Test Case Exercises (10 Study points - mandatory)

Equivalence classes

1. Make equivalences classes for the input variable for this method:

```
public boolean isEven(int n)
```

2. Make equivalences classes for an input variable that represents a mortgage applicant's salary. The valid range is $1000 pr. month to $75,000 pr. month

3. Make equivalences classes for the input variables for this method:

```
public static int getNumDaysinMonth(int month, int year)
```

Boundary Analysis

1. Do boundary value analysis for input values exercise 1
2. Do boundary value analysis for input values exercise 2
3. Do boundary value analysis for input values exercise 3

Decision tables

1. Make a decision table for the following business case:

   No charges are reimbursed (DK: refunderet) to a patient until the deductible (DK: selvrisiko) has been met. After the deductible has been met, reimburse 50% for Doctor's Office visits or 80% for Hospital visits.

Conditions				
Actions				

2. Make a decision table for leap years.

    Leap year: Most years that are evenly divisible by 4 are leap years.
    An exception to this rule is that years that are evenly divisible by 100 are *not* leap years, unless they are also evenly divisible by 400, in which case they are leap years.

Conditions								
Actions								

State transition

State transition testing is another black box test design technique where test cases are designed to execute valid and invalid transitions.

Use this technique to test the class `MyArrayListWithBugs.java` (find code on last page). It is a list class implementation (with defects) with the following methods:

Method / Specification

```
public MyArrayListWithBugs ()  // Creates new empty list

public void add(Object o)   // Inserts object in the end of list

public int size()                  // Returns number of objects in list

public Object get(int index)       // Returns reference to object at position
index                              // Throws IndexOutOfBoundsException

public void add(int index, Object e)  // Inserts object at index position

                                   // Throws IndexOutOfBoundsException

public Object remove(int index)    // Removes object at index position

                                   // returns a reference to removed object
                                   // Throws IndexOutOfBoundsException
```

1.
Make a state diagram that depicts the states of `MyArrayListWithBugs.java` and shows the events that cause a change from one state to another (i.e. a transition).

2.
Derive test cases from the state diagram.

3.
Implement automated unit tests using the test cases above.

4.
Detect, locate (and document) and fix as many errors as possible in the class.

a. Define (more) relevant test cases applying black box and white box techniques
b. Use xUnit to implement and run the same tests cases again after fixing
c. Study the implementation (code)
d. Use debugger to locate errors

5.
Consider whether a state table is more useful design technique. Comment on that.

6.
Make a conclusion where you specify the level of test coverage and argue for your chosen level:

- Percentage of states visited
- Percentage of transitions exercised

Formalities

Hand-in on Moodle: Document with text descriptions + link to code on Github

Code Deliverables: Automated unit tests for `MyArrayListWithBugs.java` + revised version of class.

Deadline: February 25th at noon

Code

```
package myBugs;
/**
 * Class with bugs.
 *
 *
 */
public class MyArrayListWithBugs {

    private Object[] list;
    int nextFree;

    // Creates a new empty list
    public MyArrayListWithBugs()
    {
        list     = new Object[5];
        nextFree = 0;
    }

    // Inserts object at the end of list
    public void add(Object o)
    {
        // check capacity
        if (list.length <= nextFree)
            list = getLongerList();

        list[nextFree] = o;
        nextFree++;
    }

    // Returns the number of objects in the list
    public int size()
    {
        return nextFree;
    }

    // Returns a reference to the object at position index
    // Throws IndexOutOfBoundsException
    public Object get(int index)
    {
        if(index <= 0 || nextFree < index)
            throw new IndexOutOfBoundsException("Error (get): Invalid index" +
index);

        return list[index];
    }

    // Inserts object at position index
    // Throws IndexOutOfBoundsException
    public void add(int index, Object o)
    {
        if(index < 0 || nextFree < index)
            throw new IndexOutOfBoundsException("Error (add): Invalid index" +
index);

        // check capacity
        if (list.length <= nextFree)
```

```
            list = getLongerList();

        // Shift elements upwards to make position index free
        // Start with last element and move backwards
        for (int i = nextFree-1; i > index; i--) {
            list[i] = list[i-1];
        }

        list[index] = o;
    }

    // Removes object at position index
    // Returns a reference to the removed object
    // Throws IndexOutOfBoundsException
    public Object remove(int index)
    {
        if (index < 0 || nextFree <= index)
             throw new IndexOutOfBoundsException("Error (remove): Invalid index"
+ index);


        // Shift elements down to fill indexed position
        // Start with first element
        for (int i = index; i < nextFree-1; i++) {
            list[i] = list[i+1];
        }
        nextFree--;

        return list[index];

    }

    //============== private helper methods ==========
    // create a list with double capacity and
    // copy all elements to this
    private Object[] getLongerList() {
       Object[] tempList = new Object[list.length*2];
        for (int i=0; i< list.length;i++) {
            tempList[i] = list[i];
        }
        return tempList;
    }


}
```