Génie logiciel pour la conception d'un Système d'Information
CSC4521
Voie d'Approfondissement Intégration et Déploiement de Systèmes d’Information (VAP DSI)

Design Problem - Faulty Stacks and Queues

paul.gibson@telecom-sudparis.eu

http://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/CSC4521/CSC4521-DesignFaultyStackAndQueues.pdf

Architecture as compositional design

Design Problem -
Faulty Stacks and Queues

Today's session is about architecture and design (and team work).

You will be asked to design a solution to a problem (explained on the whiteboard).

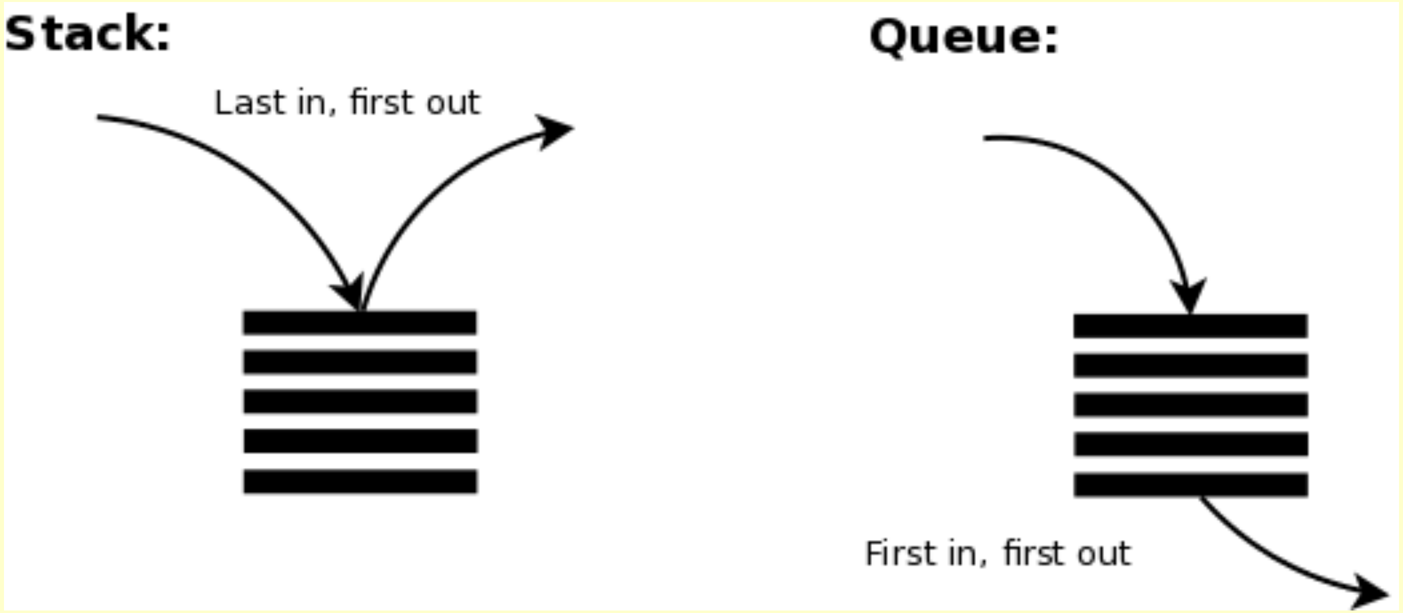
In summary, the problem will be to implement Queues using Stacks (and Stacks using Queues) and Queues using Stacks.

You will then be asked to evaluate and adapt your designs when the components are faulty.

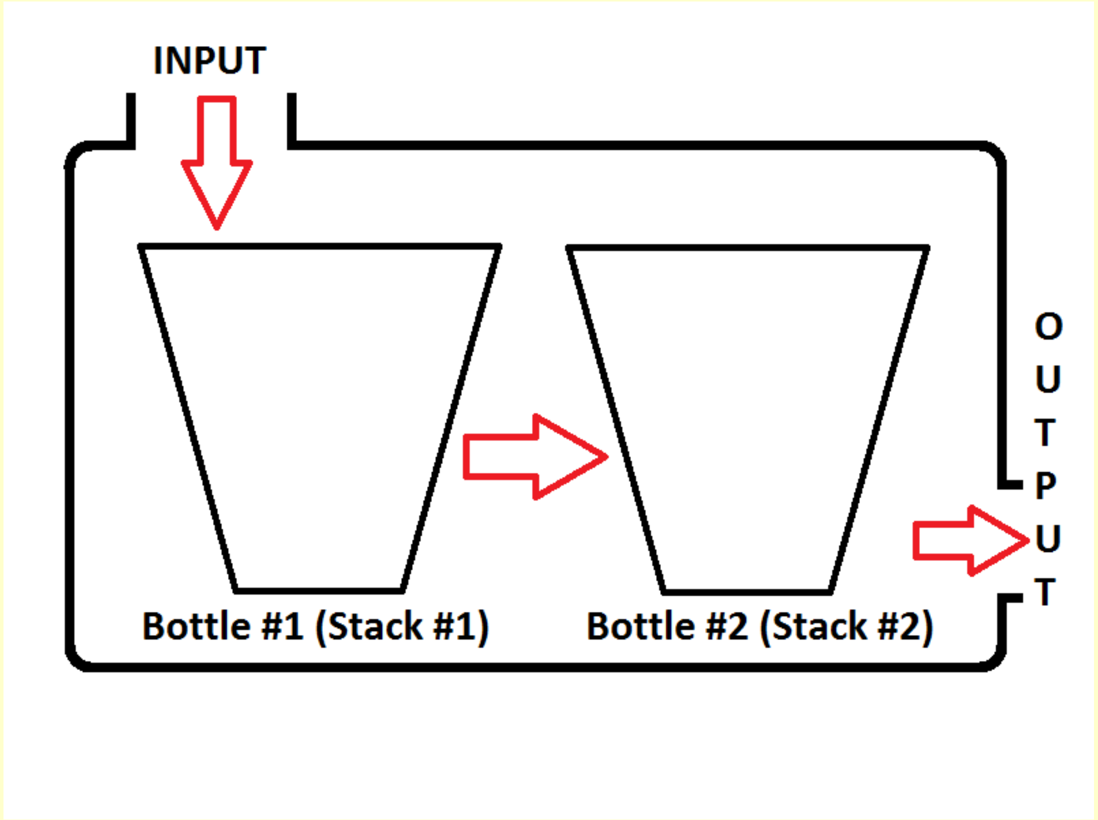
This will become clear during the session.

I will share code with you (as you request it), but you can also code everything yourselves if you wish

Given the requirements of a bounded queue (FIFO) of integer values, can we implement it using only 2 Stack components for storing the elements??

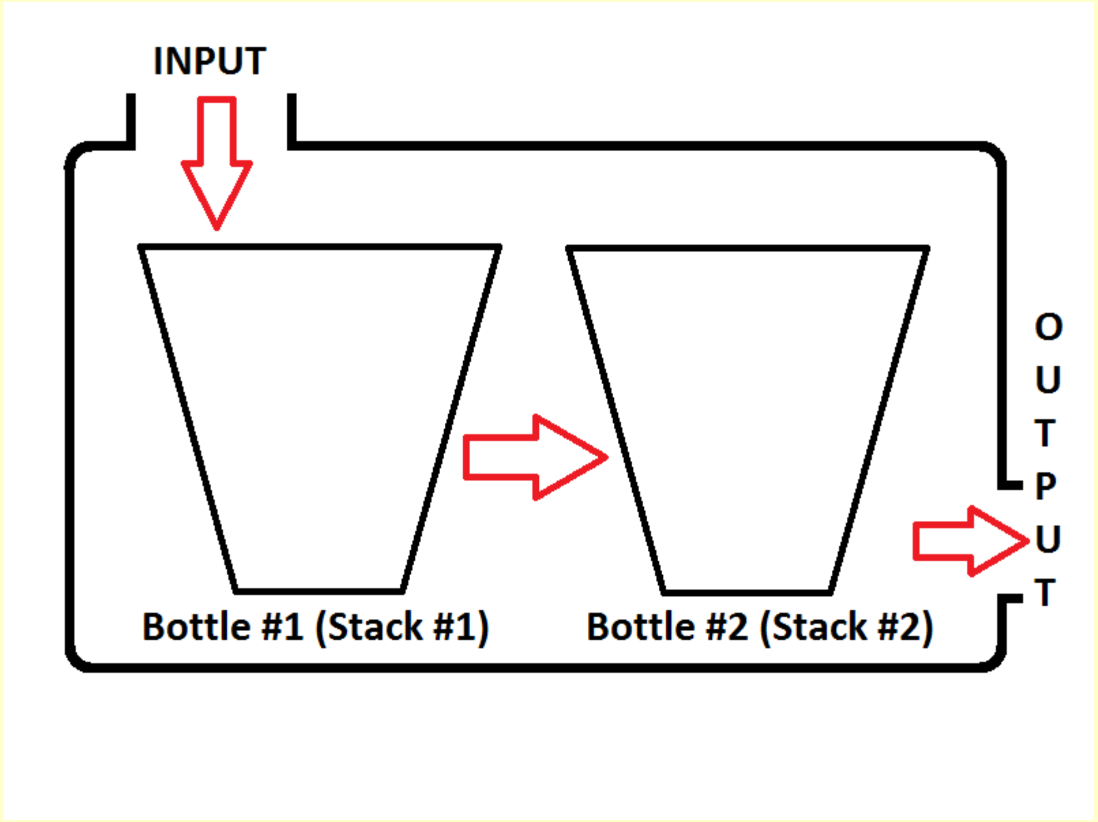


HIGH LEVEL CONCEPTUAL DESIGN



https://stackoverflow.com/questions/69192/how-to-implement-a-queue-using-two-stacks

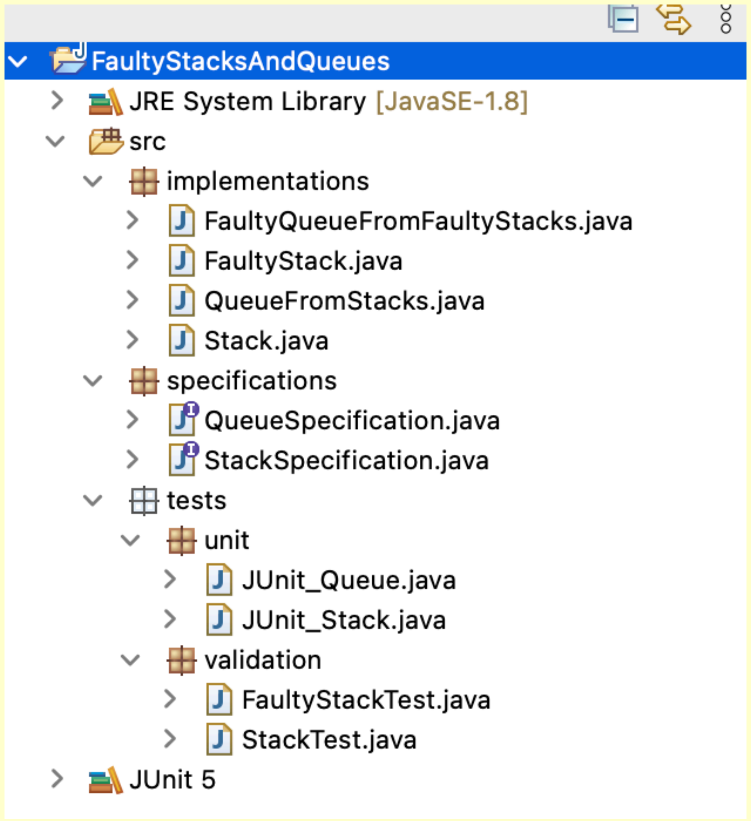
Can we improve the design if the component Stacks are **faulty**?



Fault Specification-

There is a percentage chance that on every interaction with the stack that changes its state (i.e. the pushes and pops) that the state of one of the other elements of the stack will be corrupted (incremented by 1). The element that is corrupted is randomly chosen.

Java Code project Structure



```
FaultyStacksAndQueues
  JRE System Library [JavaSE-1.8]
  src
    implementations
      FaultyQueueFromFaultyStacks.java
      FaultyStack.java
      QueueFromStacks.java
      Stack.java
    specifications
      QueueSpecification.java
      StackSpecification.java
    tests
      unit
        JUnit_Queue.java
        JUnit_Stack.java
      validation
        FaultyStackTest.java
        StackTest.java
  JUnit 5
```

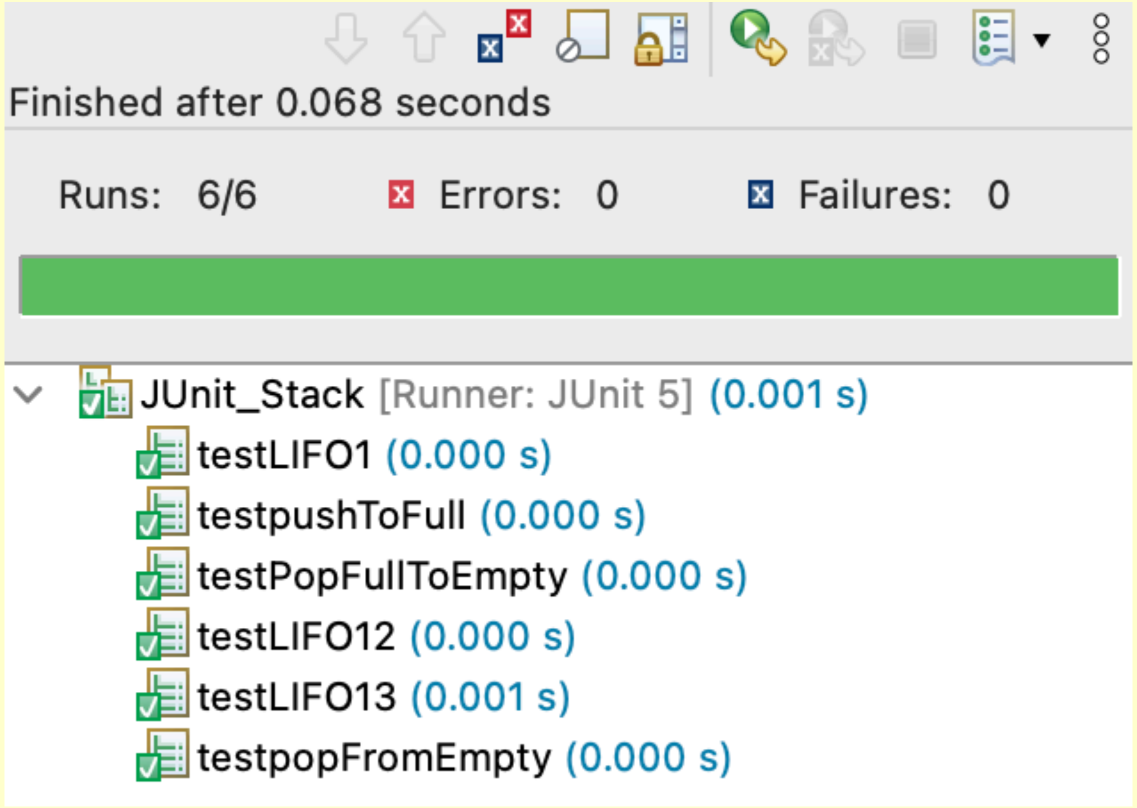
The Stack Specification - An interface + the unit tests

```
package specifications;

/**
 * A LIFO bounded stack of integers with a maximum number of elements (that must be at least 1)
 * @author J Paul Gibson
 */
public interface StackSpecification {

	/**
	 *
	 * @return the maximum number of elements that the stack can hold
	 */
	public int getSize();

	/**
	 *
	 * @return the number of elements that are currently stored on the stack
	 */
	public int getNumberOfElements();

	/**
	 *
	 * @return if the stack is full
	 */
	public boolean is_full();

	/**
	 *
	 * @return if the stack is empty
	 */
	public boolean is_empty();
```

The Stack Specification - An interface + the unit tests

```
        /**
         *
         * @param x is value being pushed onto stack
         * @throws IllegalStateException if we try to push onto a stack that is full
         */
        public void push (int x) throws IllegalStateException;


        /**
         *
         * @return  head of stack without changing state,
         * where the head is the element that has been in the stack for the shortest time
         * @throws IllegalStateException if we try to read the head of an empty stack
         */
        public int head () throws IllegalStateException;

        /**
         *  remove head of stack if it is not empty,
         *  where the head is the element that has been in the stack for the shortest time
         *  @throws IllegalStateException if we try to pop from an empty stack
         */
        public void pop () throws IllegalStateException;

        /**
         *
         * @return if the stack is in a safe state
         */
        public boolean invariant();

    }
```

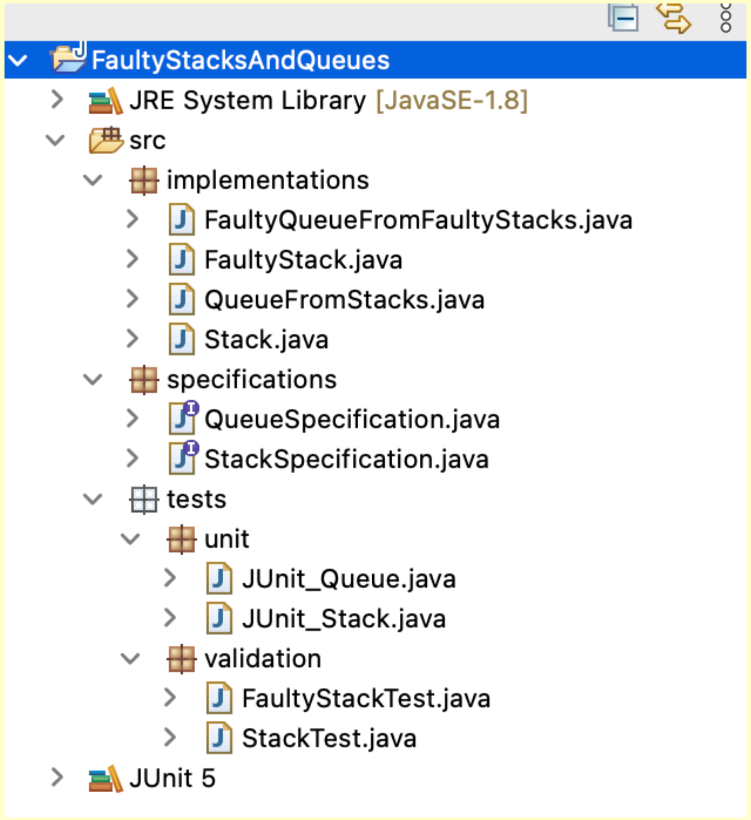
The Stack Specification - An interface + the unit tests



```
Finished after 0.068 seconds

Runs:   6/6        Errors:   0        Failures:   0

JUnit_Stack [Runner: JUnit 5] (0.001 s)
    testLIFO1 (0.000 s)
    testpushToFull (0.000 s)
    testPopFullToEmpty (0.000 s)
    testLIFO12 (0.000 s)
    testLIFO13 (0.001 s)
    testpopFromEmpty (0.000 s)
```

The Stack Specification - With a simple validation test (console I/O)

```
Stack s = new Stack (3);
Stack: head -> .
s.push(1); s.push(2); s.push(3);
Stack: head -> 3-> 2-> 1-> .
s.head() = 3
s.pop();s.pop();
Stack: head -> 1-> .
s.head() = 1
```



```
FaultyStacksAndQueues
  JRE System Library [JavaSE-1.8]
  src
    implementations
      FaultyQueueFromFaultyStacks.java
      FaultyStack.java
      QueueFromStacks.java
      Stack.java
    specifications
      QueueSpecification.java
      StackSpecification.java
    tests
      unit
        JUnit_Queue.java
        JUnit_Stack.java
      validation
        FaultyStackTest.java
        StackTest.java
  JUnit 5
```

Problem 1:
implement a Queue from 2 stacks inside the class QueueFromStacks, and test it.

Problem 2:
Implement a faulty queue from faulty stacks, and test it.

Problem 3:
Design and implement a different solution.
Analyse and Compare the 2 designs.

Problem 4:
If allowed more than 2 stack components, how could you use redundancy to make the composed system less faulty

Optional Challenge

Can you do the design (implementation and tests) where the system requirements are for a Stack and the component parts are Queues?