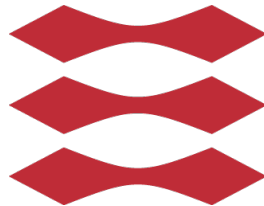


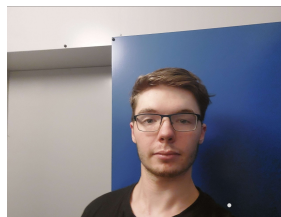
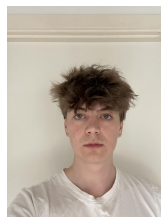
DTU



Danmarks Tekniske Universitet

02362: Projekt i software-udvikling (f24)

Project by



Noah Nissen (s235441), Andreas Jensen (s235455), Alex Lundberg (s235442),
Julius Søndergaard (s234096), William Ajspur (s236165)

02362 Project in Software Development Spring 24,

Technical University of Denmark

Contents

1	Group participation	1
2	Introduction	2
3	Conceive	3
3.1	Problem statement and analysis	3
3.2	Important terms	3
3.3	Project Definition	3
3.3.1	Use cases:	3
3.3.2	GUI improvements	7
3.4	Analysis	7
3.4.1	Requirements	7
3.4.2	Features and Game logic	8
3.4.3	Database requirements	9
3.4.4	Domain Model	10
4	Design	11
4.1	Softwaredesign	11
4.1.1	Activity diagrams for actions	11
4.1.2	MVC architecture	13
4.1.3	Class diagram	14
4.1.4	Observer-design pattern	15
4.2	Database design	17
4.2.1	Saving game configuration	18
4.2.2	Loading game configuration	19
5	Implementation	20
5.1	MVC-Class architecture	20
5.1.1	Model-Classes	20
5.1.2	View-classes	20
5.1.3	Fileaccess-Classes	20
5.1.4	DAL-Classes	21
5.1.5	Controller-Classes	21
5.2	Game flow	23
5.2.1	State change	23
5.3	Exceptions	25
5.4	Field-Actions	26
5.5	Json(Gson)	26
5.5.1	Resource-File-Lister	27
5.6	Deviations and special implementations	27
5.7	GUI	29
5.8	Generics	29
5.9	Recursion	30
5.10	JavaDocs	31
5.11	Database implementation	32
6	Operate	34
6.1	Development process	34
6.2	Tests	34
6.3	Evaluation	37
7	Guide	38
7.1	Downloading the zip file from github	38
7.2	Installing the database	38
7.3	Building the jar file	38
7.4	Opening and starting the game	39
7.5	Creating the java doc	45
8	Glossary	45

9 Conclusion 46

10 References 47

10.1 Website for deck size: 47

10.2 Roborally rules 47

1 Group participation

Miscellaneous

Introduction	Andreas
Conclusion	Andreas
Guide	Andreas
Glossery	Andreas
Javadocs	Primary: Alex

Conceive

Problem statement	Group Effort
Usecases	Noah
Requirements	Julius
Features and game logic	Noah

Design

MVC	Alex, Noah
Class diagram	Alex
Observer design pattern	Julius
DB design	Alex
Game logic	Alex

Implementation

Gameflow	Alex
Exceptions	Andreas
Field Actions	Noah
JSON	Andreas
Special implementation	Noah
GUI	Julius
Generics	Alex
Recursion	Andreas, Noah
DB implementation	Alex

Operate

Developmen process	Julius
Tests	Julius, Noah
Evaluation	Julius

2 Introduction

This report outlines the work conducted in the "Projekt i Softwareudvikling" (Pisu) course. It encompasses the analysis, design, implementation, and operation of the RoboRally game, programmed in Java 21, alongside a MySQL database integration.

The graphical interface is initially provided by the task provider and subsequently customized to suit our requirements.

RoboRally, conceptualized by Richard Garfield and Michael Davis, is a board game featuring robots programmed to navigate through various spaces on a board. These spaces influence robot movement, proximity to victory, and potential hazards. The game accommodates multiplayer engagement, requiring a minimum of two participants and supporting up to six. With five distinct phases dictating gameplay progression, victory is achieved by reaching all checkpoints sequentially, starting from checkpoint 1. The complete rulebook provided for the board game RoboRally is detailed in the RoboRally ruleset.

Our workflow adheres to the CDIO concept (Conceive, Design, Implement, Operate), comprising the following phases:

- Conceive: Incorporating analysis, including activity diagrams and domain models.
- Design: Evaluating design choices and creating diagrams to explain class functionalities.
- Implementation: Code development, with the report highlighting specific choices and specifications.
- Operate: Testing and evaluation to assess implementation effectiveness.

Although the workflow is agile and phases may not strictly follow sequential order, our approach integrates the "what-how" concept. "What" corresponds to analysis (Conceive), while "how" corresponds to implementation. Our model incorporates additional steps to articulate our thought process more comprehensively.

3 Conceive

3.1 Problem statement and analysis

In the "Project in Software Development (PiSu)" course, our goal is to refine the RoboRally game's existing code. Our main objective is to create a fully functional and intuitively playable game. We'll progressively incorporate new game functions and mechanics, focusing on features that enhance the overall gaming experience. Additionally, we'll ensure seamless integration between the game's code and its graphical user interface (GUI), employing Java exclusively for coding purposes throughout this project.

3.2 Important terms

A few important terms to know are a robot, which is the term for the figure each player moves, and the machine carrying out the programming.

Programming entails assembling a set of programming cards, which are subsequently executed in sequence. The objective is to win by reaching all checkpoints in the correct order. Each checkpoint is designated with a number indicating the required sequence of touches.

3.3 Project Definition

The following section presents the use cases derived from the game's rules and the typical desires of players. Additionally, it includes suggestions for enhancing the graphical user interface (GUI).

3.3.1 Use cases:

Use case: Start New Game
ID: UC1
Actor: Player
Brief description: When roborally is running the first thing the player will see is a small tab with a button "file" the player must press the button which then shows a drop down with 3 button "new game", "load game" and "exit" the player must click on the button new game to start a new game. the player then gets a new pop up where they must pick the board they want to play on and afterwards press ok. player is then prompted with another tab where they have to chose the amount of players.

Use case: Load Game
ID: UC2
Actor: Player
Brief description: When the player clicks file on the first tab show it must pick "load game" which then prompts player to pick between saved games. The player picks the wanted game save and the save is loaded.

Use case: Program Robot
ID: UC3
Actor: Player
Brief description: While it is the players turn the player picks the programming cards they want to play for the coming turn, and places them in their registers in the order in which they wish them to be played

Use case: Execute Program
ID: UC4
Actor: Player
Brief description: When all players have placed down their cards they press the "finish programming button" and then presses the "execute program". The game does the rest for them.

Use case: Win Game
ID: UC5
Actor: Player
Brief description: When a player has collected all checkpoints that player wins the game

Use case: Save Game
ID: UC6
Actor: Player
Brief description: When a game is running the player can save the current state of them game by pressing "file" button and then save game. The player then chooses a name for the game

Use Case diagram

Actors

- Primary actor: Player

The player is the primary actor as they are responsible for executing all the use cases. The Database Management System (DBMS) serves as the secondary actor, requiring access to only a limited set of specific use cases.

The diagram:

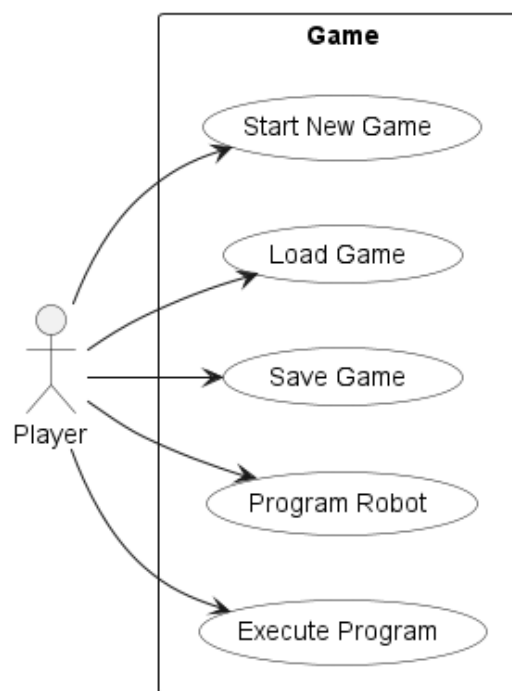


Figure 1: Use case diagram

As the diagram shows the player is interacting with all the use cases that have been selected. These use cases are different from the original ones since a boss test was carried out and most use cases were combined to make each use case significant.

We've selected "Program Robot" and "Execute Program" as the fully-dressed use cases to highlight, as they represent the core functionalities crucial for an engaging player experience. Below are examples of the fully-dressed use cases:

Use case:	Program Robot
Scope:	Roborally - Game
Level:	User Goal
Primary actor:	Player
Stakeholders and Interest:	Player
Preconditions:	Game has started and it is the given players turn
Main Success Scenario:	<ol style="list-style-type: none"> 1. Player choses 5 programming cards from their hand and places them in the registers 2. The turn goes to the next player and they pick their cards repeats for all players 3. After all players have placed their cards the button finish programming is pressed 4. the programming cards of the first register is then executed for the first player 5. repeat for every player and every register 6. the programming cards are all executed for the player and the player has successfully programmed their robot
Extension:	<p>a* if the player does not complete their programming during the 30 seconds random cards from the players are placed into their empty registers. (not implemented)</p> <p>There are no other extensions as every extension or alternate flow only happens during the execution of the cards.</p>
Frequency of Occurrence:	(very frequent) every time a player has their turn or has to place programming cards into their register.

Use Case:	Execute Program
Scope:	Roborally - game
Level	User Goal
Primary Actor:	Player
Stakeholder and Interests:	Player
Preconditions:	All players have placed the cards they wish to play.
Main Success Scenario	<ol style="list-style-type: none"> Any player presses the finish program button Any player presses the execute program button The program executes the card/register in the correct order according to antenna The robot moves depending on the card executed. The robot lands on a new space Repeat 3, 4 and 5 for all registers. Execution phase ends
Extensions:	<p>3a: if any of the cards were an "option" card such as left OR right</p> <ol style="list-style-type: none"> the player is prompted to pick left or right the robot moves according to the choice made <p>3b: if the card is a SPAM damage card.</p> <ol style="list-style-type: none"> the program draws the top card of the players, placing it into the register and executes the drawn card <p>4a. if the direction the robot wishes to move is blocked by wall or antenna</p> <ol style="list-style-type: none"> the robot is stopped by the wall and nothing happens <p>5a: if the space landed on is a "field action"</p> <ol style="list-style-type: none"> the robot "performs" the field action <ol style="list-style-type: none"> if the space is a conveyorbelt <ol style="list-style-type: none"> the robot is pushed according to the heading of the arrow if the space robot is pushed onto is occupied by another players robot. <ol style="list-style-type: none"> push the other players robot according to heading of conveyor arrow if the other players robot cannot be moved because of wall etc. <ol style="list-style-type: none"> do nothing the players robots stand still if the space is a turngear <ol style="list-style-type: none"> the players robot turn according to the direction of the gear if the space is a checkpoint <ol style="list-style-type: none"> the player gets the checkpoint of which they landed on if the space is a pit <ol style="list-style-type: none"> the player dies and loses their turn all cards on player hand is discarded and player pick up 2 card of type SPAM the player must reboot after all other players cards have been executed <ol style="list-style-type: none"> if multiple players dies same turn or if another player is on reboot space <ol style="list-style-type: none"> the players reboot in order of their player number and pushes any other player on the reboot space north <p>(Note: all of this is largely automatically done by the software but relevant to see how the expected actions of the player might be changed)</p>
Frequency of Occurrence:	Happens everytime all players have placed down cards (often)

3.3.2 GUI improvements

The GUI serves as a way of giving players all relevant information about the game and the current state, so improvements would come mostly in the form of enhancing the graphics and improving the clarity of each space.

3.4 Analysis

This section details the requirements derived from the use cases.

3.4.1 Requirements

Requirements are organized, tagged, and prioritized by comparing with MOSCOW. This means there are must-haves, should-haves, could-haves and won't-haves.

Functional requirements

Gameboard features:

- FR1: Checkpoints: Essential to collect for progression and victory determination. Represented by a space with a number on the board indicating the order in which a robot must touch in order to win. (Must-have)
- FR2: Priority Antenna: Dictates turn order via robot proximity calculating rows and then columns to determine which robot is in closest proximity. (Should-have)
- FR3: Player Mats: Organize programming cards and decks. (Must-have)
- FR4: Upgrade Shop: Allows energy cube exchange for robot enhancements. (Could-have)
- FR5: Damage Cards: Four types for hazard encounters (Could-have)
- FR6: Walls: Obstruct robot paths. (Must-have)
- FR7: Sand Timer: Caps programming phase at 30 seconds for each player. (Wont-have)
- FR8: Activation Phase Elements: Includes pits, push panels, and gears being activated during the activation phase. (Must-have)
- FR9: Push Panels: This feature pushes the robot 1 field in the direction the push panel faces. (Could-have)
- FR10: Gears: Rotate robots 90 degrees as indicated. (Should-have)
- FR11: Board Lasers: Lasers that can damage players, but not penetrate walls. (Wont-have)
- FR12: Energy Bank: Awards energy cube for landing on an energy field in the fifth register, if available. (Wont-have)
- FR13: Programming Cards and Deck: Players need to be able to draw programming cards from a command cards deck to be able to move on the board during the activation phase. (Should-have)

Gameplay features:

- F14: Age-Based Robot Placement: Enable age input to determine initial robot placement, starting with the youngest player and proceeding left. (Wont-have)
- FR15: Energy Cubes: Start players with five energy cubes in their reserve, requiring the implementation of an energy balance/reserve. (Could-have)
- FR16: Upgrade Purchases: Allow players to buy upgrade cards with energy cubes at each round's start. (Wont-have)
- FR17: Energy Collection: Provide opportunities for players to collect energy cubes by landing on energy spaces during their move. (Wont-have)
- FR18: Programming Phase Penalty: Penalize players who exceed the 30-second programming phase limit by requiring them to shuffle and draw new programming cards randomly for unfinished registers. (Wont-have)
- FR19: Activation Phase Movement: Base robot movement during the activation phase on programming card instructions, with turn sequence determined by proximity to the priority antenna. (Must-have)

- FR20: Laser Fire: Enable robots to fire lasers in their facing direction, not penetrating walls, with hits resulting in SPAM damage cards to the targeted player. (Wont-have)
- FR21: Reboot Mechanism: Allow players to reboot their robot, such as after being pushed off the board, using reboot tokens. (Must-have)
- FR22: Concurrent Player Occupancy: When a robot moves to a field already occupied, the robot on that field shifts one space in the direction of the moving player (Must-have).

Usability requirements

- UR1: The program/game should be playable by the average person with little to no coding experience given that they receive an instruction manual and a rule set. (Must-have)
- UR2: The experience should be seamless for the user. This means that the user should not experience anything they would classify as hiccups or abrupt stops in the game nor anything that would seem "clunky". (Should-have)
- UR3: Command cards should be intuitive to the user meaning that without any specific instructions, the user should be able to figure out what a command card does just by reading it. (Must-have)
- UR4: The interface and GUI should be relatively easy to use for the user which translates to that no special instructions should be needed for the user to navigate the interface (Should-have)
- UR5: The user should be able to figure out what actions on any given space are/do either intuitively or given a rule set/guide. (Should-have)

System Maintenance Requirements

- SMR1: The program should be designed to easily accommodate potential changes and upgrades, by having a clear and consistent architecture. (Must-have)
- SMR2: Design code to enable map adjustments according to user interest. (Should-have)
- SMR3: Ensure stable user experience and optimal functionality within the project scope by using JUnit tests, such as assertions, to validate robot movement. (Must-have)

System Performance Requirements

- SPR1: User inputs must be processed within 333 milliseconds, since if it takes longer the user will notice a delayed response time from the program (Must-have)

3.4.2 Features and Game logic

To bring the game to life in a digital environment, numerous features and mechanics need to be implemented. Unlike the physical version where players manually move pieces, in the digital space, much of the movement and other game mechanics are managed by the game logic. Here, we'll outline the key features and mechanics essential for the game, primarily executed by the program itself:

Checkpoint

Checkpoints play a pivotal role in providing direction and purpose to the game. Each checkpoint distributed across the board is sequentially numbered from 1 onwards. Players can "pick up" a checkpoint by landing on its space, thereby adding the corresponding point to their score. The objective is for players to acquire all checkpoints scattered throughout the board. Upon successfully obtaining all checkpoints, the player achieves victory.

Moving the robot

Players decide how the robot moves by playing cards. But the actual moving part is done by the program. So, players tell the robot what to do, and the program makes it happen.

Playing turn in correct order

In the game, there's a specific turn order for players, which changes based on the priority antenna. Instead of players manually executing their cards in the correct order, the game handles it. It determines the sequence in which each player's cards should be executed.

Priority antenna

The antenna can calculate the distance between itself and all players. If players are at an equal distance from the antenna, it also calculates the angle between them. Using this data, it establishes a priority or turn order for players, arranging them from least to most distance and angle.

Conveyor belt

When a player lands on a conveyor belt space, a simple action occurs automatically: the player is moved in the direction of the conveyor belt.

Wall

Walls positioned around the board act as barriers, preventing players from passing through them.

Lasers

Lasers pose a threat to players, causing damage if they pass through them. Players receive damage in the form of damage cards when they are affected by lasers.

Turn gear

The turn gear, found on the board, rotates players in its direction when they land on its space.

Landing on any field actions

When a player lands on a field action space, the program will automatically execute the action on behalf of the player, eliminating the need for manual intervention.

Illegal moves

If a player attempts an "illegal" move, such as trying to pass through a wall, the game will autonomously handle the situation. The player won't need to intervene or realize that the move is invalid; the game will automatically prevent the "illegal" move from occurring.

In case of death

If a player's robot should be eliminated, whether due to lasers, collision with another robot, or falling into a pit, the player won't need to manually terminate their robot. The game will handle the elimination process on behalf of the player. Additionally, after elimination, the robot will need to reboot. Again, this rebooting process and any associated actions, like picking up damage cards, will be managed automatically by the program, without requiring player intervention.

Winning the game

In addition to a player collecting all checkpoints, they shouldn't need to manually recognize their victory and close the game. The program will automatically check if a player has collected all checkpoints and won the game. If so, the program will initiate any necessary actions associated with winning the game without requiring player intervention.

3.4.3 Database requirements

The linked database has a set of requirements, specified by the data it should be able to contain. The data is:

- GameID
- Number of players
- Current board
- Player position
- Checkpoint tokens for each player
- Current programming cards
- Current registers
- Priority for antenna

3.4.4 Domain Model

Below is the domain diagram for the RoboRally program, showing the interactions within the model. We have implemented special spaces as distinct types rather than as attributes of the space. This approach aids the extension of spaces with new functionalities as development progresses

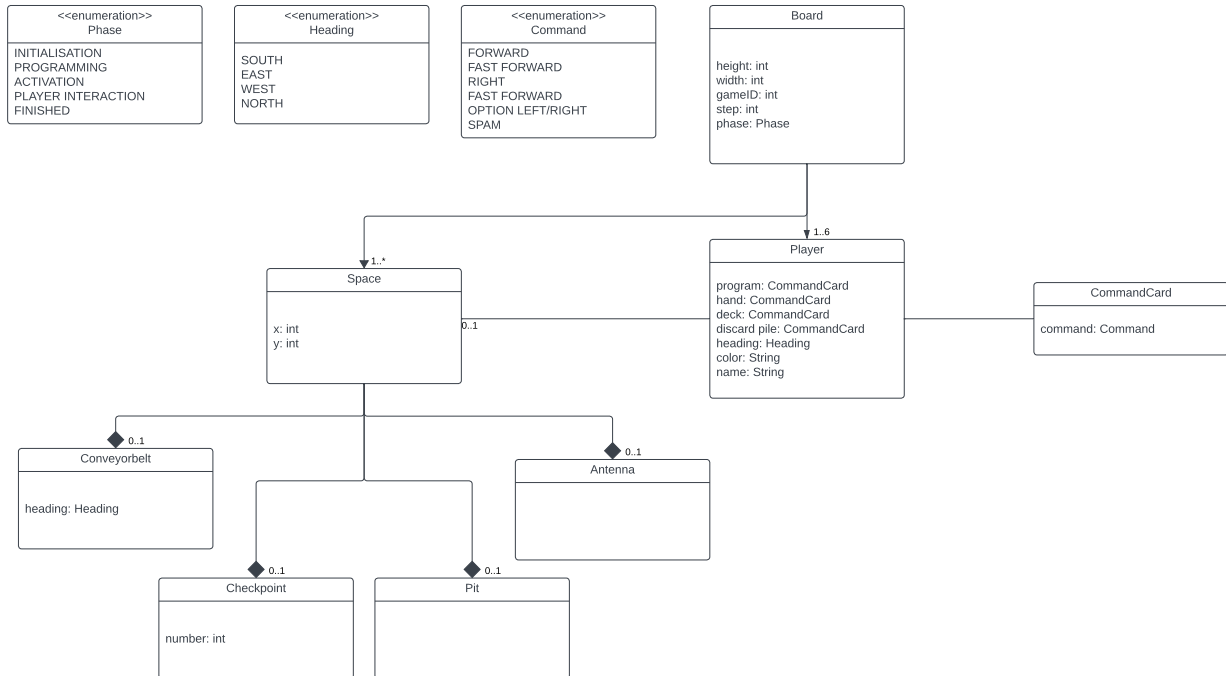


Figure 2: Domain Diagram

The special Spaces are shown as compositions of Space, meaning that the Special spaces are not able to exist independently of a space, though the actual space can be standalone from special spaces.

The directed association between the Board and the player shows that the Spaces and board are associated where spaces exist "within" the board, and a board can not exist without the spaces. This is not an actual requirement in the code, however, it's a logical requirement, as the game will likely break, without this being a flaw in the program.

Enumerations are shown in the top left and allow a basic "datatype" to be used without confusion in the domain diagram. Note that they are only kept because they all are required for the game state which is stored as attributes in the diagram.

Multiplicities where it's 1 have been omitted and thereby used as the default value. As for the command cards, no multiplicity is illustrated as they are different for each player attribute using it.

4 Design

4.1 Softwaredesign

4.1.1 Activity diagrams for actions

Player program execution

The activity diagram shows how the player moves, and what needs to be taken into consideration.

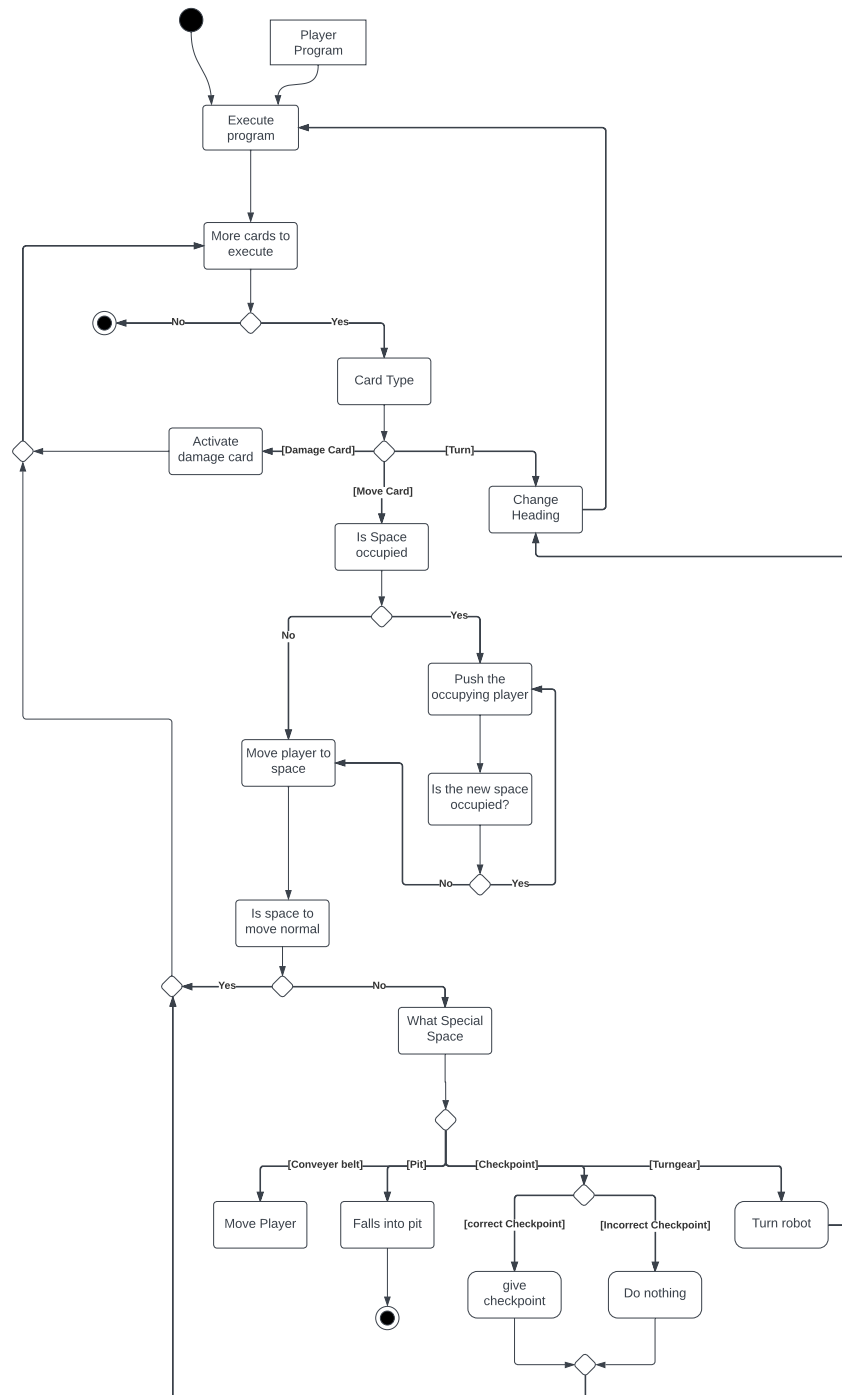


Figure 3: activity diagram for moving the robot

The Player Program consists of five command cards to be executed, represented with hardened edges to indicate it as an object in the program. To enhance readability, scenarios where a player cannot be pushed during movement are omitted. Falling into a pit terminates the player's turn.

It's important to note that the diagram does not consider interactions with other players when executing these cards, so an exit node in this context does not necessarily signify the end of the programming phase

Starting a game The activity diagram demonstrates the initiation of the game, depicting options for starting a new game or loading a game from the database.

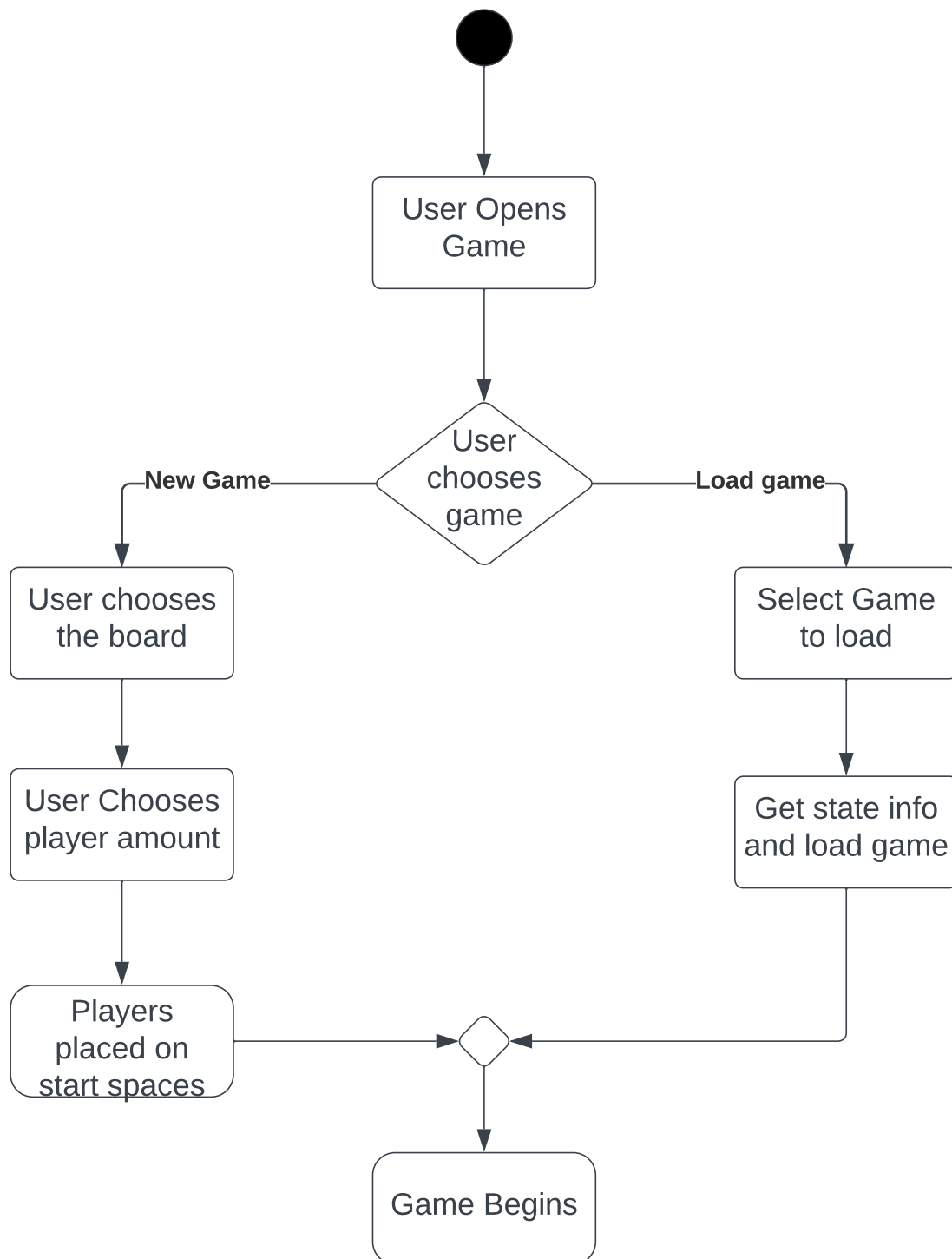


Figure 4: activity diagram for the start of a game

While simple, it outlines the necessary decision points for starting the game. For further details on loading the

game, please refer to figure 10.

4.1.2 MVC architecture

For the project, we employ an extended MVC (Model-View-Controller) pattern, augmented by a DAL (Data Access Layer). The DAL facilitates the translation between the model and the database, enhancing data management and access. This architecture is detailed in the visual below:

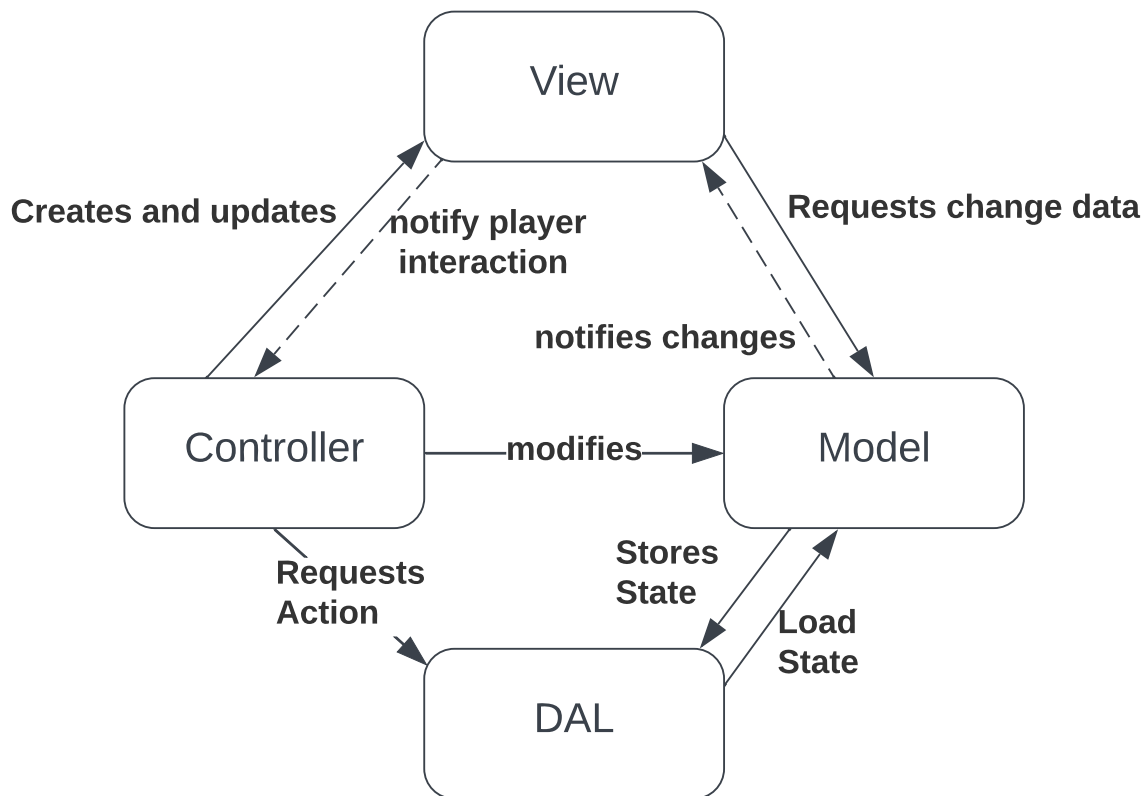


Figure 5: MVC with DAL

Please note that the diagram does not depict a return path from the request action, although, in the actual implementation, the controller calls the DAL and then updates the model. The diagram illustrates that data is retrieved from and saved to the model, with the controller facilitating this process. The inclusion of 'load' and 'store state' arrows clarifies the DAL's role more effectively than merely showing 'requests' and 'get data' interactions between the controller and DAL.

The idea here is that when a game is saved the view notifies the controller of the player interaction to save the game. The controller requests a change in the DAL, which retrieves the game state from the model and then loads it into the database, figure 9. If the player closes and then reopens the game, the controller creates a menu which responds to player interactions. If the player chooses to load a previous game, the controller retrieves the state from the DAL. The model is then updated to reflect this new state and notifies the view of the changes. The game is now loaded and ready, as detailed in figure 10.

4.1.3 Class diagram

Below is a comprehensive class diagram for the model and controller layers. These layers were selected for their complex and engaging elements. The diagram facilitates a deeper understanding of the interactions between different objects and provides insight into the connections between layers discussed in the previous subsection.

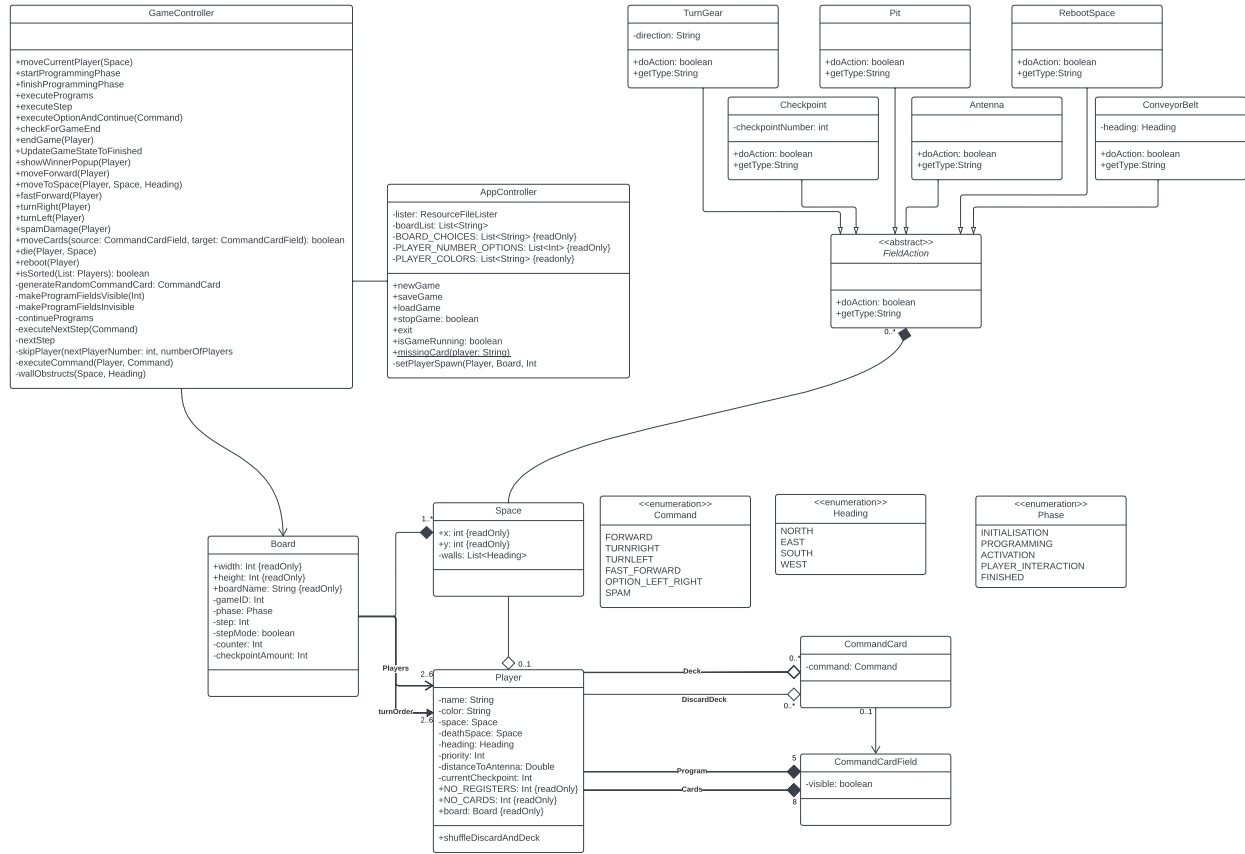


Figure 6: Class diagram for model and controller packages

The cardinalities are not stated for the standard of 1, meaning that for example, 1 space can have 0-* field actions associated with it, but we only state the 0..* cardinality, as the cardinality 1 is verbose.

The diagram highlights the divisions within the controller and model layers. Notably, there is a distinct separation within the controllers, largely due to the implementation of field actions as part of the controller, yet maintaining a close association with the model via the spaces that predominantly manage the program's actions.

Field actions are designated as part of the controller primarily because they encapsulate minimal information and serve to express functionality within the game's spaces. This categorization stems from the implementation of a FieldAction class that acts as a strategy pattern to handle the diverse functionalities of a field. Similarly, the antenna is also classified as a field action. Although it never implements a doAction() method, which is typically required, the implementation as a fieldAction allows for easy integration and avoids further possible abstractions.

4.1.4 Observer-design pattern

Our design relies on the Model-View-Controller architecture organizing our application logic into designated responsibilities which carry out different tasks.

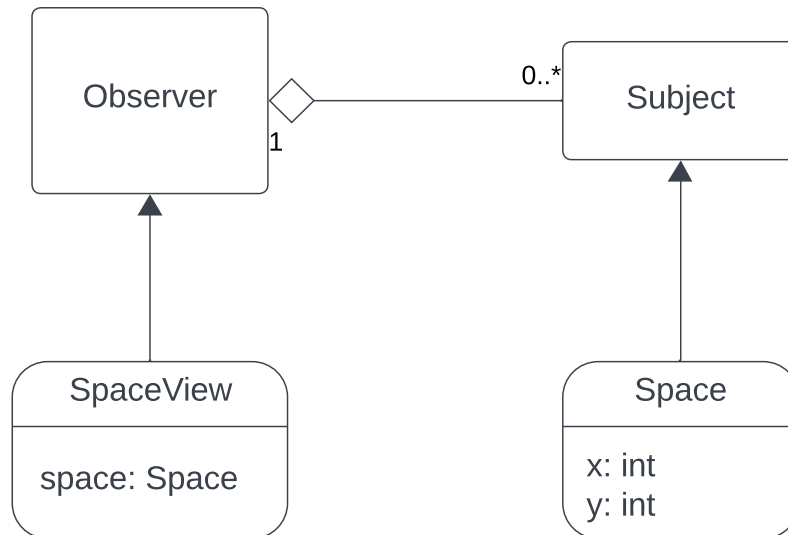


Figure 7: Observer pattern, Space - SpaceView

Our observer-design pattern aids the communication between the View and Model classes. The pattern is a one-to-many dependency between the observer and its subjects in the game. If an observer subject calls the function, `notifychange()`, the observer sees this, updates the View class and the GUI is updated.

In the code down below, the relation between the space in our Model layer and Spaceview in the View layer is illustrated:

```

1  public void setPlayer(Player player) {
2  Player oldPlayer = this.player;
3  if (player != oldPlayer &&
4      (player == null || board == player.board)) {
5      this.player = player;
6      if (oldPlayer != null) {
7          // this should actually not happen
8          oldPlayer.setSpace(null);
9      }
10     if (player != null) {
11         player.setSpace(this);
12     }
13     notifyChange();
14 }
15 }

```

The Space class acts as a Subject within our observer pattern framework. It encapsulates critical properties such as coordinates (x, y), the player currently occupying the space, and any game actions like walls or dynamic elements such as conveyor belts or pits.

The example above shows that if a different player is assigned to this space, the old player is removed from this space (`oldPlayer.setSpace(null);`). This ensures the old player no longer has a reference to this space.

The new player is then linked to this space (`player.setSpace(this);`), establishing a two-way relationship where the space knows which player is currently occupying it, and the player knows which space they are on. After

updating the player-space relationship, the method calls `notifyChange()`. This is crucial for the Observer pattern, as it notifies all observers about changes in the state of the Space. Observers can then update their state or perform actions based on this notification.

So in brief, whenever changes occur within a Space, e.g. when a player moves to a new space or when the game actions associated with the space are triggered, the Space class calls `notifyChange()`.

On the other hand, the SpaceView is an observer of the Space class. It provides a graphical representation of a space on the board. Additionally, it listens for updates from the Space subject it's associated with, ensuring that any changes in the model are immediately reflected visually. During its construction, SpaceView registers itself as an observer of the corresponding Space object by calling `space.attach(this)`.

```
1 // This space view should listen to changes of the space
2 space.attach(this);
3 update(space);
```

This step is important for setting up the observer relationship where SpaceView needs to listen for updates.

SpaceView implements the `updateView` method from the `ViewObserver` interface from the view class. The interface provides a blueprint for implementing common behaviours across the relevant classes, facilitating abstraction, and polymorphism and allowing for easy integration of new functionalities without altering existing code. The `updateView` method is triggered whenever Space calls `notifyChange()`. Inside `updateView`, SpaceView clears its current graphical elements and redraws them based on the new state of the Space. This includes rendering players, walls, and any special actions like belts or pits.

In regards to the GUI, the SpaceView uses JavaFX components to visualize elements like walls (`drawWall`), gears (`drawGear`), and checkpoints (`updateCheckpoint`). We've written more about the GUI in a designated section 5.7. These methods are called conditionally based on the type of field actions in the space, ensuring that the view always matches the game's current state.

In our Space and SpaceView class, our implementation primarily focuses on a static setup where observers are attached once and remain listening for the entirety of their lifecycle. Since SpaceView needs to consistently reflect changes in Space, there is not an apparent need to detach these observers once they are attached. However, for larger applications or those with dynamic UI components that may be created or destroyed based on user interactions or specific conditions, introducing a `detach` method would be crucial.

The benefits of our application architecture include reduced coupling, improved responsiveness and more flexibility as new types of views can be added as observers without modifying the Space class aiding the ease of application maintenance.

4.2 Database design

The database has been kept as simple as possible to keep up with the requirements, and mirroring the model state.

The Relational Schema Diagram below provides a clear view of the database structure. Each game is uniquely identified by a string primary key in the Game table. The Player table uses a PlayerID to identify players within a game and includes a foreign key to indicate the game they are part of. Consequently, both the gameId and PlayerID together form a composite weak primary key for the Player table.

Game states such as phase, step, and current player are all represented as integers in the database. The phase indicates the overall state of the game, with further details provided in figure 17. The step corresponds to the specific register currently being checked, and current player identifies which player's card is being activated at any given time. The boardName represents the name of the board on which the game is being played, which allows us to only store key data for the game state, and not the entire board.

The player's name and colour, which are displayed in the UI, are stored in the database to ensure the correct representation of players when loading the game.

The player's positions, both x and y coordinates, are stored in the database to accurately place the player back on the correct space on the board upon loading the game.

All card fields are stored as a sequence of Strings which are encoded and decoded for the program. This is done instead of keeping a separate table for all the cards, as this makes the save files easier to deal with and while not important, also decreased the size of the database. Let it be known that this does set a cap on database normalization, but we saw it as a trade-off for ease of development, which allowed us to quickly add more functionality.

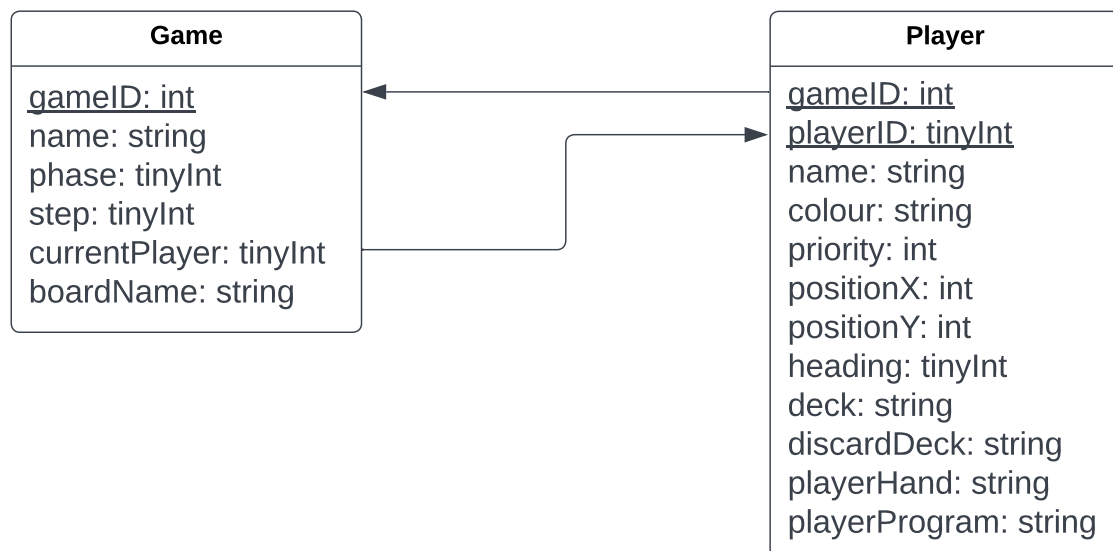


Figure 8: Relational schema diagram

This design adheres to A.C.I.D principles, ensuring that all database operations are atomic within a transaction. It strictly upholds database constraints, with the 'truth' of each game element, such as letters or card representations, maintained distinctly, rather than using numerical representations for cards. Isolation and durability are managed by our DBMS, as we do not handle backups or manage concurrent database interactions. This approach simplifies our focus, relying on the database system to maintain data integrity and persistence.

4.2.1 Saving game configuration

When a player saves the game, the process is captured in the database. The sequence diagram below describes this operation in detail:

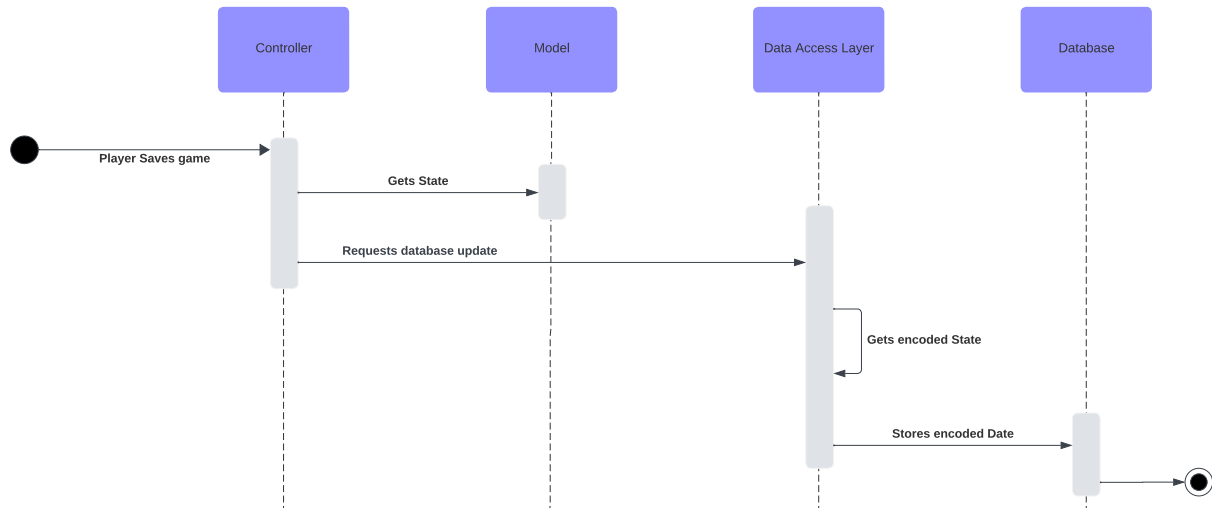


Figure 9: Sequence diagram for Saving from database

Here the program is shown as Model, Controller, DAL, and DB. This is chosen simply because we never actually notify the player of the change in the save file, and therefore, the view is not included. Please note that in the software, the controller method is initiated from the view layer, so the representation here is a simplified version of the actual process.

Going through the process, the controller retrieves the information to the Data Access Layer where the data is encoded to adhere to the database constraints. Then a transaction is started allowing the game to be saved atomically.

4.2.2 Loading game configuration

The game is loaded from a save file generated earlier.

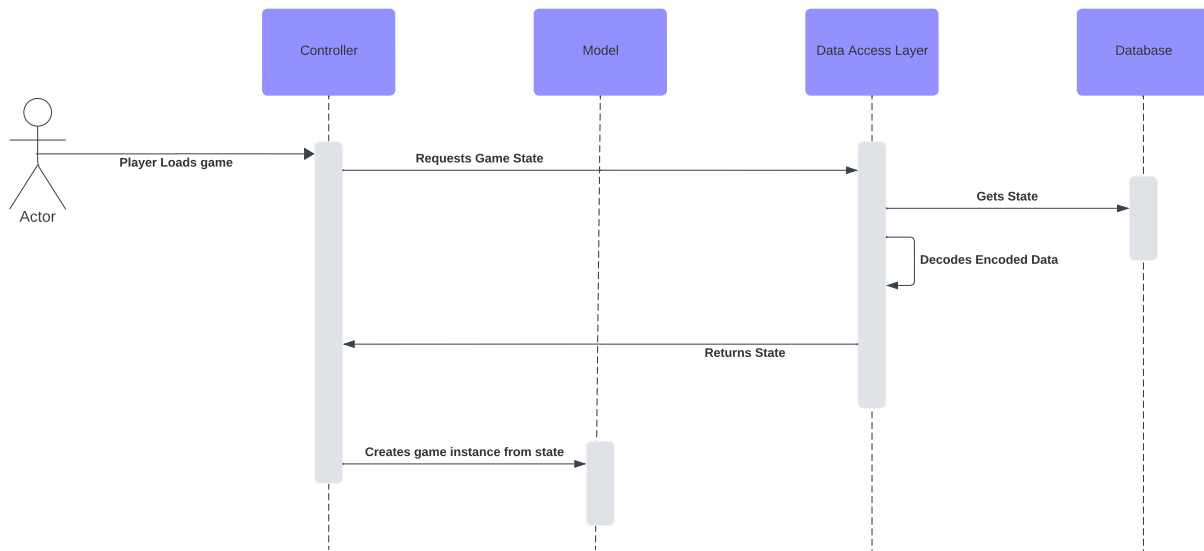


Figure 10: Sequence diagram for loading from database

Upon player initiation, the Controller requests data from the Data Access Layer (DAL), which decodes the requested information. The Controller then constructs the model using this data, recreating the state of the saved game.

5 Implementation

5.1 MVC-Class architecture

To begin the implementation chapter, we extend the previous discussion on MVC by detailing the specific files involved in this project, as outlined in the following section.

5.1.1 Model-Classes

The model class files store all essential information regarding the game's current state. Only the game controller class files have the authority to modify this data.

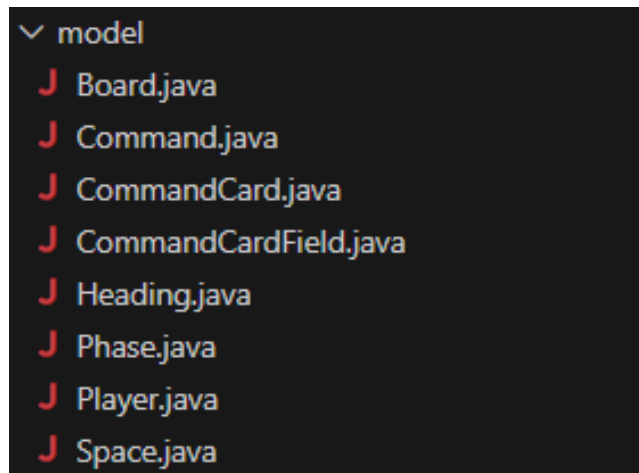


Figure 11: The Model-Classes

5.1.2 View-classes

The view class files are responsible for displaying data to the player, sourced from the model. This forms the basis of the GUI. Whenever a player acts within the view, the game controller is notified and subsequently executes any relevant methods.

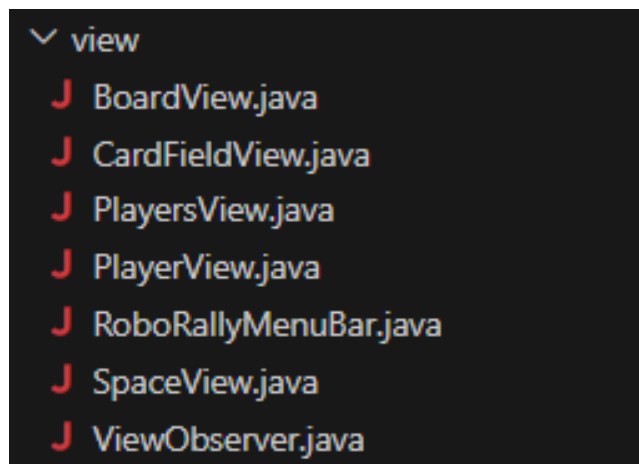


Figure 12: The View-Classes

5.1.3 Fileaccess-Classes

The file-access directory has another directory inside it, containing all the templates used by the loadboard class, to load in the board from json.

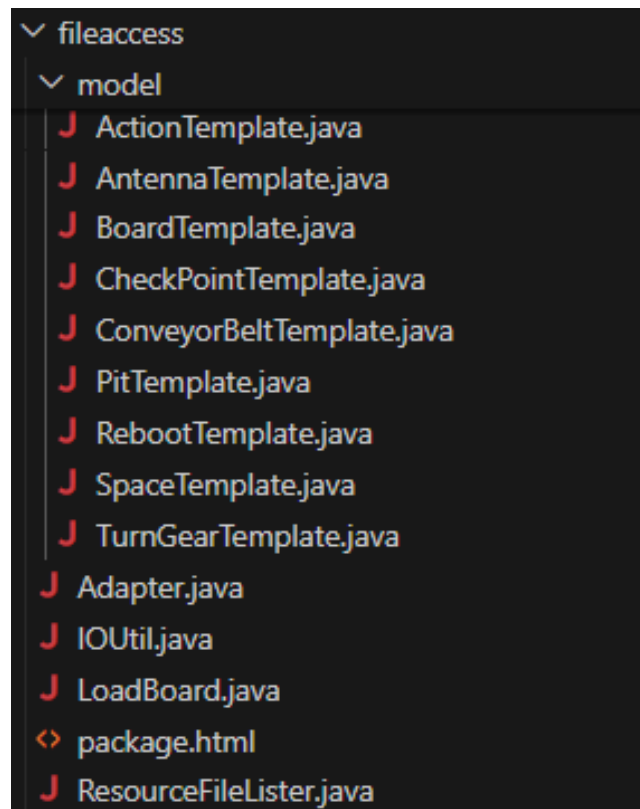


Figure 13: The Fileaccess-Classes

5.1.4 DAL-Classes

A note to this directory is that this is where the DBAcces file is located, which should be configured before the use of the program.

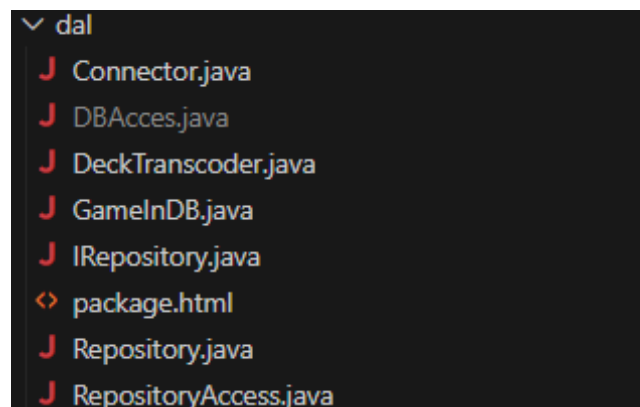


Figure 14: The Database Acces Layer Classes

5.1.5 Controller-Classes

The controller classes are the ones that contain all the methods that change the game and these classes are essentially the ones that control the entire game.

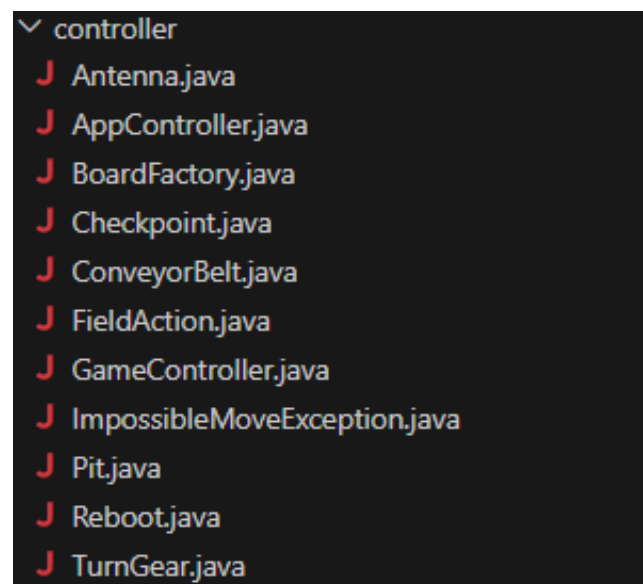


Figure 15: The Controller-Classes

5.2 Game flow

Here is the general flow of the game. The diagram below does not consider the loading and saving of the game as these things do not interfere with the actual game logic.

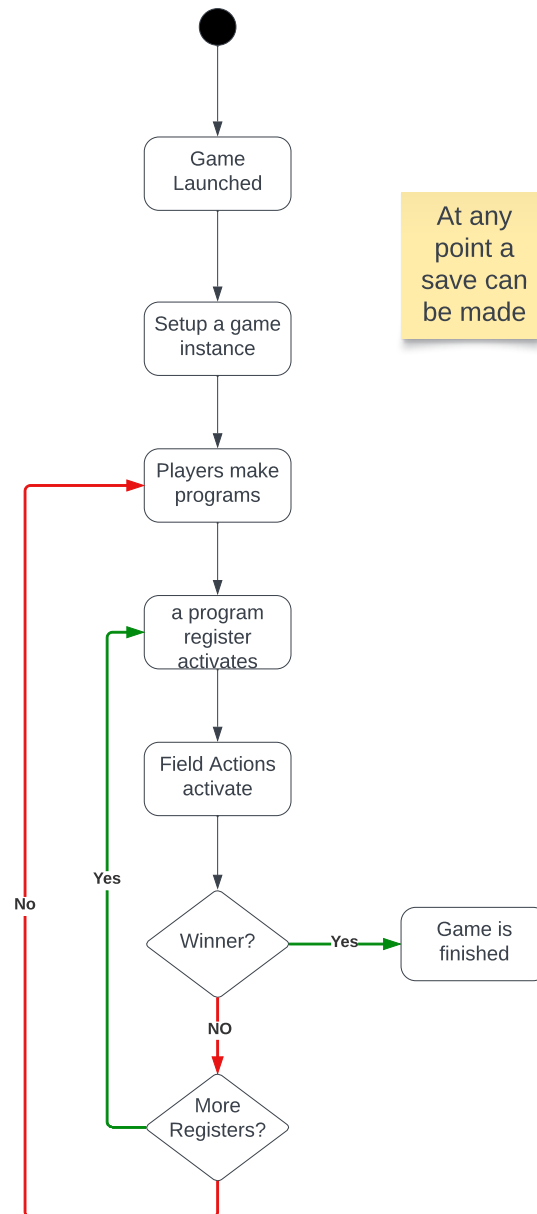


Figure 16: game flow

The game has a game loop in which the players program the cards and then activate them, see figure 3 for details on the turn activation. The field actions are activated after each register, in the order which is outlined in the official RoboRally rules.

As a checkpoint is one of these fields, the actual check is done after the actions activate to make sure the most recent state is considered. If no winner is found we run the next register, and if there are no other registers, we make programs again. The details on phase changes are found in figure 17

5.2.1 State change

State Diagram

Below is the state machine diagram for phase changes in the Roborally game.

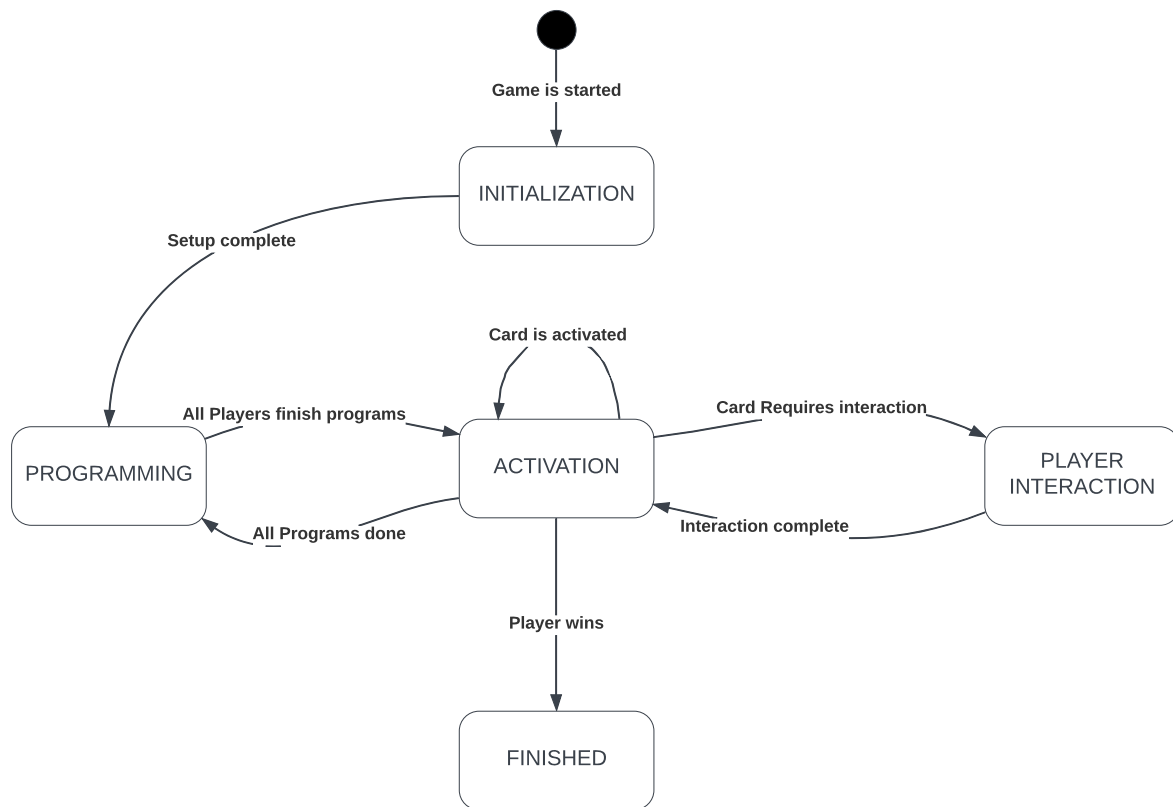


Figure 17: State Machine for game phases

We start with the initialisation phase where the game is set up to be played, see figure 4 for details about this step.

Note that for a game that's loaded from the database, the phase change from initialisation will vary depending on the stored state.

During the programming phase, players create the programs that will be executed in the subsequent phase by filling their program fields with five cards each. Once all players have completed their programs, the activation phase is initiated.

In the activation phase, cards are sequentially activated. After each register is executed, the corresponding field actions are triggered, and the game checks if a player has won. If a victory is declared, the game transitions to an immutable final state. If a card necessitates interaction, the phase pauses until the interaction is completed. For additional details on the activation phase, refer to the figure 3.

5.3 Exceptions

Exceptions in Java are interruptions to a program's natural flow. These interruptions can be thrown by individual functions, in the case of an error, and can be passed up the call stack, until they are caught. If they are not caught they potentially terminate the program, or make it unresponsive. Exceptions can be handled or caught, meaning that the programmer can prepare for the event of an error, and set up some instructions to be carried out in case of the error happening.

Types of exceptions

- 1. Checked exceptions: These are exceptions that the programmer has prepared for, and a method must throw the exception or it should be handled inside a try-catch block for it to be a checked exception.
- 2. Unchecked exceptions: These are unexpected exceptions, that usually occur as a result of programming errors, logic errors or wrong use of API's.

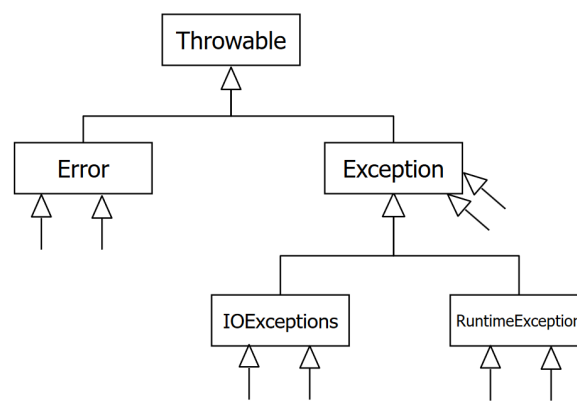


Figure 18: Hierarchy of exceptions

The graphic above shows the hierarchy of exceptions, and to categorize them, one would say that Error's and Runtime exceptions are checked, while the rest are unchecked exceptions. Classes that inherit from Error or Runtime exceptions are unchecked as well.

Control flow

The method of handling exceptions is with try-catch blocks. An example is provided below:

```

2      try {
3          ...
4      } catch {Exception1 e} {
5          ...
6      } catch {Exception2 e} {
7          ...
8      } finally {
9          ...
10     }
11 }

```

Figure 19: An example of a try catch finally statement

There are 3 main components. The content inside try is the code you are trying to run, which might throw an error. If it throws an error this would be caught by the catch statement, and the code inside the catch would be executed. This could include printing a stack trace for the error. Finally, the code always executes the code inside the final statement, even if it doesn't encounter an error in the try block. In the case of this being in

a method where the try block returns something, the code finally will still be run after the return statement. The final block is not necessary, and there can be multiple catch blocks. Each of these catch blocks can be responsible for an individual error type, or it can be made to fit all errors.

Exceptions in RoboRally

In RoboRally an exception was implemented to handle the case of trying to push another robot, onto either an invalid space or through a wall. This exception is called `ImpossibleMoveException` and takes a player, a space and a heading. When called it passes the message "Impossible Move" to its super, which is just the class exception. This message can be retrieved if relevant, but in this case, the exception shouldn't cause the program to freeze or stop working. This exception comes into play in the method `moveForward`, where the method tries to call `movetoSpace`, which is the method that throws the impossible move exception. This means that if the exception occurs, nobody will be moved and the move will simply be ignored, which makes sense in the context of the game.

5.4 Field-Actions

The way we have chosen to implement our field actions such as pit, conveyorbelt, turngear and checkpoints is by having an abstract class called `FieldAction` located in the controller folder this class has 2 methods one called `doAction()` and another called `getType()` that all field action classes inherits from the `FieldAction` class. The `doAction()` is responsible for performing the actions that need to happen to the game or player whenever a player lands on the given field action and is called for all spaces containing a field action that a player is stood on whenever a command card is executed. The method `getType()` is used to identify what field action the space currently contains so when `getType()` is called on a field action inside a space it will return a string with a short description of what the field action is such as "belt" symbolising a conveyor belt.

5.5 Json(Gson)

For this project, multiple boards can be selected, each stored as a JSON file in the resource folder. JSON's key-value structure allows each space on a board to be defined with specific attributes, such as wall locations or the type of space.

Connecting json to the game

When the game tries to show the board, it can't purely take the JSON board, since its input is a set of space objects, with properties added. This means that a translator is required to transform the data in the JSON file into the format of an array of space objects, with their properties added.

This translator is made with the help of GSON, which is a class provided by Google. This is a class in which an object can be created that has various of useful elements such as a simple builder. Before creation, it's configured to use the abstract `ActionTemplate` as an adapter, which translates to all classes that extend the `actiontemplate` can be processed. Afterwards, the builder is created. Now it can take an input stream of the JSON file. The stream is processed according to the adapter the builder was given, hence why it requires that each space has a template that extends `ActionTemplate`.

Now that it can read the contents of the JSON file, and extract information according to what matches each template, it's possible to create a new empty board and then loop through all spaces based on the height and width given in the JSON file.

When looping through it retrieves what template it should utilize from the JSON file, and then loads the information as properties of each space.

Here's an example of the start of a JSON file, and the following is the explanation using the example:

```

1  {
2    "width": 8,
3    "height": 8,
4    "spaces": [
5      {
6        "walls": [
7          "SOUTH"
8        ],
9        "actions": [
10       {

```

```

11         "CLASSNAME": "dk.dtu.compute.se.pisd.roborally.fileaccess.model.ConveyorBeltTemplate",
12         "INSTANCE": {
13             "heading": "WEST"
14         }
15     },
16     ],
17     "x": 0,
18     "y": 0
19 },

```

At the start, the width and height are defined and since they are in quotes they are the key while the colon signifies the value. This means that when converting the template into a board these can be found by simply using `template.width` and `template.height`. Since this returns the values, the board created has the size given in the JSON file. Next is a list of spaces, in this case, spaces are the key, and then a list of spaces follows. Each space then has additional keys, walls, actions, x and y. Each of these keys has the values after the colon, and again there are two lists, as both walls and action, can have multiple instances. Each action has a subset of keys, that are parsed into a separate function that deals with actions. The Instance key, is used as a way of parsing a variable, and its value to the individual action. This action is the `FieldAction` that is explained above. Basically, the converter takes the template from the GSON reader, then it creates a template of the action, with the parameter from the reader, so in this example, it would create a `conveyorbeltemplate`, with the heading of west. Afterwards, it creates and returns the conveyorbelt, and gives the conveyorbelt the heading, that it received from the template.

There's another part of the `loadboard.java` file, where it does this in reverse, creating a template from a class, but that's not used in the RoboRally program, but could be used to make a new board without having to make the JSON file yourself.

5.5.1 Resource-File-Lister

In a Maven project like this, all resources must be stored in the resource folder and are accessed using the classloader, which can read the contents of files even when the project is compiled as a jar file. Unlike the `java.util.file` import, which can't access files in a jar, the classloader is essential for this purpose.

However, accessing the names of the files, rather than their contents, presents a challenge. For example, in the `AppController`, where users select a board for a new game, the file names of the JSON board files need to be dynamically listed. This list should automatically update to include any new boards created by users without manual intervention. To achieve this, we utilize Spring as a dependency, which enables the search for files matching a specific pattern. All board files are JSON files, allowing Spring to identify and list all relevant files. This functionality is implemented in the `ResourceFileLister` class, which returns a list of file names without the `'json'` extension. These names are then directly imported into the GUI from the `AppController`, allowing users to select their desired board

5.6 Deviations and special implementations

Programming robot

In RoboRally, players must program their robots by placing 5 cards on their registers within 30 seconds, after which any unfilled registers are automatically filled with random cards from their hand. In our digital adaptation, we've omitted the 30-second timer to accommodate the challenges of dragging and navigating cards with a mouse or mousepad. Instead, we've implemented a method that activates when a player hits the 'finish programming' button. This method checks all registers of all players. If any register is found to be empty (null), an alert notifies which player has not completed their programming. This modification ensures that all players still play 5 cards, maintaining the game's integrity without the original time constraint

Falling off of the board

In traditional RoboRally, if a player's robot moves beyond the board's boundaries, it is considered as falling off and results in the robot's 'death,' requiring a reboot similar to falling into a pit. However, in our adaptation, moving off the board doesn't result in death. Instead, the player reappears on the opposite side of the board. This is achieved by taking the player's position modulo the board's height or width. If the result is zero, the

player is moved to the corresponding opposite side, introducing a wrap-around effect rather than a fatal outcome.

Antenna

Our implementation of the antenna has managed to fulfil all the functions of the antenna and works seamlessly with the database. The way the antenna works is by first calculating the distance between the player and the antenna by adding up the x and y amount of spaces to the player. Next, it calculates the angle to the player in case two players are an equal distance away the size of the angle will be used to determine who goes first. The player then has an attribute called distanceToAntenna that is set after calculating both. The way we calculate both angle and distance is shown below:

```
1 double py = (double) playerspace.y;
2 double px = (double) playerspace.x;
3 double ay = (double) space.y;
4 double ax = (double) space.x;
5 distance += ((1.0 + (Math.atan2(py - ay, px - ax) / (2 * Math.PI) + 0.25)) % 1.0);
6 distance+=(Math.atan2(py-ay,px-ax))/(Math.PI*2)+0.5;
7
8 board.getPlayerTurn(i).setDistanceToAntenna(distance);
```

The method used to determine player turn order starts by calculating the distance and angle from a fixed northern heading, which cannot be altered. After calculations, the players are sorted by most to least distance and angle using bubble sort. This sorting process occurs just before each programming phase begins and when a new game is initiated. Player turn order is indicated in the status message as P1, P2, P3, etc. Unlike the automatic progression to the next player found in some games, our adaptation requires players to manually switch to the next player by referencing the turn order in the status message.

Player decks

The way we have decided to implement player decks deviate slightly from the way that RoboRally game rules dictate. Players each have their "own" deck in the sense that each player is assigned a set deck of random cards at the start of the game but the deck itself is not visible to the player and the player itself does not have to draw the cards manually from the deck. furthermore, the discard pile isn't physical but only exists within the code where each card gets put after the execution phase the cards are then shuffled back into the deck when the deck is empty.

SPAM damage

In our adaptation of RoboRally, each player is assigned a randomized deck of cards at the start of the game. Unlike the original game rules, these decks are not visible to players, nor do players manually draw cards from them. Additionally, the discard pile is virtual, existing only within the code. After cards are used in the execution phase, they are placed in this virtual discard pile. When the deck runs out, the discarded cards are shuffled back into the deck for future use.

Spawn locations

In the traditional RoboRally game, players are required to spawn on designated black and white gears, choosing their starting position based on a specific order. In our adaptation, we have not implemented black-and-white gears. Instead, we use a method to generate a random space for player spawning. After selecting a space, we conduct an 'if check' to ensure it is not a restricted area like checkpoints, pits, or antennas. If the space is deemed suitable, the player will spawn there.

Rebooting

The way reboots work in roborally rules is that there is one reboot per board and that players reboot at the one corresponding to the board they died on. In our roborally we can have multiple reboots on the same board and players will reboot on the reboot space closest to where they died we do this by storing the player "death space" and calculating the the distance to all the reboots and then spawning at the one with the least distance. Furthermore player will reboot in the order of the player list so who dies first has no effect on the spawn order. In case a lot of players die they will all spawn facing north and will push the player already standing on the space north, this means that if 5 players die and there is a wall 4 spaces up it will not spawn the 5'th player since it cannot push the rest of the players further up and player 5 will be spawned next round.

5.7 GUI

The core focus in our report has been on the functionality of the game, and not the GUI in particular. However, the GUI can help facilitate gameplay and user interaction delivering a user-friendly experience which is desired.

To build the graphical user interface, we use JavaFX combined with PNG images to strengthen the presentation. The images help to provide a more appealing look compared to purely programmatic shapes. An example of this can be seen down below:

```

1  private void drawBelt(){
2      if (space.getActions().size() != 0){
3          FieldAction action = space.getActions().get(0);
4          if (action.getType().equals("Belt")){
5              ImageView belt = new ImageView(new Image(getClass()
6                  .getResourceAsStream("/images/belt.png")));
7              belt.setRotate((90 * ((ConveyorBelt)action).getHeading().ordinal()) % 360);
8              this.getChildren().add(belt);
9          }
10     }
11 }
12 }

```

In the drawBelt() method in our SpaceView class, we use an ImageView to display a belt PNG, which visually represents the conveyor belts on the game board. This method checks for a ConveyorBelt action in the space, loads the appropriate image, sets its rotation based on the belt's heading, and displays it in the GUI, enhancing the visual cues that guide player movement.

The Space class, part of the Model in the MVC architecture, manages a list of FieldAction objects that can include various game elements such as conveyor belts. It does not hold specific details about these elements but knows which actions are present on it. Changes to these actions, such as adding or removing a conveyor belt, trigger notifications to the SpaceView (part of the View), which updates the visual representation in response. This communication is facilitated through the observer pattern, ensuring the GUI reflects the current state of the game board.

The GameController (controller) interacts with this setup by managing how actions like the movement of players along conveyor belts are executed. It calculates the player's new position based on the belt's direction and updates the model, which in turn updates the view. This separation ensures that the game logic and GUI are loosely coupled, making the system easier to maintain and extend.

However, it's worth noting an element with our GUI: it currently only supports displaying one action field per space, whereas the underlying code is robust enough to handle multiple actions simultaneously. This limitation could potentially impact the visual representation of complex game boards, and future improvements to the GUI may need to address this issue for a more comprehensive user experience.

5.8 Generics

Generics are used in multiple places in the codebase. It offers a way to establish methods that are generally useful and can be used with different types. One generic type we have used frequently is a List. In our project, we specify the types of elements that should be included in the list. For the actual implementation, we primarily use ArrayList to manage these elements.

We implemented a method using generics which is being utilized to find instances of a fieldAction in the board making it easier to implement future features where this is a required logic step.

```

1  public <T extends FieldAction> ArrayList<Space> getSpaceByActionSubClass(Class<T> filter){
2      List<Space> output = new ArrayList<>();
3      for(Space[] spaceList : spaces){
4          for(Space space : spaceList){
5              if (space.getActions().size() == 0){
6                  continue;
7              }
8              for(FieldAction fieldaction : space.getActions()){

```



```

9         if (filter.isInstance(fieldaction) == true){
10             output.add(space);
11         }
12     }
13 }
14 }
15 }
16 return output;
17 }

```

This is using a concept known as reflection, though we will only discuss the use of generics here. We use `T` as a generic type where we employ a bounded type because we want subclasses of `FieldAction`. Afterwards, the method is executed with the bounded generic type as the parameter. This makes it easy to use when further developing the game, but also robust enough to make it seamless and dynamic with a developing codebase. Also used within the method is another type of generic datatype. The list takes a specific class which is contained in the actual list. It's then implemented as an `ArrayList` where the generic type to be used here is inferred from the `List`.

5.9 Recursion

Recursion is a term in programming for having a function continuously call itself until some condition is met. This could be if you were trying to compute the factorial of a number.

```

1 public static int factorial(int n) {
2     if (n == 0) {
3         return 1;
4     } else if (n > 0) {
5         return n * factorial(n - 1);
6     } else {
7         throw new IllegalArgumentException();
8     }
9 }
10

```

In this example, if `n` is greater than 0, the function recursively calls itself with `n-1`. This continues until it reaches 0, at which point it returns 1. Following this, each recursive call resolves by returning the product of `n` and the result of the factorial of `n-1`, ultimately calculating the factorial of the original number.

Recursion in roborally

In `RoboRally`, recursion is utilized to manage scenarios where a player needs to push others in front of them. For example, even if a player is positioned behind a line of five players, they can push all five forward. Recursion comes into play when a player tries to move forward and encounters another player in their path. Now, you attempt to call 'move forward' on that player to check if movement is possible. If they can move but encounter another player ahead, the function is called again on the third player to assess their ability to move. This process continues until either there are no more players in front and a space is available for the last player to move into, or an obstacle such as a wall or antenna blocks a robot's path. In the latter case, an 'impossible move' exception is thrown, which cancels the entire sequence of moves, preventing any movement. The code for this functionality is implemented as follows:

```

1 public void moveForward(@NotNull Player player) {
2     Space space = player.getSpace();
3     if (space != null) {
4         Heading heading = player.getHeading();
5         Space newSpace = board.getNeighbour(space, heading);
6         if (!wallObstructs(player.getSpace(), player.getHeading())) {
7             if (newSpace != null) {
8                 try {
9                     moveToSpace(player, newSpace, heading);
10                    player.setSpace(newSpace);
11                } catch (ImpossibleMoveException e) {

```

```

12         }
13     }
14 }
15
16
17 }
18
19 }
20 public void moveToSpace(
21     @NotNull Player player,
22     @NotNull Space space,
23     @NotNull Heading heading) throws ImpossibleMoveException {
24     Player other = space.getPlayer();
25     if (other != null) {
26         Space newspace = board.getNeighbour(space, heading);
27
28         if (newspace != null && !wallObstructs(other.getSpace(), player.getHeading())) {
29             moveToSpace(other, newspace, heading);
30         } else
31             throw new ImpossibleMoveException(player, newspace, heading);
32     }
33     player.setSpace(space);
34 }
35 }

```

In the implementation, the `moveToSpace` is the function that is called recursively as can be seen by it calling itself. The necessity for two functions arises partly because each robot can have a distinct heading. To maintain the original heading for each robot throughout the recursive process, the first function does not include a heading as a parameter. Consequently, a second function is required, specifically designed to accept heading as a parameter, ensuring accurate direction management across recursive calls.

Possible bugs

If a board is only six spaces wide and all players are lined up in a row, pushing from the last player could result in an infinite loop of recursion, due to the fact that the board wraps on leaving edges. This would happen if each robot continuously attempts to push the one in front, effectively creating a cycle where there is always another robot to push, leading to endless recursion and causing the game to freeze. The fix is to add a counter that increments each time the function is called, and when the counter is above the amount of players, it should simply throw an `impossiblemoveexception`.

The again card

There is a card called `again`, that repeats the last card, and this one also uses recursion in the case of having multiple `again` cards in a row. It is implemented as:

```

1 public void playAgain(Player player, int step){
2     if(step > 0){
3         CommandCard previousCard = player.getProgramField(step-1).getCard();
4         if(previousCard.command != Command.AGAIN){
5             executeCommand(player, previousCard.command);
6         } else if(previousCard.command == Command.AGAIN) {
7             playAgain(player, step-1);
8         }
9     }
10 }

```

In this case, there is a variable that will get decreased each time it calls itself, so it could potentially call the same fast forward 4 times given there are 3 `against` cards in a row after the fast forward.

5.10 JavaDocs

In this project, JavaDocs are used for commenting, allowing us to generate documentation as we develop. This approach is extremely beneficial both during the development process and at project handoff. JavaDocs provide

easy access to function details through language server protocols (LSP) available in most modern Java IDEs. This reduces the need to manually review all functions in a growing codebase, offering a time-efficient way to abstract and develop documentation.

Adopting JavaDocs as a standard practice eases the burden of code documentation by enabling developers to quickly explain their code. This prevents the original coder from needing to address issues other people may have with the code later on. Additionally, when code is updated, the corresponding documentation should also be revised to reflect these changes. This approach helps avoid the problem of outdated documentation, ensuring that the documentation remains relevant and accurate.

5.11 Database implementation

The database as described in 4.2 needs to be implemented to work. The SQL code used to make the required tables can be seen below:

```

1  CREATE TABLE IF NOT EXISTS Game (
2  gameID int NOT NULL UNIQUE AUTO_INCREMENT,
3
4  name varchar(255),
5
6  phase tinyint,
7  step tinyint,
8  currentPlayer tinyint NULL,
9  boardName varchar(255),
10
11 PRIMARY KEY (gameID),
12 FOREIGN KEY (gameID, currentPlayer) REFERENCES Player(gameID, playerID)
13 );
14
15 CREATE TABLE IF NOT EXISTS Player (
16 gameID int NOT NULL,
17 playerID tinyint NOT NULL,
18
19 name varchar(255),
20 colour varchar(31),
21
22 priority int,
23 positionX int,
24 positionY int,
25 heading tinyint,
26 deck varchar(255),
27 discardDeck varchar(255),
28 playerHand varchar(255),
29 playerProgram varchar(255),
30
31 PRIMARY KEY (gameID, playerID),
32 FOREIGN KEY (gameID) REFERENCES Game(gameID)
33 );
34

```

The types for the different attributes have already been listed and remarked upon in 4.2, and only a select few will be explained.

The gameID is the primary key for the Game table and also the Player table. It is set as not null meaning that this specific attribute must always have a value. It is unique, meaning that at no point can 2 tuples exist within the database that have the same gameID. Auto.increment ensures that we're not dealing with the gameID constraints in the code we use to insert values into the database, thus allowing consistency in ACID terms. The rest of the Game table has been remarked on, hence there is nothing more here.

The player table has its primary keys as gameID and playerID. By referencing gameID it's easily understood what player belongs to what game. Naturally, neither can be null as they are primary keys.

The deck, discardDeck, playerHand, and playerProgram are all using a varchar as the type. The reason behind this was in the design section on the database therefore proceeding without the why, and explaining the how. The cards are stored by representing each card as a Char in a string. This allows us to keep the database mirroring the state which it represents, where the Game contains the board, currentPlayer, and generally all the information stored on the board. While the player contains the attributes associated with the player. It's done by using a transcoder to encode and decode these Strings by using a map for each which can be easily extended when adding new cards into the game. Currently, it's adding to it when a transcoder object is instantiated, and if ever new requirements arise this could be changed out by a builder or something similar to create differing interpretations of the data. Once complete, the information is retrieved from the ResultSet for the Player table and then converted into a stream to enable functional data processing.

```
1 String string = deck.stream()
2   .filter((CommandCard card) -> card != null)
3   .filter((CommandCard card) -> card.getName() != null)
4   .map((CommandCard card) -> this.getCardMap().getOrDefault(card.getName().toString(), "n"))
5   .collect(Collectors.joining());
```

The first two filters are applied to ensure the deck is free of any anomalies or unusual values. Following this, a map function is used to convert the card objects into a string representation suitable for database storage. To ensure all errors can be found and minimized the get for the map is doing a defaulting to a not-in-use card letter to make sure this defaulting does not mean that the decks are changed in a way that is not noticeable as an error. Finally, the stream is collected by `Collectors.joining()` which takes all the stream elements that are strings or chars and collects them into a final string. Then this data is sent to the database to be retrieved at a later point.

A repository is used to make the actual operations. The repository is a way of making it easy to implement different versions of a save file allowing us to quickly change between using an online database, or local, or even changing database structure or type. Without touching the other Layers. The only implementation we have of the repository is the local repository which uses a DBAcces file to store the connection details. Here an interface is used effectively making a strategy pattern for the DAL. Note that this is more complex than not using it, and if the project is not specified to expand beyond the current database design. The current layer of abstraction could be removed to lessen the program complexity. This is not done in our case as the program is supposed to be maintainable and able to be iterated on later.

6 Operate

In this section, we will outline our development process and delve into our testing methods aimed at ensuring stable gameplay.

6.1 Development process

The project is developed in Java we have used different IDEs, such as IntelliJ IDEA and Visual Studio Code, depending on the preferences of the individual developers in our team. This versatility in development environments applies to all the backend code as well as the database connections, which are implemented via the JDBC API.

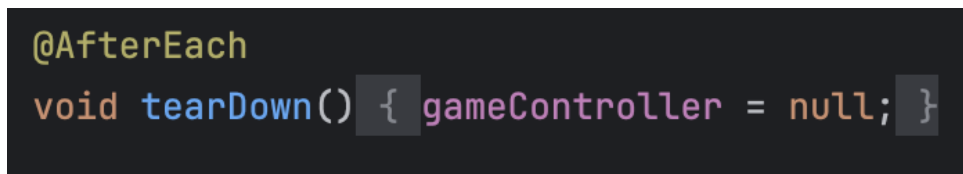
Maven is also employed to manage and import the GUI components, leveraging dependencies such as JavaFX to ensure a robust graphical user interface. The database is a relational database, created and managed using MySQL Workbench, which supports our backend services by providing a stable and scalable storage solution. Additionally, we used GitHub to collaboratively manage our project, utilizing feature branches for each new development. Once a feature was thoroughly tested and integrated, we merged it into the master branch. The development process can be categorized as an agile-inspired approach. This has allowed for concurrent progression in analysis, design, and implementation phases throughout the project, facilitating dynamic adaptation to changing requirements and rapid iteration.

6.2 Tests

To ensure stable gameplay, we've made JUnit tests to test the code functionality along with user tests to make sure the gameplay matches their expectations. This combined qualitative and quantitative optimizations to provide a better user experience.

JUnit Tests

For our JUnit tests, we set up the board which is predefined in a `@BeforeEach` function which we use for the tests. Moreover, we use an `@AfterEach` function. Its purpose is to clean up resources or reset the state that might have been modified during the test execution.



```
@AfterEach
void tearDown() { gameController = null; }
```

Figure 20: `@AfterEach`

The `gameController` object is set to null, indicating that it's being cleaned up after each test. This ensures that each test method starts with a fresh or properly initialized `gameController` instance, preventing any unintended side effects or dependencies between test cases.

The `@BeforeEach` function on the other hand is important for establishing a controlled test environment in our JUnit tests.

```

@BeforeEach
void setUp() {
    Board board = new Board(TEST_WIDTH, TEST_HEIGHT);
    GameController gameController = new GameController(board);
    for (int i = 0; i < 6; i++) {
        Player player = new Player(board, color: null, name: "Player " + i);
        board.addPlayer(player);
        player.setSpace(board.getSpace(i, i));
        player.setHeading(Heading.values()[i % Heading.values().length]);
    }
    board.setCurrentPlayer(board.getPlayer(i: 0));
}

```

Figure 21: Enter Caption

As the image shows, the function creates a Board with specified dimensions and a GameController for managing game logic. The function iterates six times to add six players to the game, each initialized with a unique identifier ("Player " + i). It finally sets the first player in the list as the current player. This standardizes the point of action initiation for tests, particularly those that depend on the sequence of play or specific player interactions. The @BeforeEach method sets up essential components and initial conditions for each test ensuring consistency.

Therefore, the @BeforeEach method is crucial for setting up a consistent and isolated test environment for each test function. It initializes the game board and sets up players and game controllers, ensuring that each test begins with a fresh, predictable state. This method helps prevent interference between tests by configuring essential components and initial conditions at the start of every test. It's about preparing the test environment rather than resetting it, which is the role of the @AfterEach method, responsible for cleaning up resources or states post-test execution.

Our JUnit tests are generally structured with an AAA-framework (Arrange-Act-Assert) to ease the setup and readability. In other words, we start by setting up the test (arrange), followed by performing the desired action (act) and wrapping it up by verifying the expected outcome (assert).

Down below, we've given an example of an actual unit test using the AAA-framework:

```

@Test
void testPlayerCollisionFastForward() {
    Board board = gameController.board;
    Player player1 = board.getPlayer(i: 0);
    Player player2 = board.getPlayer(i: 1);

    player1.setHeading(Heading.EAST);
    player1.setSpace(board.getSpace(x: 1, y: 0));
    player2.setSpace(board.getSpace(x: 2, y: 0));

    gameController.fastForward(player1);

    assertEquals(player1, board.getSpace(x: 3, y: 0).getPlayer(), message: "Player 1 should be at (3,0)");
    assertEquals(player2, board.getSpace(x: 4, y: 0).getPlayer(), message: "Player 2 should be at (4,0)");
}

```

Figure 22: Player Collision Test

In the example, we set two players on the board giving player1 an EAST direction to ensure it collides with player2 if player1 executes a moveForward card. The act step is executed when we move player1 forward followed by our assert tests to ensure we experience the expected outcome.

To double-check each step in the test, we use the debugger function in the IntelliJ IDE. This is particularly helpful in pinpointing the root cause of failures or discrepancies between expected and actual outcomes in our tests - but also in general when debugging our code. By leveraging the debugger, we can ensure that our tests are accurately assessing the behaviour of our game controller and detecting any deviations from expected defined behaviour. This is followed up by running the GameControllerTest class.

We also use the debugger in IntelliJ to find the code coverage metric for various classes, methods and lines:

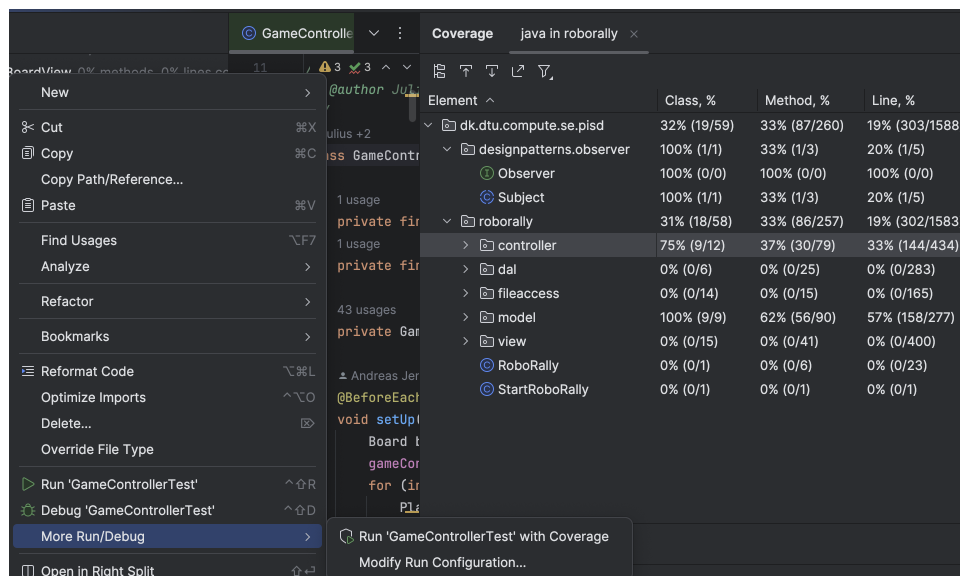


Figure 23: GCT coverage

As the picture illustrates, our tests cover 32% of the classes, 33% of the methods and 19% of the lines. However, we've decided to test the most crucial gameplay functions such as move logic, game-end conditions and vital action fields like checkpoints, antenna, conveyor belts and other features. Therefore, we haven't tested classes such as the AppController and similar no tests have been performed in the dal and fileaccess folder. We've likewise excluded GUI tests focusing on Model and in particular Controller classes. This is e.g. why we've tried avoiding implementing JavaFX in the GameController logic hence certain methods have been divided into multiple methods to make it more testable, for example, the endGame method.

It's an active choice to primarily focus on the GameController as a lot of the crucial logic is applied here along with the action fields. If each class in the controller had various tests, it could also have made sense to set up a more organized folder structure, however, since we primarily execute 1 to 2 tests per class or function we test, we've kept all tests in 1 GameControllerTest class.

Code coverage can be a useful way to find untested sections in a project. However, with that being said, it also has its limitations. A relevant example is that code coverage does not usually reflect whether all logic paths are tested or if the tests handle edge cases effectively. It also doesn't cover scenarios that could occur due to user interactions or integration with external systems unless specifically designed to do so. It doesn't account for the quality of assertions in our tests either. A section of code could be executed by a test without any effective assertions checking that the code behaves as expected. This even means bugs can still exist in "covered" code.

One way to discover and combat edge cases or unexpected scenarios is through user tests. We'll discuss that briefly in the next section.

Acceptance/user test

After our JUnit tests, we decided to do user tests with a small test group of people with varying coding experience some had little coding experience and some had none however no one had experience coding in Java. The entire test group were either friends or relatives. The purpose was to discover additional bugs and potential gameplay lifts.

To facilitate the user test, participants were provided with an instruction manual detailing how to play the game and its rules. The primary goal of the test was to ensure a seamless user experience, free from any unusual behaviours such as abrupt game interruptions, broken gameplay mechanics—like players disappearing from the board, being forced through walls, or conveyor belt malfunctions—that would detract from the expected game experience.

We had users play the game and think out loud to monitor their interactions, especially their use of command

cards and how intuitive the gameplay was for them. While most aspects of the game functioned smoothly, we identified an issue where the antenna did not correctly calculate the starting player in a four-player scenario—our original JUnit tests only covered two-player scenarios. Upon discovering this bug, we revised the scenario and corrected the calculations. We also updated our JUnit tests to include a scenario with up to six players to ensure more robust testing coverage.

The GUI was an important element for user interaction, though not the primary focus of this project. It was important for the GUI to be user-friendly, enabling players to navigate and control their characters without needing additional guidance. Key functions like starting a new game and executing command cards needed to be straightforward. Additionally, making the action fields easily recognizable was crucial, as it helped users quickly grasp the different functionalities of each field.

The test group found the GUI intuitive and easy to navigate, successfully using the command cards and completing games without issues. However, they noted that the GUI appeared outdated, which could affect player interest if the game were published online. While updating the GUI goes beyond the current project scope, it is an important consideration for future development of the MVP.

During our manual testing, a colour-blind tester reported difficulty distinguishing between player colours, sometimes confusing his player position with others. Initially, we hadn't considered colour blindness, but this feedback led us to change the player colours to accommodate those with color vision deficiencies, enhancing accessibility and gameplay clarity.

By combining automated and manual testing, we ensured that our game operates correctly across different scenarios and provides a seamless, error-free gameplay experience. However, there are limitations to our current testing strategies. We will explore potential improvements and alternative testing methods in the next section.

6.3 Evaluation

Due to our JUnit tests, user tests and continuous use of debugging, we've been able to discover and fix various bugs throughout the gameplay, e.g. the antenna issues but also smaller glitches such as the checkpoints weren't being counted. This ensures a stable user experience.

In evaluating the testing approaches used for our RoboRally game project, we recognize that while JUnit and user testing frameworks are in place to assess various aspects of the game, we could've implemented other tests to cover additional aspects of the gameplay.

An example is that JUnit tests look at each unit and not how the units interact with each other. Therefore, we could've considered an integration test. Integration testing would involve combining individual software modules and testing them as a group. This is crucial because it helps identify issues that occur when units interact, which might not be captured in unit testing.

For instance, an integration test could simulate a sequence of game turns involving multiple players to ensure that the game logic correctly handles turn orders, rule enforcement, and game state transitions. This would e.g. involve testing interactions between the GameController, Player, and Board classes, particularly focusing on how changes to the game state in one module affect the operations in others.

This approach aids in identifying and fixing the inter-module bugs. This would enhance confidence in the stability and reliability of our game, ensuring a smoother user experience.

7 Guide

7.1 Downloading the zip file from github

Using this link, download the zip file from GitHub, but be aware that the file DBAcces is not included and should be added by yourself. Link to Github The zip file provided has the DBAcces file included.

7.2 Installing the database

To install the database, first log into MySQL workbench and set up a connection. After creating the connection, go in and open a window for executing SQL queries. Here type:

```
1 create database PISU;
```

And press this button:

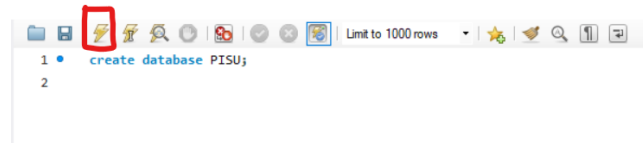


Figure 24: Create Database

This should create an empty database. Now you are ready to move on to connecting the database to RoboRally.

Following this path:

roborally\src\main\java\dk\dtu\compute\se\pisd\roborally\dal\DBAcces.java

you'll find 5 empty variables that need to be updated with your database information. Here, you will need to add the database user (often 'root'), the password you set during database creation, the database name (for example, 'PISU'), the port (usually 3306), and the host (commonly 'localhost'). After updating these details, save and close the file. The filled-in variables should reflect the specific credentials and settings of your database setup:

```
1 package dk.dtu.compute.se.pisd.roborally.dal;
2
3 public class DBAcces {
4     public static final String USER = "root";
5     public static final String PASSWORD = "YOUR PASSWORD";
6     public static final String dbName = "PISU";
7     public static final int Port = 3306;
8     public static final String Host = "localhost";
9 }
10
```

7.3 Building the jar file

Now you should open a command line of any kind. To illustrate git bash will be used. First, navigate to the directory where you have RoboRally and enter it. Typing the following:

```
mvn package
```

Followed by:

```
cd target/
```

Now you should have built the project into a runnable .jar file.

7.4 Opening and starting the game

To start the game, run the command while in the target folder:

```
java -jar roborally-1.1.3a-SNAPSHOT.jar
```

This should open up a window (You might have to alt+tab to see it), called RoboRally.

Here press on file and then new game to start your first game:

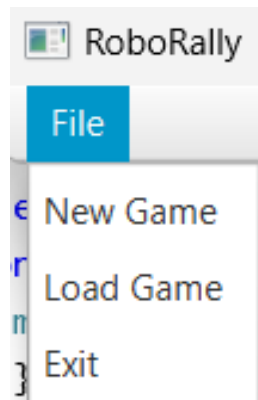


Figure 25: New Game

Afterwards, use the dropdown menu to select the board and amount of players. You are now ready to play. The following images describe the most important things to remember when playing:

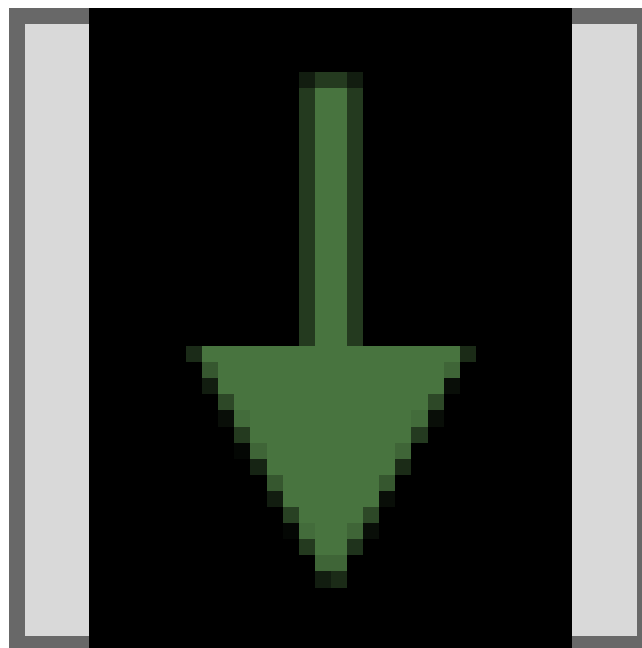


Figure 26: Conveyor belt. This space pushes you in the direction of the arrow after each register has been completed.



Figure 27: Checkpoints. These should be touched in order of their numbers, and after touching all you win.



Figure 28: Reboot. This is the space were your robot reboots after dying.

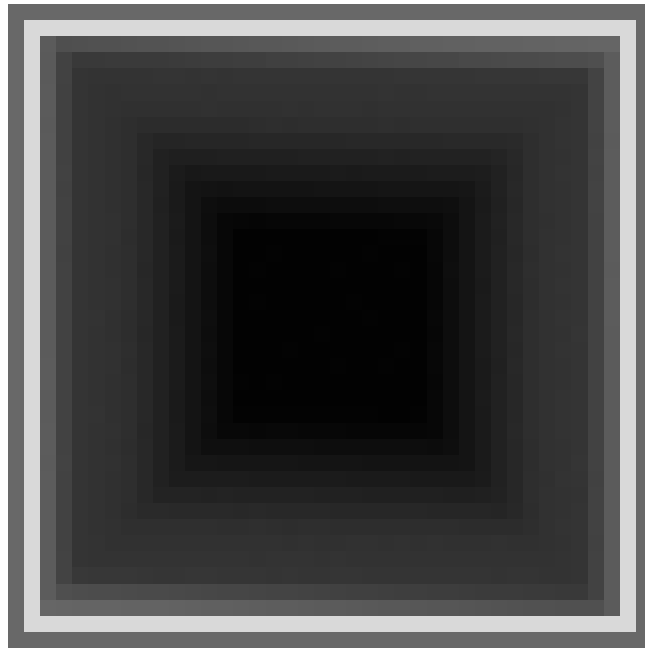


Figure 29: Pit. This is where your robot dies if it falls into it.



Figure 30: Wall. This is an impassable wall that you cannot pass through.



Figure 31: Antenna. This is the space that dictates the order each player gets to move in.

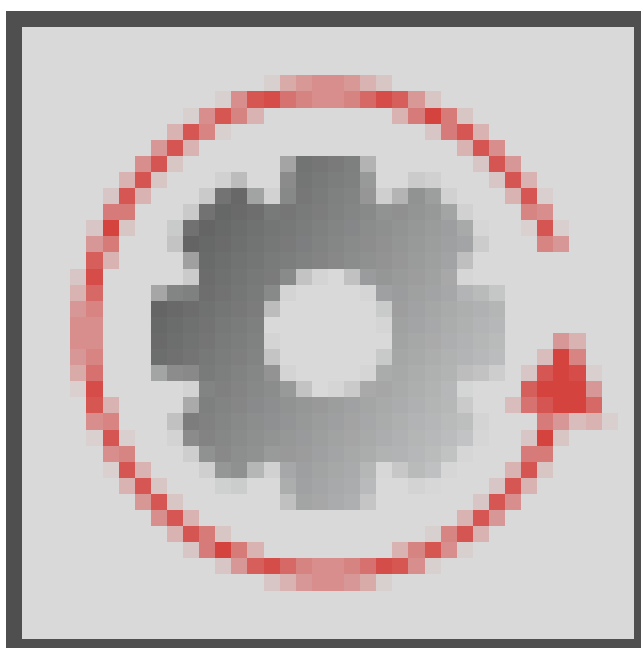


Figure 32: Gear. This gear turns your robot to the left when it stands on it

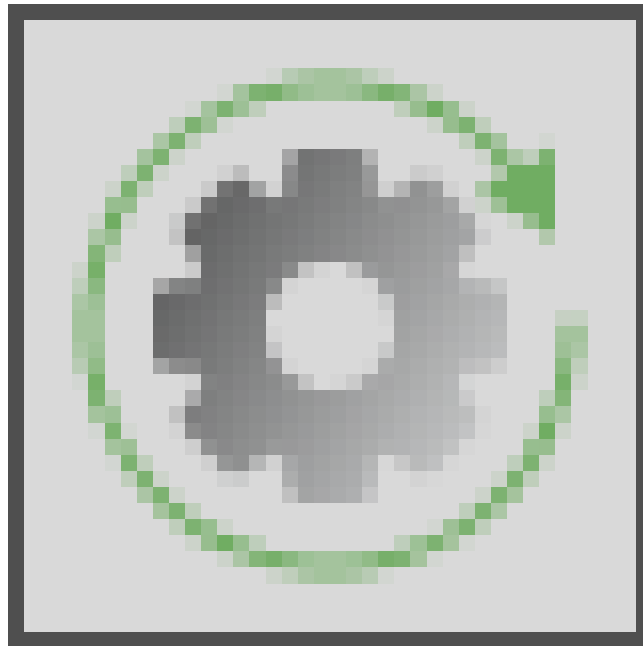


Figure 33: Gear. This gear turns your robot to the right when it stands on it

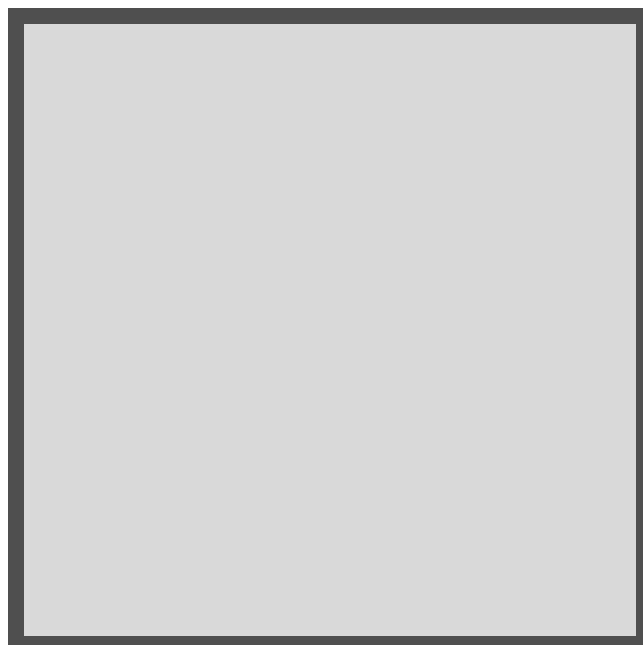


Figure 34: Space. An empty space, that a robot can freely move across



Figure 35: Indication of what each thing in the tabs are.

- **Blue** = tabs. Here you switch between each player.
- **Orange** = Program. Here is where you drop off your cards that you want your robot to carry out.
- **Green** = Command Cards. This is where you will be assigned your cards that need to be dragged to the program to be ran.
- **Brown** = Finish programming. This is where you press when you are done programming all the robots.
- **Light blue** = Execute program. This runs all programs of all players at once.
- **Yellow** = Execute current register. This runs the next step of the current players program and is used to step through every move of every robot.

Now you are ready to play the game. GLHF! (Good Luck Have Fun!)

7.5 Creating the java doc

Go to the directory of roborally again, and write:

```
mvn javadoc:javadoc
```

Now in File Explorer locate the target folder, and inside that there is a folder called site, with an index.html inside. Run this in your browser and it will open up the java doc.

8 Glossary

The glossary contains essential terms for understanding both the game rules and its scope.

Main Items

- **Robot:** The programmable entity manipulated during gameplay.
- **Player:** The individual programming and controlling the robot.
- **Board:** The playing surface comprises diverse spaces where the game unfolds.
- **Start Board:** This distinct section, apart from the main board, houses the black and white gears for players to position their robots at the game's outset. Additionally, it incorporates the priority antenna.
- **Spaces:** Defined locations on the board designed to accommodate various objects.
- **Checkpoint:** An item placed on a space that, when traversed by a player's robot, serves as a victory condition by requiring the completion of all checkpoints on the board.
- **Checkpoint token:** The token awarded upon reaching a distinctive checkpoint.
- **Pits:** A space where a robot falls and is required to reboot, before continuing.
- **Priority Antenna:** A space determining player turn priority based on proximity; the closest player starts. This space is impassable to other objects..
- **Walls:** Impenetrable spaces preventing robot movement and blocking lasers.
- **Blue conveyor belts:** Spaces with directional arrows; landing on them moves a player in the indicated direction.
- **Push panel:** A space that will push a robot in the direction of the panel, if the number on the panel corresponds to the current register number.
- **Gears:** Spaces either rotating a robot 90 degrees (green for right, red for left) or serving as starting locations.
- **Start fields/spaces:** Spaces marked by gears where each player begins; the youngest player selects their starting spot.
- **Energy Spaces:** Spaces hosting energy cubes, pickable only as the last action of a turn, disappearing permanently after collection.
- **Energy cubes:** Pickable and storable resources for each player.

- Energy Bank: A collection of energy cubes, that are given out when a player lands on an energy space.
- Register: A player's programmable sequence with 5 slots executed in order, situated on the player mat.
- A robots program: The whole register that the robot will carry out.
- Player mats: Each player's own register and programming deck.
- Programming deck: A set of programming cards that can be played into the register.
- Programming card: There are different kinds of cards that each make a player's robot do something different. These are distributed randomly between players at the start.
- Phases of the game: There are 3 different phases in the game. The programming phase is the phase where each player is assigned a programming deck and creates their register of cards with instructions to be carried out. After this, the activation phase starts where instructions are carried out, in the order of players, with each player carrying out one card before the next player gets to do an action. After all cards/instructions are carried out the phase ends. The upgrade phase occurs at the beginning of rounds, where the players can spend energy cubes in exchange for upgrades for their robots.
- Upgrades: There are two types, permanent and temporary.
- Reboot token: A reboot token is an item that can be placed in a specific space within the game. This allows a player to restart or 'reboot' from that location
- Lasers: Board: Lasers are a kind of space where a laser is present and facing a direction. Moving into the line of the laser at the end of a register means taking one tick of laser damage which is shown as a SPAM card. A space can have up to 3 lasers on it.
- Damage cards: There are four different kinds of damage cards. The SPAM card deals damage. The TROJAN HORSE, makes the robot take two SPAM cards. The WORM card immediately requires a reboot for the player to be able to move again. The VIRUS card is a card that makes others around you in a six-space radius pick up the VIRUS card.
- Racing Courses: Courses are premade boards made up of different types of spaces these courses can be put together to create even bigger courses.

Functions

- Move robot: Moves a robot according to what the programming card states.
- Move card: Move a card from the player's deck into the register.
- Take damage: Taking damage in the game involves picking up damage cards and adding them to your discard pile. When you shuffle your deck, these damage cards are incorporated into your current deck and hand, affecting your gameplay.
- Reboot robot: When rebooting a robot, the player must add two SPAM cards to their discard pile. Any cards remaining in the player's register are not executed and are instead moved to the discard pile. The player must then skip programming for the current round and wait until the next round to program again. The robot is moved to a reboot token that has been placed on the board; if no reboot token is available, the robot returns to the starting position. After rebooting, the robot can face any direction.
- Purchase upgrade: Having the ability to buy upgrades to a player robot.
- Win game: Touching all checkpoints and acquiring all checkpoint tokens.

9 Conclusion

The development of RoboRally succeeded in creating a single-computer version closely resembling the original board game. The requirements outlined in the conceive phase, have been evaluated, and the most important requirements have been satisfied. Additionally, we've tested gameplay with JUnit, acceptance and user tests to ensure a proper, functional experience.

The implementations made during this project are missing certain parts of the original game, that were deemed unnecessary or too complicated to implement in the given time frame. This translates to all features deemed a

"must" or "should" have been implemented, while some "could" have been left out, together with all "won't". The "could" and "won't" include features such as lasers, energy cubes and robot upgrades.

The database is designed according to relational database principles, ensuring it can store all data related to a RoboRally instance for saving and loading purposes. Additionally, the GUI has been enhanced to improve clarity and overall quality.

By leveraging Maven as a build tool, the game is compiled into a standalone JAR file, inclusive of all necessary dependencies, enabling it to run independently.

10 References

10.1 Website for deck size:

Roborally Website

10.2 Roborally rules

Rules