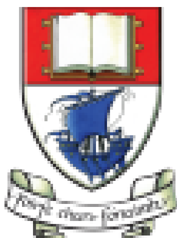
Fragile Base Class - Example

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/

Fragile Base Class Problem

A common problem with Implementation Inheritance

Fragile Base Class

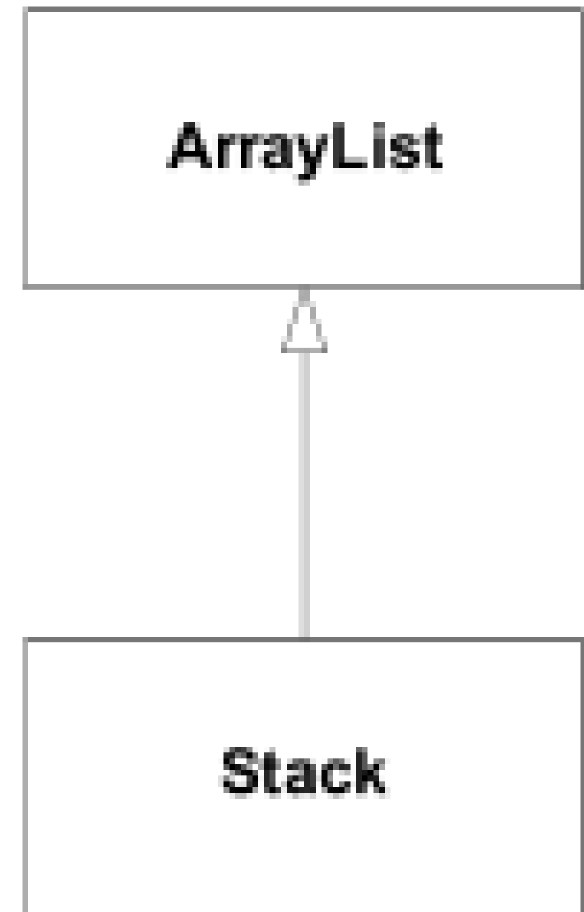
- A problem in object-oriented systems.

- “Happens when base classes (superclasses) are considered ***fragile*** because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction. The programmer cannot determine whether a base class change is safe simply by examining in isolation the methods of the base class.”

https://en.wikipedia.org/wiki/Fragile_base_class

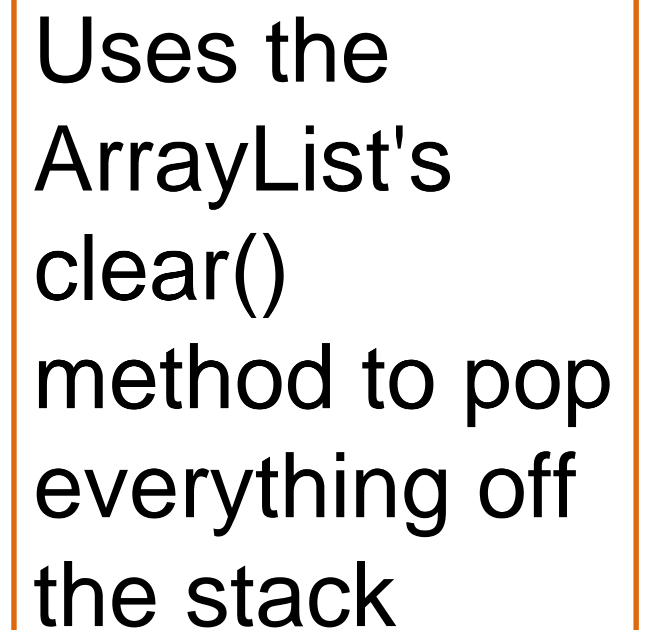
Fragile Base Class Problem

```
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] );
  }
}
```




```
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();
```

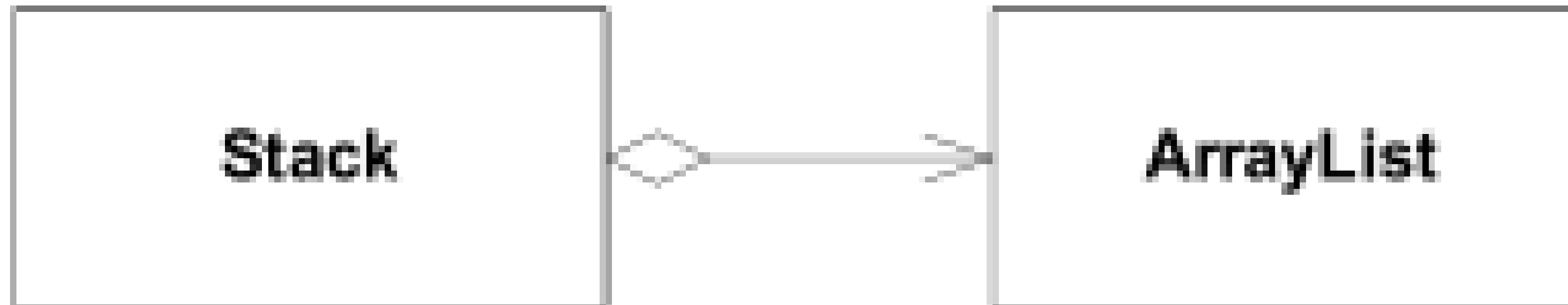
Fragile Base Class Problem

```
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();
```

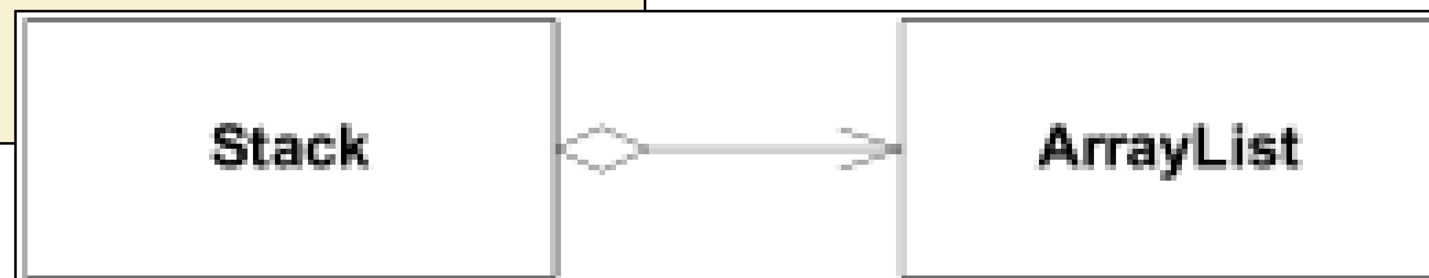
Use Composition instead of Inheritance



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] );
  }
}
```



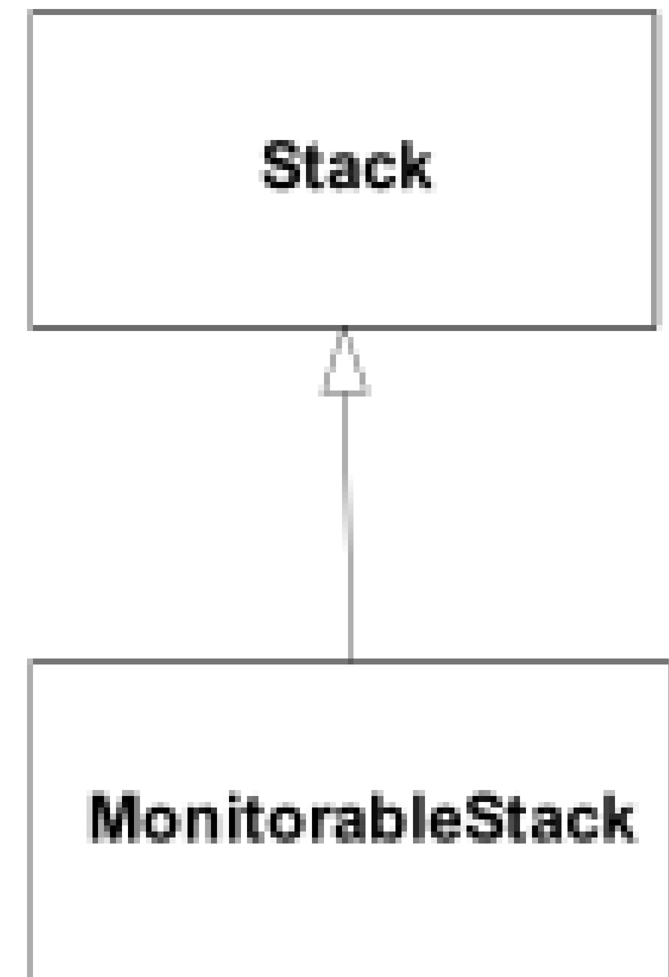
There's no clear() method now
(so far so good)….

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);
  //...
}
```

Stack implements the `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

Imagine this scenario:

- 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;
  }
}
```

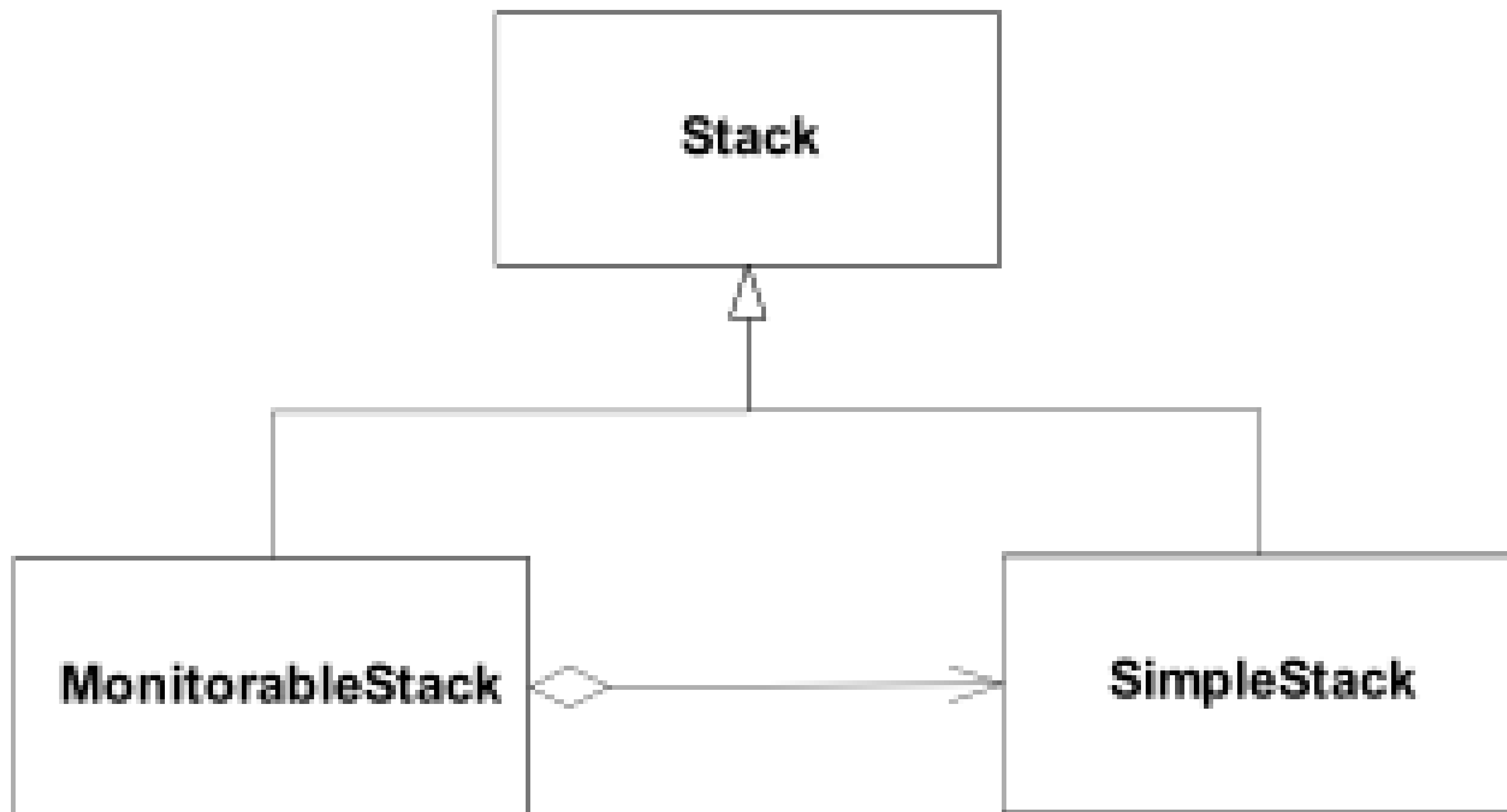
Problems?

```
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.

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

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.

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

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

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 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);
  }
  //code omitted
}
```

Simple_Stack

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

```
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;
  }
}
```

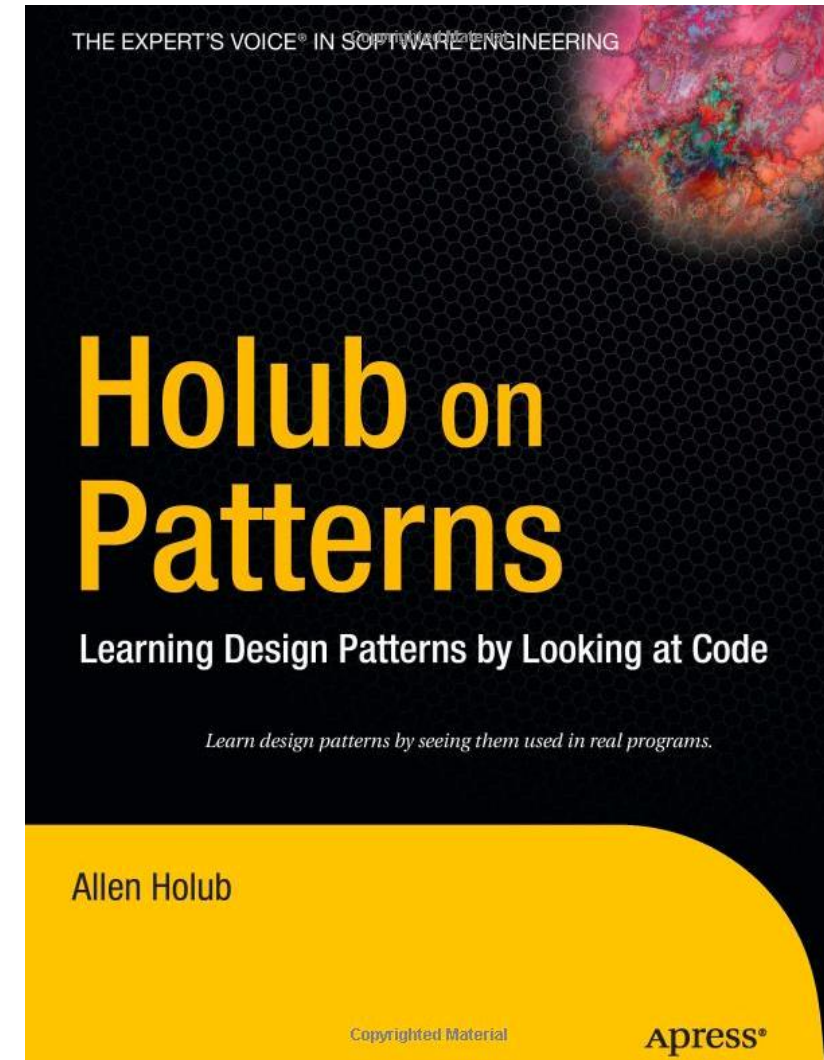
```
class Monitorable_Stack implements Stack
{
  private int high_water_mark = 0;
  private int current_size;
  Simple_stack stack = new Simple_stack();

  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;
  }
}
```

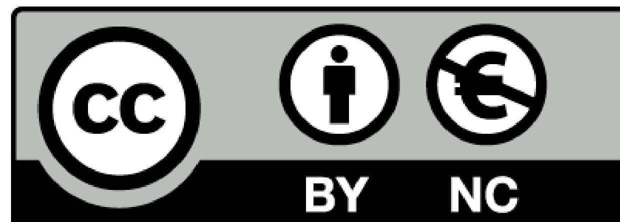
We delegate to SimpleStack which could be a base class but isn't.

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

Holub's Advice

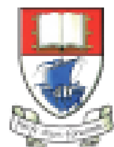
- In general, it's best to avoid concrete base classes and extends relationships in favour of interfaces and implements relationships.
- Rule of thumb : 80 percent of code at minimum should be written entirely in terms of interfaces.
  - e.g. never use references to a HashMap, use references to the Map
- 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.





Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see http://creativecommons.org/licenses/by-nc/3.0/



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit