Aston University

Birmingham B4 7ET http://bb9.aston.ac.uk/

Coursework Specification 2019-20 v1.1

Antonio García-Domínguez

Module CS1410 February 27, 2020

# Change history

* V1.0: first version.
* V1.1: clarify number of floors in the building: it should be ground floor + 6 floors above it, given the mention of “floor 6” in point 4.

# Introduction

This document contains a specification of the software and other documentation that form the assessed coursework assignment for the module CS1410: Java Program Development.

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. GitHub has free educational licenses if you use your Aston email account, and Gitlab provides unlimited private projects for free.

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 sessions 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 contact us. You can also ask the Programming Support Officer for questions about Java itself.

# Problem Description: Basic Requirements

The task is to develop a simulation of the elevator of an office building. Developers, other employees, clients, and maintenance crew will all attempt to use the elevator. The purpose of the simulation is to see how much internal and external activity we can handle in the building.

3.1 Types of people

Several types of people will queue up to use the elevator:

Employees who are not developers will attempt to go to any other floor with the same probability.

Developers only work in the top half of the building. Developers may randomly decide to move to another floor in the top half.

Clients will enter the building and go to one of the floors in the bottom half of the building (which may include the ground floor). After they complete their business, they will return to the ground floor and leave.

Maintenance crews will arrive and go to the top floor, taking up 4 spaces with their equipment and materials. After they complete their work, they will return to the ground floor and leave.

3.2 Main features of the simulation

Your simulation should model the building with a time resolution of 10 seconds (1 “tick”). It should not be a real-time simulation: it is not that a tick should take 10 seconds to run, but rather that it *represents* 10 seconds of the simulation. You should simulate a working day (8 hours).

The simulation is set up as follows:

1. The building has 7 floors (ground floor + floors 1–6) and 1 elevator with capacity for 4 people or 1 maintenance crew.
2. The elevator starts at the ground floor. If there are no pending requests, the elevator will go by itself to the ground floor and rest there.
3. The elevator only moves when its doors are closed, and moving up or down one floor takes 1 tick. When the elevator reaches one of the requested floors, it will open its doors. The doors will stay open until the elevator detects nobody entered or left in the last tick, and then they will close. It takes 1 tick to open or close the doors.
4. The elevator keeps going in the same direction (either up or down) until there are no more requests in that direction or it cannot go any further.

For instance, assume that the elevator is currently at floor 2 going up, and has requests for floors 1 (person A waiting to go to floor 6), 3 (person B riding the elevator) and 4 (person C waiting to go to floor 5). The elevator could do this:

* + - Stop at floor 3: B leaves. Keep going up.
    - Stop at floor 4: C enters and asks for floor 5. Keep going up.
    - Stop at floor 5: C leaves. Since there are no requests further up, start going down.
    - Stop at floor 1: A enters and asks for floor 6. Start going up.

1. Any number of people can leave and/or enter the elevator in 1 tick. If the elevator reports that there is not enough space for someone to come in, they will have to wait and request the elevator once more after it has left the current floor.
2. The user sets the number of employees and developers in the building. These enter the building at the beginning of the simulation through the ground floor and stay within the building for the entire simulation. No new employees or developers enter the building during the simulation.
3. Upon entering the building, employees and developers will immediately pick a floor to move to. After they reach that floor, each employee and developer may independently decide to change floor with probability *p* for each tick. Note that you can simulate probabilities by generating a random value through the nextDouble() method in the java.util.Random class, and checking that the generated value is less than *p*.
4. Clients will arrive at the ground floor with a probability of *q*. Between 10 and 30 minutes after they arrive at their desired floor, they will go to the ground floor and leave the building.
5. Maintenance crews will arrive with a probability of 0*.*005 per tick. Between 20 and 40 minutes after they arrive at their desired floor, they will also go to the ground floor and leave the building.

For the most basic version of this simulation (for a maximum mark of 69%), you can assume that the elevator has only one “call” button (no “up” or “down” buttons), and the interface will be entirely in text.

3.3 Study requirements

As part of this coursework, you will need to run a study to decide the largest combinations of *p* and *q* that can be handled by one elevator for a certain number of floors. In other words, on average users should not be waiting more than 10 minutes (60 ticks), and if you have impatient clients (see below), there should not be more than 5 complaints from the clients in a day. You should also study how *p* and *q* relate to the number of complaints and the average waiting time.

To do this, the simulation should be run for a working day (8 hours) for all possible combinations of these parameters:

* *p*: 0*.*001, 0*.*002, 0*.*003, 0*.*004, or 0*.*005.
* *q*: 0*.*002, 0*.*004, 0*.*006, 0*.*008, or 0*.*01.
* Seed: 10 different seeds, i.e. 10 different values for the constructor of the java.util.Random pseudo-random number generator. You should ensure that a single instance of Random is reused throughout your simulation, and that results are repeatable as long as you use the same seed.
* Developers: 10 (for a level 1 submission, 5 work for Goggles and 5 work for Mugtome).
* Employees: 10.

# Problem Description: Advanced Requirements

There are four advanced requirements that you could meet to achieve higher marks. You can achieve full marks by implementing three out of these four:

1. You can achieve up to 10 extra marks in the implementation component if you can simulate a smarter solution than using a single elevator: you could distribute the work over multiple lifts, or provide two buttons for the elevator (requesting to go “up” or “down”).
2. You can achieve up to 10 extra marks in the implementation component if you provide an alternative “playback” mode where the requests from the people in the building are read from a file, rather than being generated on the fly. You will need to create your own file format and request file generator as well.
3. You can achieve up to 10 extra marks in the implementation component if you create rivalries between developers and make clients a bit more impatient.

The developers would be now divided between two companies: Goggles and Mugtome. Their rivalry is so fierce that their developers will not ride the elevator at the same time. If a developer sees a rival riding the elevator, they will go back to the end of the queue and wait for the next elevator.

Clients will only wait for 10 minutes to take the elevator: if it takes any longer, they will file a complaint and leave. To reduce the number of complaints, clients will now have priority when entering the elevator: you may find priority queues useful to model this fact.

1. You can achieve up to 10 extra marks in the implementation component if your game has a graphical user interface (preferably JavaFX, but you can use another toolkit at your own risk). The GUI should allow the user to set values such as *p* and *q*, the number of floors of the building, the capacity of the elevator, the number of employees and developers (perhaps divided between Goggles and Mugtome), and the period of time that the simulation should be run for.

After clicking on the “Run” button, the GUI should visualize the simulation as it is running. The GUI should show an abstract representation of the building and the people inside as it is going through the simulation: for instance, you may represent people as items in list boxes (which would be the floors and the elevator), or as icons within geometric figures. You are not required to provide animations of any sort for this visualization.

# 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. Don’t jump into coding straight away. Think about the problem, use CRC cards to draw out a good distribution of responsibilities, come up with a good set of classes, create a skeleton with only empty methods, and once you have agreed on the skeleton have different team members fill in the blanks.
2. 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 (different number of floors, more types of users, more than one elevator) without too many changes to the library classes. This is one of the assessment criteria for the design.
3. You should start with the simplest version of the problem: an elevator with no employees. Once you have that, then extend it to include one employee that does nothing but enter and exit the elevator. After that, you can start introducing developers, clients and maintenance crew into the simulation.
4. You should use exceptions and assertions (discussed later in the module) to detect inconsistent states as early as possible. An exception is ideal for reporting invalid “untrusted” (e.g. from a class it doesn’t know) parameters that your method is given. Assertions are good for checking things that should be true before some code (preconditions) or after some code (postconditions).
5. You should focus on the hardest parts of the design first. In this case, it would be the elevator itself. Make sure you test it thoroughly before implementing the various types of people in the building: a dummy person with no behaviour of its own could be useful. Use exceptions and assertions to detect inconsistent states as early as possible: exceptions are ideal for checking preconditions, and assertions are good for checking postconditions.
6. For debugging and visualization, it will be useful to give each user of the elevator their own unique identifier, so you may track them as they go around the building.
7. You should keep the core of the game separate from the actual UI. For a text UI, have different classes run the simulation and render it into text. For a graphical UI, have the interface redraw the building between ticks, or make the grid *observable* so the GUI reacts naturally to any changes. The second option is more elegant to use but more challenging to implement.
8. You must develop all the classes in your team, with the exception of the Java standard library classes (e.g. collections, JavaFX, math routines).
9. You will want to talk amongst yourselves to find your relative strengths and weaknesses, and play on those. Some of your team members may be better at doing the analysis, while others may do better at coding, designing the UI, coming up with tests to run or writing good reports.
10. You should think of a way to cleanly divide the work across the team, in a way that allows for easily integrating what you do on your own, and which allows for several people to work on things at the same time. For instance, you could break the work into more or less these “chunks” or subsystems:
    * The elevator.
    * The different types of people in the building.
    * The integration of the elevator and the actors into a simulation.
    * Any extra requirements (smarter elevators, playback mode, complex actors).
    * (Graphical and/or textual) user interface for configuration.
    * (Graphical and/or textual) user interface for visualisation of the running simulation.
    * The study on the values of *p* and *q*.
11. Read each other’s code, and test the classes of other team members! A bit of pair programming will be great, especially if you have people with very different levels of experience in coding in the team. The more experienced person can learn a lot when teaching another: it’s not a one-way street, and in your future job you will eventually be mentoring your juniors anyway.

The idea is that one person types the code (the driver) and focuses on the low-level details, while another person thinks about the overall task and how the code fits into the intended design (the co-pilot). It’s best if the more experienced person acts as the co-pilot, though you may want to rotate roles every 20-30 minutes.

1. The submission should provide evidence that all team members contributed roughly equally although they may have had different roles. If team members do not contribute equally, then there should be evidence that they were invited to do so. The marks for the teamwork component will be reduced if there is evidence that the team did not work effectively.

In terms of evidence, you can provide a list of project meeting minutes with dates, attendees, and evidence that the meetings were announced to everyone in the group. If one of your members is not confident about programming, try including them in a session of *pair programming* with someone who is more confident.

# Deliverables

The coursework involves several types of submissions:

* Every week on Fridays from the week the coursework starts, we will have a team submission asking to submit your weekly meeting minutes, where you should mention what has been done since the last meeting, who has done it, and who has attended the meeting.

You should also make notes of what has been decided or worked on during the meeting. Failure to provide these reports will impact your teamwork component, and also your individual mark (as we will have no evidence of your individual contribution).

* We may have some interim submissions, asking you to upload your early drafts of some of your artifacts. These interim submissions will only be checked to provide you with early feedback, and will not count towards your final marks. However, failure to meaningfully engage with the interim submissions will impact your teamwork mark.
* You will have to do a team submission of a ZIP file with the final version of a number of artifacts by the Friday after the Easter holidays (Friday 1st May) at 23:59 UK time. We will list down the expected contents of that ZIP file below.
* By Friday 1st May 23:59 UK time, you will have to complete an individual submission with a reflection of your experience doing the project, a statement of your contribution towards the coursework, and a statement of the contributions made by the other team members. Your submissions will be kept private and confidential from your other team members.

Your individual mark will be based on the team mark, with a correction factor based on your individual contribution to the team. Please note that this correction factor can be large enough to give a First to some members and fail others.

The team mark is computed from a combination of different factors:

* Teamwork: evidence of collaboration, engagement and joint learning (20%)
* CRC cards: good use of responsibility-driven design (5%)
* UML diagrams: good use of notation and strong link to CRC cards (10%)
* Design of your code: is it good, object-oriented code? (10%)
* Implementation: working complete simulation (40%)
* Testing: unit tests in JUnit with good coverage (10%)
* Evaluation: study on *p*/*q* with good insights (5%)

To provide evidence of these components, we will ask for the following deliverables in the final ZIP your team will send to us:

* A set of clear photographs of your CRC cards. You do not need to copy their contents to a document: we only require that the cards are readable and that both the front (with name/purpose/roles) and back (name/responsibilities/collaborators) are shown.
* A set of pictures with UML class diagrams of the intended domain classes for the system. We heavily suggest you use a “proper” UML tool (e.g. PlantText, Eclipse Papyrus, GenMyModel) or at least a UML-aware drawing tool (Umlet, draw.io). Generally, extracting it from your code does not produce good results.

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.

* A short (2 pages max) 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?
* An executable Eclipse project with the final version of the simulation, with high-quality Javadoc comments 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.

You do not need to test trivial parts (e.g. getters/setters). You should have tests for the “happy” scenarios, and for failure scenarios as well. Using a test coverage tool to find tests (e.g. EclEmma in Eclipse) you may be missing is recommended.

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!

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.

Please see the subsections below for the mark descriptors.

6.1 Teamwork

40–49 Most team members have contributed to the submission, and the team has made a credible attempt at engaging the non-contributing members.

50–59 All team members have contributed to the submission, and there is evidence of regular attendance to meetings and coursework sessions.

60–69 The team takes into account strengths of each member, distributes work accordingly, and engages meaningfully with draft submissions and interim feedback.

70–79 The team is aware of its performance, and members are learning from each other.

80+ The team actively experiments with ways to improve how they work, and reflects on those experiments.

6.2 CRC cards

|  |  |
| --- | --- |
| 40–49 | The CRC cards enumerate a part of the relevant concepts, but a few are missing. |
| 50–59 | The CRC cards capture most of the relevant concepts, and role stereotypes are used effectively. |
| 60–69 | The CRC cards capture all the relevant concepts, roles and stereotypes are well specified. |
| 70–79 | CRC cards have a good distribution of responsibilities, with a distributed collaboration style. |
| 80+ CRC cards show a clear grasp of responsibility-driven design, with well-designed collaborations. | |

6.3 UML classes

40–49 A UML diagram has been submitted which is related to the CRC cards. Notation has noticeable flaws.

50–59 UML diagram is mostly complete, and the notation has only minor flaws.

60–69 Multiple UML diagrams have been used to describe the various subsystems, and classes are laid out to aid readability.

70–79 UML diagram is flawless, and packages and comments have been used effectively.

80+ UML diagram is complemented with meaningful explanations for the various decisions taken among several alternatives.

6.4 Design

40–49 An object-oriented approach has been used, with different classes representing the elevator and the various types of people. A description of the design was provided.

50–59 Classes have been organised into subsystems, with some minor issues (e.g. dependency cycles). Some classes may be larger than necessary.

60–69 Subsystems are organised meaningfully and without dependency cycles. Polymorphism and composition have been used effectively to avoid code repetition.

70–79 Logic and presentation are explicitly separated, and some thought has been given to testability. Design patterns are used effectively.

80+ The architecture allows for new types of people, multiple lifts and different arrival simulation approaches to be added without disrupting the rest of the code. Cohesion and coupling are good across all classes.

6.5 Implementation

40–49 The code runs, and most of the expected functionality is available. The program may crash under some (but not all) normal configurations.

50–59 The program does not crash in normal circumstances. Code is well formatted and key classes are documented with Javadoc. The elevator operates as expected, ensuring requests are eventually honored.

60–69 The code follows Java naming conventions well, and makes good use of static/final and access control. All basic types of people work as expected. A text-based UI has been achieved.

70+ The code meets the conditions above, and also implements some of the advanced requirements:

* Up to +10 for smarter/multiple elevators.
* Up to +10 for playback mode.
* Up to +10 for rival developers and impatient clients.
* Up to +10 for a GUI.

You may not score more than 100 here: that means that you can achieve full marks by implementing only three of these four advanced requirements.

* 1. Testing

40–49 There are test cases for the lift. Most of the test cases pass, though a few corner cases may fail.

|  |  |
| --- | --- |
| 50–59 | All test cases pass. Test cases include the different types of people as well. |
| 60–69 | Test cases also cover invalid inputs. There is evidence of a methodical approach to testing. |
| 70–79 | Test cases cover corner cases (e.g. full elevators, rival devs). A test plan has been derived |

from the requirements and executed meaningfully.

80+ Test-driven development has been applied in a disciplined manner, and testability is explicitly considered in the design.

* 1. Evaluation

40–49 There is a report on the behaviour of the simulation, but it has not been done in a reproducible way: the results cannot be easily repeated. This usually means that you have not been careful in using a single Random instance throughout the program, or that you have failed to provide a way for the user to specify a seed (e.g. via program arguments).

50–59 The report has been done in a reproducible way, but the discussion is merely a statement of facts and does not offer any insights.

60–69 The report shows a well-executed study, with some insights offered on the generated data.

70–79 The insights show ways in which the elevator could be made better, or the building could be better designed.

80–89 Insights are truly illuminating, and draw from other similar simulations in the literature, outside this one.

# Milestones

Experience has shown that the most successful groups are those that work together in a structured way and follow a sensible lifecycle. To encourage this, I propose the following milestones.

Requirements Analysis: you need to analyse the requirements defined in this document to identify candidates and distribute responsibilities between them. You should start this after the introduction to PlantText on February 24th, and submit an early draft by the end of the day so it may be picked for early feedback on the tutorial the next day.

Design: you need to create UML class diagrams for your detailed design. You should start this within the February 24th lab as well, and discuss it in the coursework design tutorial on Tuesday 3rd March. The design should be evaluated by running some scenarios on paper.

Text-based implementation: I suggest that you start with the basic requirements first, then move up into the advanced requirements. As soon you have the UML class diagrams, you should do a first skeleton of your Java code and distribute the work across your team. You should be able to start coding by March 9th, so you can ask the lab staff for guidance. You should have a first implementation of the basic requirements before Easter.

Advanced requirements: you will most likely need a few weeks to do the basic requirements. If you have the basic requirements before Easter, you should be able to use the Easter break to polish them and do some of the advanced requirements (e.g. the GUI). You will probably want to wait until after the assessed GUI lab on March 24th to start on the GUI, but try to keep your simulation separate from its presentation so it is easy to add a GUI later on.

Testing: once you have gone through the testing and I/O lab on March 31st, you should start adding unit tests to your program. It will be best to combine black-box (specification-based) and whitebox (code-based) approaches to come up with tests, and use coverage measurement tools (e.g. EclEmma) to see if you’re missing any important parts of your code.

Study: your study on *p* and *q* combinations could be done in parallel with the testing, as it is quite likely that the intensive running of your simulation parameters may uncover some situations that you are not handling entirely well.

Again, try to have a first basic version fully working before going on the Easter break — hopefully, the break should be only used to polish the final submission! Avoid at all costs doing “big bang” integrations on the last week: that always ends badly!