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

# Coursework Specification 2018-19 v1.3

**Antonio García-Domínguez**

Module CS1410 March 11, 2019

## 1 Change history

- V1.0: first version.
- V1.1: fixed date for design tutorial (it is on April 2nd).
- V1.2: board format can be reused in TUI, specify what to show in TUI if there is both a robot and something else at the same position on the board.
- V1.3: laser firing phase happens after gears and belts have activated. Flags and pits do not matter, as their behaviour happens when the robot enters their positions on the board.

## 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. 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 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: Basic Requirements

The task is to implement a grid-based board game, where you race with your robot to touch all the flags on the board in order (first player to do that wins). It is essentially a heavily simplified version of *Robo Rally*, from Wizards of the Coast: look it up if it sounds interesting!

The robots start on predetermined positions on the board facing north, and a player gets the first player marker. Each round consists of these steps:

1. The players program their robots with a sequence of 5 different actions. These can be "move Forward one space" (F), "Back up one space" (B), "rotate 90 degrees Left" (L), "rotate 90 degrees Right" (R), "do a U-turn" (U), or "Wait" (W).

   You may not repeat an action twice in a row: "FFLWF" is not valid, but "FWFLF" would be valid. It would be boring and unfair if everyone simply played "FFFFF" all the time!

2. Robots will operate in player order, executing the action in the first slot, then the locations where the robots are will activate, and the first player token pases to the next player. This is repeated with the second, third, fourth and fifth slots.

   As an example, with player A and player B having decided on "FLFWF" and "WFWFL", and with A starting with the first player token, this will happen:

   (a) A "F" is executed, then B "W". Board activates. First player token passes to B.
   (b) B "F" is executed, then A "L". Board activates. First player token passes to A.
   (c) A "F" is executed, then B "W". Board activates. First player token passes to B.
   (d) B "F" is executed, then A "W". Board activates. First player token passes to A.
   (e) A "F" is executed, then B "F". Board activates. First player token passes to B.

There are some additional rules for movement:

- If a robot moves into a position occupied by another robot, it pushes that other robot one space away in the same direction. The robot being pushed may push another robot, and so on.
- If a robot moves outside the board, it is destroyed. Move the robot back to its starting position. If the starting position is occupied, try the position adjacent to the north, then east, south, and finally west.

There are two types of locations that react to robots entering them:

- **Flags** will notify the game that a robot entered it. The game will track if that was the next flag in the sequence for the robot, or not.
- **Pits** will destroy your robot. Move it back to its starting position, just as if it had moved outside the board.

There is one type of location that manipulates robots upon activation, after all robot moves in an action slot have been resolved: **gears**. They can rotate your robot either clockwise or counter-clockwise by 90 degrees.

To activate the board, go through the robots from the top-left corner to the bottom-right, row by row. For each robot, find any special locations where they are and activate them.

A basic version of the game should read the board and the programs from a file. The specific formats for these files are described in Section 5. The basic version of the application would use a text-based user interface (TUI).

The basic requirements allow you to go up to the 60-69 band in the implementation component of the final submission, according to the marking scheme on Table 2. Section 4 describes a set of advanced requirements which will give you more marks in this component, which includes the creation of a JavaFX-based graphical user interface.

Remember that whether you use a TUI or a GUI, you should show the evolution of the board over the game. This should be done at least once per round, but you could do it between action slots or even individual actions.

# 4 Problem Description: Advanced Requirements

There are a number of advanced requirements that you could meet to achieve higher marks:

- You can achieve up to 10 extra marks in the implementation component if the game supports both reading programs from files, and reading them directly from the players.

  For a TUI, you will need to add an appropriate way to pick the game mode (whether through a menu or through a command-line switch). For a GUI, you will need to add the UI to enter the actions. In both cases, you will need to tweak the game to run in a stepwise manner, so players may see the consequences of each action.

- You can achieve up to 10 extra marks in the implementation component if the game has a graphical user interface in JavaFX. The GUI should expose all the available functionality in the game. If you do a GUI, you do not need to develop a TUI, but it may be easier/safer to start with a TUI and refactor the code later.

- You can achieve up to 10 extra marks in the implementation component if the **game supports conveyor belts**. Conveyor belts will move your robot one space towards a certain direction, unless there is already a robot there. Conveyor belts in corners will rotate your robot 90 degrees clockwise or counter-clockwise as well, if the robot is carried onto them by a conveyor belt (and *not* if pushed by a robot).

  There is a number of sample boards for this: have a look at them and let us know what you think!

- You can achieve another 10 extra marks in the implementation component if the **game allows robots to be damaged**, regardless of whether you use a TUI or a GUI. Robots will start with 5 health, and they will be equipped with lasers.

  The game will be extended with a laser firing phase, after the gears and conveyor belts (if you have them) have been activated:

**Listing 1:** Example `.brd` board file

```
format 1
.....+......
.1.s<<w.E>>S
...v.2^.^..v
...e>>n.N<<W
...-..[..].(
.x....x.....
.....CABD..)
```

**Listing 2:** Example `.prg` program file

```
format 1
Alice Bob
FLFWF WFWFL
RFWFL FRWRF
```

1. First, robots will fire their lasers in player order: the first robot standing on the line of sight of the firing robot will receive one point of damage.
2. Second, if a robot is standing in the middle of a board laser beam, they will also take 1 point of damage. If two or more robots are standing in the way of a board laser beam, only the robot closest to the emitter is affected: it is stopping the beam before it reaches the others.

Robots with $n$ health can only do their $n$ first actions. If a robot's health reaches zero, it is destroyed and it returns to its starting point (or a free adjacent position in NESW order, if not available) with full health and facing north.

For this one, you will have to come up with your own sample boards. Make sure to write about how you designed those sample boards in your design report!

## 5 Design Notes

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

- The program should allow for setting up the board, the number of players, the names of the players and the programs given by the players to their robots. You could do this through a textual interface, through a graphical interface, and/or by loading text files. In fact, we have prepared a collection of sample `.brd` board files and `.prg` program files. Examples are shown on Listings 1 and 2.

  You are heavily recommended to reuse these formats for your coursework. This would allow you to share boards between groups and check against each other if you have implemented the game behaviour correctly.

  For board files, we are going to follow a simple text-based format. The file will start with a line declaring the format version (a common practice for future-proofing the format), which we can ignore. After that, we will have a sequence of lines of characters: each character is a cell in the grid, and all lines must have the same number of characters. The available characters are as follows:

  - A period (`.`) represents a position with no locations (it is empty).
  - `A` to `D` are the starting positions of the maximum 4 players we can have in the game.
  - `+` and `-` are gears (`+` rotates clockwise, `-` rotates counter-clockwise).
  - `1` to `4` are the maximum 4 flags you can have, to be visited in this order.
  - `x` is a pit, which destroys your robot and returns it to its starting position.

  - `v`, `>`, `<` and `^` are straight conveyor belts which upon activation, move your robot one space down, right, left or up, respectively.
  - (Optional: if doing conveyor belts) `N`, `E`, `S` and `W` are the clockwise corner conveyor belts, and `n`, `e`, `s` and `w` are the counter-clockwise corner conveyor belts. The clockwise conveyor belts will turn your robot to the right *if and only if* it is carried onto that location by another conveyor belt. Likewise, the counter-clockwise conveyor belts will turn the robot to the left on that situation.

    Upon activation, `N`/`n` will move the robot north, `S`/`s` south, `W`/`w` west, and `E`/`e` east.

    In the example above, the conveyor would have the robot go on a counter-clockwise loop on the left, and a clockwise loop on the right. You would need a carefully timed entry and jumping off the loop on the left to reach the second flag!
  - (Optional: if doing lasers) Lasers will span from an emitter "[" to a receiver "]" horizontally (left to right), or from an emitter "(" to a receiver ")" vertically (top to bottom).

  You can reuse the text-based format from above if you want to show the board in a text-based user interface. If there is a robot and something else at the same position on the board (e.g. a gear), show only the robot as "A", "B", "C" or "D", depending on the player.

  The format for the program file is much simpler. You have the same format version declaration line, a line with the player names separated by spaces, and then a sequence of lines with the programs from the different players in each round. Each line is formed of groups of 5 characters, separated by single spaces. The first group will be the program of player A, then player B, and so on. This means that the order of the groups is based on the identities of the players, not their turn order in that round!

- 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 will want to create a Grid class, and have a base GridEntity type (which may be a class or just an interface) that represents anything that can go on the grid and can be asked to act() at a certain point. The various location types and the robots themselves should be based on this common type.

- You may want to have the actions that can be picked by the players as their own objects, in their own hierarchy. This would make it easier to support new player actions in the future without having to modify the rest of the codebase.

- You may want to support a single-player mode to make it easier to test the basic board logic. Do not forget to test the main multiplayer mode, though!

- You should keep the core of the game separate from the actual UI. For a TUI, have different classes read the grid and render it into text. For a GUI, have the graphical interface redraw the grid from action to action, 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.

- Beyond the provided sample boards and programs, you should try creating your own small boards and programs that demonstrate that the game works as it should. Same goes for robots pushing each other in various directions. Do not forget to test when the robot falls off the board — it should be destroyed and go back to its starting position (or a free adjacent position in NESW order).

- 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 for each type of file.
  - The different types of locations you can have on the grid.
  - The ROBOT entity and the management of the global game state (e.g. flags touched).
  - Any extra requirements (lasers, conveyor belts).
  - (Graphical and/or textual) user interface for configuration.
  - (Graphical and/or textual) user interface for visualisation of the running game.
- 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.
- 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.
- Please avoid the "hero coder" figure, when one person is doing 90% of all the coding "because he/she is much better at it than us". This imposes a large burden on that hero which may backfire spectacularly, makes everyone else disengage from the process and it curbs the growth of the rest of the team. It is a very fragile situation: you do not want your project to depend on a single person's wellbeing and motivation!
- You must print each of the resulting board states, and the final board state. This information should be printed to the screen and/or a file.

# 6 Deliverables

The coursework will be divided into two submissions:

- The first submission is due on **Friday 5th April**, 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). You do not need to worry about the TUI/GUI classes, only the domain of the problem itself. For instance, a GAME class aggregating the state of the game makes sense at this analysis stage, but not a MAINWINDOW class nor a TEXTMENU class.
  - A PDF with UML class diagrams of the intended domain 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). Again, do not bother with the TUI/GUI classes at this early stage either.
    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.

| 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 several alternatives. |
| Code skeleton (20%) | The code runs and shows a representation of the board 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 board 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)

  - 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 game, but not necessarily its behaviour. The main() method should be able to set up a board and show it on screen, whether through a TUI or a GUI.

  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 **Friday 3rd May**. 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? Which approach(es) have you followed during testing? Using a test coverage tool (e.g. EclEmma, in the Eclipse Marketplace) to ensure you have not missed anything obvious would be a good idea.
  - 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.
  - 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 out of bounds robots detected and handled correctly?
    * Are robots moving as expected?
    * Are gears rotating robots as expected?
    * Is the winning condition detected correctly?
    * Are conveyor belts rotating and carrying robots as expected?
    * Are lasers firing correctly, and are beams being blocked as expected?
    * ... 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. Using a test coverage tool to find tests 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!

  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.

## 7 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.

| Component | 40–49 | 50–59 | 60–69 | 70–79 | 80+ |
|---|---|---|---|---|---|
| Design (20%) | An object-oriented approach has been used, with different classes representing the robots and the various types of locations. 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 without dependency cycles. Polymorphism and composition have been used effectively to avoid code repetition. | Logic and presentation are explicitly separated, and some thought has been given to testability. Design patterns are used effectively. | The architecture allows for new actions, location types and rules to be added without disrupting the rest of the code. 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 with Javadoc. | The code follows Java naming conventions well, and makes good use of static/final and access control. Board and program files can be read and all basic location types (flags, pits, gears) work as intended. A batch-style TUI has been achieved. | Up to +10 for interactive hot-seat play, in addition to file input.<br>Up to +10 for a JavaFX-based GUI that exposes all available functionality.<br>Up to +10 for conveyor belts.<br>Up to +10 for damage model (robots have health, board lasers and robot lasers are implemented). | |
| Testing (20%) | There are test cases for each location type. 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 location type, and the winning conditions. | Test cases also cover invalid inputs and failure scenarios. There is evidence of a methodical approach to testing. | Test cases cover 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. |

**Table 2:** Marking scheme for the last submission (after Easter)

**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 after the tutorial on week 7 (Tuesday March 5th) and be ready to discuss it on the week 8 coursework lab (Monday March 11th) and the week 9 coursework analysis tutorial (Tuesday 19th).

**Design:** you need to create a UML class diagram for your detailed design. You should start this as soon as the analysis is ready, and be prepared to discuss your design in the coursework design tutorial on week 10 (Tuesday 26th March). The design should be evaluated by running some scenarios on paper.

**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 after the week 10 tutorial. You can also write all the class skeletons during weeks 10 and 11.

**GUI Design:** some of your team members could sketch your GUI design as a "wireframe" during the coursework design tutorial in week 11 (Tuesday 2nd April) — feel free to show it to us then!

You *must* have a solid and detailed design specification to share amongst the team by the end of week 11 (April 2nd 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!