CS5001 Object-Oriented Modelling, Design and Programming
Practical 2 – OO and Testing

School of Computer Science
University of St Andrews

Due 9pm Friday week 8, weighting 35%
MMS is the definitive source for deadlines and weightings.

Objective

To gain experience with OO programming, coding to interfaces, and unit testing.

Learning Outcomes

By the end of this practical you should understand:

- how to produce a robust implementation according to a specified interface
- how to design and implement unit tests.

Getting started

To start with, you should create a suitable assignment directory structure with a `src` directory for your source code, such as `~/Documents/CS5001-p2` in your network home space (your network home directory) on the Linux lab clients.

Once you have set up your assignment directory, you should decompress the zip file at

http://studres.cs.st-andrews.ac.uk/CS5001/Coursework/p2-oo-testing/code.zip

to your assignment directory. From a terminal window on the lab machines, you can do this using the following commands:

```
mkdir -p ~/Documents/CS5001-p2
cd ~/Documents/CS5001-p2
unzip /cs/studres/CS5001/Coursework/p2-oo-testing/code.zip
```

Please note that the zip file contains a number of files in the src directory, some of which are blank or only partially implemented. Once you have extracted the zip file and have started your implementation, please make sure you don't accidentally unzip the file again as this would overwrite your src directory (and thereby your own implementation) with empty files contained in the zip.

Requirements

You are to develop an implementation of the interfaces for a simple vending machine, product records and products in the `interfaces` package in the `src` directory and also provide unit tests (using JUnit 5) to test your implementation.

You may find information useful on unit testing and JUnit in week 4 lectures (slides 18-26), the week 4 example `CatTestExample` on studres (which shows how to write a simple unit test) and the `CS5001_W04_Exercise1_Overriding` exercise (which included a part on practicing writing unit tests).

Much like many simple vending machines, this vending machine should be able to store a number items belonging to the same product line in a lane. You might imagine the machine having a lane for Ready Salted Crisps, a lane for cans of Orange Juice etc. and each lane is identified by a code such as A1, A2, A3, B1 etc. You might take inspiration from looking at the vending machines in the John Honey building.

You are encouraged to develop your implementation and associated functionality for your methods at the same time as developing a set of unit tests that you try to keep short and such that (as best possible and as mentioned in lectures) each test method only tests one specific thing. That way, you are loosely following an approach called Test-Driven Development (TDD) in which the tests you write for a system feature specify the actual requirements/behaviour of the system (the behaviour of the methods for a given input) and you keep adding short, new tests and the corresponding minimal implementation to satisfy any new tests, while ensuring that all the previous tests are satisfied as well.

As such, for this assignment, it is your job to implement all the classes in the `impl` package and write suitable tests in the `test` package (such as in the supplied `Tests` class) which specify the required behaviour of your methods and demonstrate how your implementation passes these tests. If you want to follow a strict TDD process, you could even develop tests before writing the implementation to satisfy those tests. Obviously if you do that, you will see that any tests you write before the corresponding implementation will initially fail and then your job would be to subsequently write the minimal implementation in the classes in the `impl` package to satisfy the new tests and all your previous tests as well.

Parts of the `impl.Factory` class have already been implemented, for this class you only need to implement the methods containing `// TODO` comments. These methods should be very short and should merely return the result of calling the correct constructors that you decide to implement in the associated classes in the `impl` package. The only reason for providing factory methods here is to allow code to be handed out to you before you have actually implemented any constructors in the classes in the `impl` package, but which nevertheless compiles and runs (albeit initially failing any tests).

Please note that you must **NOT** change the interfaces. Coding to a given interface is an important aspect of OO development and is seen as a valuable exercise for this assignment. That said, you can provide any number of `private` helper methods in your implementation that are not listed in the given interfaces (and as private methods shouldn't be listed). You can also add additional helper classes according to your own design if you want to as long as the classes handed out in the `code.zip` comply with the interfaces that were handed out.

The Javadoc comments in the interfaces in the `interfaces` package give more information on what the individual methods in the corresponding classes should do and how they should

behave. However, should you find some aspects of the interfaces ambiguous, you will need to make a decision as to how to implement the interface, which your tests should make clear and which you should also mention in your report.

When writing your implementation, you should consider:

- what attributes and `private` helper methods you may need in your classes
- what constructors you should provide
- what data structures you may want to use (you can make use of anything in the java Standard Library including any collection classes in `java.util`).

When writing your tests, you should consider:

- normal cases, which you would expect to occur routinely during operation
- edge cases, such as empty collections at startup, potentially full collections, or a collection only containing one element, or generally when your implementation is operating at some edge / limit ...
- exception cases such as dealing with unexpected conditions, `null`s, duplicate lane codes for different products, products being unavailable, in general, conditions under which your tests need to expect (`assertThrows`) that your method implementations `throw` certain exceptions.

You should not add any additional *checked* exceptions beyond those already included in the `exceptions` package. Note that declaring additional *checked* exceptions (i.e. ones that extend `java.lang.exception`) as being thrown by a method would involve changing the interfaces which is not allowed as indicated above. That said, if you feel the need to make use of additional exceptions and can suitably explain and justify their use in your report, Javadoc and verify these in your tests, you can make use of *unchecked* exceptions that extend `java.lang.RuntimeException` and which you do **not** need to declare as being thrown by a method (so the interfaces can remain unchanged).

Note: For this assignment, you will not need to provide a class with a `main` method to try out your vending machine implementation. The `main` method already exists within the JUnit framework and the tests that you write should check whether the functionality you have provided is correct.

The easiest way to run your JUnit tests is by using the auto-checker (see section below).

Running the Automated Checker

The tests for the auto checker have been set up so as to run your own JUnit tests. You can run the auto checker and your JUnit tests on your vending machine implementation by opening a terminal window connected to the Linux lab clients/servers and executing the following command:

```
cd ~/Documents/CS5001-p2
stacscheck /cs/studres/CS5001/Coursework/p2-oo-testing/Tests
```

assuming `~/Documents/CS5001-p2` is your assignment directory.

A single test method is included in a `Tests` class in the `test` package that was handed out. However, it merely checks that the factory method for creating a vending machine product (which you will have to implement) calls a suitable vending machine product constructor (which you will also have to implement in the `VendingMachineProduct` class). The test also demonstrates how to use the factory methods to instantiate objects. Your first step could be to make this failing test succeed before going on to add the next failing test(s), adding the corresponding implementation, re-running all tests etc. if you decide to follow a strict TDD process. The final test `TestQ_CheckStyle` runs a program called Checkstyle over your source code using the Kirby Style as mentioned for the previous assignment.

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/programming-style.html

Deliverables

Hand in via MMS a zip file containing:

- Your entire assignment directory (i.e. containing all your source code and report).
- A PDF report describing your implementation, explaining and justifying any design and implementation decisions (for example which data structures you used and why), describing test cases, how you chose those test cases, and how your choice of test cases might have influenced your implementation. In case you are interested, there is general report template at

https://studres.cs.st-andrews.ac.uk/CS5001/0-General/

Marking

The submission will be marked according to the mark descriptors at:

https://studres.cs.st-andrews.ac.uk/CS5001/0-General/descriptors.pdf

which focus on quality and clarity at the top end in place of enhancements.

A very good attempt in an object-oriented fashion achieving almost all required functionality, together with a clear report showing a good level of understanding, can achieve a mark of 14 - 16. This means you should produce very good code with sensible method decomposition and use of data structures and provide a very good set of tests with clear explanations and justifications of design and implementation decisions in your report.

An excellent attempt achieving full required functionality, written in good style, together with a very clear and well-written report showing real insight into the subject matter can achieve a mark of 17 or above. Quality and clarity of design, implementation, testing (with a comprehensive set of test cases, testing all aspects of your design and covering any cases), and quality and clarity of your report are key at the top end.

Lateness

The standard penalty for late submission applies (Scheme A: 1 mark per 24-hour period, or part thereof):

http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/education/handbook/good-academic-practice/