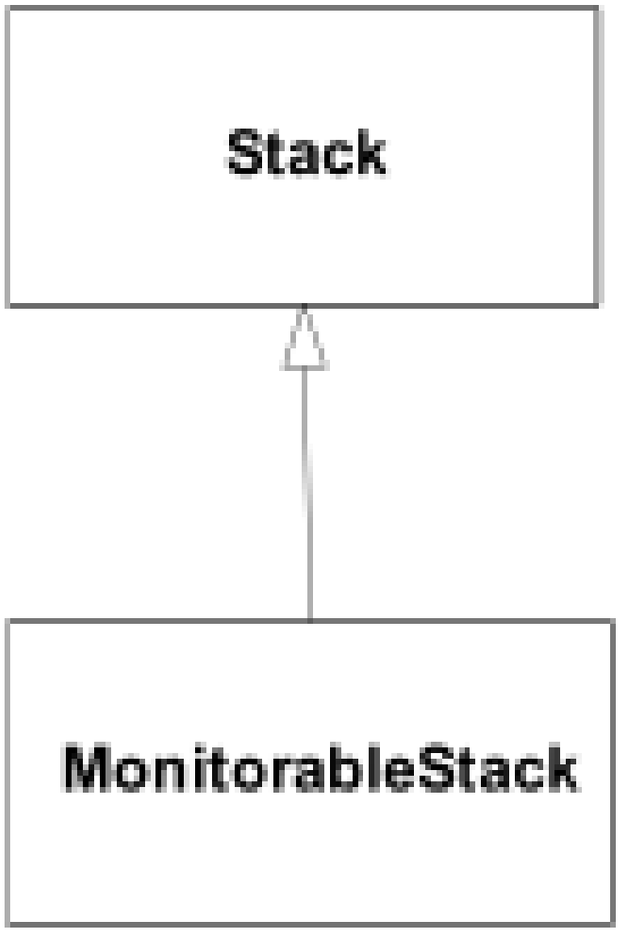
There’s no clear() method now
(so far so good)….

BUT…let’s extend the behaviour…

Monitorable Stack

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  @Override
  public void push( Object article ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }

  @Override
  public Object pop(){
    --current_size;
    return super.pop();
  }

  public int maximum_size_so_far(){
    return high_water_mark;
  }
}
```

Tracks the maximum stack size over a certain time period.



push_many Implementation

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

Which class implements `push_many()` method?

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  @Override
  public void push( Object article ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }

  @Override
  public Object pop(){
    --current_size;
    return super.pop();
  }

  public int maximum_size_so_far(){
    return high_water_mark;
  }
}
```

push_many Implementation

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

If **`f()`** is passed a **`MonitorableStack`**, does a call to **`push_many()`** update **`high_water_mark`**?

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  @Override
  public void push( Object article ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }

  @Override
  public Object pop(){
    --current_size;
    return super.pop();
  }

  public int maximum_size_so_far(){
    return high_water_mark;
  }
}
```

push_many Implementation

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

Polymorphism ensures that **MonitrableStack's push()** method is called, and **high_water_mark** is appropriately updated.

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  @Override
  public void push( Object article ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }

  @Override
  public Object pop(){
    --current_size;
    return super.pop();
  }

  public int maximum_size_so_far(){
    return high_water_mark;
  }
}
```

push_many Implementation

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

This is because **`Stack.push_many()`** calls the **`push()`** method, which is overridden by **`MonitorableStack`**

```
class Stack
{
  private int stack_pointer = 0;
  private ArrayList the_data = new ArrayList();

  public void push( Object article )
  {
    the_data.add( stack_pointer++, article );
  }

  public Object pop()
  {
    return the_data.remove( --stack_pointer );
  }

  public void push_many( Object[] articles )
  {
    for( int i = 0; i < o.length; ++i )
      push( articles[i] );
  }
}
```

```
class Monitorable_stack extends Stack
{
  private int high_water_mark = 0;
  private int current_size;

  @Override
  public void push( Object article ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    super.push(article);
  }

  @Override
  public Object pop(){
    --current_size;
    return super.pop();
  }

  public int maximum_size_so_far(){
    return high_water_mark;
  }
}
```

Revised Stack

- A profiler is run against an implementation using Stack.
- It notices the Stack isn't as fast as it could be and is heavily used.
- The base class Stack is improved i.e. rewritten so it doesn't use an ArrayList and consequently it gains a performance boost…

Revised Stack using Arrays

```
class Stack
{
  private int stack_pointer = -1;
  private Object[] stack = new Object[1000];

  public void push( Object article )
  {
    assert stack_pointer < stack.length;
    stack[ ++stack_pointer ] = article;
  }
  public Object pop()
  {
    assert stack_pointer >= 0;
    return stack[ stack_pointer-- ];
  }
  public void push_many( Object[] articles )
  {
    assert (stack_pointer + articles.length) < stack.length;
    System.arraycopy(articles, 0, stack, stack_pointer+1,
                                          articles.length);
    stack_pointer += articles.length;
  }
}
```

No longer calls push();

Problems?

```
void f(Stack s)
{
  //...
  s.push_many (someObjectArray);
  //...
}
```

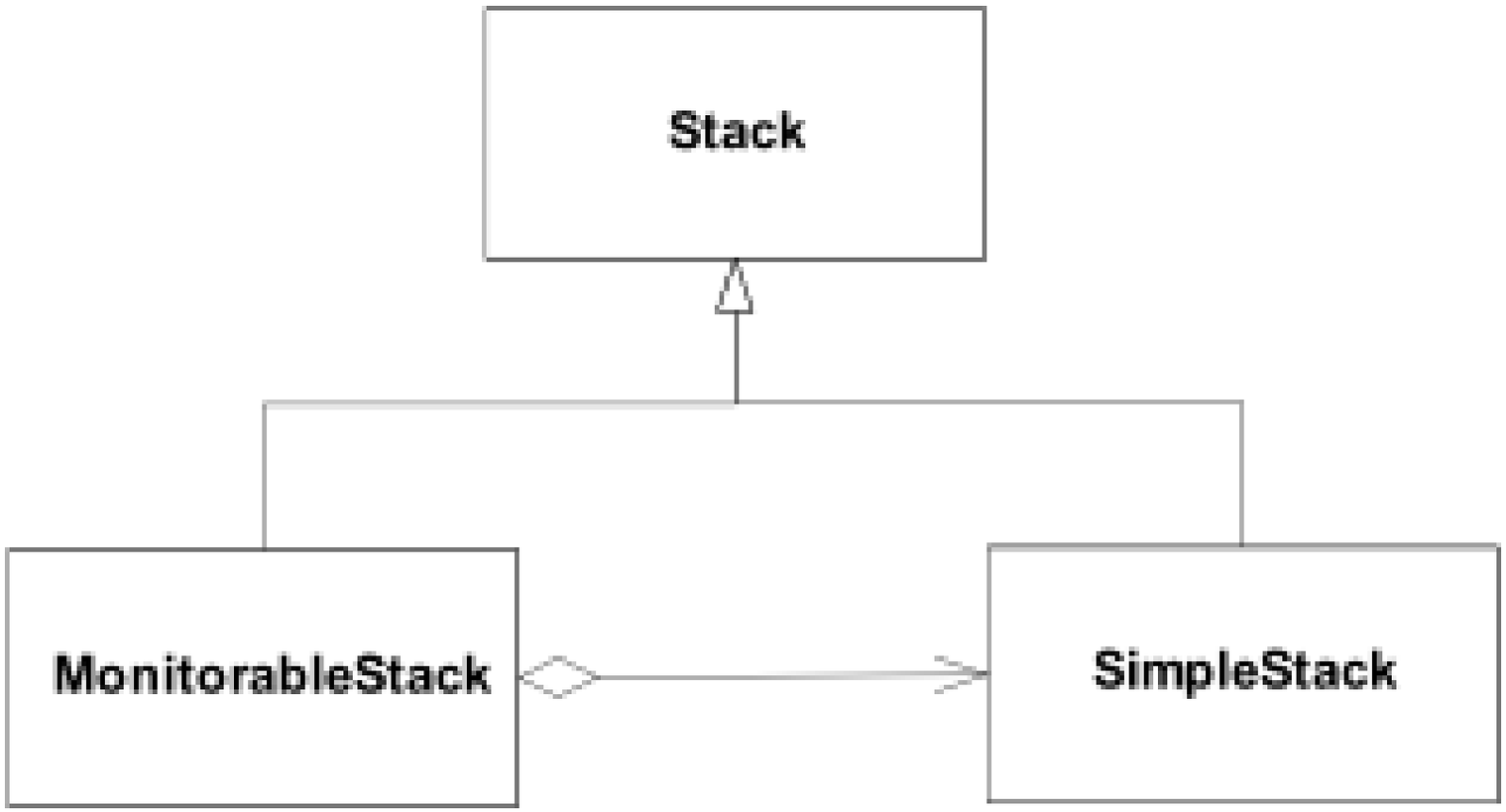
- If **`s`** is a **`MonitorableStack`**, is **`high_water_mark`** updated?

- No – because the new **`Stack`** base class **`push_many()`** implementation does not call **`push()`** at all

- LSP Violation: i.e. function **`f()`** will not appropriately operate a **`Stack`** derived object.

Solution to our Fragile Base Class

```
interface Stack
{
  void push( Object o );
  Object pop();
  void push_many( Object[] source );
}
```



MonitorableStack now USES a SimpleStack; it IS NOT a SimpleStack

Delegation / Inheritance Pattern

- Create an interface, not a class.
- Instead of implementing methods at base-class level, instead, provide a “default implementation” of those methods.
- Instead of extending the base-class, implement the interface. Then, for every interface method, delegate to a contained instance of the “default implementation”.

```
interface Stack
{
  void push( Object o );
  Object pop();
  void push_many( Object[] source );
}
```

```
class Simple_Stack implements Stack
{
    // code omitted
}
```

```
class Monitorable_Stack implements Stac
{
  private int high_water_mark = 0;
  private int current_size;
  Simple_stack stack = new Simple_stack

  public void push( Object o ){
    if( ++current_size > high_water_mar
      high_water_mark = current_size;
    stack.push(o);
  }
  //code omitted
}
```

Simple_Stack

```
class Simple_Stack implements Stack
{
  private int stack_pointer = -1;
  private Object[] stack = new Object[1000];

  public void push( Object article )
  {
    assert stack_pointer < stack.length;
    stack[ ++stack_pointer ] = article;
  }
  public Object pop()
  {
    assert stack_pointer >= 0;
    return stack[ stack_pointer-- ];
  }
  public void push_many( Object[] articles )
  {
    assert (stack_pointer + articles.length) < stack.length;
    System.arraycopy(articles, 0, stack, stack_pointer+1,
                                          articles.length);
    stack_pointer += articles.length;
  }
}
```

```
class Monitorable_Stack implements Stack
{
  private int high_water_mark = 0;
  private int current_size;
  Simple_stack stack = new Simple_stack();

  public void push( Object o ){
    if( ++current_size > high_water_mark )
      high_water_mark = current_size;
    stack.push(o);
  }

  public Object pop(){
    --current_size;
    return stack.pop();
  }

  public void push_many( Object[] source ){
    if( current_size + source.length > high_water_mark )
      high_water_mark = current_size + source.length;
    stack.push_many( source );
  }

  public int maximum_size(){
    return high_water_mark;
  }
}
```

We delegate to SimpleStack which could be a base class but isn't.

We are forced to implement push_many which also delegates to SimpleStack.

Holub's Advice

- In general, it's best to avoid concrete base classes and extends relationships in favour of interfaces and implements relationships.
- Rule of thumb : 80 percent of code at minimum should be written entirely in terms of interfaces.
  - e.g. never use references to a HashMap, use references to the Map
- The more abstraction you add, the greater the flexibility.
- In today's business environment, where requirements regularly change as the program develops, this flexibility is essential.