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

Coursework Specification 2017-18 v1.2

Antonio García-Domínguez

Module CS1410 April 23, 2018

1 Change history

- V1.0: first version.
- V1.1: added note about timing of item pickup for robots arriving at packing stations.
- V1.2: clarify some points about the simple path-finding algorithm.

2 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.

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 use the discussion forums or book some office hours on WASS (the links are on Blackboard). I prefer using the forums over email, since everyone will be able to see my replies as well. You can also ask the Programming Support Officer for questions about Java itself.

3 Problem Description

The task is to develop a simplified simulation of a part of an automated warehouse, such as those used by Amazon[1].

[1] https://www.youtube.com/watch?v=6KRjuuEVEZs

The simulation is run in "ticks": each tick, all the actors will perform a bit of their behaviour. By running one tick after the other, we can see how the warehouse would operate given some parameters. The simulation starts with a predefined list of orders, which may be read from a file (more details in Section 4) or be randomly generated. The simulation continues until it fails or all orders are dispatched, reporting the number of ticks and the final result.

In these warehouses, the floor is divided into a grid of cells. Each cell may contain entities of various types, all with unique identifiers (UIDs):

- Idle **packing stations** (identified as P1, P2, ...) take up the next order from the list, and then ask the robots to bring items from certain storage shelves. Once a packing station has all the items, it takes a number of ticks packing it (which depends on the order), and then dispatches it for delivery.
- A **storage shelf** (identified as S1, S2, ...) has the items which can be ordered by customers. Robots take items from these shelves[2]. In this simulation, we do not care about the specific items themselves: orders simply list the UIDs of the shelves that we should take items from. Shelves are mostly passive markers for the robots to go to.
- A **charging pod** (identified as C1, C2, ...) charges the batteries of a robot C power units per tick. Each robot has its own charging pod: robots can go through other pods, but they only charge at their own pod.
- Once per tick, a **robot** (identified as R1, R2, ...) can move up/down/left/right once on the grid, or it can take items from the shelf it is currently at. They can move under storage shelves, charging pods and packing stations. If two robots end up in the same cell, they have crashed: after notifying the user, the simulation must be stopped.

  Robots are battery-powered. A robot starts with B power units in its battery: to move 1 space, the robot uses 1 power unit when not carrying anything, and 2 power units otherwise. Robots do not use power if they are not moving. If a robot runs out of battery power, this must be reported back to the user and the simulation must be stopped.

  When a robot reaches its designated charging pod, it continues charging until the battery is half full. Robots should also return to the charging pod if they detect that they will not be able to reach their next destination and return to the charging pod with their current battery level.

  When a robot reaches a packing station, as a simplification, we assume that it picks up all the desired items on the same tick. On the next tick, it will be able to move again.

  A packing station will ask a robot if it can "bring items from shelf X". The robot will decide if it wants to accept the assignment or not: this will depend on the current battery level and how far the shelf and the packing station are. If all robots reject the assignment, the packing station will retry on the next tick in case one of the robots becomes available.

  The exact way in which you move the robot and estimate distances is up to you. Here are some ideas that we have tried out, and how well they do:

  - The simplest strategy is to go towards the target column first, and then move vertically to the target row. This is easy to implement, but you will have robots running into each other if you are not careful.

    Even if you prevent collisions, with this strategy a robot going to the right will block a robot going to the left indefinitely. We only recommend this strategy for the simplest configurations (e.g. one shelf, one robot/pod, one station), when you are starting out.

[2] This is a simplification from the real robots, which take the entire shelves with them and later return them to where they were. In the real world, staff in the packing stations take the items from the shelves.

- A slightly better strategy is to check which of the four directions (up/left/down/right) brings you closer to the destination. To measure distance, you can use *Manhattan distance*, which is the sum of the absolute differences of the source/target columns and the source/target rows. Given two positions $a = (x_a, y_a)$ and $b = (x_b, y_b)$, the Manhattan distance $D(a, b)$ is:

$$D(a, b) = |x_a - x_b| + |y_a - y_b|$$

  As an example, the distance between $(1, 3)$ and $(2, 2)$ is $|1 - 2| + |3 - 2| = 1 + 1 = 2$. It is essentially the number of city blocks that you'd have to traverse in Manhattan (New York), depending on your position. (Hence the name.)

  Note 1: the Manhattan distance is very optimistic. It does not take into account that there could be obstacles in the way (e.g. other robots). When using Manhattan distance to figure out if you have enough battery power, it may be good to add a safety margin to your estimation (say, 20%).

  Note 2: you still have to check against collisions - don't try going into a direction that has a robot or goes outside the warehouse!

- If you want to take those into account, you can follow a dedicated path-finding algorithm. Videogames tend to use A*[3], but it might be difficult for you to implement at this stage. Section 7 describes a simpler algorithm which can be implemented in 70 lines or so.

You are free to make the system smarter in how it assigns missions to robots, or how the robots deal with them. Extra marks may be given in those cases. Be sure you explain your strategy and how you have tested it!

You should provide some representation of the warehouse floor and the main metrics of the simulation. A text representation can work, but to achieve full marks you will need a graphical representation in JavaFX.

There should be a way to watch the simulation step by step. This will help with debugging and it is needed to mark your submission.

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.

- The program should allow for configuring manually the various parameters of the simulation. You could do this through a textual interface, through a graphical interface, and/or by loading a text file. In fact, we have prepared a collection of sample configurations for you in a specific `.sim` file format. An example is shown on Listing 1.

  You are heavily recommended to reuse this format for your simulation. If you do so, you could opt into a fun competition with other groups on who has the smartest warehouse. We will give extra marks for your additional efforts, but do not worry if you are first or last — your final position is not part of the marking scheme. We will announce on Blackboard the top 3 teams, though!

  The exact format is as follows:

[3] https://en.wikipedia.org/wiki/A*_search_algorithm

Listing 1: Example of `.sim` file

```
1  format 1
2  width 3
3  height 3
4  capacity 20
5  chargeSpeed 1
6  podRobot c0 r0 2 0
7  shelf ss0 2 2
8  station ps0 0 2
9  order 9 ss0
10 order 15 ss0
11 order 7 ss0
```

1. The file always starts with "format 1". It is common to indicate a format version number for future compatibility. You can safely ignore this.
2. "width X" and "height X" indicate the width and height of the warehouse grid, in cells.
3. "capacity X" and "chargeSpeed X" indicate the capacity of the battery of all the robots and the number of power units that a charging pod recharges per tick, respectively.
4. "podRobot IDPOD IDROBOT X Y" place a paired set of a charging pod and a robot with the specified UIDs at column X, row Y.
5. "shelf ID X Y" and "station ID X Y" do the same, but for a single storage shelf and a single packing station, respectively.
6. "order PT IDSHELF+" place an order that requires PT ticks to pack, and that needs items from the shelves with the listed UIDs.

- It may be useful to have all the entities of a certain type tick, then the ones from the next type, and so on. Particularly, we recommend having all packing stations tick, then the charging pods, the shelves, and finally the robots. This way, the code in your robots can assume that all packing stations have done what they had to for this tick.

- You may want to divide your orders into unassigned, assigned and dispatched orders. You can do this by adding fields to your orders, or by dividing them into three separate lists in the simulation. The simulation should finish when all orders have been dispatched.

- You must develop all the classes in your team, with the exception of the Java standard library classes (e.g. collections, JavaFX, math routines).

- 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.

- You may want to separate two things from the Robot class into their own subclasses: the PathFindingStrategy that decides how to move, and the CostEstimationStrategy that estimates how much it will cost to follow a certain route. If you do so, you will be able to swap between different strategies cleanly, without disturbing the rest of the code.

  For instance, you could define PathFindingStrategy as an interface with one method, and then have different implementations. A Robot could then contain a strategy and delegate on it for deciding in which direction it should go.

This might also be helpful for testing: you would be able to test various cost estimation / pathfinding strategies separately from the rest of the logic of the ROBOT.

- 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:
  - File input and output (if you want to read the `.sim` files).
  - Most of the entities within the simulation.
  - The ROBOT entity.
  - Path-finding / cost estimation routines.
  - (Graphical and/or textual) user interface for configuration.
  - (Graphical and/or textual) user interface for visualisation of the running simulation.
- Don't jump into coding straight away if you are not experienced with object-oriented programming. Analyze the problem with the noun/verb method, 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 of different classes.
- Read each other's code, and test the classes of other team members!
- You must track for how long the simulation ran before the first crash or power loss, and how long did it take to dispatch each order. This information should be printed to the screen and/or a file.

5 Deliverables

The coursework will be divided into two submissions:

- The first submission is due on **Friday 23rd March**, before Easter. 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 (e.g. draw.io), 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 Java classes based on the UML diagram, with enough code to contain the state of the simulation, but not necessarily its behaviour. The main() method should be able to set up a warehouse grid and show it on screen (whether through a textual or a graphical user interface).

  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.

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 a representation of the grid 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 (before Easter)

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. `.sim` files can be read, and simulations succeed for all given samples.	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.	The simulation is engaging to watch, and there are no gaps in the Javadoc documentation. There is no unnecessary code repetition or undesired coupling.
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 AI and file input and output. 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 last submission (after Easter)

- The second submission is due on **Friday 27th April**. 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. For instance, how do your robots react to configurations that have more overlap between their usual paths? How much of a safety margin do you need for the batteries in your robots?
  - 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:
    - Does the system reject invalid configurations?
    - Are collisions and power failures detected and reported?
    - Are robots moving as expected?
    - Are orders coming in as expected?
    - Do charging stations power up the robots correctly?
    - Are packing stations dispatching deliveries when they are ready?
    - ... and so on. Think of other things that ought to work.

    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.

    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.

6 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 generate an overall system architecture (e.g. using the noun-verb method). You should start this in week 6 (Tuesday 27th February).

Design: you need to create a UML class diagram for your detailed design. You should start this in week 7 (Tuesday 6th March) at the latest and be prepared to discuss your design in the lab on Monday 13th March. The design should be evaluated by running some scenarios. This can be done in the lab.

Unit Tests: you can write some of your unit tests before implementing the body of the code. This can start once the design is agreed, preferably in the lab on Monday 12th March. You can also write all the class skeletons during weeks 8 and 9.

GUI Design: you can sketch your GUI design as a "wireframe" at the tutorial in week 8 (Tuesday 14th March).

You *must* have a solid and detailed design specification to share amongst the team by the end of week 8 (Friday 16th March) so that the system can be developed over the Easter break with some hope of integrating the software successfully on your return. Avoid at all costs doing "big bang" integrations on the last week: that always ends badly!

7 Simple path-finding algorithm

As an example, suppose we have this map, where X is an obstacle, O is an empty space, S is the starting point and D is our destination:

	i_1	i_2	i_3
j_1	S	O	O
j_2	O	X	D
j_3	O	O	O

Looking only at the Manhattan distance, it doesn't matter if we start going to the right or going down: it is 2 between (i_2, j_1) (column 2, row 1) and (i_3, j_2), and it is also 2 between (i_1, j_2) and (i_3, j_2). However, you can clearly tell that it takes 3 steps if you go to the right first, and 5 steps if you start by going down.

One way to do this is by creating a table of your own with an estimation of how many steps it takes to reach the destination from each cell. Like this:

	i_1	i_2	i_3
j_1	3	2	1
j_2	4	N/A	0
j_3	3	2	1

You will notice that the destination has 0, whereas our starting position (i_1, j_1) has 3. Once we have this table, we can just look at the adjacent cell that has the lowest value, and go in that direction. Out of all adjacent cells to (i_1, j_1), the one to the right, (i_2, j_1), has the lowest value: for this tick, we will go right.

If you want to code this strategy, you'll have to follow these steps:

1. Create an empty queue of positions (e.g. a LinkedList<Position>).
2. Create an empty map from the positions to the shortest distance from the destination seen so far (e.g. a HashMap<Position, Integer>). This map will act as our table.
3. Add the destination to the end of the queue.
4. Set the table cell of the destination to 0.
5. While the queue is not empty:
   (a) Retrieve and remove the first position p_f from the queue. This is very efficient, thanks to our use of a LinkedList.
   (b) Retrieve the table cell c_f for that position.
   (c) For every orthogonally adjacent position p_a, if you can move into it (i.e. not an obstacle):
      i. Retrieve the table cell c_a for p_a, if it has been set. If it has not been set, set c_a to a very large number (e.g. Integer.MAX_VALUE).
      ii. If $c_f + 1 < c_a$, then:
         A. Set the counter for p_a to $c_f + 1$.
         B. Add p_a to the end of the queue.
6. You should have the table now. Go in the direction of the adjacent position that has the smallest table cell.

Note that in order for you to use a HashMap from Positions to Integers, you will need to have implemented hashCode and equals appropriately. Check your slides, and check the "Source" context menu of the Java code editor in Eclipse: it can generate those methods for you. You can invoke it by pressing Alt+Shift+S from the editor, or right-clicking on it.