Aston University
Birmingham B4 7ET
http://vle.aston.ac.uk/

Coursework Specification 2018 v1.1

Antonio García-Domínguez

Module DC2300 June 8, 2018

1 Change history

- V1.0: first version.
- V1.1: clarify what we mean by "show it on screen" for the code skeleton of the first submission.

2 Introduction

This document contains a specification of the software and other documentation that form the assessed coursework assignment for the module DC2300: Java Program Construction.

The project is to be carried out in small teams: you will see your assignment on Blackboard. I recommend that you set up communication and sharing channels as soon as possible: instant messaging is a good idea, plus some way of sharing files (e.g. Google Drive or Dropbox). If you can, learning a version control system such as Git quickly pays off over using Drive/Dropbox when working on code.

Regarding marks, the project will be first marked as a whole, and then your contribution will be factored into the group mark to obtain your individual mark. For that reason, it is important that you provide evidence as to what your contribution was. If you do *not* contribute and attend coursework webinars regularly, I may remove you from the group: in that case, you will have to complete the coursework by yourself.

If you have any questions on how the coursework runs or what to do, feel free to use the discussion forums or contact us. I prefer using the forums over email, since everyone will be able to see our replies as well.

3 Problem Description

The task is to develop a simulation of a self-service petrol station with a certain number of pumps and a shop. Different types of vehicles may come to the station to top up, (optionally) pick up some food and necessities and then pay for everything at one of the tills. The purpose of the simulation is to study what level of demand can be handled with a certain number of pumps and tills, keeping customers happy and net income high. All value ranges mentioned are uniformly randomly distributed.

You will have various types of vehicles:

- **Small cars** have 7–9 gallon tanks, and take up 1 unit of space in the queue.
- **Motorbikes** take up 0.75 units of space in the queue, and their gas tanks are smaller (5 gallons).
- **Family sedans** take up 1.5 units of space in the queue, and their gas tanks are larger (12–18 gallons).

Note that the probabilities of the purchases, the sizes of the gas tanks and their behaviour are subject to change in the future. You should make your classes flexible enough to adapt to this without major alterations to the software.

Your simulation should model the petrol station with a time resolution of 10 seconds (1 "tick"). The system has the following configuration:

1. There is a configurable number of pumps. Each pump has a queue that can fit up to 3 units of space (e.g. 2 sedans, 3 small cars or 4 motorbikes). Pumps provide 1 gallon per tick of fuel.
2. There is a configurable number of tills in the shop. Paying at the till takes 2–3 minutes.
3. Small cars and motorbikes arrive with a probability of p per tick.
4. Family sedans arrive with a probability of q per tick.
5. Customers always go to one of the least occupied queues first (both for pumps and for tills).
6. If a vehicle arrives and does not fit into any of the queues, the vehicle will simply leave.
7. Vehicles always fully top up their tanks. By default, one gallon is £1.20 for the simulation.
8. Vehicles stay in the queue for the pump while the driver goes to the shop to pay at the till.
9. The vehicle starts topping up on the next tick after it gets to the front of the queue for the pump, and the driver goes to the till on the next tick after it tops up.
10. You must track how much money was earned, and how much money was lost because of vehicles being unable to enter the station.
11. You must write test classes in JUnit 4 for at least **five** of your classes.

There is a second set of requirements that you will need to complete for full marks:

1. A graphical user interface should be provided and allow the user to set values such as p and q, the price of the gallon, and the period of time that the simulation should run for.
2. You must also simulate a shopping area in the gas station. Happy customers that refill quickly will spend some time looking around (away from the tills), before queueing up at a till on the next tick and paying some additional money:
   - For small cars, if the refill is done in less than 5 minutes since arriving, there is a probability of 0.3 that the driver will shop for 2–4 minutes to spend an extra £5–£10.
   - For family sedans, if the refill is done in less than 10 minutes since arriving, there is a probability of 0.4 the driver will shop for 2–5 minutes to spend another £8–£16.
   - Motorbike drivers in the area are thrifty and will never go to the shopping area.

You will need to track the money lost from missed sales as well. It may be interesting to track it separately from the money gained from selling fuel.

3. Trucks may also arrive at the station. They take up 2 units of space. After refilling their 30–40 gallon tank, a truck driver that refilled in 8 minutes or less will always do a purchase of £15-£20 after browsing for 4–6 minutes.

   An unhappy truck driver will let the other truck drivers know about the bad service. This will make it less likely that they come. Trucks arrive with a probability of t, which is initially $t_0 = 0.02$. An unhappy truck driver will reduce t by 20% of its current value: $t' = 0.8t$. A happy truck driver will increase it by 5% of its current value, up to the original value of t: $t' = \min\{1.05t, t_0\}$.

The aim of running the simulation is to decide for each station configuration (number of pumps and tills) which level of activity it is best suited for: that is, which are the values of p and q that report the highest net income (raw income minus missed sales).

To do this, the simulation should be run for four hours (1440 ticks) for all independent combinations of the values below, and the results should be averaged over 10 different seeds for the random number generator:

- p: 0.01, 0.02, 0.03, 0.04, and 0.05.
- q: same options as p.
- Pumps: 1, 2 and 4.
- Tills: 1, 2 and 4.
- With and without trucks (if implemented).

For full marks, you should run separate studies with and without trucks. The owners of the gas station are currently wondering if they should allow trucks to refill at the station or not, so you should compare for each station configuration if it is better to allow trucks or not.

4 Design Notes

The following information and ideas may be useful in developing the simulation system. You will also find some of the ideas and structures from the lab classes helpful.

1. Think generically! You should aim to write a small library of classes that can be used to build the scenario described above, but that would also support similar scenarios (more types of vehicles, smarter queueing) without change to the library classes. This is one of the assessment criteria for the design.
2. When deciding which type of vehicle will arrive, first generate a random number between 0 and 1. If it is less than or equal to p, it will be a small car. If it is between p and $2p$, it will be a motorbike. If it is between $2p$ and $2p + q$, it will be a sedan. Otherwise, no vehicle will arrive. **Note**: trucks would add another range between $2p + q$ and $2p + q + t$.

3. It is easier to start developing the simulation as a console application, while keeping a clean separation between the core logic (the "model"), the the part that shows the current state of the simulation (the "view"), and the "controller" that coordinates the two. The GUI could be added later. The GUI would be used to set values such as p and q, the number of pumps, and the period of time that the simulation should be run for.

4. You must develop all the classes yourself, with the exception of the standard library classes (e.g. random number generation, collections, GUIs, standard exceptions, and I/O).

5. It will be easier to run the study as a console application. The results should be printed to standard output (`System.out`) or written to a file. If they are printed to standard output, in Unix-based systems (GNU/Linux or Mac) you can then use output redirection to put the values into a file. The command

   ```
   java bar > foo
   ```

   overwrites the contents of `foo` each time the program `bar` is run. To append the results of each run you should use the syntax

   ```
   java bar >> foo
   ```

   If you are running the program from Eclipse, you can use the options in the "Common" tab to redirect the output to a file.

5 Deliverables

The coursework will be divided into two submissions:

- The first submission is due on **Saturday 16th June**. This submission is worth 30% of the marks, and focuses on the early analysis and design steps rather than on the code. The marking scheme is on Table 1. Your group must upload to Blackboard a ZIP with:
  - A PDF with the noun-verb analysis of the problem, with Class/Responsibility/Collaboration (CRC) cards detailing what each class should do (at the analysis level).
  - A PDF with UML class diagrams of the intended classes for the system. You may hand draw and scan this, use a drawing program with a UML stencil (e.g. draw.io, Visio or UMLet), use a UML modelling program (e.g. Papyrus or GenMyModel), or extract it from your code (e.g. ObjectAid).

    Usually, it is better to use one UML diagram per subsystem rather than trying to put everything in the same diagram. The same class can appear in multiple diagrams if it makes sense: usually it will be fully detailed in one and simplified in the others.
  - An executable Eclipse project with a first version of the sources of the Java classes. These should be based on the UML diagram and have enough code to contain the state of the simulation, but not necessarily its behaviour. The main() method should be able to set up a gas station and show its components on a screen (whether through a textual or a graphical user interface).

    Basically, we should be able to see a static snapshot of the pumps, tills and shopping area of a station, with some vehicles and/or customers in them (depending on your choice of internal representation). The static view should be derived from real domain classes, rather than just

Component	40–49	50–59	60–69	70–79	80+
Noun-verb and CRC (40%)	A noun-verb analysis identifies the key concepts, and the CRC cards organise and relate them.	The noun-verb analysis has been done in a methodical manner, and the CRC concept is followed meaningfully.	Noun-verb analysis is thorough, only missing some minor requirements. There is evidence of an agreed vision in the team.	Duplicate and unclear requirements have been resolved, and CRC cards lend themselves well to a UML class diagram.	Roles and responsibilities have been distributed while taking into account good system design properties.
UML classes (40%)	A UML diagram has been submitted which is related to the CRC cards. Notation has noticeable flaws.	UML diagram is mostly complete, and the notation has only minor flaws.	Multiple UML diagrams have been used to describe the various subsystems, and classes are laid out to aid readability.	UML diagram is flawless, and packages and comments have been used effectively.	UML diagram is complemented with meaningful explanations for the various decisions taken among various alternatives.
Code skeleton (20%)	The code runs and shows an object-oriented representation of the gas station in text form.	The code is organised meaningfully into packages, and corresponds to the UML class diagram.	The code is well documented with Javadoc comments, and exhibits high cohesion and low coupling.	The grid is presented in graphical form, and some unit tests are present but may not be exhaustive.	There is good separation between graphical presentation and internal state, and unit tests cover well available functionality.

Table 1: Marking scheme for the first submission

drawn on screen. For a console application, you should have a text view component that loops over the state of the simulation and prints it. For a GUI, a window with one list per till/pump/shopping area would be enough.

You could make something more engaging than that, but we would rather have you spend your time doing a good noun-verb/CRC analysis than drawing a realistic gas station at this point. Keep an eye out for the weights in the marking scheme!

Additionally, each group member must submit a PDF with an individual reflection on their contribution to the overall work, and how the team has operated during this first half.

- The second submission is due on **Thursday 26th July**. This submission is worth 70% of the marks. The marking scheme is on Table 2. You will upload to Blackboard a ZIP with:

  - A short (2-3 pages) description of the overall design of your system. What are the various parts, and how are they related to each other? Have you followed any design patterns? How have you kept cohesion high and coupling low?
  - Javadoc documentation in HTML format, generated from your comments. Make sure you provide comments for all your classes and your methods. In general, Javadoc comments should focus on *what* the code does, rather than *how* it does it.
  - A report on what happens in the simulation as you change the different parameters. Check the end of Section 3 for details.
  - An executable Eclipse project with the final version of the simulation, and a "good" test suite written with JUnit. With "good", we mean that it covers the most important parts of the functionality of your system and that it also checks that failure/invalid states are handled correctly. Please do not write obvious tests that simply set a value and check it is returned.

    We *heavily* recommend you do these at the same time you develop the rest of the system. It is usually better to gradually build and test your system up, rather than building everything and then praying that it works!

  As in the previous submission, each group member must individually upload a PDF with a reflection on their contribution and how the team has operated.

Late submissions will be treated under the standard rules for Computer Science, with an **absolute** deadline of one week after which submissions will not be marked. This is necessary so that feedback can be given before the start of exams and to spread out coursework deadlines. The lateness penalty will be 10% of the available marks for each working day.

Component	40–49	50–59	60–69	70–79	80+
Design (20%)	An object-oriented approach has been used, with different classes representing the actors. A description of the design was provided.	Classes have been organised into subsystems, with some minor issues (e.g. dependency cycles). Some classes may be larger than necessary.	Subsystems are organised meaningfully, and inheritance and composition have been used effectivelly to avoid code repetition.	Logic and presentation are explicitly separated, and some thought has been given to testability. Design patterns are used effectively.	Components can be reused beyond this simulation, and cohesion and coupling are good across all classes.
Implementation (60%)	The code runs, and most of the expected functionality is available. The program may crash under some (but not all) normal configurations.	Canonical form has been applied meaningfully, and the program does not crash in normal circumstances. Code is well formatted and key classes are documented.	The code follows Java naming conventions well, and makes good use of static/final and access control.	A graphical representation of the simulation is implemented in an effective manner, and the program is easy to use. Internal data structures are well chosen. Either trucks or the shopping area work.	The simulation is engaging to watch, and there are no gaps in the Javadoc documentation. Code is concise and kept decoupled. Both the trucks and the shopping area work correctly.
Testing (15%)	There are test cases for each type of actor. Most of the test cases pass, though a few corner cases may fail.	All test cases pass. Test cases take into account the most common scenarios for each type of actor.	Test cases also cover invalid inputs and failure scenarios. There is evidence of a methodical approach to testing.	Test cases cover the implemented optional requirements. A test plan has been derived from the requirements and executed meaningfully.	Test-driven development has been applied in a disciplined manner, and testability is explicitly considered in the design.
Evaluation (5%)	There is a report on the behaviour of the simulation, which may have gaps in presentation or argumentation.	The report evaluates multiple values of one parameter as the others remain fixed, with some evidence.	The report evaluates the impact of each parameter separately, and provides strong evidence.	The study is done in a planned manner, with hypothesis, experiments and discussion of the results.	The study is engaging to read, and the conclusions of the study are insightful and well written.

Table 2: Marking scheme for the final submission