Testing Assignment A1

Points 10

Testing Assignment A1

Background

You are a consultant who has been hired by 40 Hectare Forest Ltd to create a test suite for the Therapeutic Goods Register module (TGR) they require to be integrated into their large Enterprise Resource Planning infrastructure. They have also requested you to implement a small component as a 'proof of concept'.

Note: The development and testing processes you will be using here are for assessment purposes – they do not resemble normal industry practice. The API design has also been modified from best practices in order to support assessment.

Your assignment will be automatically marked by a script. This places strict requirements on your classes and filenames, as well as on what you are able to assume in your code. You will be provided with package structure that you must follow along with the API. Your marks will be drawn entirely from the marking script – failure to follow these instructions will lead to automatic loss of marks (up to 100% depending on script output).

Assignment Requirements

For this submission, you must use the information contained in the 40HF System API V1 documents to complete the following tasks:

Product Module

- Implement a full micro/unit test suite for a concrete implementation of the Product Module covering the following classes:
  - ProductListImpl implements ProductList
  - ProductListFactory
  - ProductImpl implements Product

- Your test suite must test all methods implemented from the public facing API – do not include tests for methods not specified in the public facing API (or reference them in your test suite in any way – this would cause the marking script to fail to compile)
- Your test suite will be marked using some different implementations of this Product Module
- Implement this module and ensure it passes your test suite
- Your implementation will be marked using a known working test suite

TGRFacade Module

- Implement a test suite for a concrete implementation of the TGRFacade covering the TGRFacadeImpl class only.
- Again, your test suite must test all methods implemented from the public facing API but nothing else – this would cause the marking script to fail to compile.
- For this preliminary submission your tests should use a concrete instance of your fully implemented Product module to create your test fixtures (consider these larger integrated tests rather than micro/unit tests). This means you may use a concrete instance to both set up the test, and inspect that instance to determine whether the test is successful.
- You do not need to implement this module.

For your reference there are a few elements here:

- 'Your implementation' - this is your implementation of the Product module (which is all contained within the Product package).
- 'The implementation(s) you will be testing' - these are the marking implementations of both the Product and TGR packages - one working, several with bugs.
- 'Your test suite' - this is the test suite you write for both the Product and TGR packages.
- 'The marking test suite' - this is a test suite that covers the Product package which will test your implementation of the Product module.

Submission Requirements

You will be submitting your code as a GitHub Repository. This GitHub Repository must be created under your unikey account on the **https://github.sydney.edu.au** **(https://github.sydney.edu.au)** platform, with the repository name *exactly matching*:

- SCD2_2021_A1

You must create this repository as a PRIVATE repository. Make sure it is private, otherwise you could get in trouble for academic dishonesty!

You must add the following unikeys as 'collaborators' so both the marking script and the teaching team can access your work. Do not add any other collaborators, and make sure you get the spelling/numbers correct to avoid releasing your code to somebody else:

- jbur2821

- mmcg5982
- bsch0132
- ttho6664

Your repository should match the file structure of the provided 'skeleton' gradle project (keep the package and directory structure exactly as it is given to you). You can add other files if you like to help you while you work (for example, an implementation of TGRFacadeImpl), but only the following files will be assessed:

- TGRFacadeImplTest.java
- ProductImpl.java
- ProductImplTest.java
- ProductFactory.java
- ProductFactoryTest.java
- ProductListImpl.java
- ProductListImplTest.java

You can see a public repository example of this submission structure at **https://github.sydney.edu.au/JBUR2821/SCD2_2021_A1 (https://github.sydney.edu.au/JBUR2821/SCD2_2021_A1)**

Marking Mechanism

The marking for this assignment is done by a script. This script will run each night at around 3am, based on your last pushed commit to your repository's master/main branch. Below is a simplified description of the process the marking script will follow so you can better understand the feedback it gives you. Feedback will be available through this Canvas assignment. Note that some feedback will be hidden until the due date!

- First, it checks to see if it has access to a correctly named repository for your unikey. If it does not, it terminates.
- If it has access to a repository, it will clone the repository, and retrieve the latest pushed commit you have made to the master/main branch (most likely HEAD). Don't do anything like deleting or renaming the master/main branch, but working on other branches is perfectly fine. The script will only look at master/main though.
- Once it has the latest commit, it will parse the directory tree to see if it looks like it should (i.e. it will look for the 7 assessable files in the directories they should be in. If they are not there, it terminates.
- If it has found the files, it will move them into the test harness. Your other files are ignored (your assessable code cannot rely on them!). This test harness includes:
  - An environment for testing your implemented code (i.e. **my** tests will be run on **your** code). You will not be told whether your code passes or fails these tests until after the due date (you will be told if it fails to compile)
  - Multiple environments for testing your test cases (i.e. **your** tests will be run on **my** code)

    - One version will have no bugs. If you reject this version as being bugged, the script will terminate. You MUST pass the working version in order to gain any marks at all.
    - 20 versions will have one bug each. You gain marks based on the number of bugged versions you reject as bugged. Most bugs (75%) will be 'hidden' until after the due date.
- Each of these will be checked with the gradle command 'gradle test' - using the same build.gradle file you have been provided.
- Your code may fail to compile. If this is the case the script will terminate. Your code MUST compile in order to gain any marks at all. This can occur separately depending on which files are being tested (that is, your implementation might compile and run, but a test file might fail).
- Once all of the above completes successfully a mark will be calculated and the script will terminate.

Your feedback will include some of the following, depending on how far the script got:

- If the script terminated prematurely, you will be given a message indicating when it terminated. Any errors generated (such as compile-time errors) will be included.
- If this is a 'before the due date' marking run, and the script completed, you will receive the following:
    - A message indicating your code structure appears to be ok and your code compiled successfully
    - A message indicating the number of 'open' bugs you have caught vs missed
- If this is the 'after the due date' marking run, and the script completed, you will receive the following:
    - A message indicating your code structure appears to be ok and your code compiled successfully
    - A message indicating how many tests your implemented code passed vs failed, including the JUnit report
    - A message indicating how many open AND hidden bugs you have caught vs missed, including what those bugs were
    - A mark derived from the above based on the marking guide.

Resources

40HF_System_API_V1.zip - this is the API you must target for your tests and implementation.

skeleton.zip - this contains the repository package structure - you must follow the package structure indicated. The included build.gradle file indicates the only externally imported libraries that will be available in the marking environment (no Apache Commons, Google Guava, alternate test frameworks/versions).

Some important notes:

- Ensure you stick to the folder structure, package structure, and filenames required for this assignment – the marking script will not know the difference between a typo in the filename and a syntax error and will fail you either way! In particular do not reference methods not declared in the public API documents – the code that is swapped in will not implement any other methods and this will cause a compilation failure.

- Pay attention to what classes you are supposed to test – you do not need to test any of the given interfaces, you will be testing concrete implementations of those interfaces based on the requirements the interface javadocs specify.
- You will be testing the defensive programming elements of these modules as well as their actual operations – that is, do they correctly identify and reject input that breaches their preconditions in the way their API says they will. However – YOUR implementation may not rely on the defensive programming of other elements (for example, the ProductListImpl class may not assume the ProductImpl class will handle rejecting input which breaches ProductListImpl preconditions, it needs to be making its own checks). However the implementation you are given to test may (and will) rely on correct *outputs* from its dependencies and you can may this in your implementation also. (Much more on this in the next assignment!)
- Something to make things easier: You may assume that the implementations you are given are entirely deterministic – there is no use of any pseudorandom functions, the system does not react to the system clock anywhere it doesn't tell you it will in the API, it does not query the network, and it does not look at the current hardware. This is obviously NOT something you can assume when doing real testing! (be careful though - some Java in-built classes do not offer guarantees you might assume - for instance the order of certain collections)
- All postconditions should be considered to have an implicit 'and no unrelated externally observable effects' requirement. That is, for example, Product.setApprovalStatus does not explicitly say that this operation should not modify the output of Product.getName from what it was before setApprovalStatus was called, but this and all similar cases should be assumed. You do not need to test for breaches of this requirement (none of the bugs you need to catch are like this), but breaching it in your implementation may result in your implementations failing assessment testing.
- Unless otherwise specified, this API does not make any guarantees of concrete implementing class – that is, where List is specified, ArrayList or LinkedList or a custom List would all be valid. Do not make more detailed assumptions of behaviour when testing.
- In the Javadocs, preconditions are the combination of the 'Precondition', 'Parameter', and 'Throws' sections. Postconditions are the combination of the 'Postcondition' and 'Return' sections.
- Do not rely on the copied information in the concrete class javadocs (not all of the documentation gets copied) - refer to the interface specification.
- You may find 'best practice' information that says you should not test simple methods like basic getters and setters - this is correct, it's usually a waste of time. For the scope of this assignment however (and the next one) you should be testing everything, even the simplest methods.
- Sanity note: If your tests passing mean you can say with certainty that the API is adhered to, then you are 100% guaranteed to pick up all of the marking bugs. Each of them directly breach something said in the API - however a detailed and correct comprehension of this API will be required!
    - To give you an idea of just how much easier this makes things (and how hard real world testing is), there is no bug that only occurs if a Product object has an ID of 1337.

- That also means you don't need to test for some things that you normally should - such as integer over/underflows. Stick to the API. The test suite used for your implementation follows the same mindset.
- Conversely, DON'T add things not required by the API - either in your implementation, or your test suite. e.g. if the API doesn't say to throw an exception, then don't (and don't have your tests expect the implementation you are given to throw one either). There is at least 1 deliberate gotcha here where a well designed and consistent system would act differently - but we're just here for verification, not validation.
- Lists that do not guarantee order in the API should not be tested with a required order.

Marking Guide

Note: For each bug section, marks are only available IF your test suite accepts the given working example. If your test suite marks the working example as bugged, the total score for that section will be 0%. The marking is not linear - passing the working example and detecting 50% of the bugs will yield more than 50% of the available marks, but hidden tests are worth 75% vs 25% for the open tests.

- 0% Does implementation & test suite compile, and is the working version passed? (must achieve this for any marks in the relevant sections)
- 2% for your implementation of the Product module passing the marking test suite (you will only be told if your code compiled before the due date, all of the tests run on your implementation are 'hidden')
- 3% for detecting 8 TGR bugs (1% for open, 3% for closed - 2 open bugs)
- 3% for detecting 6 ProductListImpl bugs (1% for open, 3% for closed - 1 open bug)
- 2% for detecting 4 ProductImpl and 2 GodsFactory bugs (0.5% for open, 1.5% for closed - 1 open bug each)