OOP2018_c17johha_assignment6

Assignment 6 report

Object oriented programming IT308G – spring term 2018

Johan Häggmark

C17johha@student.his.se

0920

Institution of communication and information

University of Skövde

Table of Contents

Chapter 1: Introduction

The problem is to develop a working interactive board-game, dice-game or something similar, in the Java programming language. The system has to be developed with the Object-Oriented paradigm.

The solution must have a graphical-interface, with at least five classes. The solution uses inheritance with encapsulation and polymorphism, or/and it uses interface.

The solution is a Turn-based board-game, which requires two players, that will be playing against each other. The game is about deliver packages to the correct post box. To do this each player have a number of trucks, that is able to pick up packages and drive to the correct post box. Each player needs to wait for its turn, to be able to move a truck. A player can move each truck once, until the turn moves on to the next(other) player. The player with the highest score when there are no more rounds left, wins the game.

This is report will describe the solution in different levels.

In Chapter 1: Requirement specification, explains the game mechanics, how to play the game and an explanation of the rules. This will give all requirements about the game on a user experience level.

In Chapter 2: Design, the code structure will be explained and motivated. How components are used and how they are interacting with each other and why some techniques are relevant in the solution.

Chapter 3: Example on execution and analysis, is about the discussion of the solution, are the requirements fulfilled? Are the techniques working well or badly? how could it be done otherwise?

Chapter 4: Summary, this is my personal view of the whole project and my experience while working with it.

Some of the words that are used in the report might be a bit confusing without explaining the syntax for them. Here are the principles of the key-words:

System - the program and the design structure.

Program – the Java application.

game – the playable content a user is experience.

Game, Truck, etc. – some words has a Capital first letter, these targets classes or are in some way code related.

Function – is designed code to reach a goal.

Method – multiple functions to reach a bigger goal or a function that is imported from another sources but the developed project, example paint(Graphics) comes from JComponent.

Chapter 2: Requirement specification

The Game is called Truck Delivery. It operates on one computer. Two players are racing against each other to collect points, there are the red team and the blue team. They are to take turns to move the trucks that belongs to their own team.

A player has a marker/crosshair, which he (or she) is able to move around the board, by pressing the "Arrow-keys". The marker's color indicates who's turn it is. The blue player has a blue marker and vice versa. If a red marker is visible on the board, then it means it's the red player's turn.

A player may move the marker to a truck. If the player is able to move the truck, then the possible moves will indicate as bright gray color on those squares. When the player decides to move a truck, he (or she) must first mark the truck by pressing the "Space-key" (while the marker is on the same position as the truck). Then the player must move the marker to desired square and press "Space-key" again. If the move was possible, the truck should have been moved. A truck can only be moved by the player once each round. When a player wishes to finish its round, he (or she) can simply press "Enter-key", then the turn will switch to the other player.

To earn points, a player must move a truck to a package. A package has its own ID as a character, there will be a post box with the same ID. A truck will pick up a package if the truck moves to its position. Then the truck has to move to the post box position to accomplish a delivery. That will generate a point to the player. For example, the package 'A' is on the board. A truck of the red team is picking it up by moving to the same square as the package is shown. The package then disappears from the board, and the truck now indicates that it's carrying a package with the ID 'A'. When the player of red team has its next turn, he (or she) moves the truck to the post box with the ID 'A'. Red team then earned one point! See Figure 1 Deliver a package.

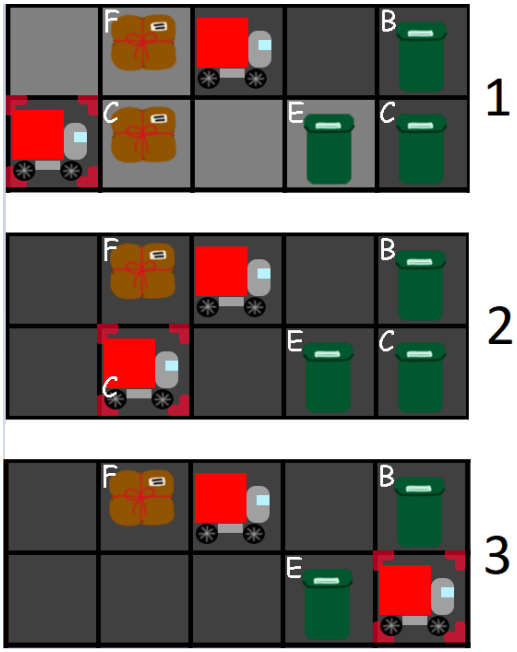

Figure 1 Deliver a package

A Truck can be carrying a maximum number of two packages at once. But be careful. For each package a truck is carrying, it has one square length less to move.

In the main menu, the players must decide which game type they want to play. Simply by left-click one of the shown buttons. There is a quick, normal and a long game. A long game gives the players

more rounds and has a bigger board with more trucks and packages available, compared to the quick and normal game. Quick game has the smallest board and smallest amount of trucks and packages with the least number of rounds. See Figure 2 Main Menu.

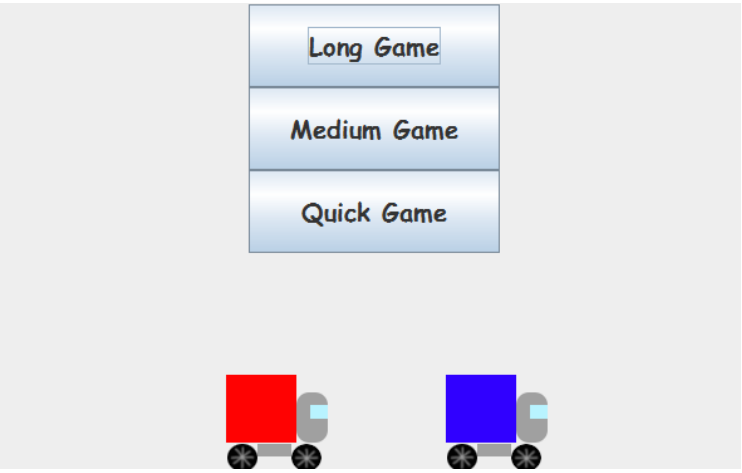

Figure 2 Main Menu

When the game is over, it will indicate which team won the game, or if it's a tie. To play again, restart the program. See Figure 3 Red team won the game. To the left, red team has a score of 2 points. This is the last round. To the right, the game is over.

Figure 3 Red team won the game

Functional Requirements:

- Two people can play the game against each other on one computer.
- Both users have the equal preconditions in the beginning to pick up packages and deliver them.
- Users interacts with the system through mouse and keyboard.
- A user can move the crosshair/marker with the "Arrow"-keys.

- A user can select a truck with the “Space”-key while the truck is in scope of the crosshair/marker and if there is no other truck selected.
- A user can move a selected truck with the “Space”-key to the crosshair/marker, if the Crosshair/marker is on an up lighted Square.
- Users take turns to move the trucks.
- The game has a limited number of rounds/turns.
- Users are informed about the game statistics.
- The system gives response to the users through a graphical interface.
- Packages and post boxes are placed in equally positions for both teams.

Non-functional Requirements

- The game does not have a time limit.
- The system does not include unnecessary heavy calculations that will slow down the program.
- The system does not react on other inputs but the current controller settings.
- The system is safe for the computer to run.
- Structure of Code is designed for further content implementation.
- Hard code is avoided in the highest degree.
- The system is developed with an object-oriented design.

Chapter 3: Design

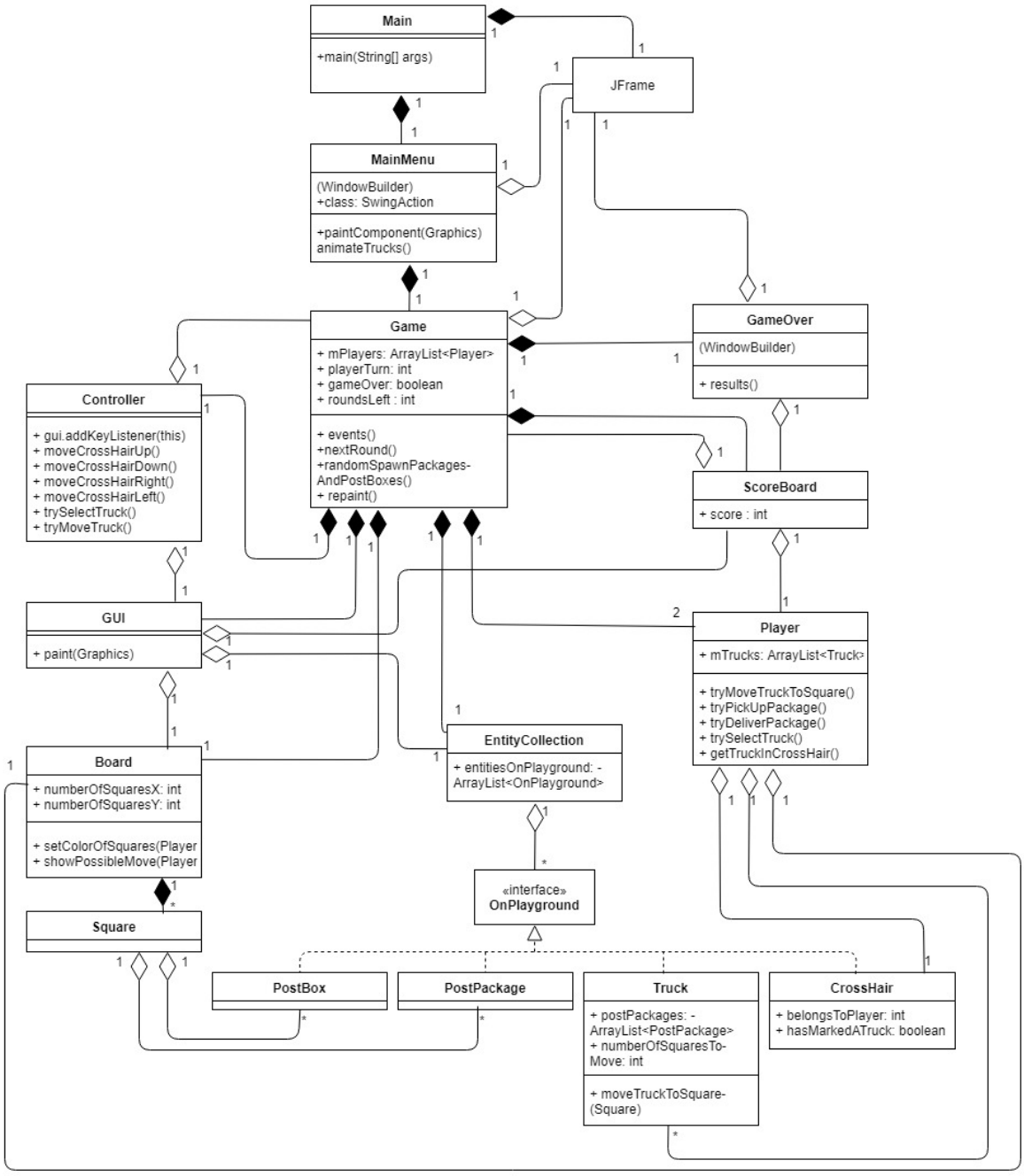

Figure 4 Class diagram

See Figure 4 class diagram. Classes: MainMenu, Game, GameOver, all receives the JFrame, because JFrame must be on the current state of the. MainMenu creates an instance of Game. To create a Game instance, these parameters (which decides the size of the game) must be fulfilled:

- numberOfSquaresX, numberOfSquaresY
- numberOfTrucksForAPlayer
- rounds

the creation of Game triggers the games awakening. Once Game is created, the game is alive.

Game adds an instance of GUI(which is a JPanel) to the JFrame. The instance of Controller (Which implements a KeyListener) adds itself to the instance of GUI. Controllers KeyListener is the thread that listen to any KeyBoard inputs. There is no running process in the program in the state of waiting for input, except for "Thread[DestroyJavaVM], Thread[AWT-EventQueue-0], Thread[Thread-2]. When the KeyListener detects an input signal, Controller will call keyPressed() method. Depending on what key was pressed, Controller will call different functions. For example, moving the crosshair. At the end of KeyPressed() method, Controller calls game.events().

Game's function event() is responsible to give the user the updated state of the game, as the final function in event() calls gui.repaint(). As Controller has access to the instance of Game, Controller directly get hold to the instances of Player. The game knows which of the Player objects turn it is, in that way Controller is always adapted to the current Player, and manipulates the Player's CrossHair without using for-loops or if-statements.

There are a number of objects from different classes that are going to interact with each other. The solution is to check the position of objects. Objects are only in positions of pixels that can be divided by 100. This makes it easy to check possible moves, or when a truck should pick up a package.

The Game is about moving the trucks to pick up and deliver packages. The design of the code of doing this can be explained in this way:

Change position of CrossHair:

- 1. The game knows which of the players turn it is.
- 2. Controller asks Game which of the Player's CrossHair it should manipulate
- 3. CrossHair is a OnPlayground item, all OnPlayground items exists inside EntityCollection.
- 4. When step 2 is happening, Controller calls the GUI's repaint(). repaint() draws the updated positions/states of all OnPlayground items.

This is how to move a truck:

- 1. Controller's KeyListener notifice a KeyEvent from the button "SPACE". This will call the functions tryMoveTruck() and trySelectTruck(). Let's say there were no Trucks with "boolean select == true" in the current Player, then tryMoveTruck() was not possible to perform, because tryMoveTruck() only moves a selected Truck. Instead trySelectTruck() sets a Truck's (Boolean select = true), this was possible, since the CrossHair and the Truck was on the same position.
- 2. As game.event() was called from the Controller's KeyEvent, game.event() then called Board's function: setColorOfSquares(Player). Game sends in the current Player to the function.
- 3. Board's setColorOfSquares(Player) will call functions that will calculate what color of each Square will have. For example, Board finds out that the Player has selected a Truck. Board's job is now to calculate which Squares the Truck is able to move to. Board checks the selected Truck's variable: numberOfSquaresToMove and calculate which of the squares are in the distance of this variable.
- 4. When this is done, game.event() finally call gui.repaint().

New round:

- Whenever there is an input from the “Enter” key. Controller will call Game’s function nextRound()
- nextRound() will change the variable playerturn. The new current Player’s Trucks will be reset, so they are able to move again.
- Game’s variable roundsLeft is degreasing.

The GUI is using polymorphism inside the paint(Graphics) method, for OnPlayground items. This is to get the positions, images, and information, such as if a Truck is carrying a PostPackage, or the ID of a PostPackage.

Classes:

- **Game** is the owner of most of the objects of the system. Game decides if the current match should be alive or if it is game over and who’s turn it is. Game’s capital function is event(), which makes the program run for the user. To create a Game, it must be decided how big the board should be, how many rounds the game should have, how many Trucks each player should have, and how many PostPackages should be spawned. When Game is created, each PostPackages and PostBoxes are attached to a random Square.

  When a PostPackages is added, there must be added another PostPackage with an opposite position, it is a requirement to give both users equal preconditions of the game. The function randomSpawnPackagesAndPostBoxes() will call other functions to check if there are unoccupied pair of Squares. A pair of Squares with opposite positions is defined as (the first element of an Array + X), and (the last element of the same Array – X).
- **Player** is responsible for the rules of what a user is able to do, any manipulation of objects a user is doing, is going through the Player. It also owns its score variable. Player must get access to Board and get Board’s Squares content, while it is managing its Trucks to pick up PostPackages and deliver PostPackages to PostBoxes. Player checks with Board whether a Truck should be able to move to a position that is equal to a Square.
- **Square** has a position and might either have a PostBox or PostPackage inside it.
- **Board** owns a multiple number of Squares, Board decides what belongs to a Square, If a Square instance has a PostPackage, the Board is responsible of when it should be removed. A Square might be occupied by a red Truck, when a blue Truck tries to move to that square. Board will make it an impossible move. In this way, Player and Board are working together.
- **OnPlayground** is an interface. In this case it is chosen over an Base Class. The idea of OnPlayground is not what a parent Base Class is. All OnPlayground items are simply the entities, that is visible on the playground. Foremost Onplayground items exists to simplify graphics handling. It implements by:
  - **PostBox** is created inside Game but attached to a Square and EntityCollection, it contains ID and position
  - **PostPackage** is created inside Game but attached to a Square and EntityCollection, it contains ID and position
  - **Truck** is created inside Game but attached to a Player and EntityCollection, it’s responsible for it own rules (example, numberOfSquaresToMove, moveTruckToSquare()), it may contain PostPackages. The goal with Truck, is to be able to have each truck (game piece) as an object, and store valuable content in it.
  - **CrossHair** is created inside Game but attached to a Player and EntityCollection. This is the tool for the user, to interact with the games entities.

- **EntityCollection** holds all OnPlayground items in an ArrayList. The goal of EntityCollection is to have a separate place for all OnPlayground items.
- **GUI** draws every Onplayground item via EntityCollection, and every Square via Board. GUI is only used for handling the graphics.
- **MainMenu** is made with the tool WindowBuilder. It also has a separate Thread running. The thread makes an animation of moving Trucks possible, while the MainMenu is waiting for user input. The Thread's while loop is updating two Trucks position and then repaint() the Trucks in MainMenu's PaintComponent(Graphics) method. The goal is to give the users an opportunity of choosing the length and complexity of the game.
- **GameOver** is made with the tool WindowBuilder. The game results is put on a JLabel. The goal is to give the users feedback when the game is over.

Chapter 4: Example on execution and analysis

The final Product is a simple board game with a cognitive challenge against each other. Two people are able to play the game with the very most equally preconditions to each other. Though, the second player might just make the exact same moves as player one, and always end up in a tie. The users must take turns to use the keyboard, to interact with the game. The users get all responses from the game through a graphical interface. All functional requirements are fulfilled.

From a user experience point of view, the system is not always visibly for how to interact with the system properly. It might be confusing to select a truck and move a truck with the same key. Test persons also had problems to separate the function of “Enter”-key (next round) and “Space”-key (select truck, move truck).

The game does not depend on time. The program so far seems to be light weighted, in perspective of: When the program is waiting for user input in a game. The CPU usage is falling down to 0%. It is a strong advantage to not use a running update loop in this case. Almost every object in the game are in standby until they are exclusively called when there is a key event. An effective way to skip some for loops, is to let Game decide which of the Player’s turn it is. In this case Controller directly calling the correct Player’s functions. Else the program could check which of the Players that should be called, every time a “manipulation” (Key event) is happening.

There is no sign that the system is unsafe to run as a Java application on a PC. No failure of the system was found during test period.

The graphics handling could be done in an other way. The implemented method has a very understandable basic idea. An object is never responsible for whether it should be drawn on the screen. An object only possessing its physical states. The GUI asks the object of its physical states, and then GUI translate it to fit a drawing function, so it can be displayed in the graphical interface. In this case, every OnPlayground item can be drawn by GUI. GUI possess the only one paint(Graphics) method, when Game is running. The same time it was a strong readable structure of code, it also became a limitation.

For a while it was suggestions of having animations in the game, an idea was to bring the game to life. But the concept of handling the graphics limited the possibility of having animations that would run along while the game was in a passive state. The concept of only have one graphic handler could not do animations without repaint everything else. It seems that this would bring the game from a minimalistic and effective solution, to clumsy and ineffective unclean code.

One solution might be to throw away the concept of only use one paint(Graphics) for the game, and add a paint(Graphics) or paintComponent(Graphics) on each OnPlayground item. GUI would be responsible to call each object’s render() function, which would include a paint(Graphics) method. And then animations can be drawn at a specific rate in separates Threads without everything else is drawn again. Though, the time was not enough, and the requirements of the project doesn’t focus on bigger solutions, the idea had to be thrown away. But it might be a feature in the future.

There are not much hard coded implementations in the system. When Game is created the parameters: numberOfSquaresX, numberOfSquaresY, and numberOfTrucksForAPlayer must be fulfilled. They are further used with algorithms to adapt the involved objects to fit with the current size of the playground/board. For example, numberOfSquaresX of course decides how many Squares the board will have in x-led. But it also influences the width of the current JPanel, it also tells where

Trucks, PostPackages, and PostBoxes might spawn. numberOfTrucksForAPlayer tells how many Trucks a Player will have. For red team, the Trucks starts spawning in top lane and the second Square from the left and keep spawns to the right. For blue team the Trucks starts spawning in the bottom lane and the second Square from the right.

The code structure follows the object-oriented design paradigm. The program runs with multiple objects of the written classes. This structure also enables the project for a further implementation. Maybe in the future, there will be more content in the game, like other types of trucks and new missions.

Chapter 5: Summary

This report focused on the implemented ideas of how to develop a board game in Java. This is the best solution I came up with to make a board game as light weighted as possible. In my mind I found it inappropriate to use the well-known (in game systems) polymorphism update methods for all entities, and decided to have a more passive system, which only updated the affected objects.

I always find it difficult to assess the width of a project. With the resources I have and the amount of time that is reserved, what reasonable could be done? I would say that these kind of assignments are a very beneficial way to practice developing of projects. With deadlines you have to be able to finish your goal, and predict what is possible.

This was the first game idea I considered. Then another, more interesting idea came to me. I was not sure which of the game ideas I would go for, so I started to look into the design in both of the ideas. The other game would have physics, and that was something that would take much resources from me, and that would jeopardize to finish the assignment before deadline. Fortunately, I focused on the requirements, and the other game-idea was above of the requirements of the projects.

I am not completely familiar with the structure of developing projects. What I understand is that you do not look into the code until you finished the requirements and the design part. I find it hard to envision what I am able to code, and what it takes to fulfill a functional requirement. Something that sounds easy in my mind, might be a huge project to perform.

I would waste less time on figuring out the game-idea if I was to do the assignment again.

The assignment was very fulfilling, especially with all freedom. It was a great opportunity to grow as a student of system developing.