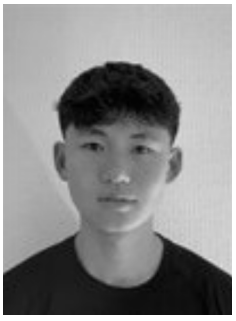
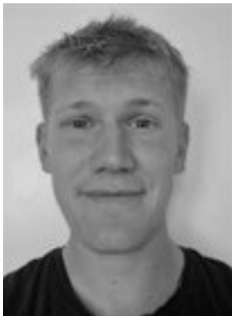
Danmarks Tekniske Universitet

DTU

Roborally

Group 3

Authors

			
Martin Dahl Lund s235454	Anton Fu Hou Dong s235460	William Wegener Kofoed s235451	Jonas Woetmann Larsen s235446

1.Table of contents

2. Responsibilities for the report

Diverse	**Names**
Introduction	Martin
Conclusion	All
Proofreading	All
Handbook - Installation of the game	Anton
Handbook - How to play the game	Jonas
Glossary	Martin
Javadocs (The documentation)	All

Analysis	**Names**
Taxonomy	Jonas
Requerments - FURPS+	William
Baseline - GUI og MVC	William
Database requirements	Martin
Use cases	Martin
Domain model, State diagram and activity diagram	Anton

Design	**Names**
Softwaredesign	
- Observer-design patterns	Jonas
- System sequence diagram	Martin
- Sequence diagram	William
Databasedesign	Anton

Implementation	**Names**

Model-classes	William
The flow of the game (game controller)	Jonas
Exceptions	Anton
Checkpoints + end of game	Martin
JSON	Anton
Database connection	Anton
Observer	Jonas
Recursion	Anton
Javadocs	All

Operate	**Names**
Development process	Martin
Test	Anton + Jonas

3. Introduction

We, Group 3, have programmed a version of the 1994 board game *RoboRally* as an assignment in our course: Project in Software Development 02362

In this report, the analysis, design and implementation will be shown by the board game *Roborally* which is programmed in IntelliJ. This project also includes a Database implementation which is done in MySQL.

We have been given the board for Roborally by our teacher and it is our responsibility to further develop on the project.

Roborally is a board game for 2-6 players, the game was designed by Richard Garfield in 1994. The purpose of the game is to move robots around the board and collect all the checkpoints in a certain order. The player that collects all the checkpoints first wins the game.
The winner will be decided by the player who collects all checkpoints in the correct order.

We are following the rules given by our teacher, the rules can be seen in (Bilag 1 or Sources).

4. Analysis

4.1 Taxonomy

Nouns	
Object	**Glossary**
Game	
Board	
Player (Robots)	
Space	The fields on the Board
Card - Programming Card	**Programming Card:** Cards that program the path of your robot
Checkpoints	The players targets (Tokens are acquired once standing on the Checkpoints which are associated with spaces)
Actions	Various actions based on the space the robot is currently standing on
Token - Checkpoint Token	**Checkpoint Token:** Required to win - acquired by moving to Checkpoints

Verbs/Activity	
Verb (to..)	**Activity**
Start game	
Save game	
Load game	
Stop game	
New game	
1) The programming phase	
Draw cards	A player draws 8 random programming cards

Program robot (the robot moves)	The player plans the route the robot is going to move
2) The activation phase	
Activate robot - Move	The robot moves according to the program the player has made for the robot - Moves forward - Fast forward - Turn left - Turn right
Execute Field action	Executes the actions based on the robots location (different spaces have different actions)
Get checkpoint token	The player lands on a checkpoint token

4.2 FURPS+ requirement specification

The specification of requirement for roborally is based upon FURPS+ which stands for Functionality, Usability, Reliability, Performance Supportability and Expansion options.

Functionality:
in-App Functionality:
Running Roborally on a board with different obstacles.
Providing cards that command movement directions (left, right, up, down).
Implementing programming and activation phases.
Incorporating checkpoint tokens to win the game.
Loading board from a JSON file, load and save game from/to database.
Running Roborally on a board with different obstacles, including tiles for field actions.
Providing cards that command movement directions (left, right, up, down) and activate field actions.
Game Functionality:
Ability to stop the game.

Usability:
Roborally must be easy to use for every age group.
Roborally must have an intuitive user interface.
The documentation of roborally must be clear and unambiguous.
Roborally must show all information on the state of a game on a GUI

Reliability
Roborally must run smoothly without crashes or errors.
Roborally must be able to read the cards correctly in programming-fase.

Performance
Roborally the game must run smoothly and respond quickly to user actions.
Roborally must be able to act fast on the instructions.

Supportability
Roborally should be easy to repair and maintain.
Roborally must be compatible with different devices.
Roborally must provide documentation for players and developers.
Roborally must have a long life span.

Expansion options
Roborally should have a scalability for potential future updates or expansions.

4.3 Baseline - GUI and MVC

This project is based on a program skeleton of Roborally and functional GUI, which was handed to us at the start of this project. The program skeleton and GUI has some limitations. The Skeleton model is modeled after the model-view-controller example, which was our starting point for this project. This gave us some problems in what was able to be implemented due to the functionality of the GUI and the player cards that were already implemented in the handout.

Model:
- game logic
-data structures

view:
-GUI Components
-Visual Representation

Controller
-User Input Handling
-Event Handling
-Communication

4.4 Database requirements

The game must be able to get saved in a database. Following requirements are saved in the database

- Board
- Players position
- Checkpoints
- Walls
- Gears

4.5 Use cases

Use Case 1: Start game
Use case description: Player runs the program, a window pops up where the player pushes the button “Start game” og chooses between 2 or 6 players. A robot appears for each player on the game board.

Precondition: There are between 2 or 6 players ready to play the game. The code is up to date.
Postconditions: Robots will appear on the board for each player and then each player will be dealt 8 cards and out of those cards each player has to pick 5 cards.

Use case 2: Program robot
Use case description: A player draws 8 cards, out of those 8 cards the player picks 5 cards. The 5 cards are dragged to the player's player mat. This determines the robot's path.
Preconditions: A player has started the game and filled in how many players there are in the game.
Postconditions: The robot will move accordingly to the cards the player has picked out.

Use case 3: Save game
Use case description: A player clicks on file and after save game. The game gets saved in a database. It saves the board, checkpoint, walls and gears.
Preconditions: The game has to be started, the game can be saved anytime in the game.
Postconditions: After the game has been saved, the game is now ready to be closed.

4.5.1 Fully dressed use case

Name	Activate robot
ID	1
Description	The game goes into the activation phase
Actors	Player
Triggers	When all players have been through the programming phase
Pre-conditions	The player needs at least one programming card to complete the robots path
Post-conditions	All of the programming cards have been executed
Main flow	1. The player programs the robot 2. Player activates the robot 3. Step 1 and 2 will repeat until all the players have been through the programming phase 4. Step 1, 2 and 3 will repeat until a winner i found 5. Game will end

4.5.2 Use case diagram

This use case diagram shows the interaction between the players or users and the game Roborally.

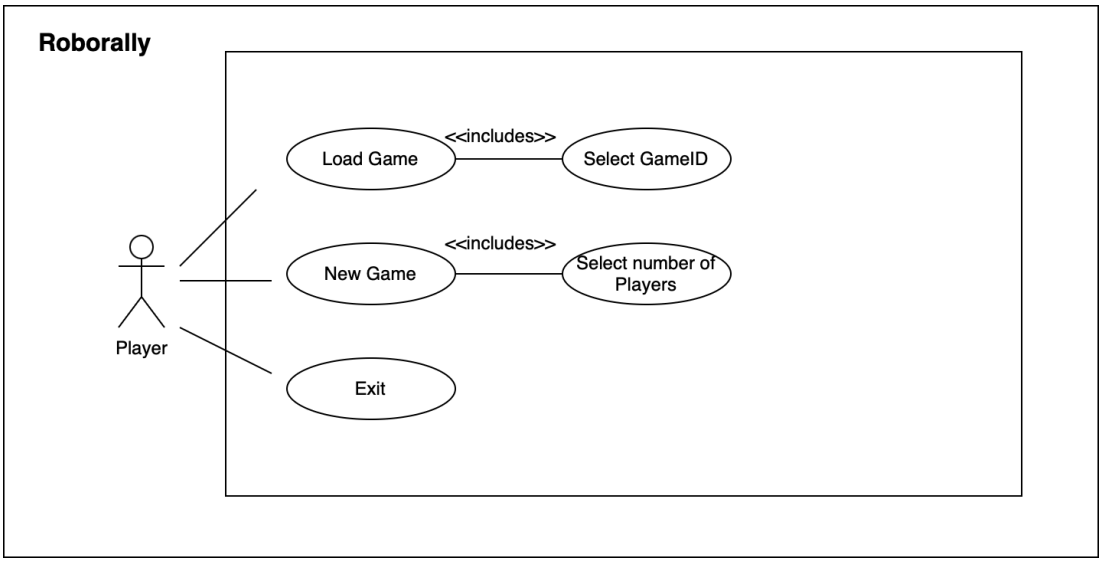


Figure 1: Use case diagram

4.6 Domain model, State diagram and Activity charts

4.6.1 Domain model

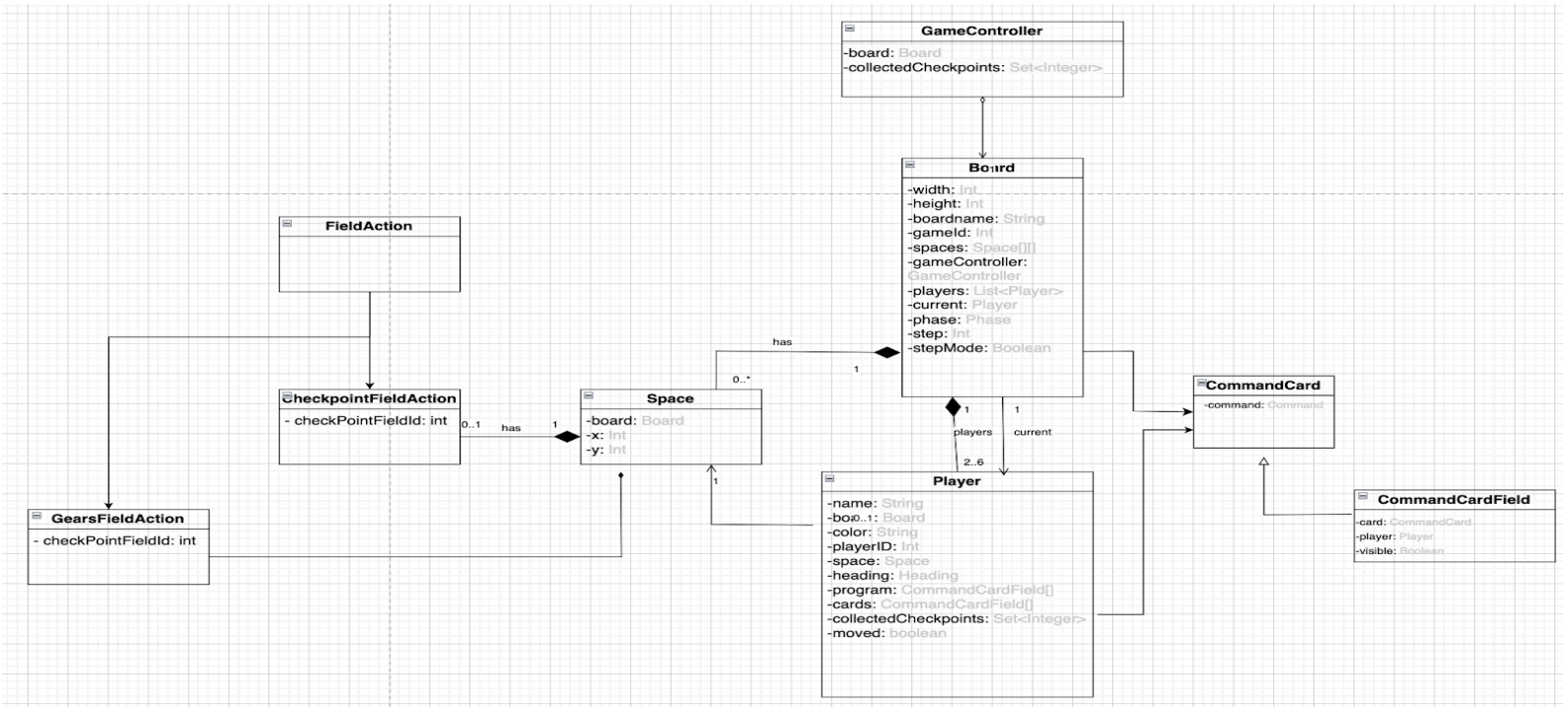


Figure 2: Domain model

This UML diagram outlines the structure of Roborally.

1. GameController
    - The GameController class is the central class that controls the overall flow of the game
2. Board
    - The board class represents the game board.
3. Player
    - Each player is characterized by their own robot.
4. CommandCard
    - The CommandCards provide commands for the robot
5. Space
    - The space represents one area on the game board, such as a fieldaction or the coordinates of the space on the board
6. CheckpointFieldAction
    - Every player must reach all the checkpoints in the specified order to win the game.

4.6.2 State diagram

The state diagram describes how the game object Roborally changes state.

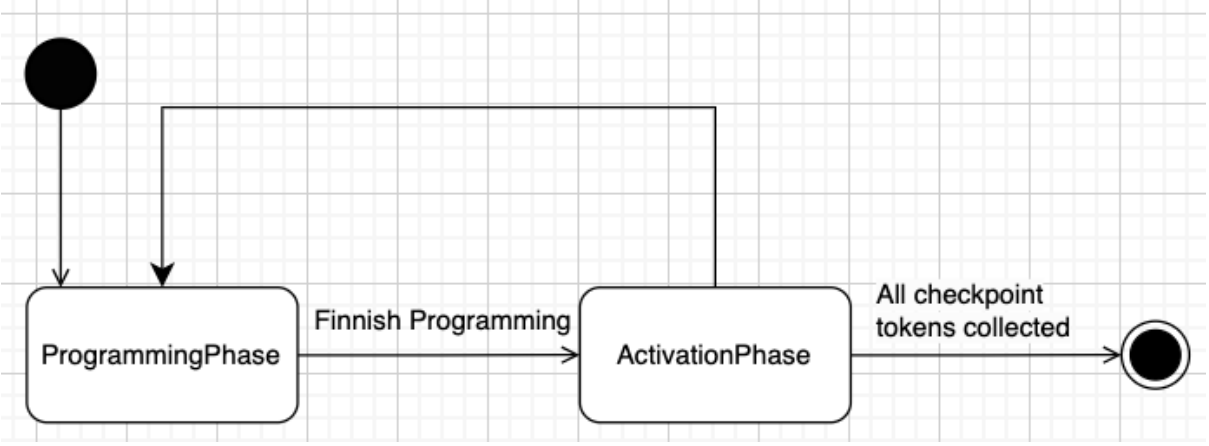


Figure 3: State diagram

Roborally contains several different states

1. ProgrammingPhase
    - This is the initial state where players actively dictate their robots' movements and actions on the game board.
2. ActivationPhase
    - Once programming is completed, the game transitions to the ActivationPhase. This phase involves the execution of the actions programmed in the ProgrammingPhase.

4.6.3 Activity charts

The activity diagram describes the activities that occur in a game of Roborally.

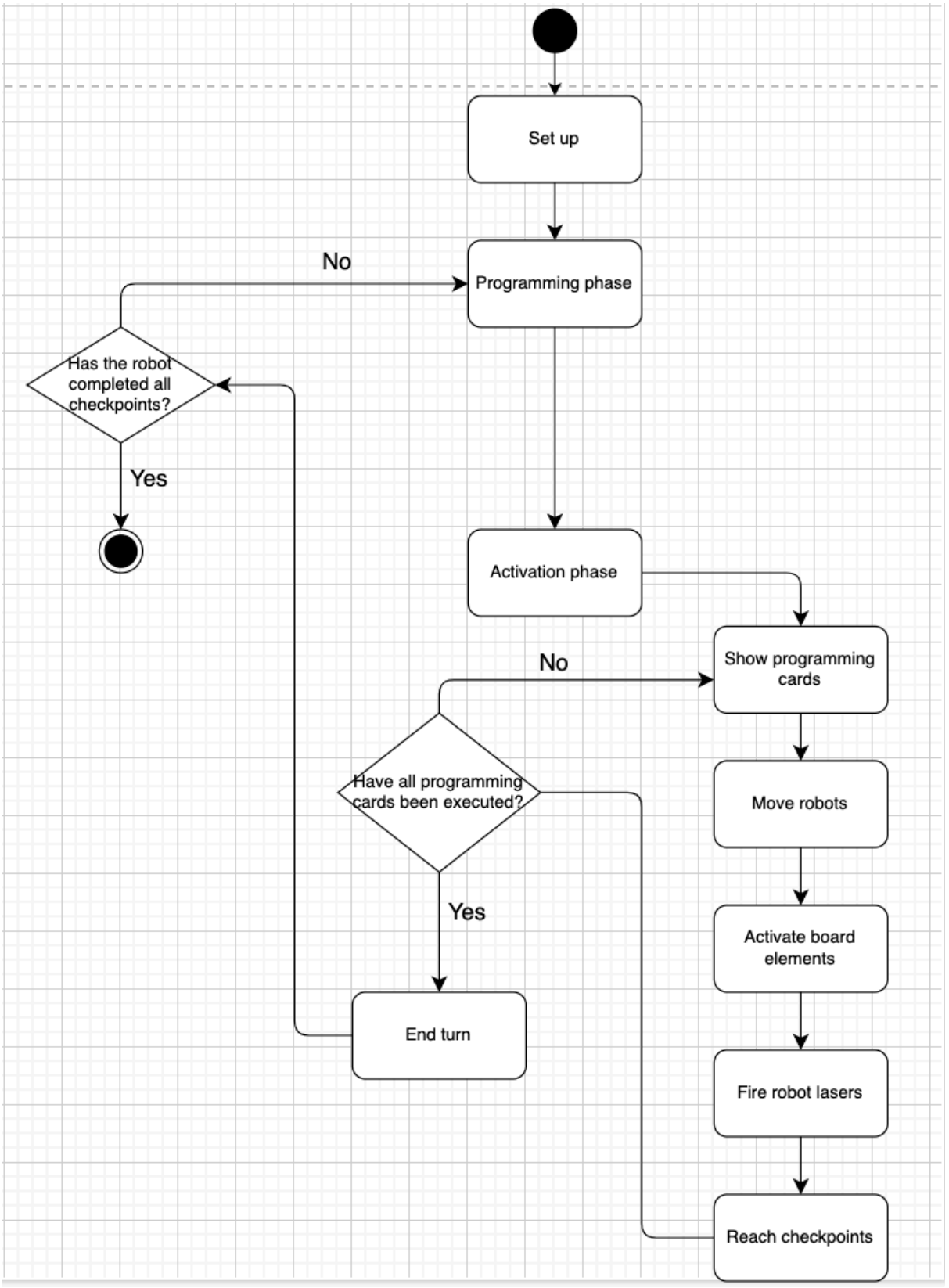


Figure 4: Activity charts

This UML diagram is a flowchart, which outlines the sequence of game phases and decisions that players make during a turn.

1. Set up
    - The game begins with a setup phase, where the board is prepared, the robots are place and programming cards are assigned
2. Programming phase
    - Players plan their moves for their robot for the turn.
3. Activation phase
    - The planned actions from above will be executed in this phase
        - Robots are moved on the board according to the programming cards
        - Certain fieldactions or walls on the board will activate, which might affect the game state or the robots
        - Robots attempt to reach certain designated checkpoints on the board.
4. Decision points
    - During the activation phase, there will be checks to see if all the programming cards have been executed.
    - There is also a decision point to check if a robot has completed all checkpoints, which will signify the end of the game.

5. Design

5.1 Software design

5.1.1 Observer design pattern

This observer design pattern acts as a way to update the visual parts of the game or the view-layer. This means that when one of the subjects in the game changes its state, the one-to-many relationship with the subjects will make sure the parent class Subject will notify the Observer to update the GUI.

The diagram below visualizes the observer design pattern. The concrete subjects inherit from the abstract class Subject, as well as the concrete observers inherit from the Observer interface through implementation of another interface (ViewObserver) in the view classes.

In the program itself, a subject, for example a player, will inherit the abstract class ‘Subject’ connecting it with an observer. This will observe the player's state, updating if the player's position or heading changes, notifying the observer that it needs to update and making sure the visual part in the view classes update as well, so the player's location and direction in the code always correlates to where the player’s stands on the board in the GUI.

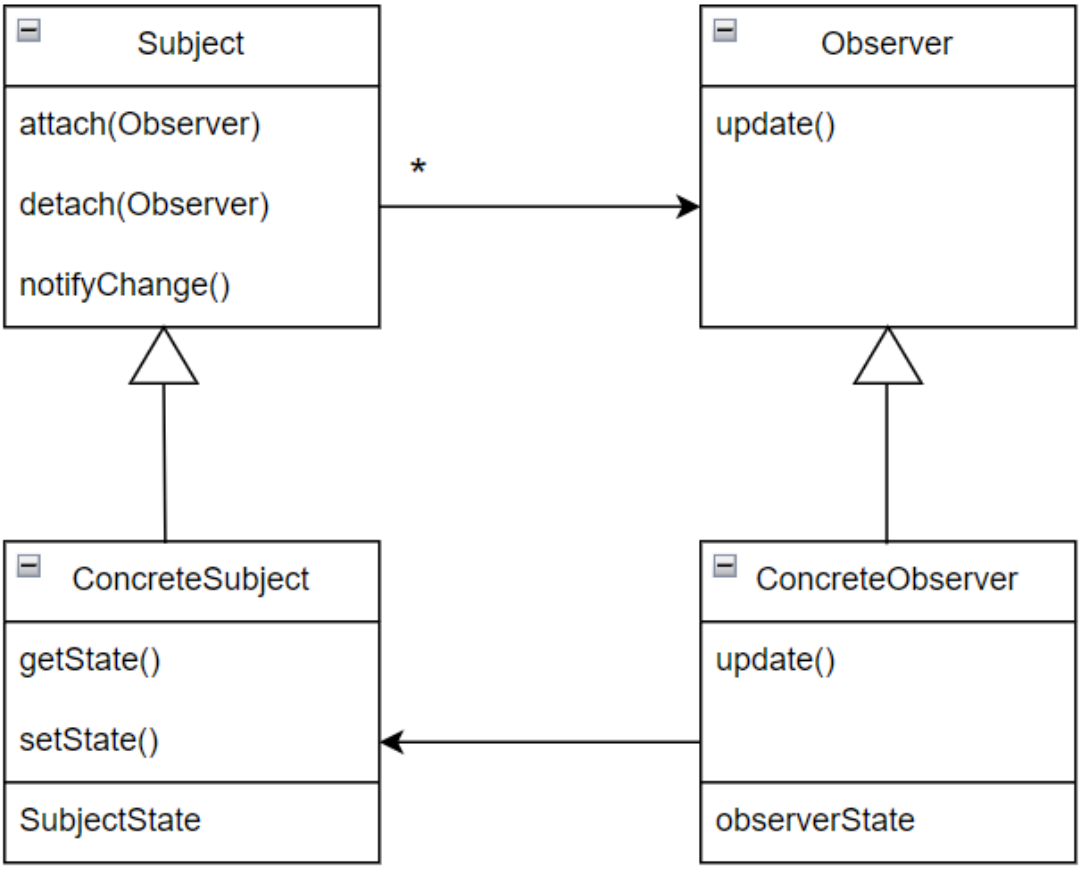


Figure 5: Observer design pattern

5.1.2 System sequence diagram

The diagram below describes how the users interact with the system.

Player actions:

- New game: The player starts a new game.
- Select number of players: The player chooses the number of participants.

Loop:

- Programming phase: The player programs the robot by the cards the system has dealt to them.
- Finishing programming: The programming ends and goes into the system.
- Activation phase: The system activates the programmed route for the robot.
- Execute Program: The robot follows the programmed route:
- Checks for checkpoint: The system checks if the player has landed on a checkpoint.

End of the game:

- Finds a winner: The system checks if a player has collected all the necessary checkpoint, if the player has collected all the checkpoint, the player wins.
- End game: Game will end and close down.

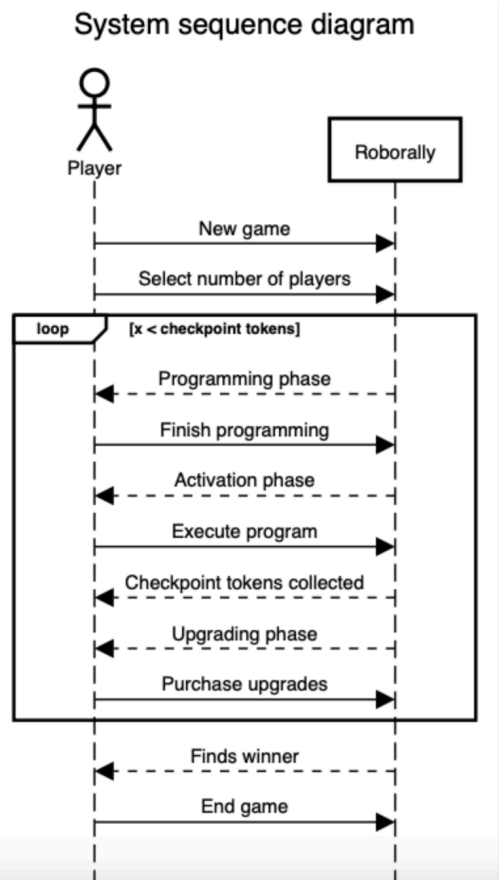
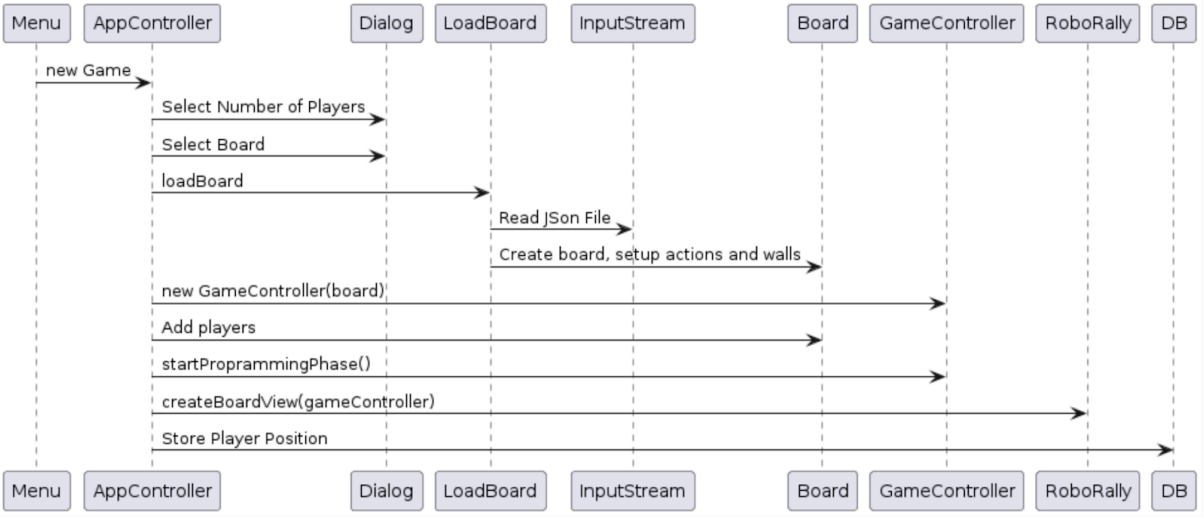


Figure 6: System sequence diagram

5.1.3 Sequence diagram

This Sequence diagram is an example of what happens when you start a new game in roborally.



5.1.4 Navigation diagram

The navigation diagram shows the step by step execution of the program from a user perspective.

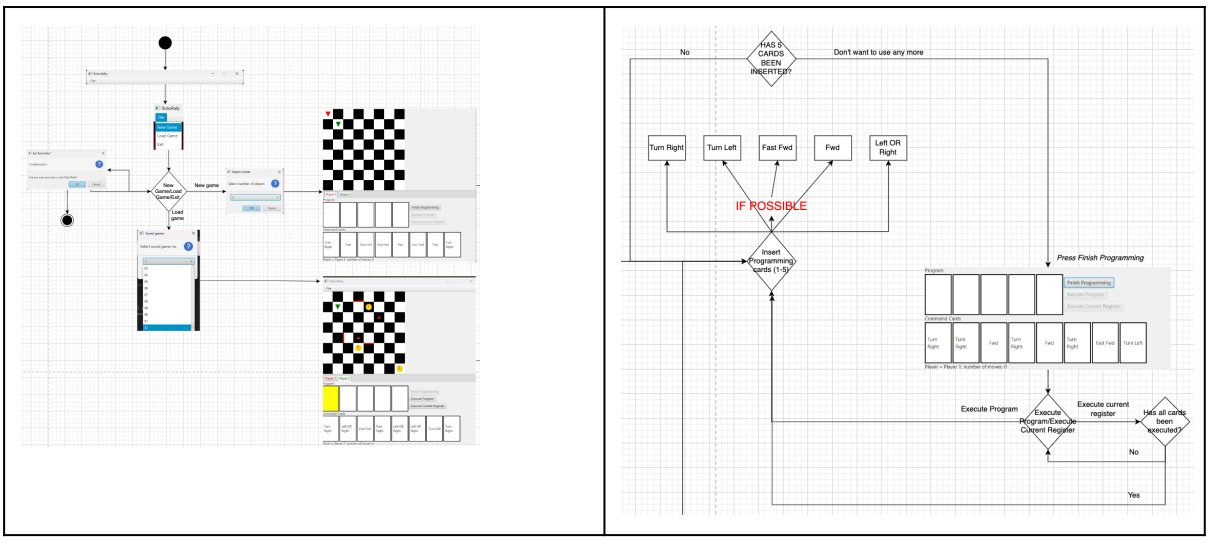


Figure 7: Navigation diagram

5.2 Database design

This ER diagram shows the relations between the MySQL database tables.

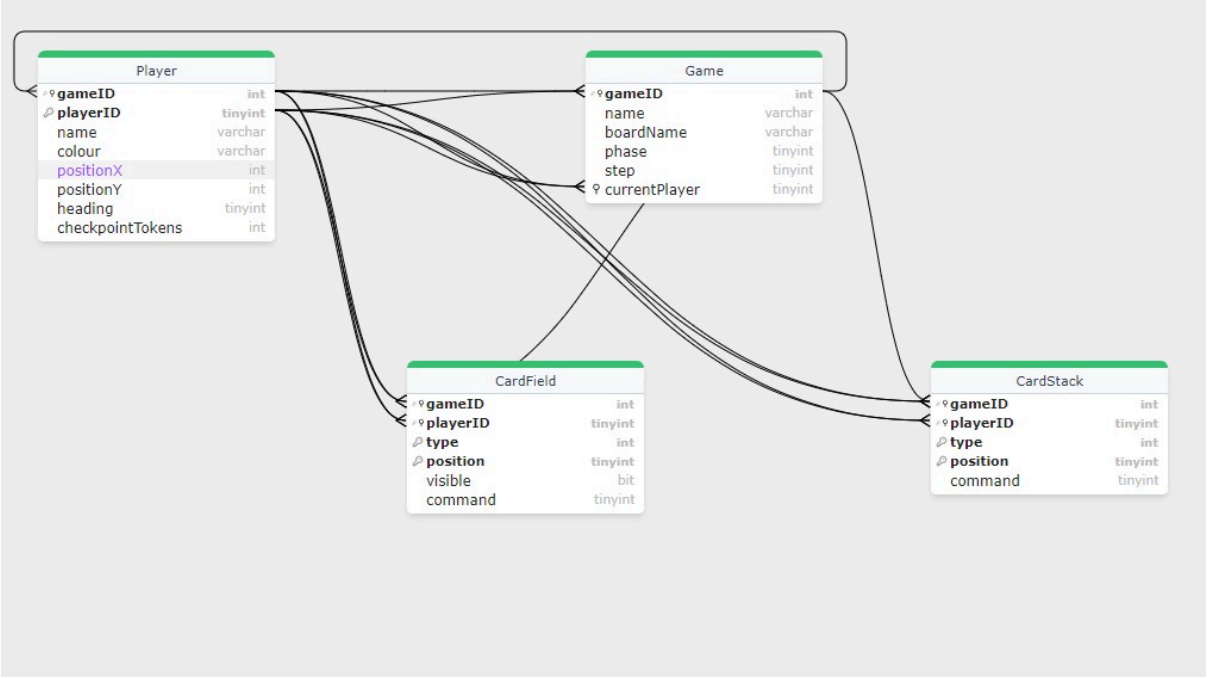


Figure 8: ER diagram of the database, created from SQL flow.

Entities and Their Attributes:

Game

- gameID (Primary Key): Unique identifier for each game.
- name: Title of the game..
- phase, step, currentPlayer: Attributes controlling the flow of the game, indicating the current phase of the game, the step within that phase, and which player's turn it is.

Player

- playerID (Primary Key): Unique identifier for each player.
- gameID (Foreign Key): Links to the Game table to indicate which game the player is part of.
- name, colour: Player's name and color representation in the game.
- positionX, positionY: Coordinates of the player's position on the game board.
- heading: The direction the player is facing.
- checkpointTokens: Number of tokens the player has collected.

CardField

- gameID, playerID (Foreign Keys): Identify the game and player associated with specific card actions.
- type, position: Type of card and its position in gameplay.
- visible, command: Indicates whether the card's effects are visible and what command/action it represents.

CardStack

- gameID, playerID (Foreign Keys): Link the card stack specifically to a game and a player.
- type, position, command: Attributes similar to CardField, detailing the stack's characteristics and the commands it issues.

6. Implementation

6.1 Model - classes

In Robally is there Implemented the following classes. Some of the classes were from the domænemodel and some were Implemented later after need for a new class.

- Game controller
    - board: This field is a static reference to the Board object associated with the game. It allows access to the game board from other parts of the program.
    - collectedCheckpoints: This field is a static set containing the identifiers of checkpoints collected by players during the game so you can win the game.

- Board
    - width: This field represents the width of the board.
    - height: Similar to width, this field represents the height of the board.
    - boardName: This field stores the name or identifier of the board.
    - gameId: This field represents the unique identifier of the game associated with the board.
    - spaces: This field is an array representing the spaces or tiles on the board.
    - players: This field represents the players on the board.
    - current: This field represents the current player whose turn it is on the board.
    - phase: This field represents the current phase of the game on the board (e.g., initialization, movement phase).
    - step: This field represents the current step or turn in the game on the board.

    - stepMode: This field indicates whether the game is in step mode.

- Player
    - name: This field stores the name of the player.
    - color: This field stores the color associated with the player.
    - playerId: This field stores the unique identifier of the player.
    - space: This field represents the current space (or position) of the player on the board.
    - heading: This field represents the direction the player is facing.
    - program: This field represents the program or sequence of commands the player plans to execute.
    - cards: This field represents the cards held by the player.
    - collectedCheckpoints: This field is a set containing the identifiers of checkpoints collected by the player during the game.
    - moved: This field indicates whether the player has moved during the current turn.

- CommandCard
    - Command: This field represents the command associated with the command cardt it stores an instance of the Command class or one of its subclasses.

- commandCardField
    - player: This field represents the player associated with the command card field.
    - card: This field represents the command card placed on the field.
    - visible: This field indicates if the command card on the field is visible.

- Space
    - board: This field represents the board that the space belongs to.
    - x: This field represents the x-coordinate of the space on the board.
    - y: This field represents the y-coordinate of the space on the board..
    - hasGear: This field indicates if the space contains a gear item.

- CheckPointFieldAction
    - checkPointFieldId: This field represents the ID of the checkpoint field.

- GearsFieldAction
    - heading: This field represents the direction in which the gears force the robot to move.

6.2 The flow of the game

The game is initialized by running the main method in the RoboRally class. In this instance that is done by running the StartRoboRally class for a small workaround involving a quirk in the Open JavaFX Project Launcher. This launches the appController, gameController and the menuBar. The first two are responsible for controlling the application and the game itself while the latter shows the menu where the player chooses to either start a new game or load a prior game. (Additional choices

can be made here). When the game is loaded, three different methods will control the flow of the game.

The startProgrammingPhase class will be initialized when starting the game automatically. This method will set up the programming phase making it possible to view the cards and play them.

```
public void startProgrammingPhase() {
    board.setPhase(Phase.PROGRAMMING);
    board.setCurrentPlayer(board.getPlayer( i: 0));
    board.setStep(0);
    for (int i = 0; i < board.getPlayersNumber(); i++) {
        Player player = board.getPlayer(i);
        if (player != null) {
            for (int j = 0; j < Player.NO_REGISTERS; j++) {
                CommandCardField field = player.getProgramField(j);
                field.setCard(null);
                field.setVisible(true);
            }
            for (int j = 0; j < Player.NO_CARDS; j++) {
                CommandCardField field = player.getCardField(j);
                field.setCard(generateRandomCommandCard());
                field.setVisible(true);
            }
        }
    }
}
```

When the finish button is pushed the finishProgrammingPhase will end this phase and lock in the choices

```
public void finishProgrammingPhase() {
    makeProgramFieldsInvisible();
    makeProgramFieldsVisible( register: 0);
    board.setPhase(Phase.ACTIVATION);
    board.setCurrentPlayer(board.getPlayer( i: 0));
    board.setStep(0);
}
```

when either of the execute buttons is pushed the program will execute either immediately or in steps but the result will be the same

```
public void executePrograms() {
    board.setStepMode(false);
    continuePrograms();
}

/**
 * This method executes the next step.
 */
1 usage  antonhtmlito
public void executeStep() {
    board.setStepMode(true);
    continuePrograms();
}

// XXX: V2
2 usages  antonhtmlito
private void continuePrograms() {
    do {
        executeNextStep();
    } while (board.getPhase() == Phase.ACTIVATION && !board.isStepMode());
}
```

When all the steps have been executed, the startProgrammingPhase method will be called again. This continues up until someone collects all three checkpoints and the game ends

6.3 Exceptions

The Roborally catches several exceptions. For example in the GameController.java, the sql exception is handled during the createGameInDB method.

```
} catch (SQLException e) {
    e.printStackTrace();
    System.err.println("Some DB error");

    try {
        connection.rollback();
        connection.setAutoCommit(true);
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

Figure 9: This code snippet shows how an sql exception is handled.

6.4 Checkpoints

The checkpoints are a key factor in the game. These checkpoints serve as markers on how far the players are in the game and also how the game will end.
At the start of the game the checkpoint (0, 1 and 2) are automatically added to the board and shows the order of which checkpoint the players should collect first. The further players progress in the game by collecting checkpoint certain actions will happen.

```
public class GameController {

    public static Board board;

    private Set<Integer> allCheckpoints;
    private List<Player> players;

    private static Set<Integer> collectedCheckpoints = new HashSet<>();

    public static Set<Integer> getCollectedCheckpoints() {
        return collectedCheckpoints;
    }
    public GameController(@NotNull Board board) {
        this.board = board;
        board.setGameController(this);
        collectedCheckpoints.add(0);
        collectedCheckpoints.add(1);
        collectedCheckpoints.add(2);
    }
```

Figure 10:

In our game the checkpoints are working as tracking progress and also increases the users experience like providing goals to achieve.

```java
public boolean doAction(GameController gameController, Space space) {
    //System.out.println("CheckPointFieldAction doAction()");
    Player player = space.getPlayer();
    if (player != null) {
       // System.out.println("CheckPointFieldAction player " + player.getName() + " reach check point " + checkPointFieldId);
        if((player.getCollectedTokens()) == checkPointFieldId) {
            player.collectedCheckpoints(checkPointFieldId);
            if (player.hasCollectedAllCheckpoints(GameController.getCollectedCheckpoints())) {
               // System.out.println("player " + player.getName() + " has collected all checkpoints");
                AppController.saveGame();
                Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
                alert.setTitle(player.getName() + " has won the game");
                alert.setContentText(player.getName() +  " has won the game" + "\n Start new game by going to file -> new game");
                Optional<ButtonType> result = alert.showAndWait();

                if (!result.isPresent() || result.get() != ButtonType.OK) {

                }
                //System.exit(0);
            }
        }
    }
    return false;
}
```

Figure 11:

In the CheckpointFieldAction class, the player landed in the space with the checkpoint will be checked if he has collected the checkpoint ID in the right sequence. If yes, the checkpointId is added to the player. If the player has collected all checkpoint IDs, the player wins the game.

6.5 JSON

JSON is a file format, which can be used for data exchange. In our Roborally, we use JSON to hold the data that dictates where our “Walls' ' elements should be placed, as well as the field actions on the space and the size of the board.

In our code we use an adapter for GSON, which deals with structures, where a statically typed element can have dynamic sub-types. This adaptor transforms an object into JSON format, and conversely takes the JSON format and transforms it back into a Java object.

The LoadBoard class reads the JSON file using the newJsonReader method from the GSON library.

```java
reader = gson.newJsonReader(new InputStreamReader(inputStream));
BoardTemplate template = gson.fromJson(reader, BoardTemplate.class);

result = new Board(template.width, template.height);
for (SpaceTemplate spaceTemplate: template.spaces) {
    Space space = result.getSpace(spaceTemplate.x, spaceTemplate.y);
    if (space != null) {
        System.out.println("space on " + spaceTemplate.x + ", " + spaceTemplate.y + " found.");
        space.getActions().addAll(spaceTemplate.actions);
        space.getWalls().addAll(spaceTemplate.walls);
    }
}
```

Figure 12: This is a code snippet from the LoadBoard.java, which reads the JSON file.

Figure 13: This is a code snippet from the defaultboard.json, which shows how the code defines the walls and the GearsFieldAction.

6.6 Database connection

The Roborally game uses a mysql-connector-j library to establish the database connection between the Java objects and the local database entries.

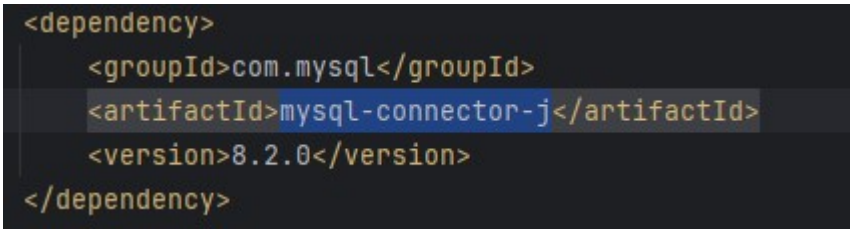


Figure 14: This is a code snippet from the pom.xml, showing the JDBC library Roborolly uses.

The database tables are created through the createschema.sql when the Roborally application is launched. The code snippet below shows the sql code for creating the Game table in the database.

```
CREATE TABLE IF NOT EXISTS Game (
  gameID int NOT NULL UNIQUE AUTO_INCREMENT,

  name varchar(255),
  boardName varchar(255),

  phase tinyint,
  step tinyint,
  currentPlayer tinyint NULL,

  PRIMARY KEY (gameID),
  FOREIGN KEY (gameID, currentPlayer) REFERENCES Player(gameID, playerID)
);;
```

Figure 15: This code snippet is from the createschema.sql, and shows the Game table in the database.

The roborolly information, such as the gameID, the phase, the step and the currentPlayer is saved into this table, when a user presses the *Save Game* button. After the user has clicked the *Save Game* button, the information will be stored in the Game table, which is shown below.

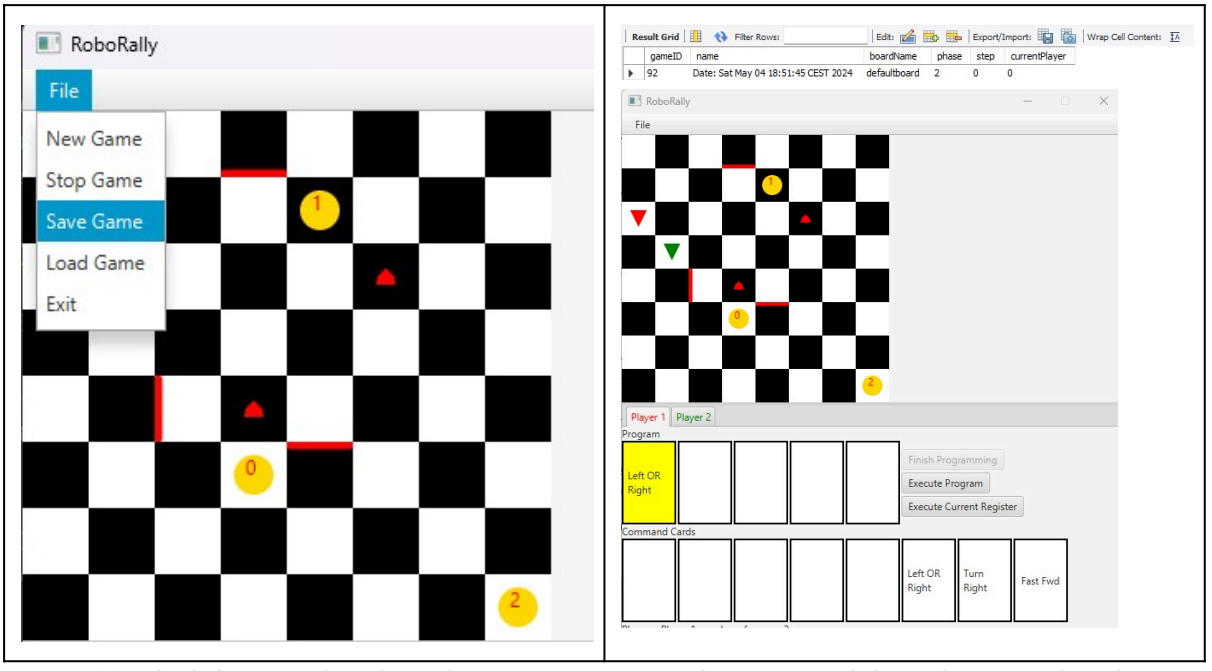


Figure 16: The left screenshot shows how a user can save the game, and the right screenshot shows how the game's gameID, the phase, the step and the currentPlayer is saved into the game table.

The logic regarding saving, updating, and loading players is implemented in the Repository class, which implements the IRepository interface. This interface is used to connect the Repository with our AppController.

6.7 How to establish connection to the database

You must create a database called pisu in mysql server, you can do it through MySQL Workbench.

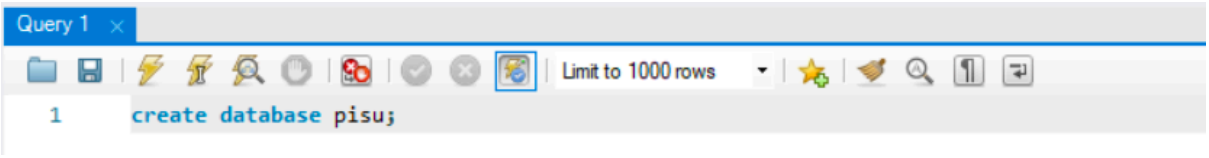


Figure 17:Creating a database through MySQL Workbench

In Connector.java, you must change the information for the username and the password, so it aligns with your MySQL server

```
public class Connector {

    1 usage
    private static final String HOST     = "localhost";
    1 usage
    private static final int    PORT     = 3306;
    1 usage
    private static final String DATABASE = "pisu";
    1 usage
    private static final String USERNAME = "root";
    1 usage
    private static final String PASSWORD = "root";
```

Figure 18:

6.8 Observer

When a subject is constructed it will need a method to keep it updated throughout the duration of the program. A subject like a player often changes its position on the board and its heading, when moving around trying to get to the checkpoints. The observer makes sure that the player's changes is also updated in the GUI.
When a player is on the board an observer is attached to it like this:

```
if (player.board != null) {
    player.board.attach( observer: this);
    update(player.board);
}
```

When something changes for the subject for example a new heading is set for the player it will call the notifyChange() method from the parent class Subject like this:

```
public void setHeading(@NotNull Heading heading) {
    if (heading != this.heading) {
        this.heading = heading;
        notifyChange();
        if (space != null) {
            space.playerChanged();
        }
    }
}
```

The notifyChange() method will then call the observer method update():

```
final protected void notifyChange() {
    for (Observer observer: observers) {
        observer.update( subject: this);
    }
}
```

which will then update the subject to match the GUI with, for example, the new heading of the player

```
void update(Subject subject);
```

6.9 Recursion

Recursion is used in our Roborally implementation in the GameController.java. The recursion is applied regarding the push player function, when there are more than one player in front of the current player. When the current player calls the moveForward method and there is a player in front of the current player, then the application will invoke moveForward method for the player in front of the current player, which is also the target player. If there is a second player in front of the target player, then the application will invoke the moveForward method for that player.

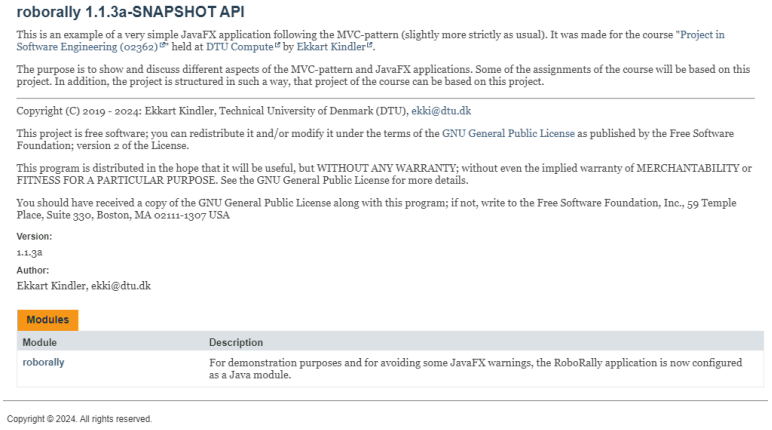
```
5 usages  ± anton dong +3
public void moveForward(@NotNull Player player) {

    Space current = player.getSpace();
    player.setMoved(true);
    Space target = board.getNeighbour(current, player.getHeading());
    Player targetPlayer;
    targetPlayer = target.getPlayer();
    if (current.hasCurrentWall(player)) {
        player.setMoved(false);
        System.out.println("Cannot move forward: Wall detected in the target space for player " + player.getName());
        return;
    } else {
        System.out.println("no wall ahead for player " + player.getName());
    }

    if(targetPlayer != null) {
        targetPlayer.setMoved(true);
        System.out.println("there is player " + targetPlayer.getName() + " on the space, push the player!");
        Heading tempHeading = target.getPlayer().getHeading();
        targetPlayer.setHeading(player.getHeading());
        if (!target.hasCurrentWall(targetPlayer)) {
            System.out.println("no wall ahead for player " + targetPlayer.getName());
            moveForward(targetPlayer);
            targetPlayer.setHeading(tempHeading);
        } else {
            System.out.println("There is wall ahead, no move for player " + targetPlayer.getName());
            targetPlayer.setHeading(tempHeading);
            targetPlayer.setMoved(false);
        }
        if(!targetPlayer.getMoved()) {
```

Figure 19: The code snippet shows the moveForward method being invoked within the moveForward method.

6.10 Javadocs

Classes and methods that are not in this report, can be found in our javadocs in our code. We have all described the method we have used in our programming.
The goal with our javadocs is, it makes it easier to follow our progress and the description helps understanding what the code does without reading the code and who has made what.

All of the Javadocs have been generated into a html website.



roborally 1.1.3a-SNAPSHOT API

This is an example of a very simple JavaFX application following the MVC-pattern (slightly more strictly as usual). It was made for the course "Project in Software Engineering (02362)" held at DTU Compute by Ekkart Kindler.

The purpose is to show and discuss different aspects of the MVC-pattern and JavaFX applications. Some of the assignments of the course will be based on this project. In addition, the project is structured in such a way, that project of the course can be based on this project.

Copyright (C) 2019 - 2024: Ekkart Kindler, Technical University of Denmark (DTU), ekki@dtu.dk

This project is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 2 of the License.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Version:
1.1.3a

Author:
Ekkart Kindler, ekki@dtu.dk

Modules

Module	Description
roborally	For demonstration purposes and for avoiding some JavaFX warnings, the RoboRally application is now configured as a Java module.

Copyright © 2024. All rights reserved.

7. Operate

7.1 Development process

7.1.1 Tools

In this project we have used a variety of different tools. The project is programmed in java and it is coded in IntelliJ. This is used in all of the code and in the connection of the database.
The database is programmed in MySQL and we have used MySQLWorkbench for the programming.

7.1.2 Process

Our process on this project has been as expected, like a rollercoaster. The work rate has been good and often organized but we have had our ups and downs. Our work on analysis has been good and we have been updating it throughout the course. The design part of our project started out great and then we had some time where we didn't update it but we pulled ourselves together and fixed it. The implementation has been also updated throughout the whole course. Over all our work rate has been good.

7.2 Test

It is important to test the features of the program to make sure that they work. In the project we have shown this by making a few relevant tests for some of the general features in the game. If the project were to progress and additional features would be added it would also need tests for these features.
The tests that have been made are for:

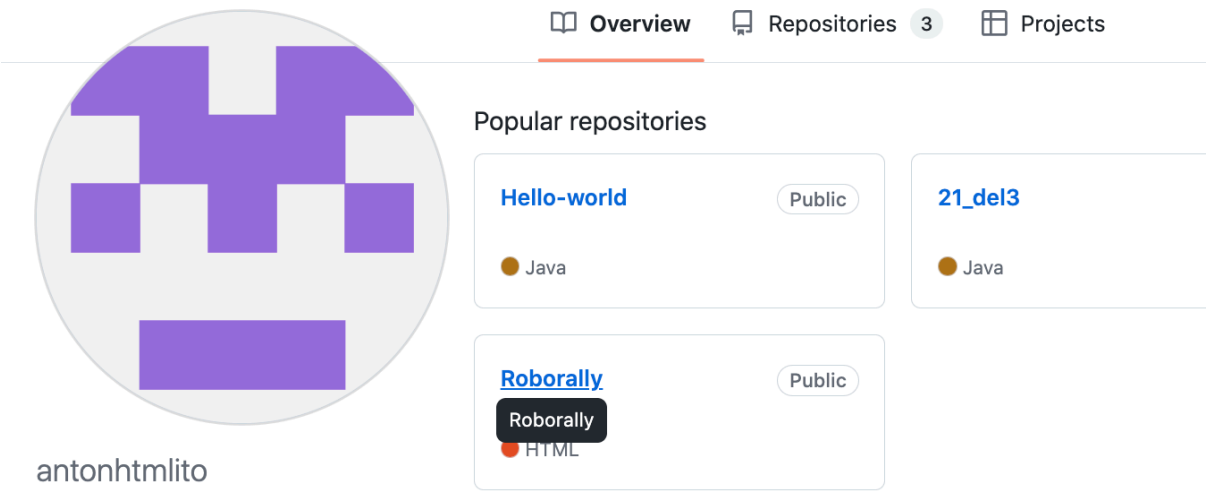
- Moving to a location
- Using forward card
- Using fast forward card
- Turning left
- Turning right
- Starting programming phase
- Ending programming phase
- Executing command option and continuing

8. Handbook

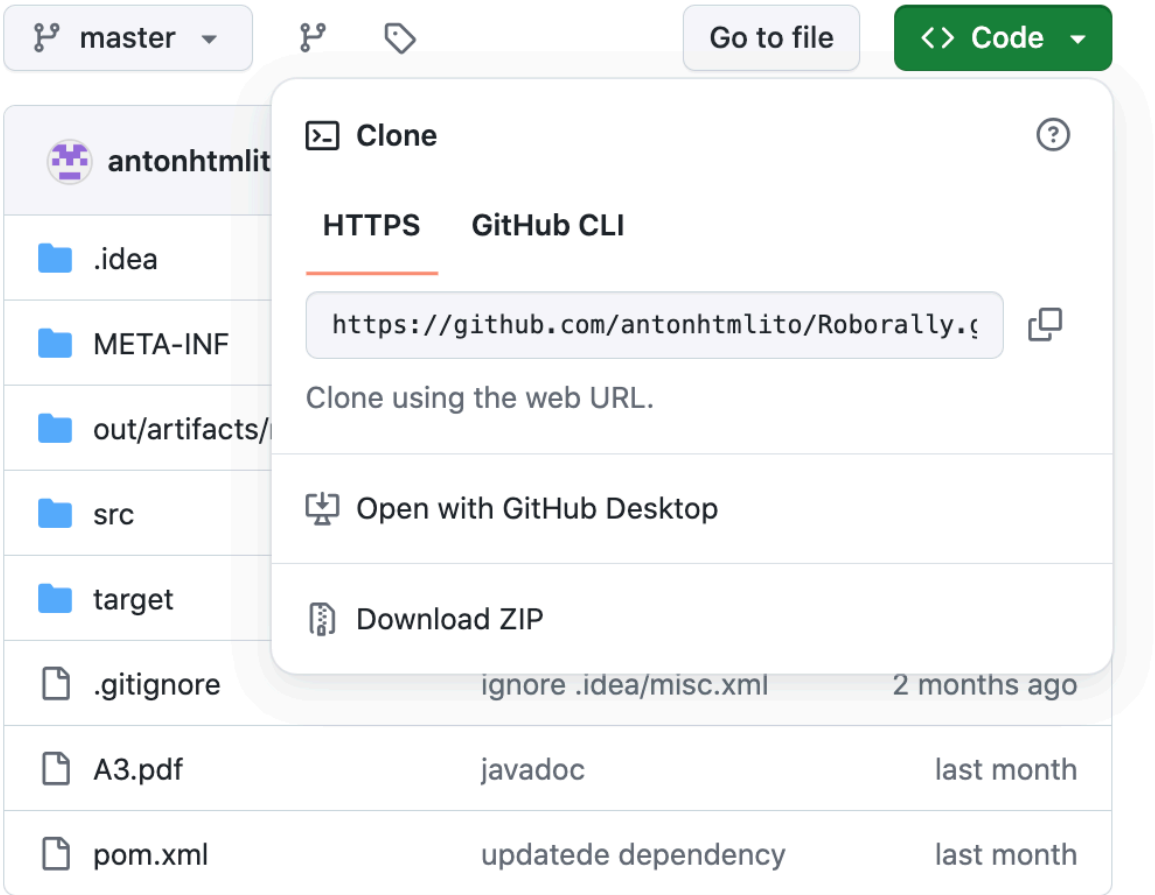
8.1 How to import from Git to IntelliJ

This is a guide on how to clone the code from a repository that is uploaded on GitHub.com

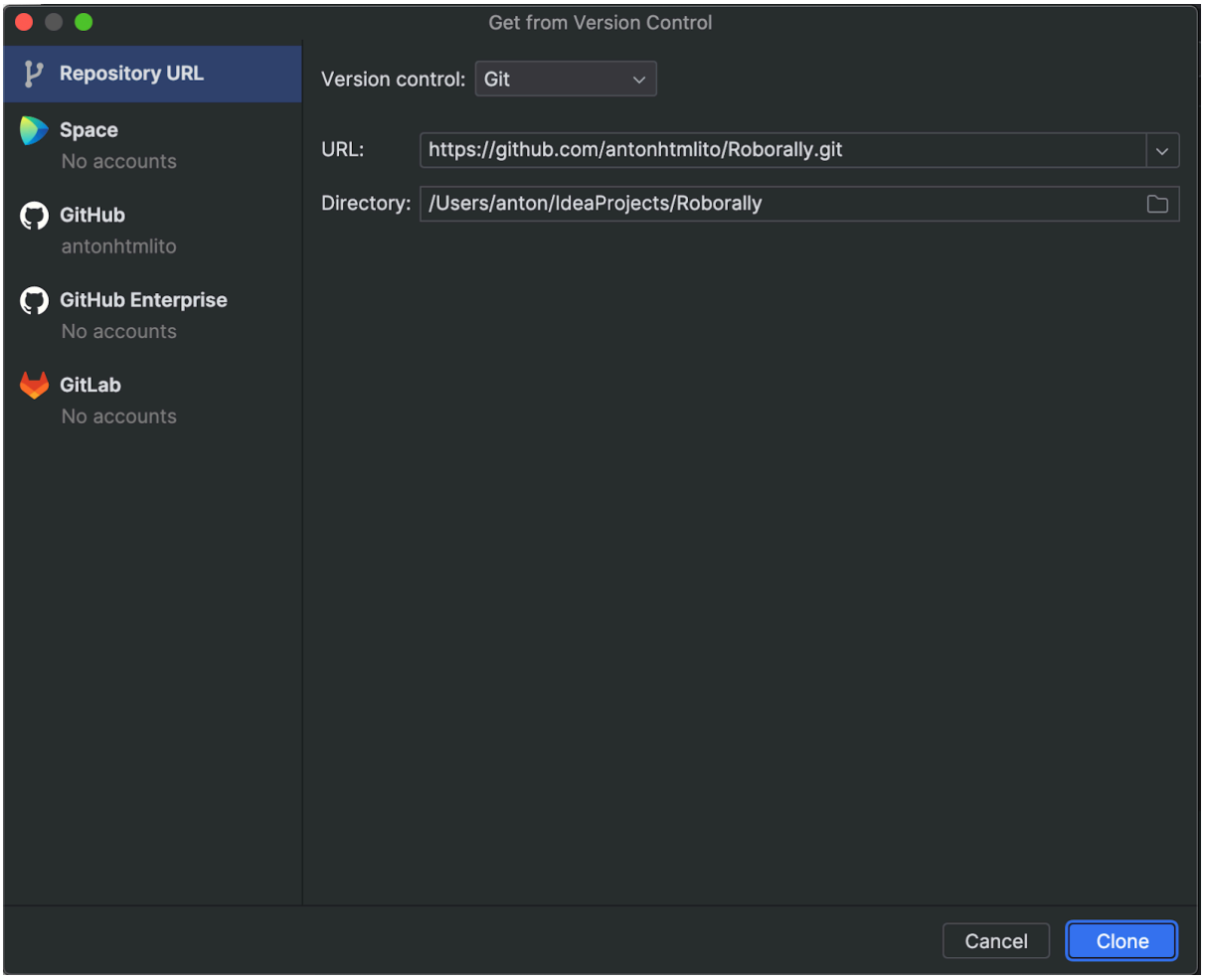
1. Find the repository you would like to clone, and find the link.
    a. Open GitHub.com, find repository Roborally and open it



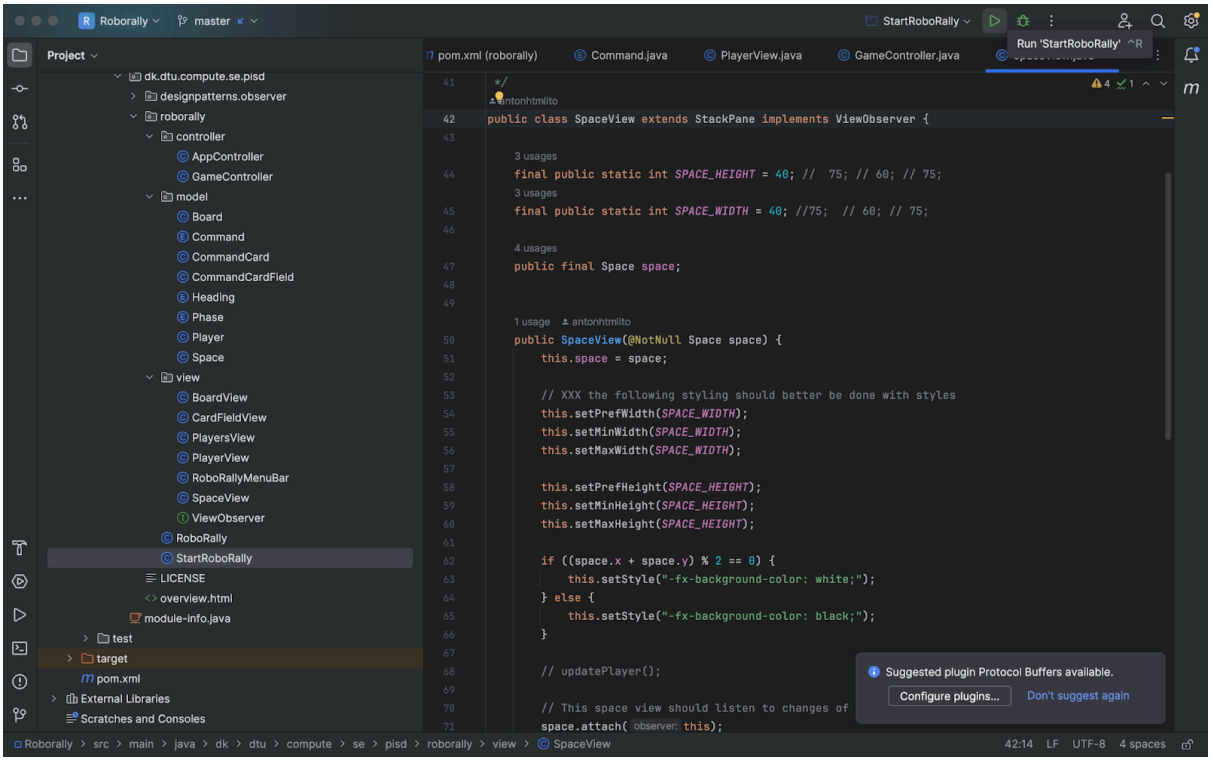
b. You must press the green button called “code”, and you will see an URL address, which you must copy:



2. Open the program “IntelliJ”, and press on “Get from VCS”. Now you will get the possibility to insert an URL address, and where you want it to be saved. When this is done you will have cloned the file.

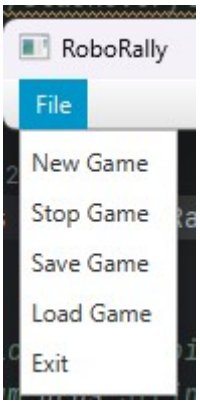


3. Find the “StartRoboRally” class and compile the application:

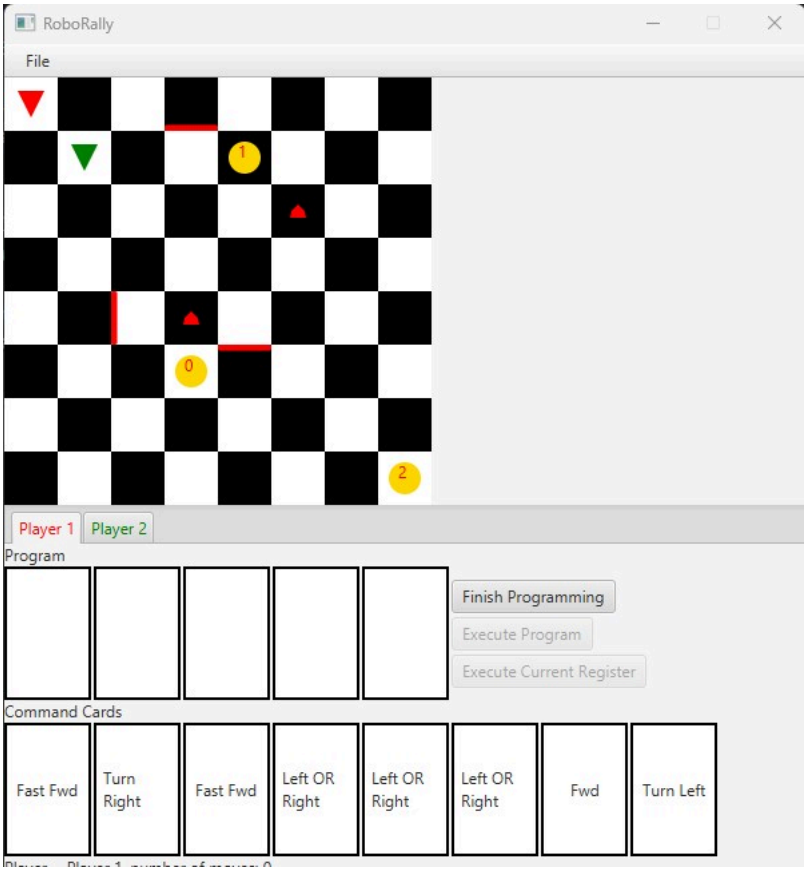


8.2 How to play the game

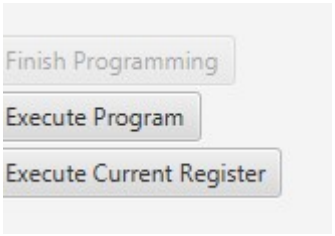
When the application has been launched, the game is started by either initializing a new game or loading a previous one. If a new game is initialized, the next is to choose a number of players between 2 and 6



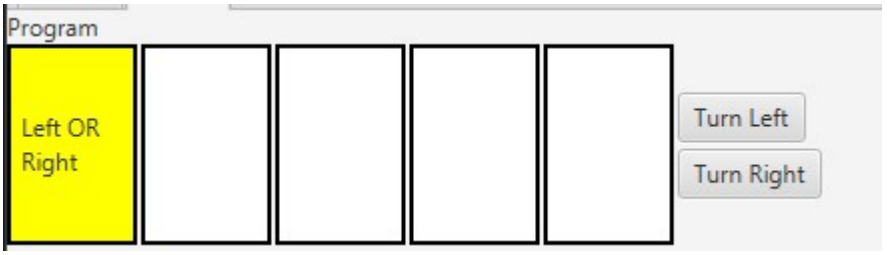
When the board is loaded, the players need to program their robot to move towards the checkpoints by dragging and dropping cards from their hand to the programming slots. The players do this one player at a time.



When the robots have been programmed, their actions will be executed by pushing one of two buttons. Either executing all commands at once or executing them one at a time



If the played card demands a choice, two more buttons will be accessible once the card is being executed. After pushing one of the two buttons it will continue executing as before



The game is ended when either the end game button is pushed or one of the robots reach all the checkpoints

9. Conclusion

We have implemented a digital version of the 1994 board game *RoboRally*, which can fit 2-6 players on a board with different obstacles.

We have provided a game that has cards, which command movement directions. Furthermore we have incorporated field actions, such as walls, checkpoints and gears.

The board for RoboRally can be loaded from three different JSON files, and the current game state can be saved, and loaded again for breaks in between playing RoboRally.

We have met all of our requirements, however RoboRally still has a big opportunity for scalability for potential future updates and expansions, such as lasers, pits, conveyors belts and much more.

10. Sources

10.1 Roborally rules

https://media.wizards.com/2017/rules/roborally_rules.pdf

10.2 Github repository

https://github.com/antonhtmlito/Roborally.git