Génie logiciel pour la conception d'un Système d'Information

CSC4521

Voie d'Approfondissement
Intégration et Déploiement de Systèmes d'Information
(VAP DSI)

Fault Tolerance : Faulty Stacks and Queues

http://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/CSC4521/

.../CSC4521/CSC4521-FaultyStacksAndQueues.pdf

Architecture as compositional design

Design Problem - Faulty Stacks and Queues

Today's session is about architecture and design (for fault tolerance).

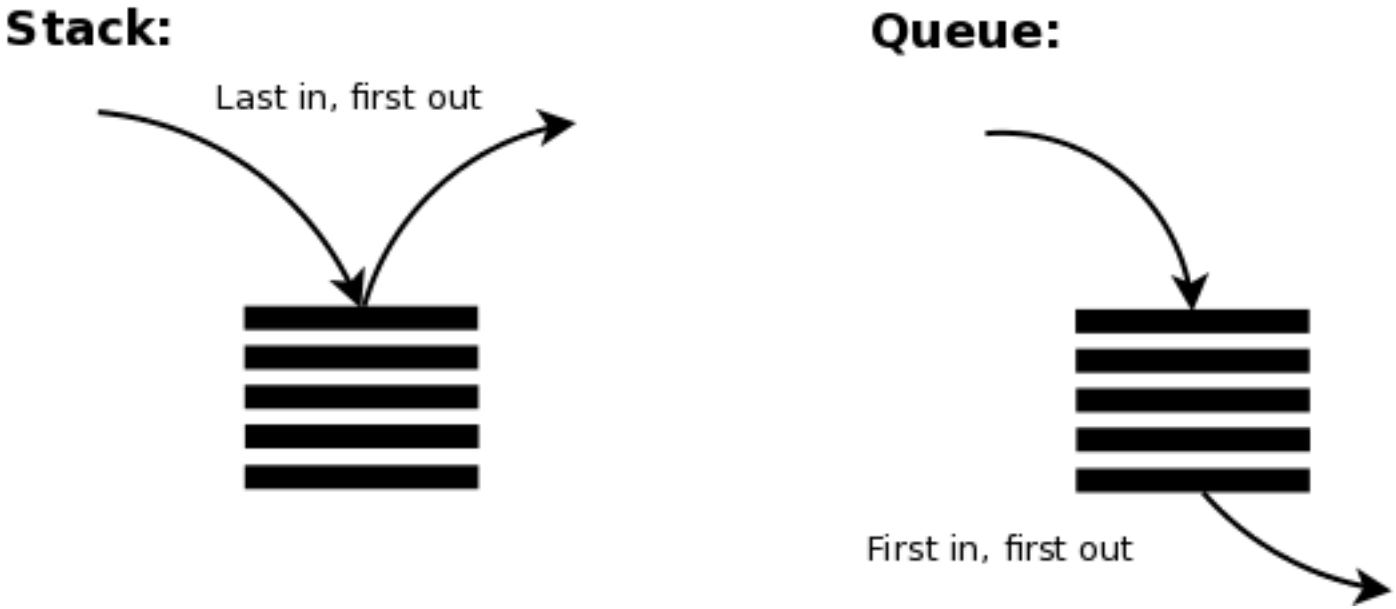
You will be asked to design a solution to a problem (explained on the whiteboard).

In summary, the problem will be to implement Queues using Stacks

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

This will become clear during the session.

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



We need a precise specification of the API offered by the **Stack**, and that required by the Queue

```
package specifications;

 * A LIFO bounded stack of integers with a maximum number of elements (that must be at least 1)
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();

     * @param x is value being pushed onto stack
    public void push (int x) throws IllegalStateException;

     * @return  head of stack without changing state,
    public int head () throws IllegalStateException;

     *  remove head of stack if it is not empty,
    public void pop () throws IllegalStateException;

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

}
```

We need a precise specification of the API offered by the Stack, and that required by the **Queue**

```
package specifications;

/* A FIFO bounded queue of integers with a maximum number of elements that must be at least 1.
public interface QueueSpecification {

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

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

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

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

    /* @param x is value being pushed onto queue
    public void push (int x) throws IllegalStateException;

    /* @return  head of queue without changing state,
    public int head () throws IllegalStateException;

    *  remove head of queue if it is not empty,
    public void pop () throws IllegalStateException;

    /* @return if the queue is in a safe state and respects the specified requirements
    public boolean invariant();

}
```

But these look almost identical

The interfaces are the same

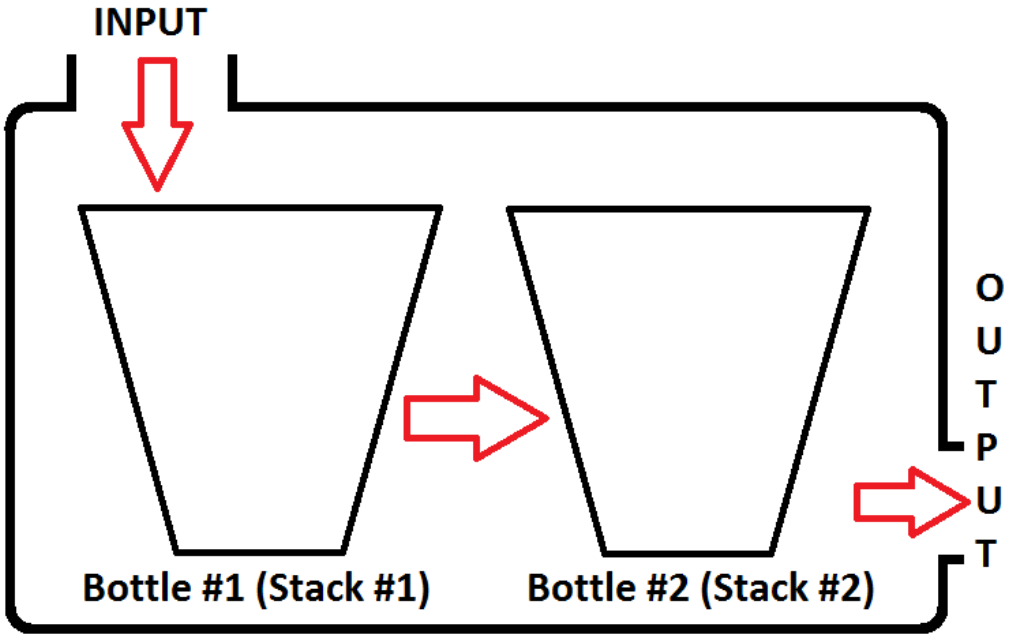
The semantics are different LIFO v FIFO

The difference is formalised in the tests that each should pass, eg:

```
@Test
public void testLIFO12(){
     testStack.push(10);
     testStack.push(20);
     Assert.assertEquals(20,testStack.head());
     Assert.assertTrue(testStack.invariant());
}
```

```
@Test
     public void testFIFO12(){
          testQueue.push(10);
          testQueue.push(20);
          Assert.assertEquals(10,testQueue.head());
          Assert.assertTrue(testQueue.invariant());
     }
```

HIGH LEVEL CONCEPTUAL DESIGN

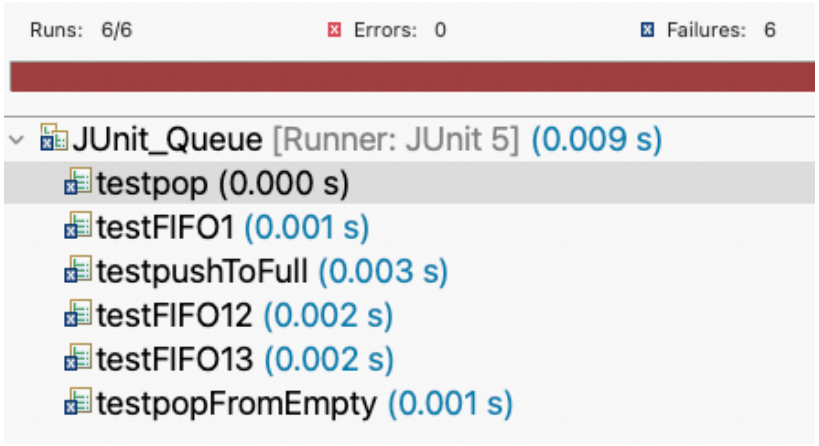


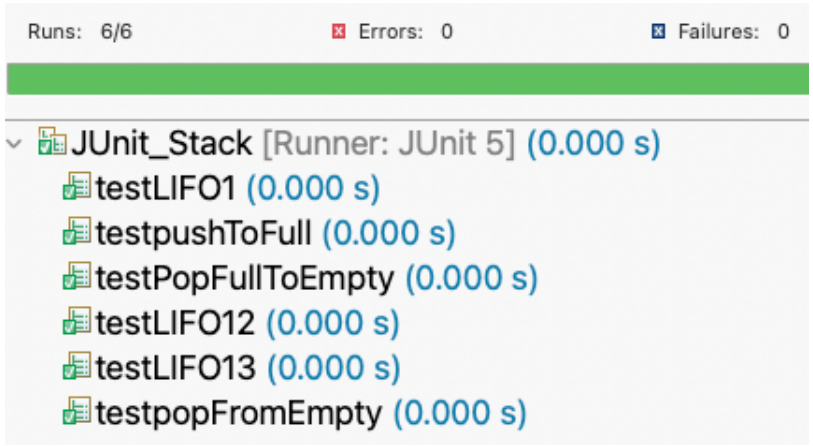
https://stackoverflow.com/questions/69192/how-to-implement-a-queue-using-two-stacks

HIGH LEVEL CONCEPTUAL IMPLEMENTATION

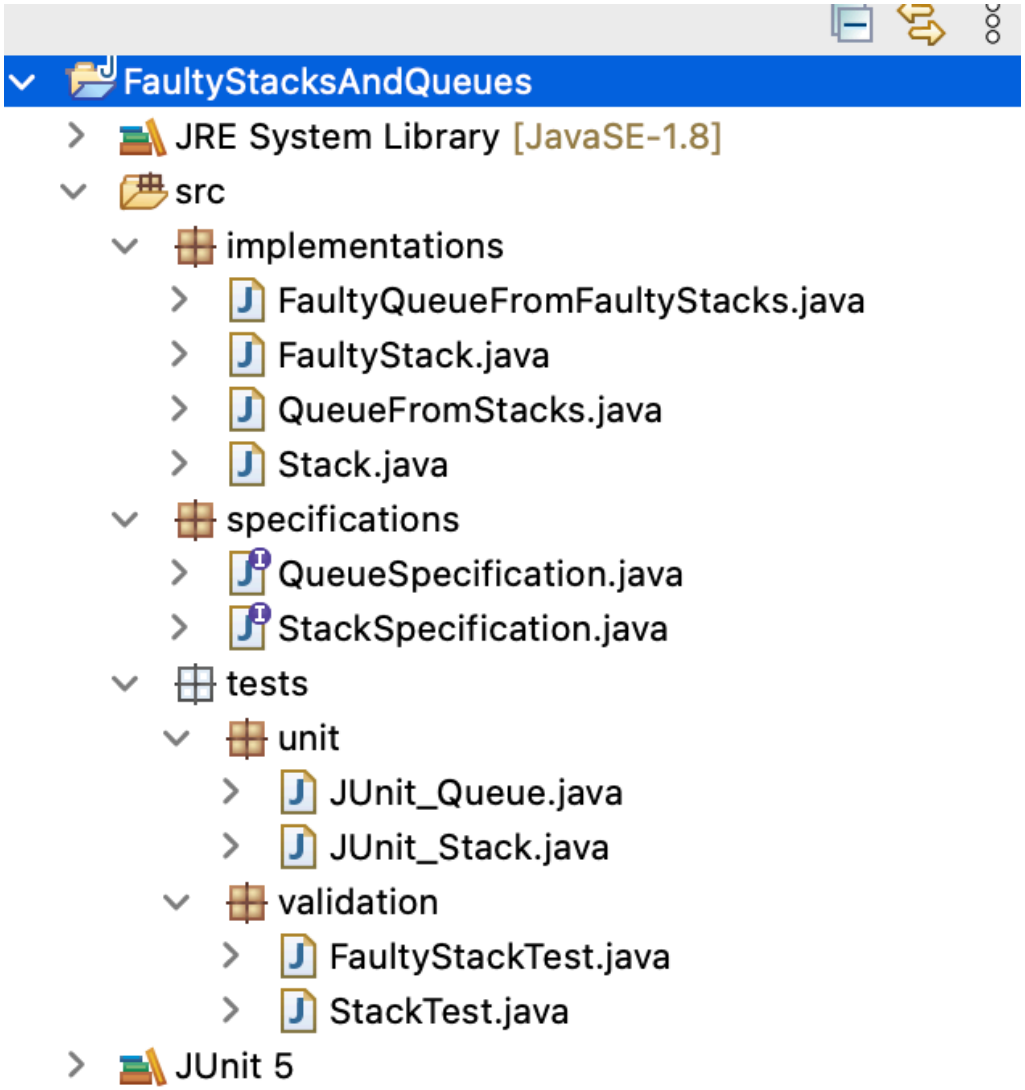
Download the Java project archive from the web site:
…/CSC4521/Code/FaultyStacksAndQueues.zip

PROBLEM 1 - Complete the design implementation so that the Queu
Unit tests, like the Stack Unit tests, pass:

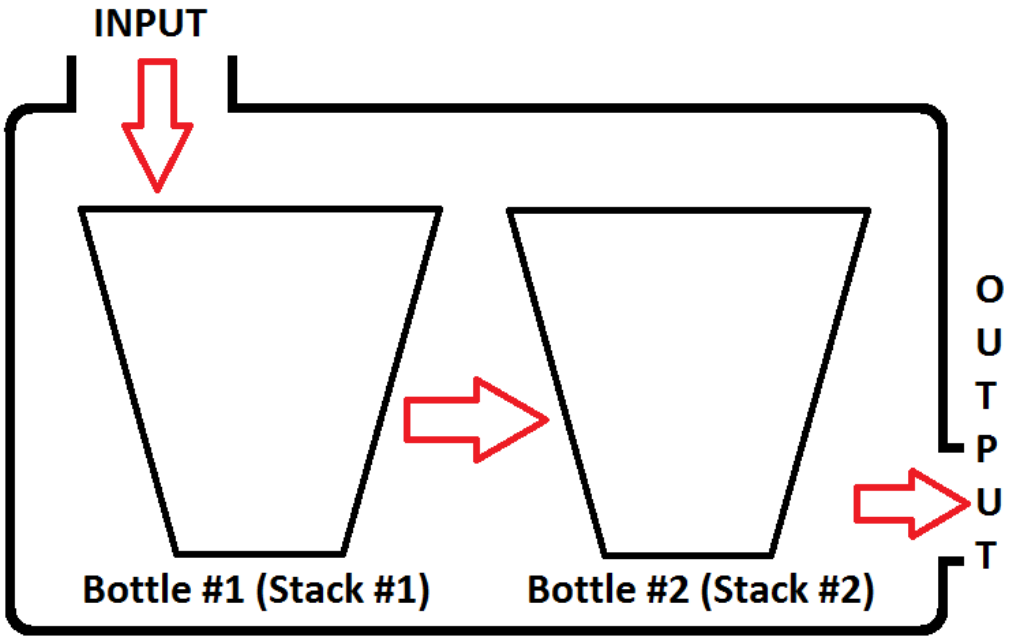




Java Code project Structure



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



Can we make a reliable system from faulty components?
Can we calculate/estimate the reliability of the system from the reliability of its components?

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

We will look at the types of faults on the blackboard

Examples (for integer data) where the fault is in the object and not in the communication with the object:

1) Faulty least significant bit - the integer data being read/ written has its LSB flipped
2) Faulty least significant bit - the integer data being read/ written is changed to 1(if it is 0)
3) Faulty least significant bit - the integer data being read/ written is changed to 0(if it is 1)
4) Random value - the integer data being read/written is changed randomly
5) A random value of the data structure is faulty (1-4)
6) A random value of the data structure is removed/ added !

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

We will look at the types of fault trigger on the blackboard

Examples (for stacks/queues):

1) When a subset of the methods are called, eg push, pop, head (pre/post execution)
2) With a probability (0..1) which is constant
3) With a probability (0..1) which is variable and depends on the state of the system (eg number of elements)
4) With a probability (0..1) which is variable and depends on the environment of the system

We can even have different probabilities for different types of fault and/or trigger!

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

In real systems, the most complex cases involve fault interactions.

- The system can tolerate faults of type F1
- The system can tolerate faults of type F2
- The system cannot tolerate faults of type F1 and F2 ‘at the same time’.

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

Some designs/architectures will be good, in general, for tolerating different types of faults

Some designs/architectures will be excellent for tolerating specific types of faults, but bad for other types

Sometimes faults are identified after deployment. Some architectures are amenable to easy change for managing this; others are not.

We can explicitly make design decisions when we know there is a possibility of faults.

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

An example provided in the project archive:

```
package implementations;
import specifications.StackSpecification;


/**
 * Implements {@link StackSpecification} interface using a linked list of {@link intLink}s
 *
 * The stack is faulty (for teaching purposes). 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.
 * @author J Paul Gibson
 */
public class FaultyStack  extends Stack {


    /**
     * Percentage value should be between 0 and 100 - the probability that there is a faulty
     * push or pop that corrupts the stack state. Default value is 10. This can be changed in
     * the constructor.
     */
     private int faultPercentage =10;

     /**
      * The random number generator used to generate random faults
      */
     private final Random RNG = new Random();
```

Problem 2:

Implement a faulty queue from the given faulty stacks, and test it.

Is it more/less reliable than the faulty stacks from which it is built?

Could a change to the design/architecture change the probability of faults in the composed system

Test your hypothesis. You will need to use a **test oracle** (correct implementation of the system) to compare results with the proposed implementation of the system

Problem 3 : Adding fault tolerance to the components

For which types of fault can we make a local change to a component that reduces/removes the probability of a fault occurring.

For example:

Faulty least significant bit - the integer data being read/ written has its LSB flipped - on the method push

TODO - replace the FaultyStack with a FaultyStack proxy that does some pre/post processing in order to correct/ reduce the fault?

NOTE - Such solutions need care if we need *thread safety*.

Problem 4 : Adding fault tolerance to the design/ architecture (when it is difficult/impossible to do it at the component level)

Redundancy is often a good way of tolerating faults.

If you had more than 2 faulty stacks to implement your queue would it be possible to use the additional redundancy to improve queue reliability?

Problem 5 (optional):

We have implemented Queues using stacks, and analysed the fault tolerance aspects.

Now do the same thing for Stacks implemented using queues