Assignment 2

Exercise 1

You have seen different ways of documenting the effects of a method in the lecture (e.g., checked exceptions, read and write effects). Purity annotations are a specific kind of write effect specification: a *pure* method i) does not modify any object that already existed in the pre-state (but it can modify newly created objects), and ii) for given set of arguments (including the receiver) the return value (if any) is always the same. For instance, in the code below method `equals` and method `hashcode` are probably pure, while method `getImage` is definitely not pure since it modifies the current object by initializing the image field.

```
class ImageFile {
  String file;
  Image image;

  ...

  public Image getImage() {
    if (image == null) {
      Image tmp = new Image();
      // load the image
      image = tmp;
    }
    return image;
  }

  boolean equals(Object o) {
    if( o.getClass() != getClass() ) return false;
    return file.equals( ((ImageFile) o).file );
  }

  int hashcode() {
    if (image == null) {
      return file.hashcode();
```

```
    } else {
      return image.hashcode() + file.hashcode();
    }
  }
}
```

1. Why are method `equals` and method `hashcode` *probably* pure? Under which circumstances are they not pure? Is is possible to change the class design such that they are pure under all circumstances?

2. Can you think of a practical solution that would catch (at runtime) violations of the first requirement? Apply the instrumentation to the code from above.

3. How would your instrumentation deal with commonly used designs, such as lazy initialization of data structures or caching?

Exercise 2

Find preconditions and object invariants for all methods of the class on page 4, and postconditions for the Add method. Express them using the syntax of C#'s Code Contracts:

- At the beginning of each method, `Contract.Requires(expr);` can be used to denote a precondition. Here, `expr` should be a pure boolean C# expression referring only to fields and methods with greater or equal visibility as the method.
- Similarly, `Contract.Ensures(expr);` can be used to denote a postcondition.
- If needed, `Contract.ForAll(lower, upper, pred);` can be used to express that the predicate `pred` holds for all integers from `lower` (inclusive) to `upper` (exclusive).
- `Contract.OldValue(expr)` can be used in a postcondition to refer to the value of `expr` before the execution of the method.
- Object invariants are denoted in a special contract invariant method. Again, the visibility of all referred fields must be greater or equal to the visibility of the contract invariant method.

```
[ContractInvariantMethod]
private void ObjectInvariant() {
        Contract.Invariant(expr);
        ...
}
```

- Methods can be marked with the `[Pure]` attribute (in the line before the method declaration) to be pure. Only pure methods can be used in contract expressions.

```
public class Bag
{
    private int[] elems;
    private int count;

    public Bag(int[] initialElements) {
        this.count = initialElements.Length;
        int[] e = new int[initialElements.Length];
        initialElements.CopyTo(e, 0);
        this.elems = e;
    }

    public Bag(int[] initialElements, int start, int howMany) {
        this.count = howMany;
        int[] e = new int[howMany];
        Array.Copy(initialElements, start, e, 0, howMany);
        this.elems = e;
    }

    public int Count() {
        return count;
    }

    public int RemoveMin() {
        int m = System.Int32.MaxValue;
        int mindex = 0;
        for (int i = 0; i < count; i++) {
            if (elems[i] < m) {
                mindex = i;
                m = elems[i];
            }
        }
        count--;
        elems[mindex] = elems[count];
        return m;
    }

    public void Add(int x) {
        if (count == elems.Length) {
            int[] b = new int[2*elems.Length];
            Array.Copy(elems, 0, b, 0, elems.Length);
            elems = b;
        }
        elems[count] = x;
        count++;
    }

}
```

Exercise 3

1. Draw a UML class diagram for the system described below:
   (a) every student is either undergraduate or graduate student. No student is both undergraduate and graduate student;
   (b) a student should register at a university, and only registered students are legal students;
   (c) every student has a unique student ID, and he or she has only one major;
   (d) students with the same major are regarded as classmates, students can have several classmates.
2. Which properties of the system above cannot be captured using UML class diagrams?