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 is 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.

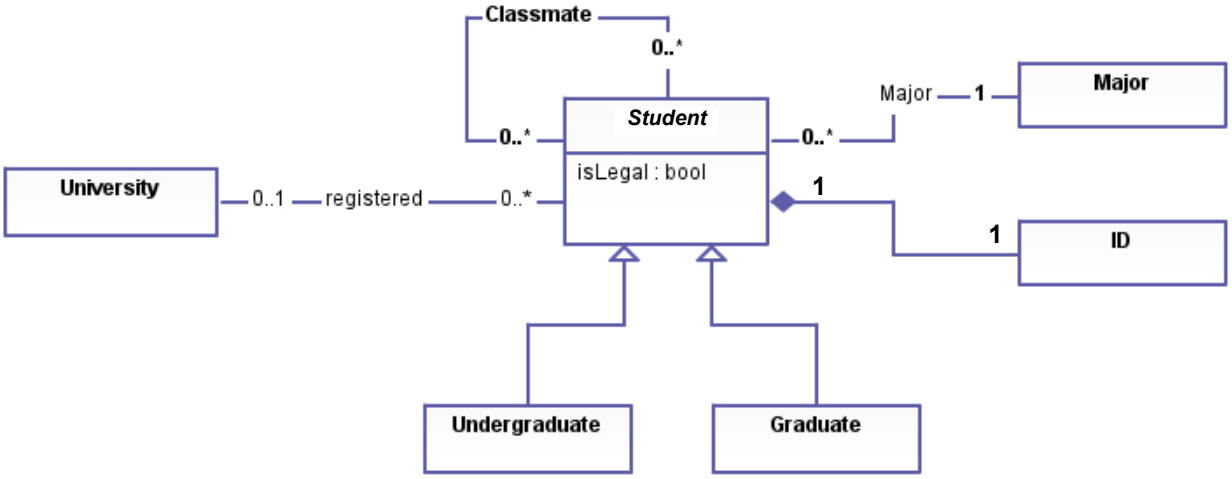
Exercise 2

```
class ImageFile {
  @invariant file != null
  @invariant file == old(file)
  private final String file;
  private Image image;

  @requires f != null
  // we can check file validity assuming that
  // following helper functions are defined
  @requires exists(f) and readable(f)
  @ensures image == null
  ImageFile(String f) {
    file = f;
    image = null;
  }
```

```
  public String getFile() {
    return file;
  }

  //either the image was null in which case it is initialized or
  //the image is not changed
  @ensures (old(image) == null and image != null) or
           (old(image) != null and old(image) == image)
  // simple version of the above (redundant)
  @ensures image != null
  @ensures result != null
  public Image getImage() {
    if (image == null) {
      // load the image
    }
    return image;
  }

  @requires o != null
  @ensures image = old(image)
  public boolean equals(Object o) {
    if( o.getClass() != getClass() ) return false;
    return file.equals( ((ImageFile) o).file );
  }

  @ensures image = old(image)
  public int hashcode() {
    if (image == null) {
      return file.hashcode();
    } else {
      return image.hashcode() + file.hashcode();
    }
  }
}
```

Exercise 3

1. Student Class diagram



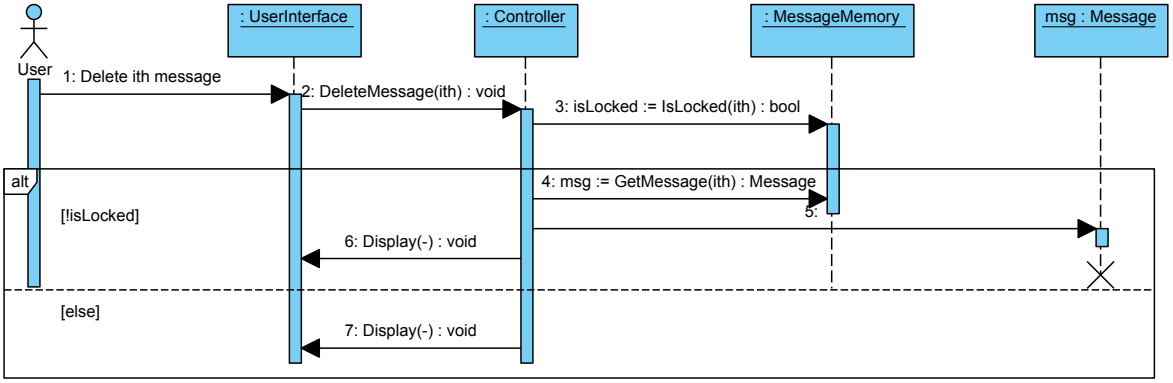
2. (a) Classmates have the same major
   (b) A student is legal iff he/she is registered

Exercise 4

1. Pseudo-code for method `PlayMessage`:

```
if (battery.isSufficientlyHigh()) {
  message = messageMemory.GetMessage(i);

  //it is not specified how to iterate over blocks in message
  //we just assume that the GetAudioBlock returns null when out of range
  AudioBlock block;
  int j = 0;
  while ( (block = message.GetAudioBlock(j++)) != null) {
    speaker.PlayAudioBlock(block);
  }
  userInterface.Display("message played");
} else {
  userInterface.Display("battery low");
}
```

2. Sequence diagram:


User
: UserInterface
: Controller
: MessageMemory
msg : Message
1: Delete ith message
2: DeleteMessage(ith) : void
3: isLocked := IsLocked(ith) : bool
alt
[!isLocked]
4: msg := GetMessage(ith) : Message
5:
6: Display(-) : void
[else]
7: Display(-) : void