

Assignment 2 (Solution)

Exercise 1

1. Q: Why are method `equals` and method `hashCode` *probably* pure? Under which circumstances are they not pure?

A: They are *probably* pure because we don't know the implementation of `String.equals` and `Image.hashCode` which might violate the purity specification.

Q: Is it possible to change the class design such that they are pure under all circumstances?

A: One possibility would be to provide a pure implementation of methods `String.equals` and `Image.hashCode`. Note that if we extract them into separate methods we then need to make them final (otherwise subclasses might override them with non-pure version).

2. One could instrument the following program operations:
 - *object allocations* to keep track of newly allocated objects that can be modified inside pure methods
 - *field writes* to check whether the method is allowed to modify given field
 - *method entry/exit* to update global instrumentation state depending on whether the method is pure or not.

The following shows a sketch of non-thread safe instrumentation:

```
class PurityChecks
{
    // Contains objects that pure method is allowed to modify
    static IdentityHashSet<Object> fresh = new IdentityHashSet<Object>();
    // Denotes whether we are restricting method
    // to modify only freshly created objects.
    // Set to true for all methods annotated as @Pure
    // and all methods called transitively from @Pure methods
}
```

```

    static boolean checking = false;
}
class ImageFile
{
    String file;
    Image image;

    public Image getImage() {
        if (this.image == null) {
            Image img = new Image();
            // whenever an object is created we add
            // it to the set of object that can be modified
            + PurityChecks.fresh.insert(img);
            ... load image

            // whenever we modify an object
            // we check if this is allowed
            + assert !PurityChecks.checking || PurityChecks.fresh.contains(this);
            this.image = img;
        }

        return image;
    }

    @Pure
    boolean equals(Object o) {
        // Since this method has @Pure annotation we:
        // 1. Save the state of the caller
        + boolean checking = PurityChecks.checking;
        + IdentityHashSet<object> fresh = PurityChecks.fresh;
        // 2. Initialize the PurityChecks
        // fresh set is initially empty as we are allowed to
        // modify only newly created objects
        + PurityChecks.checking = true;
        + PurityChecks.fresh = new IdentityHashSet<object>();

        if( o.getClass() != getClass() ) return false;
        return file.equals( ((ImageFile) o).file );

        // add newly allocated objects to the global set to allow
        // their modification after we return from this method
        + fresh.addAll(PurityCheck.fresh);
        // Restore the state of caller,
        // should be executed before each return statement
    }
}

```

```

    + PurityChecks.checking = checking;
    + PurityChecks.fresh = fresh;
}

@Pure
int hashCode() {
    + var checking = PurityChecks.checking;
    + var fresh = PurityChecks.fresh;
    + PurityChecks.checking = true;
    + PurityChecks.fresh = new IdentityHashSet<object>();

    if (image == null) {
        return file.hashCode();
    } else {
        return image.hashCode() + file.hashCode();
    }

    + fresh.addAll(PurityCheck.fresh);
    + PurityChecks.checking = checking;
    + PurityChecks.fresh = fresh;
}
}

```

3. For such designs, the instrumentation would report errors even if the effects can't be observed by a client. We could weaken the definition of purity for such designs.

Exercise 2

Main insight: `elems.Length` must be strictly greater than 0 so that `Add` works, therefore we get the preconditions `initialElements.length > 0` and `howMany > 0` in the constructors.

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

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(elems != null);
        Contract.Invariant(0 < elems.Length);
        Contract.Invariant(0 <= count && count <= elems.Length);
    }

    public Bag(int[] initialElements) {
        Contract.Requires(initialElements != null);
        Contract.Requires(0 < initialElements.Length);
        ...
    }

    public Bag(int[] initialElements, int start, int howMany) {
        Contract.Requires(0 <= start);
        Contract.Requires(0 < howMany);
        Contract.Requires(initialElements != null);
        Contract.Requires(start + howMany <= initialElements.Length);
        ....
    }

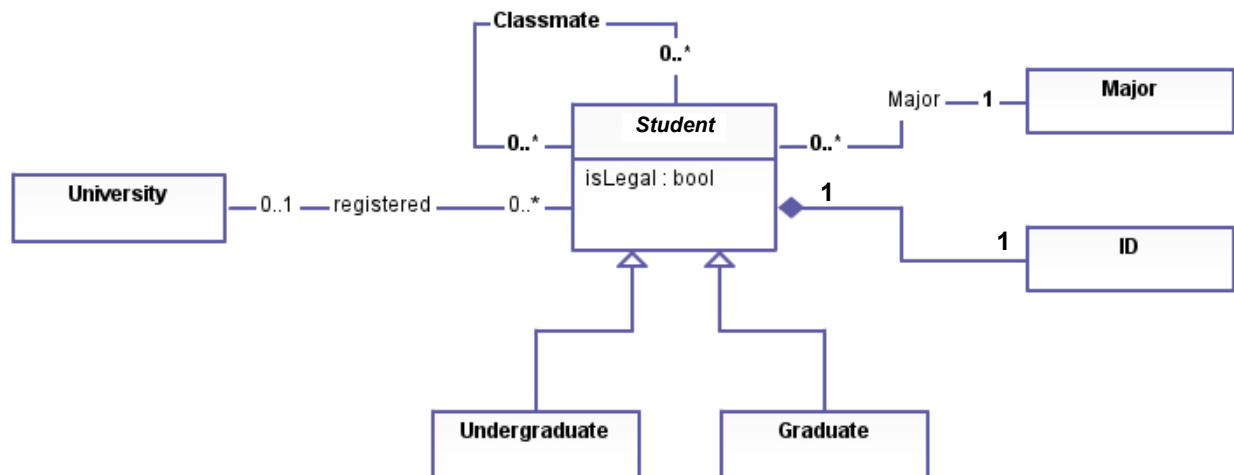
    [Pure]
    public int Count() {
        ....
    }

    public int RemoveMin() {
        Contract.Requires(0 < Count());
        ....
    }

    public void Add(int x) {
        Contract.Ensures(Count() == Contract.OldValue(Count()) + 1);
        Contract.Ensures(elems[Contract.OldValue(Count())] == x)
        Contract.Ensures(!Contract.OldValue(Count() == elems.Length) ||
            elems.Length == 2*Contract.OldValue(elems.Length))
        Contract.Ensures(Contract.ForAll(0, Count() - 1, i =>
            elems[i] == Contract.OldValue(elems[i])))
        ....
    }
}
```

Exercise 3

1. Student Class diagram



2. (a) Classmates have the same major
(b) A student is legal iff he/she is registered