RoboRally

Agile Object-Oriented Software Development 02160

Rebecca Elmquist Nielsen (s204331), Mejse Grønborg-Koch (s196050), Irene Campillo Pereda (s222704),

Mathieu Georges Kaj Jensen (s215238), Bernat Godayol Farran (s212717)

1 INTRODUCTION

The goal of the project is to develop a small video game based on the popular board game *RoboRally*. The board game consists in programming a robot to move around a factory filled with obstacles. In each round, the robot can only perform certain actions based on a deck of cards, avoiding obstacles and other robots. The goal is to program the robot so that it reaches the end point of the game board.

This is our GitHub repository: RoboRally-Group-4.

2 SOFTWARE DEVELOPMENT

Agile practices have been the core of the development of *RoboRally* and more particularly, we have been following the **Behaviour Driven Development (BDD)**. This methodology focuses on continuously defining and testing the expected outcome of our software. This approach has enabled us to easily understand and define the functionaof *RoboRally*, increase efficiency and productivity, and rapidly detect the flaws and bugs in our code.

2.1 User Stories

The first step in our development process consisted in defining the **User Stories** we would like to fulfil. Each user story has a role, a goal and a reason. The user stories constitute a key element in the agile development process as they let us comprehend which valuable features we desired to implement and how those will benefit the users of our game.

To carry it out, we first brainstormed a collection of user stories and grouped them based on the phase of the game they were linked to. More concretely, we divided the user stories into three **epics**: *Game Set-Up*, *Programming Phase* and *Activation Phase*. Each epic represents a separate stage in the game and provides the necessary actions and functionality that a player needs in order to play the game successfully.

Moreover, we ensured that the user stories were complying with the **INVEST** quality criteria and we discarded or modified those that did not adhere to it. Finally, we prioritized the user stories utilising the **MoSCoW** method, so we selected a subset of user stories that would be essential for developing a **Minimal Viable Product (MVP)** of the game.

2.2 Testing Framework

Once we defined the user stories, the development part of the code started. We began implementing the model classes of user stories that were essential for the MVP. To accomplish this, we employed the **Cucumber stack** and the **JUnit** frameworks to enable testing and behaviour-driven development. These frameworks help to maintain workable code when extensions/modifications are made and refactoring done.

We utilize our User stories to write features and scenarios using **Gherkin** syntax. These features were mapped to actual code via the *StepsDefinition.java* file and tested using the JUnit framework. We wrote scenarios that covered all the possible situations that could be encountered while playing the game. For example: When we were designing how a robot should move forward on a board, we created scenarios for every possible movement that could originate from any given starting position and direction on the game board. At the end of our project, we could cover up to 94% of the model code.

If a Cucumber scenario fails during a test run, JUnit generates a detailed report that provides information on the exact location and cause of the issue. In addition, for managing dependencies, building automation and structuring the project we operated with **Maven**.

Finally, throughout the whole stage of development, we used **Git** as a Version Control System. Every time we had to implement a new feature, or scenario or perform modifications, we generated new branches to work independently and make our changes and avoid conflicts with the *main* branch until the change was ready. We strove to merge and push changes to the main branch as often as possible, always having a working version of the game in the *main* branch.

Our use of the above mentioned practices helped ensure product readiness and continuous integration.

2.3 GUI

For the **Graphic User Interface (GUI)** we chose to use **JavaFX** since this was recommended in the course. The programming of the GUI has been done entirely from scratch and we chose not to use *SceneBuilder* because it proved easier to modify the GUI to our needs when hard coding.

We have focused on making a minimalistic GUI for the player, so the player understands the different phases of the game. To make the game more intuitive we decided to highlight the cards when they are clicked on and chosen by drawing borders around the cards. This helps the user to keep track of chosen cards but also to understand that only a limited number of cards can be chosen. We have also implemented a button that appears during the *activation phase* and disappears in the beginning of the *programming phase*. The button allows the players to see all of the moves their robot is making, so the players understands how the robots move on the board. All board tiles, robots and cards are drawn by the group with the help of the website Piskel. Lastly, we also implemented sounds in the game as we explain in detail in section 4.

2.4 Experiences and Challenges Faced

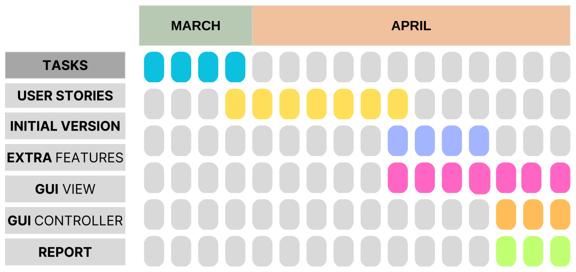

Figure 1: Gantt Chart. The boxes/dots correspond to a fraction of time relative to the whole period of time spend working on the project.

The experience of working in a behaviour-driven development and **Agile** methodology was overall quite positive. It was interesting to see how the early version of our software was continuously growing with the new implementations we added. In addition, we believe that the Agile methodology made us frequently engage in discussions about how to distribute the workload, implement new features, refactor existing code, and so on. In other words, we strongly believe that this methodology improves communication within a team.

The use of natural language within the Cucumber framework made it easy to comprehend the new features implemented at each phase. Furthermore, JUnit was quite practical for recognising and handling the issues and bugs in our code. We could monitor how each piece of new code was affecting the existing one and easily detect violations of the SOLID principles (see section 3).

On the other hand, we also faced difficulties and challenges throughout the process of developing *RoboRally*. First, it was particularly complex to understand the implementation of Git in Eclipse and work with it effectively.

Another difficulty was becoming familiar and working dynamically with the Behaviour Driven Development methodologies. It took us some time to be prepared to write code dynamically following these new techniques.

Our code has been sectioned into three packages: *controller*, *model* and *view* (see section 3.3, about model-view-controller pattern). The JUnit testing was only done for the Model code which was very helpful when generating code. It was a challenge when generating code for the View and Controller, for which we did not have the same testing framework.

2.5 Work process

Figure 1 displays a Gantt Chart showing our overall work process. As mentioned earlier (see **User Stories** in section 2) we initially created User Stories and then we moved on to creating the a MVP of the game through JUnit testing. Around the time a MVP was almost finished the work on the GUI and the extra functionalities began. For the GUI we started working on the View first and then we build the controller last.

Initially everyone worked together in one group to generate the User Stories. We also worked together when completing the first couple of scenarios in the JUnit testing. We then split up and utilized pair programming when working on implementing the next couple of User Stories. When we felt confident in the used practices we switched to working independently while still checking in and utilizing each other when needed. Pair programming was still used a lot through out the whole duration of the project - especially when working with the GUI and when implementing the more advanced rules.

3 SOFTWARE ARCHITECTURE

Our Java implementation of the RoboRally board game follows an object-oriented design approach. To ensure a well-structured software design, we utilized UML diagrams to illustrate the architecture and relationships of the various components in our *RoboRally* game implementation. This enabled us to *model* and visualize the interactions between different objects and classes, as well as define the responsibilities and dependencies of each component in an easy-to-understand manner.

In Figure 2 an overview of the UML diagram of the *model* code can be seen. It shows the attributes and methods of each class in the *model* as well as the connections between classes, such as implementations of interfaces, inheritance, and associations. Each class has a responsibility which can be seen in Table 1. The table gives an overview of the responsibilities of the main classes in the *model*.

3.1 Object-oriented programming (OOP) approach

To enhance code reusability and maintainability, we applied various OOP techniques, including encapsulation, inheritance, polymorphism, and data hiding. These techniques are reflected in our software architecture as follows:

- **Encapsulation.** We treated each class as a single entity by encapsulating the data attributes and execution logic within the class itself. For example, the *Player* class encapsulates data attributes such as *name*, *robot*, and *programmingDeck* within the class by setting all the members of the class as *private*, and implementing *set*ter and *get*ter methods to set and retrieve data.
- **Inheritance.** For instance, we extended the *Robot* class, by creating the *AI* subclass to cater to the specific type of player AI, overriding the constructor and adding a method *pickCards()* for the AI to choose random cards. Inheritance can also be useful for future versions of the game; for example, if we create different types of robots, such as flying or swimming robots, they will inherit the basic properties and functionalities of the *Robot* class.
- **Polymorphism.** We used polymorphism when we overloaded the method *setRobots()* in the *Board* class and when we defined the *execute()* method for each implementation of the *Card* interface by overriding it every time.

We considered the **data hiding principle** as well. For instance, specific movements or actions of cards are hidden from the outside and can only be accessed and executed via specific class methods. This design allows greater flexibility and modularity in the game's overall architecture. By using these techniques, we ensured that our software design was scalable and maintainable.

3.2 SOLID Design Principles

When designing our software architecture, we have kept in mind the SOLID Design Principles, in order to create a code that remains flexible, robust, and reusable over time and that could be easily modified and extended without affecting other parts.

- The **Single Responsibility Principle (SRP)** was applied when ensuring that each class has a single responsibility. We refer to Table 1 for a sum up of the responsibilities of the classes.
- The **Open Closed Principle (OCP)** presented some challenges, especially with the *MoveForward* card and the obstacles on the board. We aimed to make sure that the *execution()* method for the *MoveForward* card could be extended to handle new types of obstacles without needing to modify the code of the card. To achieve this, we designed the *Board* class to store a matrix of *Tile* objects, with each *Tile* having a *WalkableElement* and 4 *BarrierElement* attributes. Then we implemented methods in the tile class to check if a tile was exitable and accessible. In our game the *Wall* class extends the *BarrierElement* class, while the *Pit* and *Laser* classes implement the *WalkableElement* interface. Hence, this architecture allows us to add new types of barriers or walkable elements by implementing new classes that conform to the appropriate interface. In addition, the *Card* class is an interface with a required *execute()* method that takes a robot and the board as inputs and makes the robot perform an action. This design allows us to define new types of cards that can be executed in the game without modifying the existing code.
- The **Liskov Substitution Principle (LSP)** ensures that any instance of a class should be substitutable by an instance of its sub-class without affecting the correctness of the program. In our project, all the sub-classes of the *Card* interface can be used interchangeably wherever a Card object is expected, without breaking the program.
- The **Interface Segregation Principle (ISP)** ensures that no client should be forced to depend on methods it does not use. In our project, we have followed this principle by creating separate interfaces for the different functionalities required by our classes. For example, we have created the *WalkableElement* interface to have a method *execute()* required by the classes that implement it. However, it may seem like we have violated the ISP by not providing any instructions to the *execute* method in the interface implemented by the *Damage* card. But this is not the case, as the *Damage* serves as a "non-movement" card movement, which does not involve any action. Thus, the implementation is consistent with the principles of ISP. Also in the future it is possible to make different subclasses of *Damage* card with different actions overriding the *execute()* method.
- Finally, the **Dependency Inversion Principle (DIP)** states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In this way, we ensure that the high-level modules (such as the *Board* and *Deck* classes) depend on abstractions (such as the interfaces *WalkableElement* and *Card*), rather than on the low-level implementations (such as the *MoveForward* and *Wall* classes).

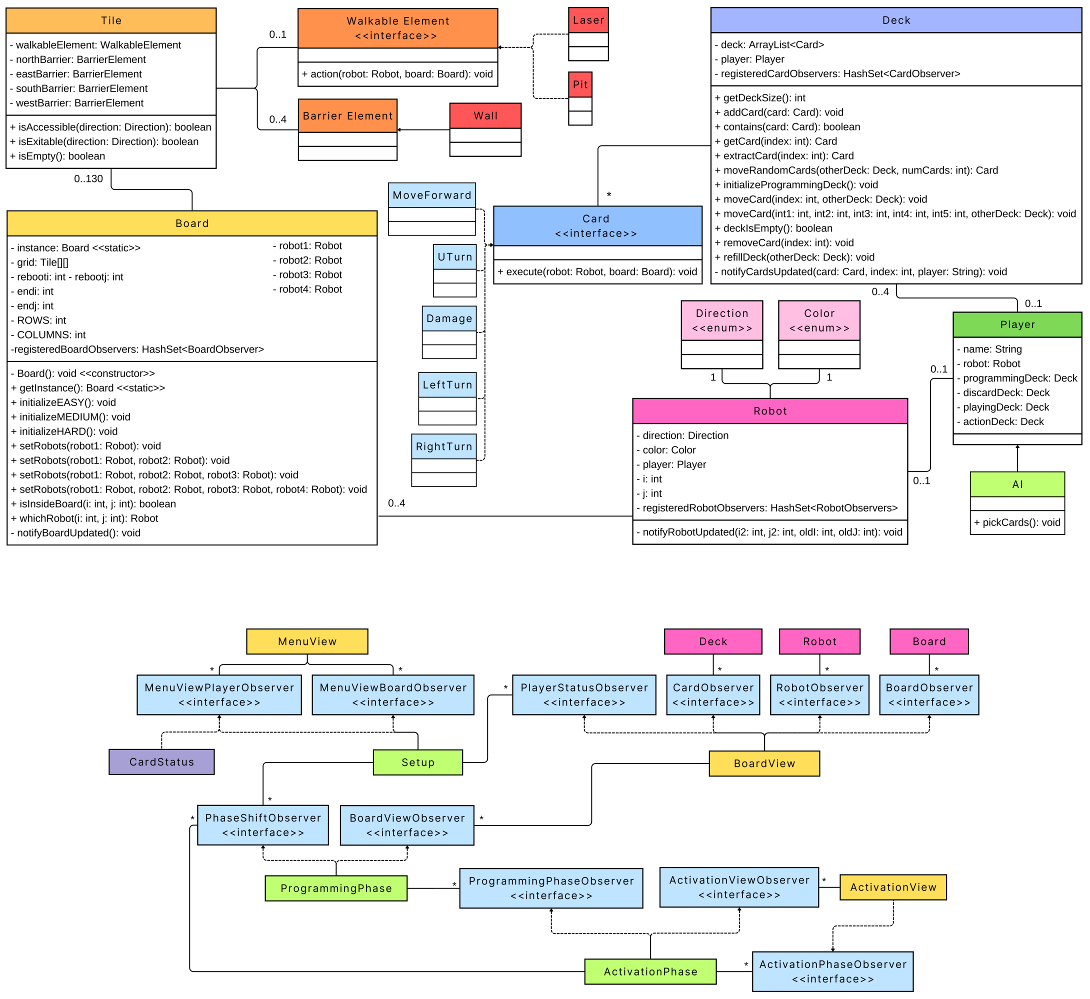
Tile
- walkableElement: WalkableElement
- northBarrier: BarrierElement
- eastBarrier: BarrierElement
- southBarrier: BarrierElement
- westBarrier: BarrierElement
+ isAccessible(direction: Direction): boolean
+ isExitable(direction: Direction): boolean
+ isEmpty(): boolean
0..1
0..4
0..130
Walkable Element <<interface>>
+ action(robot: Robot, board: Board): void
Laser
Pit
Barrier Element
Wall
Board
- instance: Board <<static>>
- grid: Tile[][]
- rebooti: int - rebootj: int
- endi: int
- endj: int
- ROWS: int
- COLUMNS: int
-registeredBoardObservers: HashSet<BoardObserver>
- robot1: Robot
- robot2: Robot
- robot3: Robot
- robot4: Robot
- Board(): void <<constructor>>
+ getInstance(): Board <<static>>
+ initializeEASY(): void
+ initializeMEDIUM(): void
+ initializeHARD(): void
+ setRobots(robot1: Robot): void
+ setRobots(robot1: Robot, robot2: Robot): void
+ setRobots(robot1: Robot, robot2: Robot, robot3: Robot): void
+ setRobots(robot1: Robot, robot2: Robot, robot3: Robot, robot4: Robot): void
+ isInsideBoard(i: int, j: int): boolean
+ whichRobot(i: int, j: int): Robot
- notifyBoardUpdated(): void
MoveForward
UTurn
Damage
LeftTurn
RightTurn
Card <<interface>>
+ execute(robot: Robot, board: Board): void
*
Deck
- deck: ArrayList<Card>
- player: Player
- registeredCardObservers: HashSet<CardObserver>
+ getDeckSize(): int
+ addCard(card: Card): void
+ contains(card: Card): boolean
+ getCard(index: int): Card
+ extractCard(index: int): Card
+ moveRandomCards(otherDeck: Deck, numCards: int): Card
+ initializeProgrammingDeck(): void
+ moveCard(index: int, otherDeck: Deck): void
+ moveCard(int1: int, int2: int, int3: int, int4: int, int5: int, otherDeck: Deck): void
+ deckIsEmpty(): boolean
+ removeCard(index: int): void
+ refillDeck(otherDeck: Deck): void
- notifyCardsUpdated(card: Card, index: int, player: String): void
0..4
0..1
Direction <<enum>>
Color <<enum>>
1
1
Robot
- direction: Direction
- color: Color
- player: Player
- i: int
- j: int
- registeredRobotObservers: HashSet<RobotObservers>
- notifyRobotUpdated(i2: int, j2: int, oldI: int, oldJ: int): void
0..4
0..1
0..1
Player
- name: String
- robot: Robot
- programmingDeck: Deck
- discardDeck: Deck
- playingDeck: Deck
- actionDeck: Deck
AI
+ pickCards(): void
MenuView
MenuViewPlayerObserver <<interface>>
MenuViewBoardObserver <<interface>>
CardStatus
Setup
Deck
Robot
Board
PlayerStatusObserver <<interface>>
CardObserver <<interface>>
RobotObserver <<interface>>
BoardObserver <<interface>>
BoardView
PhaseShiftObserver <<interface>>
BoardViewObserver <<interface>>
ProgrammingPhase
ProgrammingPhaseObserver <<interface>>
ActivationViewObserver <<interface>>
ActivationView
ActivationPhase
ActivationPhaseObserver <<interface>>

Figure 2: UML Class Diagram of *model* (top) and the Observer pattern (bottom).

By keeping in mind the SOLID principles in our software architecture, we have created a flexible and extensible design that can be easily modified and extended. Indeed, our software architecture for the *RoboRally* board game provides a solid foundation for future development and expansion.

3.3 Design Patterns

We have employed several design patterns to enhance the functionality and organization of our *RoboRally* game. Here some of the design patterns we used will be described and it will be explained how they contribute to the overall quality of our code.

- All of the classes in our project are separated into three different packages (*model*,*view*, and *controller*) according to the **Model-View-Controller Pattern**. We struggled with completely separating the *view* from the *controller* since the *View* defines what happens when the user interacts with a widget. If we had had more time, we would have liked to create a *Handler* class implementing the *EventHandler* interface in the *controller*.
- We implemented the **Observer Pattern** to enable efficient communication between the *model*, the *view* and the *controller*. For the implementation architecture we have used the push strategy, where the subject notifies its observers when changing. In our game, the *model* represents the data and logic, while the *view* and the *controller* is the Graphic User Interface (GUI). As an observer of the *model*, the *view* receives a notification from the *controller*

Class/Interface	Responsibilities
Robot	Represent a robot in the game, contain information about the robot's colour, direction, position, and the player to which it belongs. Implement the observer pattern to notify registered RobotObserver objects when the robot's position or direction is updated. Override the equals method to compare two Robot objects based on their colour.
Color	Define the four possible colours of the robots in the game: green, yellow, red, and blue.
Direction	Define the four directions that the robot can face in the game: north, east, south, and west.
Player	Represent a player in the game and manage the player's information and actions. Manage the player's name and robot, as well as their programming, discard, playing, and action decks. Ensure that these decks are associated with the player and provide methods for getting and setting them.
AI	Extends the class Player. Represent an AI player. Picks 5 cards from the programming deck at random.
Board	Arrange the game board layout, gives access to individual tiles on the board through their row and column indices, and keep track of the reboot and endpoint positions. Maintain a set of BoardObserver objects that can be registered using the registerBoardObserver() method. Notify registered observers of any updates to the board. Keep track of the robots on the board, with four robot objects (robot1, robot2, robot3, robot4) present on the board and set them to an initial position and with a specific direction.
Tile	Contain elements or not (empty) and check whether a tile is excitable and accessible. The elements can be of type WalkableElement or BarrierElement.
BarrierElement	Contain the border components of a tile. In our game, BarrierElement is only extended by a class Wall.
Card	Move the robots on the board. Interface for all types of cards in the game. Each card executes different actions.
WalkableElement	Interface for all types of walkable elements in the game. Each element performs different actions. In our game, WalkableElement is implemented by Laser and Pit.
Deck	Interface for managing a player's deck of cards. Provides methods for adding and removing cards from the deck, shuffling the deck, drawing a card from the deck, and getting the number of cards in the deck.

Table 1: Class/Interface Responsibilities

every time there is a relevant change in the *model*. For instance this happens when the robots change position or direction, the board is initialized and the playing cards are dealt. This allows the *view* to update the graphics and reflect the changes to the user. Similarly, when the user interacts with the *view*, e.g. by clicking on a button, this information is communicated to the*controller* by observers. The Observer Pattern has the advantage of allowing the *view* to observe the *model* without changing it.

- We implemented the **Abstract Factory Pattern** for the cards and elements on the board in our game. This allows us to create objects of different types without exposing their implementation details. For example, we created an interface called *Card* with an *execute()* method that is overridden in each type of card to execute its specific move. Also, the *WalkableElement* interface represents a unique action that center obstacles in the tiles on the board perform. For instance, the laser gives a damage card to the player while the pit reboots the robot and gives a damage card as well. By utilizing the abstract factory pattern, we can easily create new cards and walkable elements in the future without altering the core implementation.
- We implemented the **Singleton Pattern** for the board in our game, which ensures that a class has only one instance and provides a global point of access to it. Since there is only one board in our game, we wanted to ensure that it was unique and accessible from any point. To achieve this, we created a *Board* class that implements the Singleton pattern, making it a static attribute with a private constructor, and providing a static method to access the single instance. By doing this, we ensure that the Board class can be accessed by any other class that needs it without creating multiple instances, improving performance and reducing the risk of errors.

3.4 Data structures

We employed the following data structures throughout the development of the game:

- **Array**: We used arrays for creating the grids of the boards. These grids are unique for each kind of board and are made of an array of arrays. We decided to make use of this data structure because boards have a fixed size.
- **Array Lists**: We utilised array lists for implementing the decks of cards and some structures of the GUI. We chose array lists for structures that have varying sizes and we wanted to add or remove elements from them easily. For example, the size of the deck of cards depends on the number of players participating in the game.
- **Hash Set**: We employed Hash sets for storing our collection of Observers. We decide to use this structure because the observers are unique (no duplicates) and the order is not relevant.

4 EXTRA FUNCTIONALITIES

In addition to the basic requirements of the game, we have implemented several extra functionalities to enhance the game play experience. Let us take a closer look at what these features are and how we have implemented them in the game.

4.1 Generation of Special Terrains based on Difficulty Criteria

To give players the opportunity to play on different levels of difficulty, we have implemented three different game boards. In addition to the original board, we added two new boards with medium and hard difficulty levels, respectively. Both new boards contain more obstacles and feature a new game play element: Pits. The three game boards are illustrated in Figure 3.

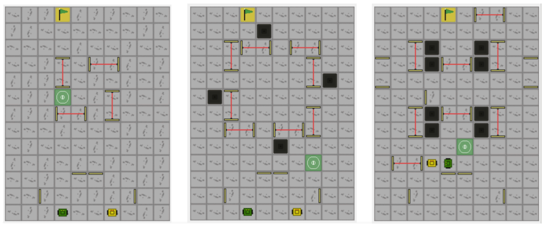

Figure 3: Easy (left), Medium (center), and Hard (right) difficulty game boards.

4.2 Advanced Rules

We wanted players to interact with each other more and to introduce elements of sabotage in the game play. To achieve this, we implemented a feature that allows robots to push each other into lasers, pits, or off the board entirely. This feature was integrated directly into the *MoveForward* card, such that whenever this card is played, it checks for the presence of another robot in front of the current robot. If there is another robot, the push mechanic is triggered, moving the other robot in the direction of the pushing robot. If there is a wall in front of the robot being pushed, both robots remain in their current positions. We also created the *MoveForward* card *execute()* method to work recursively with any number of players.

4.3 Improved Graphical Representation and Sound Effects

For the final feature, we aimed to create a more immersive experience for players. We added sound effects to various game elements to bring the game to life. For instance, when all players have chosen their 5 action cards, a sound effect saying "Robots Activate" plays. When the programming phase starts, a sound effect saying "Start Programming" is played. And when a robot is hit by a laser, a sound is also played. To play the sounds a new dependency javafx-media had to be added to the *Maven Dependencies* through the pom.xml file.

5 USER MANUAL

1. Game Set-up. When launching the game, the first step is to decide the number of players. *RoboRally* can be played by up to 4 players. If only a single player is playing, then he/she will play against an AI. Next, the player(s) need to choose a name, and each player is assigned a robot and a deck of cards. Then, a board difficulty has to be chosen: *Easy*, *Medium* or *Hard*. The boards can be seen in Figure 3. Players are now ready to play! **2. Programming phase.** The programming phase starts. 9 random cards will be distributed to each of the players from their corresponding programming decks of cards and they will be displayed on the screen. Of these 9 cards, 5 need to be selected by each of the players. There are 5 different types of cards.

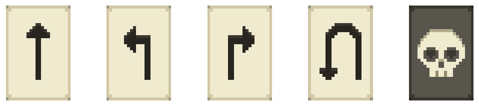

Figure 4: The 5 different types of cards in the game.

These cards can be seen in Figure 4 in the order: *MoveForward* card, *LeftTurn* card, *RightTurn* card, *UTurn* card and *Damage* card. The *MoveForward* card moves the players robot 1 tile forward, the *LeftTurn* and the *RightTurn* card changes the robots direction to the left and right respectively, the *UTurn* card turns robot 180 degrees, and the *damage* card takes up space in the playing deck and limits the players choices of cards.

It is super important the order in which these cards are selected by the player, since this is the order that the cards will be played in the activation phase. The players select the cards by clicking on them. When the cards are selected, a green frame appears around them. A player should choose his cards according to the path towards the endpoint.

3. Activation Phase. The activation phase starts. The activation phase consists of 5 registers. The *i*-th register consists on playing the *i*-th card of each player in order (player 1 → player 2 → player 3 → player 4). When a card of a player is played the players robot moves according to the card. A robot cannot go through a wall. If the robot goes out of the board it will be immediately rebooted to the reboot cell. During the register robots can push each other. If a robot goes into pit, the robot will fall into it, it will be rebooted and the robots player will receive a damage card. If a robot goes on a laser, the laser will activate and a damage card will be inserted in the player's programming deck. **Friendly advice :)** Be clever! A *damage* card is "empty", it does not perform any action, play them to get rid of them!

4. End of the Game The game ends when a robot steps on the endpoint tile. This tile is marked by a green flag on a yellow background. The owner of the robot will be the winner of the game.

6 CONCLUSION

Throughout the development of our project, we gained valuable knowledge not only about Java programming, but also about the principles and methodologies that drives it. By implementing object-oriented and agile techniques, such as Cucumber tests, SOLID principles, Object-Oriented Programming techniques, and Git, we were able to effectively manage and coordinate our team's work. One of the challenges we encountered was organizing our team's workflow in a way that allowed everyone to work on the code simultaneously without causing conflicts and Git provided a perfect solution to this problem. Though, the interaction between Git and Eclipse did cause a lot of struggles which took up valuable time to solve.

All in all, we are satisfied with the outcome of the project, although we know that there is still room for improvement. The agile object-oriented approach we took in developing the game means that modifications and additions to the game can be made easily without altering existing code. We would have enjoyed updating the game's user interface, potentially modifying the robot theme to something different, and implementing new game features, such as unique "superpowers" specific to each robot or adding new obstacles on the board. Additionally, we would have liked to explore methods for improving the game's AI, potentially using the OpenAI API.

7 CONTRIBUTION

	Contribution coefficient
s196050	0
s204331	0
s212717	0
s215238	0
s222704	0