IJP Assignment 1

Introduction to Java Programming: 2016-2017

Don't panic! This document may look long, but it includes quite a bit of material to help you understand the underlying concepts, as well as the assignment exercises themselves. It should guide you step-by-step though a fairly realistic Java application, over a period of several weeks. The preparation in section 2 should take no longer than about an hour and we recommend that you do this as soon as possible to check that everything is working correctly before you start on the following sections. A good solution should be possible with no more than about one day's work (eight hours), although you may need to spend longer if you don't have much previous programming experience. We strongly recommend that you work on the assignment steadily over this time, rather than leaving it until the final week - this will give you an opportunity to get help from the demonstrators or the Forum if you run into unexpected difficulties.

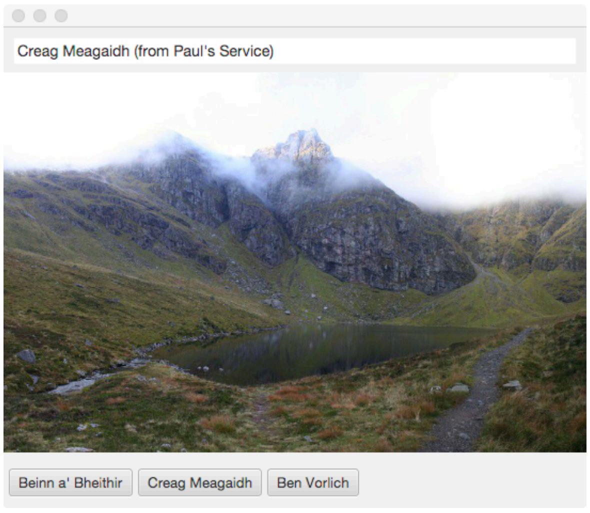
Tasks that you will need to do are marked with a ✏. *You should not spend an excessive amount of time struggling with the more advanced tasks* - especially if you do not have a lot of previous experience - it is possible to obtain a good mark even without attempting these. *Read the marking scheme carefully to understand how these will be assessed, and which tasks are essential for a good grade.*

1 The Application

For this assignment, you will be working with an application which displays images of the *Munros*. These are the 282 highest mountain tops in Scotland and are named after Sir Hugh Munro who first catalogued them. You can see the full list and hear the pronunciation of the Gaelic names on the Walk Highlands web page (you will not be tested on the pronunciation).

The application itself is quite simple, but it will give you some realistic experience with:

- ❏ Using a "professional" IDE (Eclipse).
- ❏ Creating applications by composing objects of different classes.
- ❏ Understanding and using classes created by others, as well as the standard Java libraries.
- ❏ Integrating these with your own code.
- ❏ Working with remote services.
- ❏ Working with graphical user interfaces.
- ❏ Documenting and testing your code.



The images will be taken from various web sites, including Flickr, Bing[1], and a local site with some of my own photographs[2].

[1]Bing has a limit on the number of downloads per month, so please do not use this service too heavily.
[2]`http://homepages.inf.ed.ac.uk/dcspaul/homepage/live/work/ijp_photos.html`

2 Preparation

Real Java applications are constructed from large collections of classes. And most of the code will usually be provided by other people - either by someone else working on the same (large) project, or as part of a library or framework imported from elsewhere (such as the JavaFx graphics framework). To give you a realistic experience for this assignment, we have provided a library containing some useful classes for downloading and displaying images. You will be extending some of these, and writing new classes which interact with them to produce an application for viewing photographs of the Munros.

Before starting on the assignment, you will need to install this library together with any necessary development tools, and make sure that everything is working correctly on your system. You will also need to understand how the classes in the library are used. If you have problems with any of this, ask a demonstrator for help, or post a question on the forum.

2.1 Development Environment

We have tried to design the assignment to run on a variety of different platforms so that you can work on your own preferred operating system (Linux, Mac, Windows). The FAQ section of the web site contains some hints to help you install the necessary software. However, you should ensure that you have good backups, and that you know how to use the DICE systems to compile and run your application – no credit will be given for assignments which are incomplete, or cannot be demonstrated due to problems with your own machine.

For the assignment, you will also need to use the Eclipse IDE instead of BlueJ. This does not provide the graphical representation of the classes which BlueJ provides, and you may find the range of features overwhelming at first. But, BlueJ does not support some of the features that we need for this assignment, and Eclipse is one of the most common environments that you are likely to encounter for writing "real" Java programs.

2.2 Running the Application

Download and open the zip file from the assignment web page. Look at the `README.txt` file which explains the contents. Then ...

- ❑ Open Eclipse and create a new project.
- ❑ Add the assignment1 library JAR file to your project.
- ❑ Add the `MyApplication.java` class to your project from the `templates` directory. Make sure that it goes into the default package.
- ❑ You should now be able to run the demonstration application by running the `main` method of the `MyApplication` object.

To get you started, the course web site has a short video demonstrating this process. Ask a demonstrator for help if you have problems getting this to work.

2.3 Understanding the Classes

The library includes implementations for a selection of classes which you will be able to use in your application. You can browse the documentation for these classes on the assignment web page. The zip file also includes the source for many of these, so you can read this to see how they are implemented - but do not modify the library classes for use in your application.

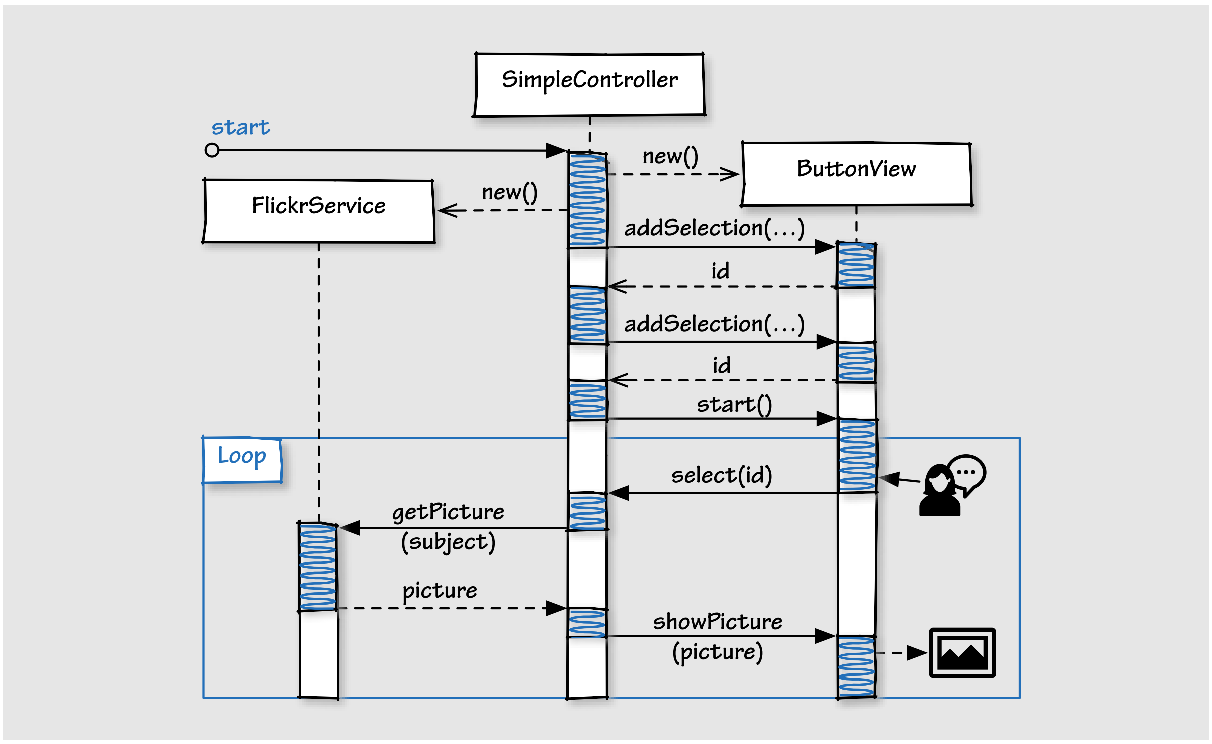


Figure 1: A sequence diagram for the simple version of the PictureViewer.

The demonstration application is structured using three main classes:

- A *View* class which handles the interface (displaying the images, and detecting user input).
- A *Service* class which retrieves a picture from the remote service.
- A *Controller* class which manages the other classes and determines the overall behaviour of the program.

Figure 1[3] is a *sequence diagram*[4] which shows a typical sequence of interactions between the objects:

- A simple Controller creates the View and Service objects.
- It then calls the View to add buttons for the required subjects to the interface. For each subject, the View returns an identifier which can be used to identify that selection.
- Control then transfers to the View `start()` method to display the interface and wait for user input.
- When the user makes a selection in the interface, the View calls the Controller's `select()` method, passing the identifier for the selected item.
- The Controller then calls the Service to fetch a picture for the corresponding subject.
- When the picture is returned, the Controller calls the View to display it.

Look at the provided template for `MyController.java` and follow the code using the above description. Notice that the demonstration application in the previous section is using a slightly more complex version of the controller – for example, it displays three buttons, rather than two.

[3] Icons from Flaticon licensed under Creative Commons BY 3.0.
[4] A sequence diagram is used to show the interactions between objects, and the order in which the interactions occur.

2.4 Using Different Implementations

The separation of classes we have just described may seem unnecessarily complicated for such a simple application. But this type of design is very important for real applications: different developers can work on different classes independently, and we can easily remove one class and replace it with another which performs the same task in a different way. For example, the library contains several different implementations of the Service class which fetch images from different services, and two different implementations of the View class which handle input in different ways. And you will be writing some different implementations of the Controller class for this assignment.

Look at the Javadoc for the library and click on `Service` (in italics in the class list at the bottom left). This is an *Interface*. It describes the methods which a class must provide if it is going to be used as a source of pictures for the application. There are only two methods: one to return the name of the service (`serviceName`), and the other to return a picture for a particular topic (`getPicture`). The application will be able to use any service class which can provide these two methods. There are several of these in the library, including `FlickrService`, `BingService`, and `PaulsService`. `LocalService` allows you to use your own images from a local directory. Look at the Javadoc for these and notice that they provide the required methods to implement the `Service` interface.

To allow you to experiment easily with different implementations, the library provides a way for you to change these without recompiling the application: they can be specified (along with some other configuration information) in a *property* file which is read by the application at runtime. The zip file includes a sample property file `my.properties`. The FAQ page on the web site explains how to install this property file in your application. If you do this, and run the application again, it should function in exactly the same way, except that the view now uses a menu to select the topic (rather than a button), and it fetches the pictures from the `FlickrService`, rather than `PaulsService`. Look at the property file and you will see that you can easily change the implementations used for the three main components. The course web site has a short video illustrating this[5]. Experiment with the different views and services, but remember to change this back to `PaulsService` before continuing[6].

This mechanism will allow you to test your own controller implementation with various combinations of other classes. To use the `MyController` class, for example, you need to copy the source file from the provided templates into your project and compile it. Be careful: the controller source file needs to go into a different *package*. The FAQ pages on the course web site explain how to do this. If you now edit the properties file to specify `MyController` as the Controller and run the application again, it should now use your controller. You can check that this is working because the interface should display the name of the controller when it starts, and the button names should have changed to the ones specified in your source file.

We are now ready to write some actual code!

3 Writing a Controller

Let's start just by making a small extension to `MyController`:

✐ [1] The supplied class provides just two possible selections. Extend the code to add a third option. Remember that the Walk Highlands web page has a full list of Munros if you don't know any. You

[5]The video uses the Picasa service which is no longer available - you will need to choose a different service, such as Flickr.

[6]You should use `PaulsService` for most of your testing because the external services are either slow, and/or have limits on the number of downloads.

may want to choose Munros that you know are available on `PaulsService`[7].

You will notice that it is very tedious to add an additional option: you need to add the selection to the view, *and* an extra clause to the conditional statement to test for it. In this case, the interface label is the same as the search string (which may not always be the case), so you have to specify the same name in two places as well. Let's modify the controller to improve this:

✐ [2] Declare a HashMap to store the correspondence between the selection identifier and the name (think about the required type for the HashMap). Write a method `addSubject()` which takes the name of a Munro, (a) adds a selection the interface, and (b) adds a corresponding entry to the HashMap (the method will be very short). You should now be able to rewrite the `select()` method very simply, and you should need only one line to add each Munro.

Having the names of the Munros "hardwired" into the source code like this means that they cannot be changed without recompiling the code. And the user cannot change them without access to the source code either. We can solve this by specifying the list of Munros in the property file. For example:

```
MyController.subjects = Ben Vorlich, Creag Meagaidh, Ben Lui
```

✐ [3] Modify your controller to read the list of Munros from the property file. The (static) `get()` method of the `Properties` class can be used to retrieve the string value of this resource - look at the Javadoc for `ijp.utils.Properties`. Having obtained this string, you will need to (a) split the string at each comma to create a list (array) of names, (b) remove any leading or trailing spaces from each name, and (c) iterate through the list adding each item. Look through the Javadoc for the standard Java class `String` to find some helpful methods for (a) and (b).

The `SimpleController` class in the assignment library is a working implementation. You can specify this in your properties file to see what the final application should look like.

You will need to submit your code for `MyController` as well as demonstrate it. You only need to submit your final version, although you may want to create a separate class file for each part of this exercise, so that you do not lose the implementations for the earlier parts. Please do not submit a final class file with earlier sections "commented out".

If you don't have much previous experience, you may find it hard to complete all three of the above tasks, in which case, you should just submit the code for whatever stage you managed to complete - it is not necessary to complete all of the stages in order to pass (see the marking scheme).

4 Proxies

In this section, we will look at another way of composing objects. Let's start with a motivating example: imagine that we are dealing with a very unreliable service, which often returns an error, but usually succeeds if the call is repeated a few times[8]. We could handle this by simply having a loop which retries failed calls a number of times. But where should we do this? If we include this code in the controller,

[7] `http://homepages.inf.ed.ac.uk/dcspaul/homepage/live/work/ijp_photos.html`
(click on a thumbnail to see the name of the munro as well as the large image)

[8] This is quite realistic. For a few months in 2014, the real Flickr service behaved in this way.

```
private Service baseService = null;

public RetryProxy(Service baseService) {
  this.baseService = baseService;
}

public Picture getPicture(String subject, int index) {
  Picture picture = baseService.getPicture(subject, index);
  int attempt = 1;
  while (!picture.isValid() && attempt<=maxAttempts) {
    picture = baseService.getPicture(subject, index);
    ++attempt;
  }
  return picture;
}
```

Figure 2: The core of the `RetryProxy` class.

then we would have to duplicate the code in every controller. If we include it in the service, then we would have to duplicate the code in every (potentially) unreliable service. Of course, in this case, there isn't much code involved, so this might not be significant. But in a real application, it may be. We can solve this in a general way using a *proxy* class:

Look at the code for the `RetryProxy` in figure 2. The constructor for this requires that we specify some base service object. The `RetryProxy` object saves this, and whenever we attempt to retrieve a picture from the `RetryProxy`, it calls the base service repeatedly until it succeeds or it reaches some maximum number of attempts. The `RetryProxy` itself conforms to the `Service` interface, so we can use a `RetryProxy` wherever we can use any other service. This means that we can take *any* service and "wrap" it with one of these proxy objects to create a version of the service which is more reliable.

The version of the `RetryProxy` in the library also has a constructor with no parameter (not shown above). If this called, the base service is determined from the properties file. This means that the following resources will configure the application to use `PaulsService` service with an automatic retry of any failed calls:

```
Service = RetryProxy
RetryProxy.base_service = PaulsService
```

Of course, if you run the application with this configuration, it will be difficult to see any difference because the service usually behaves reliably. But we can use another proxy class to illustrate that this is working: The `UnreliableProxy` adds random errors to an existing service. Try this configuration and notice the random failures:

```
Service = UnreliableProxy
UnreliableProxy.base_service = PaulsService
```

Now that we have an unreliable service, we can compose the two proxies to demonstrate that the `RetryProxy` is indeed working. You can turn on the debugging messages for the various services

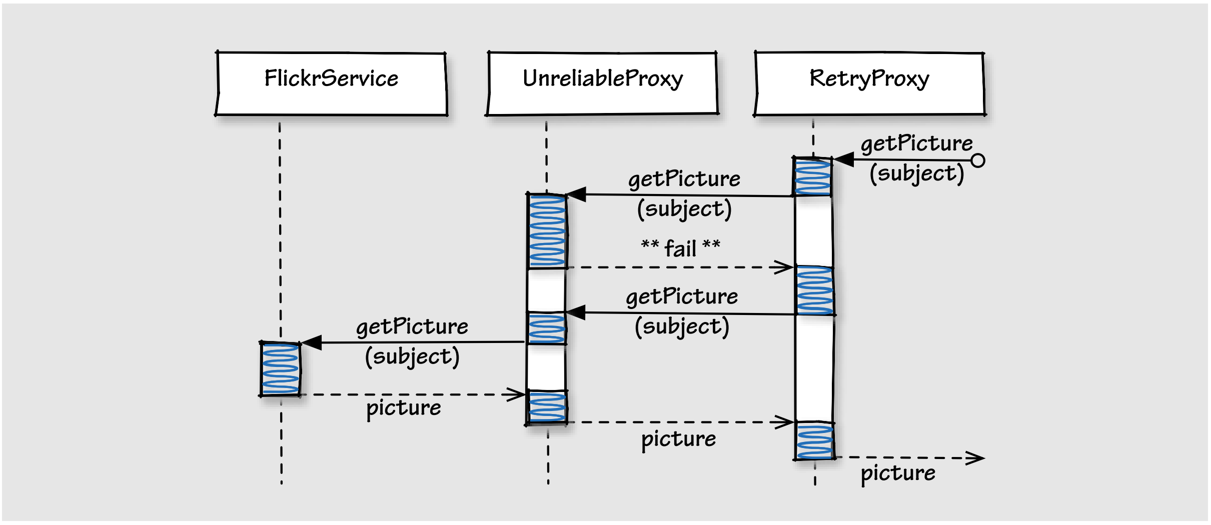


Figure 3: Recovering from a simulated unreliable service.

to display the sequence of events in the console:

```
Service = RetryProxy
RetryProxy.base_service = UnreliableProxy
UnreliableProxy.base_service = PaulsService
RetryProxy.debug = true
UnreliableProxy.retry = true
```

Figure 3 shows the sequence diagram for a typical interaction.

5 Writing a Cache Proxy

As well as being unreliable, network services can also be slow. In general, we can't do anything about this. But real applications often access the same data repeatedly, and we can improve the observed performance of a slow service by simply keeping a local copy of any picture that we download and using this local copy if the same subject is requested again. This is called *caching*. Using a cache to reduce the remote access in this way can also be cheaper for services such as Bing which charge for high-volume access. In this section, you will be writing a cache proxy for the PictureViewer application.

You have seen how the `UnreliableProxy` can be used to create an unreliable version of a service for testing. Similarly, the `SlowProxy` can be used to create a slow version of a service. Configure the properties to create a slow version of `PaulsService` and notice the reduced response time.

✐ [4] Create a new class `MyCacheProxy` in the `ijp.proxy` package. Initially, this proxy should simply forward all requests to the base service (the proxy should have no effect at all). The class should implement the `Service` interface, and the code will be a simplified version of the `RetryProxy` shown in figure 2. You will also need this constructor to get the base service from the resources:

```
public MyCacheProxy() {
  baseService = new ServiceFromProperties("CacheProxy.base_service");
}
```

✐ [5] Declare a HashMap to store the cache. The values in the cache will be the downloaded pictures. But you may need to think carefully about the keys – two requests should only be considered the same if the subject and the index are both the same. Now modify the `getPicture()` method: if the requested picture is in the cache, return the copy from the cache; if the picture is not in the cache, download it from the base service and store it in the cache before returning it. Test your cache proxy by composing it with the `SlowProxy` – the first request for a particular picture will be slow, but all subsequent requests for the same picture should be very fast.

You will need to submit your code for `MyCacheProxy` as well as demonstrate it. The `CacheProxy` class in the assignment library is a working implementation (source not included) which you can run to compare with your own.

Notice that the cache is only held in memory – if the application is restarted, then the contents of the cache will be lost. A real application would probably store a copy of the cache on disk. You may also like to think about other types of proxy that you could write – for example, the `RandomProxy` (in the library) returns a picture with the required subject from a random base service.

6 Testing

Beginning programmers sometimes think that "testing" is a chore which happens after the code has all been written. But when you come to write larger programs, you will find it incredibly useful during development to have a good test suite - this allows you to "refactor" and modify your code without worrying whether your changes have broken some other part of the program. When you are designing and coding a new class, you should think carefully about how it might be tested – otherwise it is easy to create designs which are unnecessarily difficult to test.

JUnit is a standard framework for writing tests in Java. Both BlueJ and Eclipse understand JUnit test files and can run these in a special way which automatically records any failed tests. Make sure that you understand how to run JUnit tests in Eclipse. There is a short video on the course web site, and the demonstrator will be able to help you if you are not sure how to do this – you must be able to run the tests from Eclipse without changing your application code, and Eclipse should produce a report of the failing and succeeding tests.

Look at the supplied template for `MyCacheProxyTest`. You may find it helpful to sketch a sequence diagram - the way in which the objects are configured for this test is interesting, and you may have to think carefully to understand what is happening:

Notice that no view class is necessary, since we are not displaying anything - we are only calling the service and making sure that it returns the "right" picture. But we *do* need a base `Service` object for the `CacheProxy` to call on. Rather than using a separate service class, the test class itself implements the `Service` interface and provides a `getPicture()` method. When we create the cache proxy, we tell it to use the test object as the base service by specifying `this` as the parameter to the constructor.

This means that whenever the cache attempts to call the base service, it will call the `getPicture()` method of the test object. This method simply returns a *new* (empty) picture object every time it is called.

If the `CacheProxy` returns a picture object which is equal to one that has been returned previously, then it must have been cached. If it returns a new picture object, then it must have come directly from the base service. Because the test class has control of the base service methods, we could also include other code in the `getPicture()` method to control or record other properties of the picture.

The example `equalityTest()` simply makes two calls to the `cacheProxy` object (with the same subject and index) and confirms that the same picture object is returned both times. If the pictures were not being cached, then we would expect two different objects to be returned (even though they may have the same subject and index), and the test would fail.

✐ [6] Import `MyCacheProxyTest` into the `test` package of your project and run it[9]. Notice that it is testing the version of `CacheProxy` supplied in the library, so the test should pass. Change this to test your version (`MyCacheProxy`). Hopefully, the test will still pass. Now make some change to `MyCacheProxy` so that it contains an error - perhaps commenting out the line that saves the picture in the cache - and rerun the test to check that it fails. Remember to remove the changes to `MyCacheProxy` when you have finished!

7 Testing the cache proxy

The class `BrokenCacheProxy` is a version of the `CacheProxy` which contains errors. Configure the application to use this proxy by setting the following resources:

```
Service = BrokenCacheProxy
CacheProxy.base_service = PaulsService
CacheProxy.id = NONE
CacheProxy.error = A
```

You *must replace the* `NONE` *with your student identifier* (e.g s1234567).

✐ [7] Change `MyCacheProxyTest` so that it tests the `BrokenCacheProxy` class. Write an additional test method `indexTest()` to check that the picture returned from the cache proxy actually has the index that was requested. This test should fail if you use the `BrokenCacheProxy` and pass if you use your own implementation.

✐ [8] Change the `CacheProxy.error` resource to `B`. The `BrokenCacheProxy` should now exhibit a different error. Write a test which detects this error.

✐ [9] Setting the `CacheProxy.error` resource to `C` or `D` will produce two further different errors. Try writing some tests to identify these. This is not easy, and not necessary for a pass mark, so only attempt this if you are finding the exercises easy, and are satisfied that you have completed the other sections well.

You will need to submit the code for `MyCacheProxyTest`, and demonstrate how your `indexTest()` succeeds or fails with the `BrokenCacheProxy`.

[9] If Eclipse shows errors when you import the test file, You may need to configure Eclipse with the appropriate JUnit library. This is explained in the video, and in the FAQ.

8 Worksheet & Documentation

As well as submitting your code, you will also need to submit a worksheet with answers to a few questions. You may create the worksheet using any application that you like, but please make sure that it in PDF format – the FAQ pages on the course web site explain how to generate PDF – and make sure that it has no more than 500 words:

✐ [10] Make sure that your name and matriculation number are clearly visible at the top of the worksheet.

✐ [11] Please start with a few sentences explaining your previous programming experience. This is not part of the assessment, but it will help us to tailor the course better to the overall level of the class.

✐ [12] For each part of the assignment, state the task for which you submitted code. You will be asked to demonstrate your code for these tasks, which may also be tested automatically:

(a) The controller: [1], [2], [3] or "none".

(b) The cache proxy: [4], [5] or "none".

(c) The tests: [6], [7], [8], [9] or "none".

✐ [13] Answer as many of the following questions as you are able. Some of these questions are quite difficult and *you do not need to answer them all* to obtain a good mark (see the marking scheme). Please number the answers clearly:

1. Clearly explain any errors that you found in tasks [8] or [9].
2. Explain why `MyController` is hard to test in the same way that you tested the `CacheProxy` (i.e. without relying on external services or views). Suggest how you might make a small modification to the controller class to make this easier.
3. Explain why it is difficult to compose the `RandomProxy` and the `CacheProxy`.
4. Suggest how you might change the way in which the resources are used to make this composition possible.

Hopefully, you found the Javadoc useful for the assignment library and the standard Java classes. You should create similar Javadoc for you own classes:

✐ [14] Create a directory `javadoc` containing the HTML version of the Javadoc for `MyController`, `MyCacheProxy`, and `MyCacheProxyTest`. The FAQ pages on the course web site explain how to generate the Javadoc.

Make sure that there are no errors creating the Javadoc, and that you *include all of the supporting files from the Javadoc directory*[10]. Open the `javadoc/index.html` with a browser and read the HTML files carefully to make sure that you have not left any inappropriate comments from the template files. If your own Java installation is not in English, you should generate the Javadoc on DICE to make sure that the documentation is in English.

[10] The directory should contain all of the style files and index files necessary to view your Javadoc correctly.

9 Submission

You should attempt your submission well before the deadline to allow for any problems that you may have with the submission system. If you need to update your submission (anytime before the deadline), you can simply make a new submission which will replace the previous one. But *no marks will be awarded for submissions received after the deadline* – see the School policy.

Before you submit your assignment ...

✎ [15] It is important that your code is easy for other people to read. This will form part of the assessement, so you may want to review your code style – see the relevant appendix in the textbook.

✎ [16] Please make sure that you understand about good scholarly practice – see the information on the course website for further details. We have automated tools for checking both worksheets and code for suspected plagiarism. If you are in any doubt about any part of your submission, please discuss this with the course lecturer.

✎ [17] Create a ZIP archive containing the following (please use these exact filenames):

- ❑ `MyController.java` - your source code for the controller.
- ❑ `MyCacheProxy.java` - your source code for the cache proxy.
- ❑ `MyCacheProxyTest.java` - your source code for the cache proxy tests.
- ❑ `javadoc` - the directory containing your Javadoc in HTML.
- ❑ `worksheet.pdf` - your answers to the worksheet questions.

The FAQ pages on the course web site explain how to create a zip archive. Be careful: you will not be credited for missing files, so you may want to unpack the archive again yourself to verify that it contains all of the files that you expect.

✎ [18] Use the online submission system to submit your ZIP file.

10 That's It!

During one of the lab sessions, you will be asked to show your solution to the demonstrator and a small group of students. This is a valuable opportunity to get feedback and to see potentially different approaches to tackling the same problem. The demonstrator will also be asked to check that you can demonstrate running code for the tasks which you listed on the worksheet. This will also give you an opportunity to explain the code, and any problems that it may have.

✎ [19] Make sure that you attend this lab session (on time!), and that *you have all of the necessary files to run and demonstrate the same version of your code as the one you submitted*[11].

We hope that you found this a useful and realistic exercise. Please do let us know what you think. Ask the lab demonstrators if you would like any further feedback, or use the forum to discuss any issues relating to this exercise.

[11] If you cannot demonstrate the same version of your code as the one you submitted (we can check this), then your demonstration will be discounted for the assessment.