

Assignment 1

ippo 2021-2022



Don't panic! This document may look long, but it includes the material to help you understand the underlying concepts, as well as the assignment exercises themselves. It should guide you step-by-step through a fairly realistic Java application over the next weeks. The preparation in sections 2 and ?? 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 within about 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 via the [Piazza](#)¹ forum if you run into unexpected difficulties.



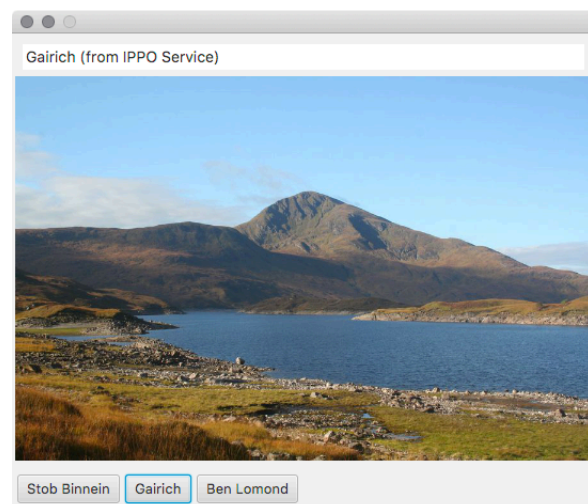
It is possible to obtain a good mark without attempting all of the tasks (📝). Especially if you don't have a lot of previous experience, don't spend an excessive amount of time on the more difficult tasks. Certain tasks are essential for the various different grades though - so you should pay careful attention to the marking criteria (appendix A).

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](#)² 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 “professional” development tools.
- ✓ Creating applications by “composing”³ 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 downloaded from [Wikipedia](#)⁴ and the following IPPO site: <http://35.178.204.69/munros>.

¹https://www.learn.ed.ac.uk/webapps/blackboard/content/launchLink.jsp?course_id=_88899_1&tool_id=_4528_1&tool_type=TOOL&mode=cpview&mode=reset


²<http://www.walkhighlands.co.uk/munros/pronunciation>

³i.e. connecting together existing objects in different ways to create different applications.

⁴<https://en.wikipedia.org/wiki/Munro>


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.

-  [1] Before starting on the assignment, make sure that you have a working set of development tools and understand how to use them. See [LEARN > Resources > Course Notes > DevelopmentTools.pdf](#).

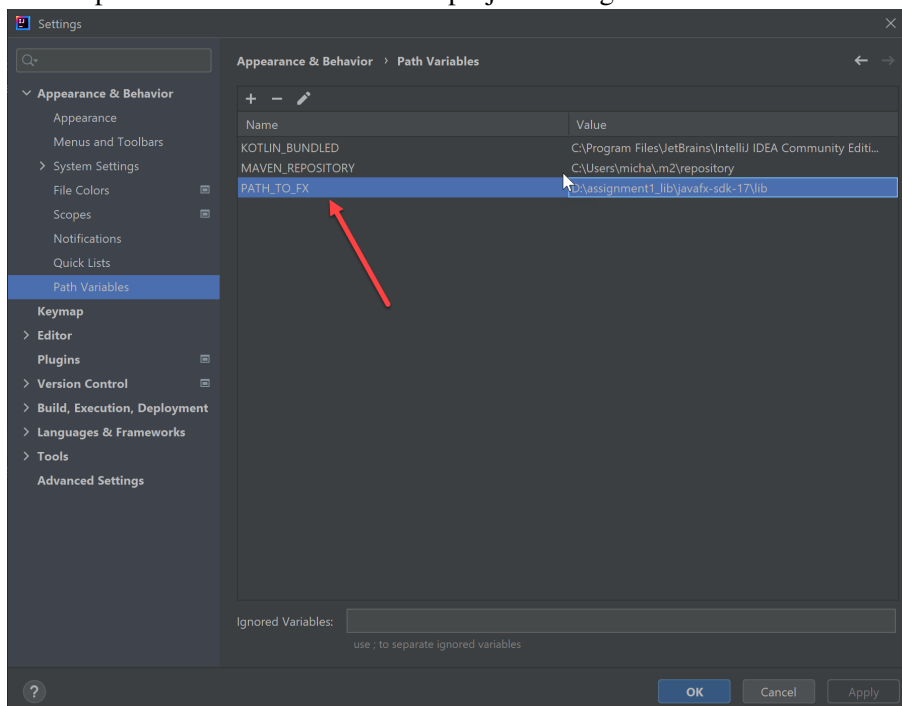
You will need:

- A Java 17 SDK
- Java FX SDK 17
- Hamrest 2.2
- JUnit

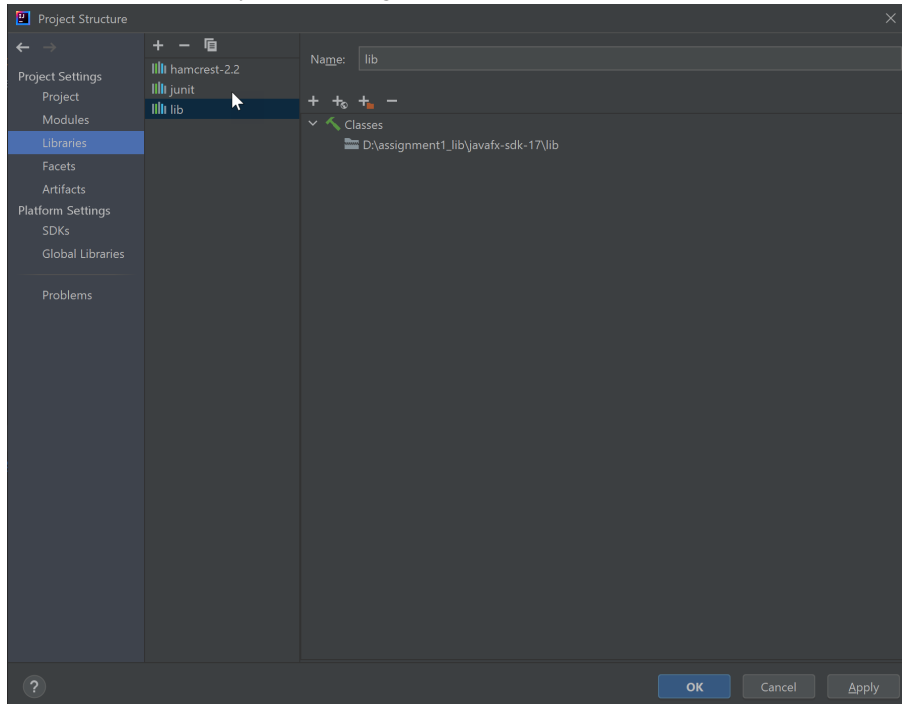
-  [2] These JAR-files and libraries should be put into a folder in your solution. To help you get started, we have provided some template files for the application.

The following illustrates the setup process:

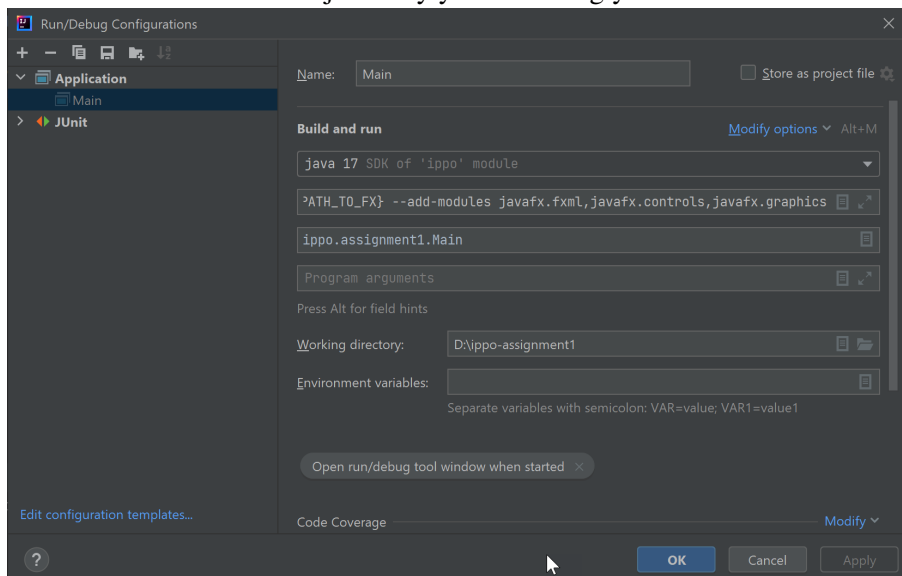
1. Download [LEARN > Assessment > Assignment 1 > ippo-assignment1.zip](#) and unzip this template project.
2. Import the template into a new IntelliJ project.
3. Set the path to the FX modules in the project settings.



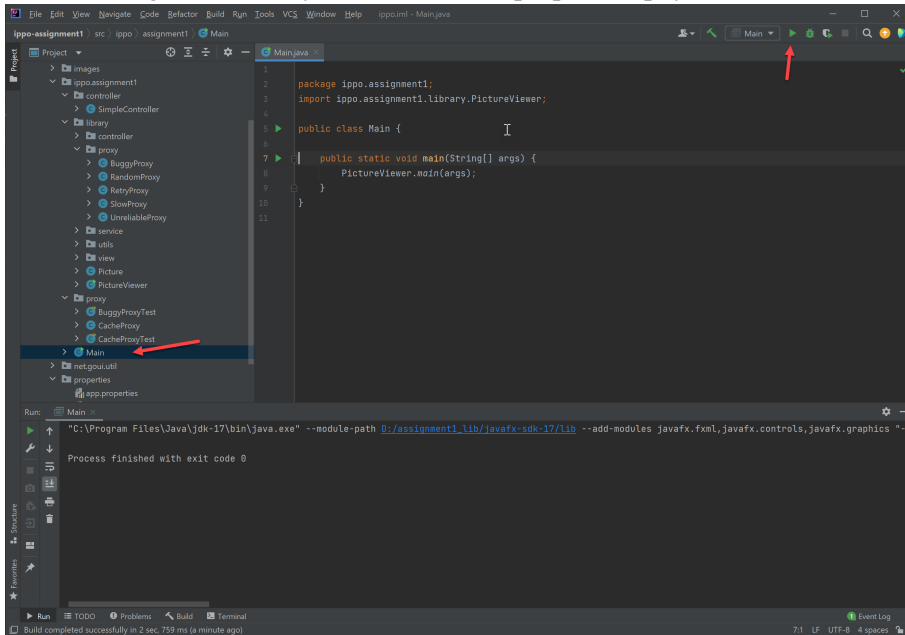
4. Define the libraries you are using.



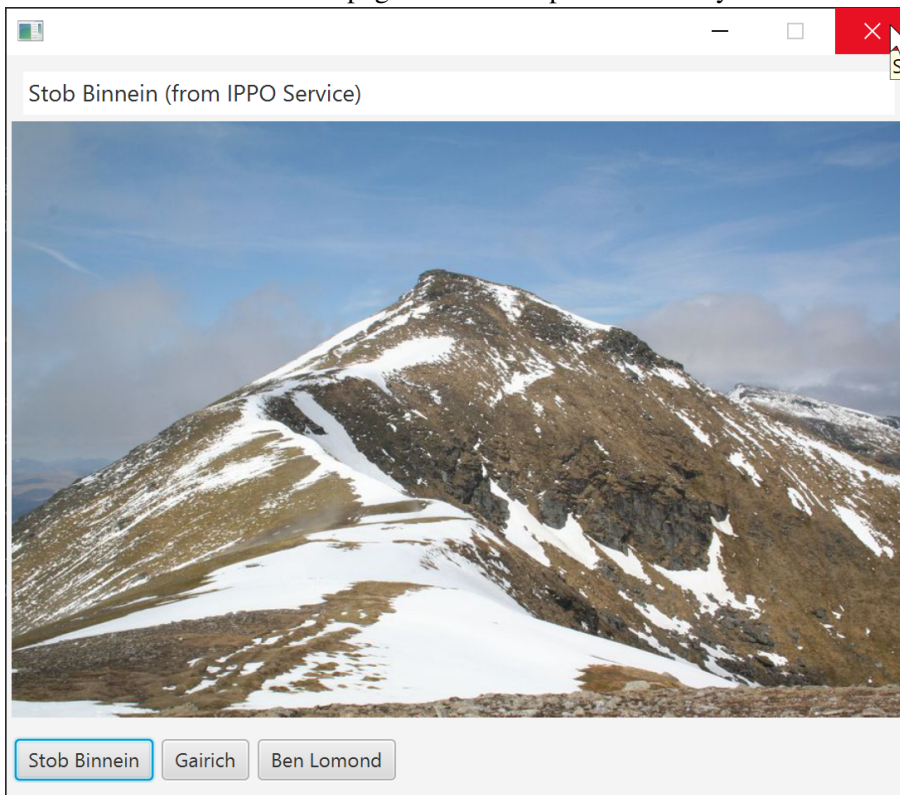
5. Specify the modules to be loaded. The template you downloaded is set up for the sample directories and has to be adjusted by you accordingly.



6. After loading the directory into IntelliJ and proper setup, your view should be similar to this:



7. After pressing run you should see an interface which allows you to view a selection of images, similar to the one on the first page. Ask for help on Piazza if you have issues getting this to work.



If you see the error message “class file has wrong version” it is probably because you are using the wrong version of the JDK - try going into the project settings and changing the JDK version to 17.

Now that you have a clean template project, you are ready to start work on the assignment ...

3 Understanding the Classes

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.


3.1 Using Different Implementations

A well-designed application will allow classes to be easily replaced with different implementations. To illustrate this, our library provides:

- Two different *View* implementations - using buttons or menus.
- Two different *Service* implementations - Wikipedia and the IPPO photographs.
- Two different *Controller* implementations - one which allows specific images to be viewed, and another which displays a random image and asks the use to guess the appropriate name (a “quiz”).

Notice that each of these (*View*, *Service*, *Controller*) has several *implementations* and an *interface*⁵ specification which defines the methods that each implementation must provide.

The library provides a mechanism for you to experiment easily with different combinations of these implementations by changing values in a “property file”:

 [3] Experiment with the different properties:

1. Open the project file `src/main/resources/properties/app.properties`. Lines starting with # are explanatory comments which are ignored by the system.
2. Change the *View* from `ButtonView` to `MenuView` and re-run the application. Notice how the interface has changed but the other components remain the same.
3. Change the *Controller* from `SimpleController` to `QuizController`. The application will show a random Munro. You can guess the name and the application will tell you whether you are correct or not. Choosing `New` will display another Munro.
4. Change the *Service* from `IPPOService` to `WikiService` (this example may be clearer if you change back to the `SimpleController`) and re-run the application. The application will fetch the images from Wikipedia, so there will be different images for each mountain (and there will be a slower response).
5. Reset the property file back to the original values for the following exercises.



This kind of flexibility is an important property of a good object-oriented design, and you should think carefully about this when you come to create your own class design.

3.2 How Does This Work?

Figure 1⁶ is a *sequence diagram*⁷ which shows a typical sequence of interactions between the objects:

⁵The term *interface* here refers to a “Java interface” which has nothing to do with the “user interface”.

⁶Icons from [FlatIcon](#) licensed under [Creative Commons BY 3.0](#).

⁷A [sequence diagram](#) is used to show the interactions between objects, and the order in which the interactions occur.

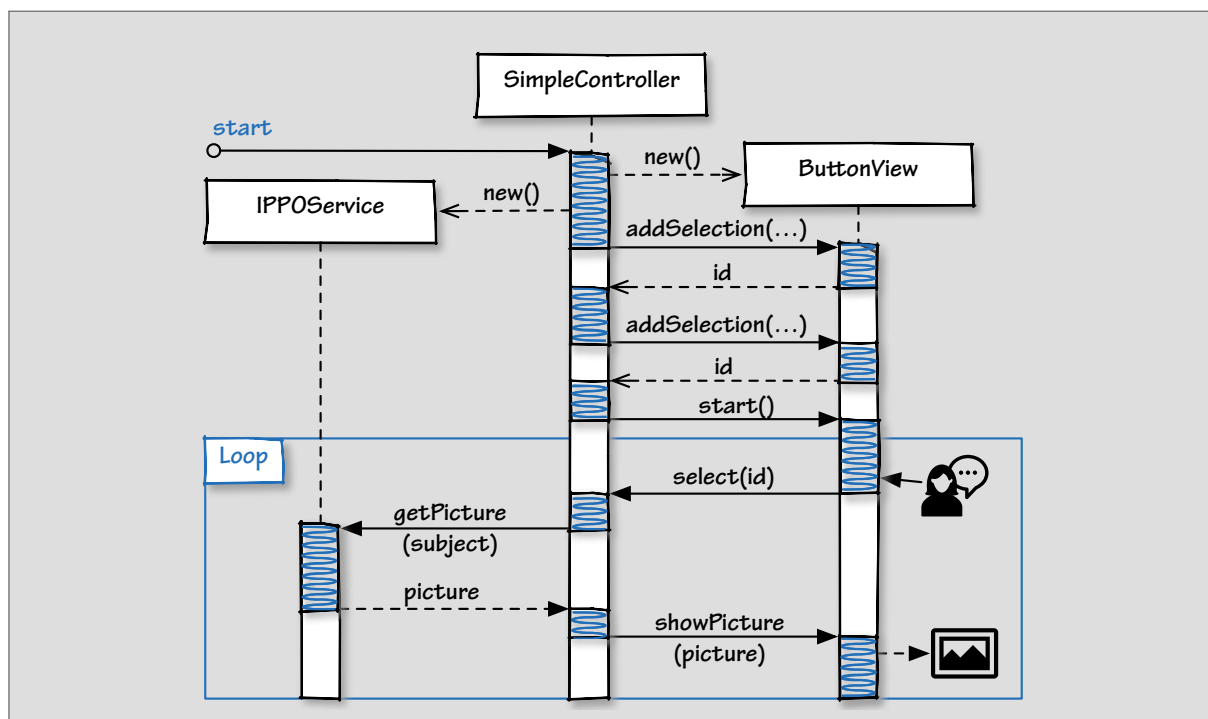


Figure 1: A sequence diagram for the simple version of the PictureViewer.

- 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.

Notice that the `getPicture()` method allows us to specify an *index* (as the second argument). In this case, we always use a value of 1 which returns the first picture for the given subject. Some services may have more than one image for the same Munro, and we could use different values for the index to retrieve these other images.


We are now ready to write some actual code!

4 Writing a Controller

The property file makes it easy for us to create and experiment with a new controller. Let's start just by making a small extension to the `SimpleController`...

4.1 Adding an Extra Option

The supplied class provides just three possible selections.

 [4] Let's add a fourth one:

1. Copy the `SimpleController.java` class to `BigController.java`, and modify the code to add a fourth option.
2. Change the property file to use `BigController` instead of `SimpleController` in the properties file, and run the application to try out your controller.
3. Keep `BigController.java` for your submission.


Look at the [IPPO page](#)⁸ to find Munros which are available on the `IPPOService`.

Remember to change all occurrences of `SimpleController` in the new source file to `BigController`. If you copy and paste the class using IntelliJ's project menu, it will automatically rename these for you, but remember to edit the comments so that they are meaningful as well!

4.2 Using a HashMap

You will notice that it is very tedious to add an extra 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 exactly the same name in two places as well (which is a potential source of errors).

We can improve this by using a `HashMap`. If you are unsure about these, read the note on `HashMaps` ([LEARN](#) » [Resources](#) » [Course Notes](#) » [Hashmaps.pdf](#)), and/or the book section on `HashMaps` before continuing.

 [5] Create a version of the controller which uses a `HashMap` to store the Munro names corresponding to each selection identifier:

- Copy `BigController.java` to `HashMapController.java` to create a second new version of the controller, and change the contents of the new file to match⁹.
- Add the following import statement:

```
import ippo.assignment1.library.utils.HashMap;
```

- Declare a `HashMap` to store the correspondence between the selection identifier and the name. You will need to think carefully about the required type for the `HashMap`.
- Make sure that you initialise the `HashMap` to an empty map, otherwise you will get an exception (`NullPointerException`) when you attempt to access it.
- Write a method `addSubject()` which takes the name of a Munro, (a) adds a selection to 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 only need to add one line (to the `start()` method) to add each extra Munro.
- Change the property file to test your `HashMapController`.
- Keep `HashMapController.java` for your submission.



You must use the IPPO implementation of the `HashMap` (above) which includes some code to be used when testing your submission. Do *not* import `java.Util.HashMap`, or `Java.Util.*` directly.


⁸<http://35.178.204.69/munros>

⁹Do not edit the `BigController` file – you will need to submit both `BigController` and `HashMapController`.

4.3 Reading Properties

Having the names of the Munros “hard-wired” into the source code means that they cannot be changed without recompiling the code. And the user cannot change them without access to the source code either. We could avoid this by specifying the list of Munros in the property file. For example:

```
controller.subjects = Beinn a' Bheithir, Creag Meagaidh, ...
```

-  [6] Create a third version of the controller which reads the list of Munros from the property file:
1. Copy the `HashMapController.java` to `PropertyController.java` and modify this to read the list of Munros from the `controller.subjects` property in the property file.
 2. Configure the property file to use this controller. The property value above is available by default from the library, so your application should display these Munros by default.
 3. Set a new value for this property in your own property file. This this should override the default and display your own list of Munros.
 4. Keep `PropertyController.java` for your submission.

The (static) `get()` method of the `Properties` class can be used to retrieve the string value of the property. 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](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html)¹⁰ for the standard Java class `String` to find some helpful methods for (a) and (b).

Notice that your code should be capable of reading an arbitrary number of Munros from the property file. Make sure that it is not restricted to a fixed number.

5 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. 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, 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 is called, the base service is determined from the properties file. This means that the following properties will configure the application to use `IPPOService` service with an automatic retry of any failed calls:

¹⁰<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>


```
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.

```
service = RetryProxy
proxy.retry.service = IPPOService
```

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
proxy.unreliable.service = IPPOService
```


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 to display the sequence of events in the console:

```
service = RetryProxy
proxy.retry.service = UnreliableProxy
proxy.unreliable.service = IPPOService
proxy.debug = true
```

Figure 3 shows the sequence diagram for a typical interaction.

5.1 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 which charge for access.

 [7] Write a cache proxy for the `PictureViewer` application:

1. 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 ser-

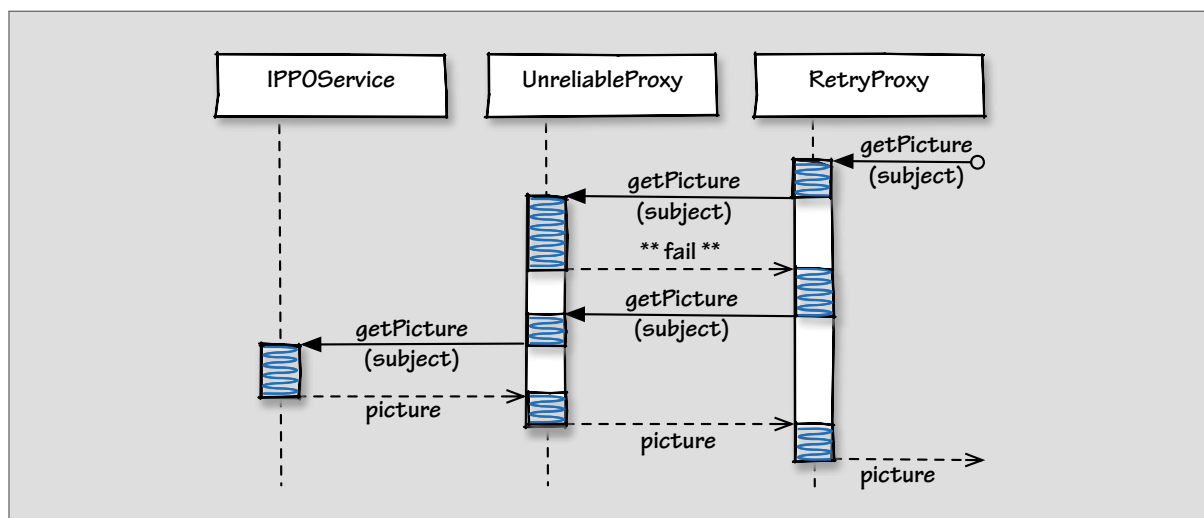


Figure 3: Recovering from a simulated unreliable service.

vice. Configure the properties to create a slow version of `IPPOService` and notice the reduced response time.

2. `CacheProxy.java` is a skeleton for a proxy which simply forwards all requests to the base service unchanged. Configure the property file to use this proxy together with the `SlowProxy`. You should not notice any difference in performance (i.e. it should still be slow).
3. Edit the `CacheProxy.java` and declare a `HashMap` to store the cache¹¹. 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 both the subject and the index are both the same.



The question of when two objects should be considered “the same” is not always easy.

4. 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 with the `SlowProxy` – the first request for a particular picture should be slow, but all subsequent requests for the same picture should be very fast.
5. Keep `CacheProxy.java` for your submission and the demonstration.

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.

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

¹¹Remember to import the `HashMap` from `ippo.assignment1.library.utils`, and not `Java.Util`

create designs which are unnecessarily difficult to test.

JUnit is a standard framework for writing tests in Java. Both BlueJ and IntelliJ understand JUnit test files and can run these in a special way which automatically records any failed tests.

6.1 Running JUnit Tests

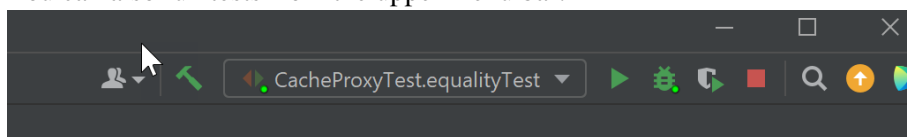
The assignment template includes a `CacheProxyTest` which is a simple JUnit test for the `CacheProxy`. This test checks that the proxy returns exactly the same object whenever it is called with the same name and index – which is what we would expect from a working cache proxy.

 [8] Experiment with this test:

1. To run a unit test in IntelliJ the easiest is to use the context menu. If your `CacheProxy` is correct, this test should pass.



2. You can also run tests from the upper menu bar:



3. Alternatively, you can use build tools like Gradle, etc. to automate tests.
4. If you (temporarily!) edit your `CacheProxy` so that it does not keep the copies in the cache, and re-run the tests, the `CacheProxyTest` should fail. The test messages printed in the IntelliJ console can be difficult to read, but you can view a formatted report by copying the URL which appears in the console, and pasting it into a browser.
5. Remember to re-edit your `CacheProxy` so that it is working again (test it), and keep the source file for your submission.



Notice that tests such as these can all be executed automatically. This is *much* easier than having to manually run the application and watch the response time (which would simply not be feasible if you had 100s or 1000s of tests).

6.2 Understanding the Cache Proxy Test

Look at the `CacheProxyTest` source file. 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 `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 `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. We say that the `CacheProxyTest` object is acting as a *mock* service object.

The example `equalityTest()` simply makes two calls to `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.

6.3 Introducing Bugs (`BugProxyTest`)

`BuggyProxyTestTemplate` is a Java class which acts as a unit testing template for several test cases - e.g. subject, index, etc.

It uses the `BuggyProxy` proxy to introduce bugs for the test. In the template implementation no bugs will be generated as the parameter `bugToIssue` is always “A” which means no bug.

The bug classes are:

- A - No bug.
- B - Does not cache any images.
- C - An obvious bug.
- D - A bug which is different for different students.
- E - A more obscure bug which is different for different students.
- F - Throws an exception after some images have been displayed.
- G - “Hangs” after some images have been displayed.

Your task is to create a copy of `BuggyProxyTestTemplate.java` class in `BuggyProxyTest.java` (which is to be submitted) which tests and finds the relevant errors in `BuggyProxy` for the test cases.

So, you will have to investigate which bugs are generated in `BuggyProxy` under which category and if these bugs match your unit test case (e.g. index). For this to work you have to set your UUID in the instance property. To see things working, just debug into an existing unit test (bug category A) and follow the flow of calls.

In the end you should utilize every bug category and have the relevant unit code testing for it. This might be quite complicated, so only attempt bug category D..G if you are finished with your other tasks.

There is no pre-defined “this is it” solution to this task as everybody might have a different approach. The most important thing is that you understand which bug category affects which unit test and how to capture/identify those errors.

It is important that your classes can be tested independently from the rest of your application. When your tests are run for marking, they will not have access to any of your other code. So ...



Before submitting, check carefully that you do not have any unnecessary imports, or other dependencies. For example:

- Do not depend on additional properties in the property file.
- Do not import additional files of your own.

7 Readability

In the “real world”, it is not sufficient for code to just “work” - it also has to be easy for other people to read and understand, easy to extend later, and sufficiently clear that there are no hiding places for obscure bugs. Clear, readable code is a core requirement of the IPPO course - code which is difficult to read will not pass, and higher marks require particularly clear and well-structured source code. So, before you submit ...



- [9] Look at the notes on code code readability [LEARN](#) » [Resources](#) » [Course Notes](#) » [CodeReadability.pdf](#) and use this to review the readability of your code.

8 Worksheet

For a higher mark (see appendix A), you will need to demonstrate your understanding of the code, and your ability to explain it clearly, by submitting a worksheet with answers to a few questions. Some of these questions are quite challenging, so only attempt them if you are confident that you have completed the other tasks well:

1. Explain why the controllers are 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.
2. Explain why it is difficult to “compose” the `RandomProxy` and `CacheProxy`. i.e. to use these class implementations to create a system which both fetches images from a random service and caches the results. Suggest how you might change the way in which the properties are used to make this composition possible.



- [10] If you think that you can provide good answers to these questions, create a document containing your answers:

1. Make sure that your name and student number are clearly visible at the top of the worksheet.
2. Make sure that the answers are clearly numbered.
3. Make sure that your answers are clear and concise, with a good standard of English.
4. Convert the document to PDF format¹² with the name `worksheet.pdf`.
5. Place the PDF document in the top-level directory of your project.



It is important that the name (`worksheet.pdf`), format (PDF), and location (top level directory) of the worksheet are all correct - otherwise we will assume that you have not submitted this task.



¹²It is good practice not to depend unnecessarily on proprietary formats (such as Microsoft Word). Ask on the Piazza forum if you are unsure how to create a PDF document from your source format.

9 Submitting your work

Submission is via Git. Instructions will shortly be made available under: [LEARN](#) > [Assessment](#)



The assignment files will be automatically extracted from your Git repository precisely on the assignment deadline. Changes made after the deadline will not be included.

-  [11] Remember to `add/commit/push` all of your files (to the main git branch), so that they appear in your remote repository. In particular, the following files will be required for the marking:
- (a) `BigController.java`
 - (b) `HashMapController.java`
 - (c) `PropertyController.java`
 - (d) `CacheProxy.java`
 - (e) `BuggyProxyTest.java`
 - (f) `worksheet.pdf`
-  [12] Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance on the [School page](#)¹³. This also has links to the relevant University pages.

We have automated tools for checking both worksheets and code for similarity with other submissions (including previous years). If you are in any doubt about any part of your submission, please discuss this with the course lecturer.

We hope that you found this a useful and realistic exercise. Please do let us know what you think on Piazza.

¹³<https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct>

Appendix A Marking Criteria

When assessing the submitted code, we will consider the following criteria:

Completion: Those with less previous experience may have difficulty completing all of the assignment tasks. It is possible to pass without attempting the more advanced tasks - and a good solution to some of the tasks is much better than a poor solution to all of them.

Readability & Code Structure: Code is a language for expressing your ideas - both to the computer, and to other humans. Code which is not clear to read is not useful, so this is an essential requirement.

Correctness & Robustness: Good code will produce “correct” results, for all meaningful input. But it will also behave reasonably when it receives unexpected input.

Use of the Java Language: Appropriate use of specific features of the Java language will make the code more readable and robust. This includes, for example: iterators, container classes, enum types, etc. But the structure of the code, including the control flow, and the choice of methods, is equally important.

Marks will be assigned according to the University [Common Marking Scheme](#)¹⁴. The following table shows the requirements for each grade. All of the requirements specified for each grade must normally be satisfied in order to obtain that grade.

Pass (Diploma only): 40-49%:

- Submit some plausible code for a significant part of the assignment, even if it does not work correctly.

Good: 50-59%:

- Submit working code for most parts of the assignment, even if there are small bugs or omissions. This must include some plausible code for each of the sections 4, 5, and 6.
- Submit code which is sufficiently well-structured and documented to be comprehensible. This includes an appropriate use of Java features and choice of methods, as well as layout, comments and naming.

Very Good: 60-69%:

- Submit working code for all parts of the assignment, even if this contains small bugs.
- Submit code which is well-structured and documented.
- Provide answers to some of the worksheet questions which demonstrate some understanding of the issues involved.

Excellent: 70-79%:

- Submit working code for all parts of the assignment, with no significant bugs.
- Provide answers to the worksheet questions which demonstrate a good understanding of the issues involved, and propose plausible solutions.

Excellent: 80-89%:

- Marks in this range are uncommon. This requires faultless, professional-quality design and implementation, in addition to well-reasoned and presented answers to the worksheet questions.

Outstanding: 90-100%:

- Marks in this range are exceptionally rare. This requires work “well beyond that expected”.

¹⁴<https://web.inf.ed.ac.uk/infweb/student-services/ito/students/common-marking-scheme>