*Jack Maguire*

*Candidate Number: 0109*

*Candidate Centre: 51518*

*apex turrets*

# Table Of Contents

[Table Of Contents 1](#_Toc33018158)

[Analysis 3](#_Toc33018159)

[Research 3](#_Toc33018160)

[Background and Problem Definition 3](#_Toc33018161)

[Analysis of Similar Systems 3](#_Toc33018162)

[Kingdom Rush 3](#_Toc33018163)

[Bloon TD5 4](#_Toc33018164)

[Brackeys Tower Defence Game 5](#_Toc33018165)

[Problem With Current System 5](#_Toc33018166)

[Identification of Users, Their needs and acceptable limitations 5](#_Toc33018167)

[Objectives 7](#_Toc33018168)

[Proposed Solution with Justification of Language 8](#_Toc33018169)

[Use of Java 8](#_Toc33018170)

[Design 10](#_Toc33018171)

[Textual Overview 10](#_Toc33018172)

[Flowchart 11](#_Toc33018173)

[Class Diagrams and Algorithm Explanations – Utility Classes 12](#_Toc33018174)

[Config File Reader 13](#_Toc33018175)

[Custom Change Listeners 20](#_Toc33018176)

[Resource Manager 21](#_Toc33018177)

[Class Diagrams and Algorithm Explanations – Rendered Classes 22](#_Toc33018178)

[Square Class 22](#_Toc33018179)

[Base Entity Class 26](#_Toc33018180)

[Turret Class 29](#_Toc33018181)

[Enemy Class 33](#_Toc33018182)

[Canvas Class 36](#_Toc33018183)

[Class Diagrams and Algorithm Explanations – Gameplay Classes 40](#_Toc33018184)

[Player Manager Class 40](#_Toc33018185)

[TurretFrame and TurretManager 42](#_Toc33018186)

[Waves 43](#_Toc33018187)

[Class Diagrams and Algorithm Explanations – Main Class 43](#_Toc33018188)

[Main Class 43](#_Toc33018189)

[Apex Turrets Class 43](#_Toc33018190)

[Technical Solution 44](#_Toc33018191)

[Testing 44](#_Toc33018192)

[Evaluation 44](#_Toc33018193)

# Analysis

## Research

### Background and Problem Definition

Currently, at KCS, we have about 1000 pupils and numerous members of staff, and other than free periods, we have two breaks, one in the morning for 20 minutes, and one in the afternoon starting at either 12:35 or 1:15 (sometimes there is an extra lesson), and ending at 1:40 (for sports), or at 2:10 (for lessons afterwards). In the other half of the school (the junior school), there was a well-equipped playground, with monkey bars, play-frames and table-tennis tables. There was also the library (homework was forbidden in the day). In the Senior School, however with vastly more pupils, and there are only spaces in which to gossip, or the library in which there are computers (homework only), or the ICT Suite (in which Games are banned).

This creates a problem, that the people who don’t gossip, can get easily bored, with only homework, reading or sitting around to do. My proposed solution, a Tower Defence Game aims to solve that, in that the IT Staff allow self-coded Games, and that no previous pupils have made large scale games, similar to any standard of Game Development (pupils are taught only python at a basic level, and most students take a dislike to the Subject due to the teacher for the school being shouty, and that you are only taught in the first year (Year 9)).

A Tower Defence Game, is a game which is 3rd person based, and one has enemies to kill, but instead of shooting them like most games, you build turrets or towers to kill the enemies. These cost money, but as enemies die, they give you money, and you can build more towers. If the enemies get to the end of their path however, then you lose lives, and if you have 0 lives, then Game Over. You win the level, if you kill all of the enemies. As you progress through the game, there are more types of enemies, and towers, and you can upgrade the towers. It resembles some other games like Factorio, in that you try to automate it so that you don’t have to do anything. In more advanced versions again, you have power-ups or heroes.

## Analysis of Similar Systems

### Kingdom Rush

One good example of a tower defence game is Kingdom Rush, a popular game that features a lots of features. The UI is clear and well-defined, and there are a lot of levels, to the point where I have not yet come close to finishing the game. It features no ads, and the only way of them making money, is buying more in-game currency (used for things like ‘Freeze all Enemies’), or buying heroes (they sit where you tell them, and kill enemies as an AI would in a typical shooting game).

However, this game may be available on Steam and Mobile, but steam and downloading .exe files has been blocked by the school, and mobile phones are banned for the whole day.

Below are 2 photos, showing the first level of the mobile version,

A picture containing indoor, grass, table, cake

Description automatically generatedA picture containing indoor, table, grass

Description automatically generated

These images show the first level of the game. The UI is clear, giving an idea of lives remaining, money and current wave. To build a tower, you tap on the spot, and it gives options. This also comes in later, for selling or upgrading towers. Link to app store:

<https://apps.apple.com/gb/app/kingdom-rush/id516378985>

|  |  |
| --- | --- |
| Pros | Cons |
| * easy to use * free * no ads * lots of levels, turrets, enemies and extras * clear ui | * **Only downloadable via steam, or phone.** |

### Bloon TD5

A screen shot of a computer

Description automatically generated

Next, is a different yet still fun style, or bloon TD5. In this version, enemies are balloons (bloons), of different speed and popping strength. Then, instead of set turret spaces, there are areas to put down balloon poppers, and some are valid, and others invalid. In this case, you could only put down turrets on land on the planets. The strategy of the positioning with limited money, makes it quite interesting. Many types of turrets exist, and keyboard shortcuts can be used to quickly grab types of turrets to place, a system I will integrate into my own game. However, the game is hosted on Kongregate, with limitations such as a need for flash, which is being discontinued by popular browser Chrome soon. Link: <https://www.kongregate.com/games/Ninjakiwi/bloons-td-5>

|  |  |
| --- | --- |
| pros | cons |
| * easy to use (slightly less so than kingdom rush, but not much) * free * lots of turrets, levels, extra | * **Ads** * **broswer-based (needs internet)** * **slightly cluttered ui** |

### Brackeys Tower Defence Game

There are several game development youtubers (for beginners), the most famous that I have found being Brackeys, a Unity Game Developer with over 900,000 subscribers. He has several series’ for making a game, one of which is a tower defence game, and while this could not be counted as a release itself, it was one of the only such games, that could be downloaded locally for free on a Mac (there is no release, follow the tutorial). This is inherently free, and has different levels, tower upgrades, different enemies, and most importantly, the easy ability to generate more enemies. The enemies are spheres, but work fine. The two turrets can be placed on any node on the map, and cost money. The UI looks clean due to a general lack of anything unnecessary.

A picture containing crossword puzzle, text

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Above is the main level (left), the level select (middle), and the start of the game (right). Firstly, and most important is the game itself. The nodes are made of cubes, and compose the entire map, other than the enemy path. Turrets can be placed, and after they are bought, a simple click is needed to place it, and they all have full rotation. Lives are shown at the top, and money, time remaining, and turrets to buy are all at the bottom. The waves are predetermined, and the level select, gives options to load different levels (all contained in similar scenes, not too different from JavaFX scenes, In that they can exist, while not being shown). Then the play screen is the model of one of the turrets, that rotates with text to either quit the game, or go to the level select screen. Link to github repository below:

<https://github.com/Brackeys/Tower-Defense-Tutorial>

|  |  |
| --- | --- |
| pros | cons |
| * free * desktop * no wifi needed * clear ui | * **have to follow tutorial / download unity to get it.** * **simplistic** * **limited assets** * **simplistic enemies / ground** |

## Problem With Current System

## Identification of Users, Their needs and acceptable limitations

Questions for Client:

**Q1: What do you currently do during our breaks?**

A: Currently, I either walk around and talk to friends or sometimes use the time to catch up on work or communicate with teachers. During the lunch break, I sometimes attend a club (which lasts around 30 minutes).

**Q2: What problems are there with this?**

A: I do agree that there are times when there is nothing to do sometimes (especially during lunch break), which can lead to boredom. Sometimes my friends may be ill/occupied with something or clubs may be cancelled, which leaves nothing to do for about an hour.

**Q3: How do you think these problems could be tackled with my proposed solution?**

A: This solution provides a system which, while tackling the obvious issue of boredom during breaktimes, may also motivate other students to create their own games for others to play.

**Q4: Do you believe that there are any essential features that are currently missing?**

A:

* As the system runs on a user login basis, there should be a way for a player to play the game (as much as possible) without signing in or connecting to the database.
* The game should have a tutorial level which explains the basic mechanics of the game to a new player.
* There should be a level list screen, which allows the user to go back an replay a level whenever they want to. Future levels should be visible but inaccessible (locked).

**Q5: Are there any features with my proposed system that you find unnecessary?**

A: While a message feature in the leaderboard is perhaps a nice addition, it would lead to possible bugs and errors in formatting. As well as this, a profanity filter would not be able to catch any implicit profanity (innuendo, racist connotations, etc.)

**Q6: Do you have any ideas about UI, or anything else to do with the Front-End?**

A: The UI should be as easy to use as possible for both old and new players:

* The player should be asked to login (or continue as guest per my request in Q4) **before** seeing the main menu.
* There should be a button the top left / top right of the main menu screen which allows the player to sign in/access their account settings.
* The main menu should then have the essential options for any game:
  + New game
  + Resume game
  + Select level
  + Leaderboard
  + Quit
  + (Anything else)

Ben has asked for reasonable things, and I will attempt to put in all of these.

## Objectives

1. The game should not be repetitive.
   1. There should be different turrets.
      1. Firstly, there should be at least two different turret templates.
      2. Then there should be upgrades available for each turrets (at least one, but more should be able to come.)
         1. An upgrade constitutes of an increase to a statistic positively, such as more damage, or shooting more often.
   2. There should be different enemies
      1. Currently, there is one enemy that is fast, but can be shot down fast.
      2. And another that is the reverse.
   3. There should be different levels
      1. First, will be set levels, in which enemies come and go, in a set order (determined by a config file), and score will be determined by money remaining, money used, and hearts remaining.
      2. Then there will be an infinite ‘gauntlet’ mode, in which enemies are thrown semi-randomly (there will be checks not to be too slow, or too fast), and score is determined based on factors like time between enemies, enemies killed and the above factors for the set levels.
2. The game should be able to differentiate users.
   1. A user can select a difficulty, and have that kept for them.
   2. Each user should be able to keep a unique account.
      1. A password will be required, of a certain length.
      2. As well as a username of a certain length.
      3. The password will be hashed using SHA256, and stored in that way.
      4. The attributes of each player scored locally:
         1. Overall Score,
         2. Levels complete and score for each,
         3. Gauntlet high score;
      5. As a new user is created, a new file is created with the attributes of that player.
3. The game should be as efficient as possible.
   1. Storage Space Efficiency
      1. All turret, enemy, and level data will be stored on a Github repository, and fetched at runtime.
4. The players should be able to share statistics.
   1. The email will be stored, so a python script could be used to email others about the score.
   2. There will be a leaderboard for set levels, and for the Gauntlet.
      1. On the leaderboard, it will be in rows, sorted by score, and only display the top 10, and it will show the username, a message, and their score.
         1. In case of any unsavoury language, each word will be checked against a database (<https://raw.githubusercontent.com/RobertJGabriel/Google-profanity-words/master/list.txt>), and censored appropriately (eg. F\*c\*).
5. Running Example:
   1. The user will be presented with a main menu, with a Welcome Screen and two buttons to either login or to see the leaderboard.
      1. If they click on the leaderboard, then it will show it, as in 4.2.1
   2. If they click login, a new window will open to login, with an simple GUI to create a user, or to login.
      1. The details are stored locally on a cfg file.
   3. Once the user has logged in, they will be able to choose to replay a level, or to progress to the next.
      1. If they choose to progress to the next level, it will be the same as a previous only a different level shown.
         1. Then the level will be shown, with a grid system for the tiles, an enemy base, a home base, and GUI for purchasing towers, as well as to inform about lives and money.
            1. The grid system is that each icon has an ID, and dependant on a cfg file, it dynamically shows different tiles to change up the level.
            2. At the enemy base, the enemies spawn, and they slowly make their way across ‘path’ squares, and attempt to get to the home base. If they reach the base, then the player loses an amount of lives, and points for the end of the level. If the player runs out of lives, then Game Over.
            3. To combat the enemies, the player can buy turrets, and these can shoot the enemies, to kill them before they reach the base. The turrets get better, but so do the enemies, and turrets can be upgraded.
            4. In the shop, the player buys the turrets.
      2. The player could also play the Gauntlet, which is the same, only the level never ends until game over.
   4. If they choose to replay a level, they play through it as a normal level, and when it finishes, the old data from the player’s file is deleted, and the new data is inserted.

## Proposed Solution with Justification of Language

After analysing the similar systems, I have decided to use Java as the language, a language famous for desktop game development, for its versatility and ease of use for all platforms.

## Use of Java

Java is a relatively easy programming language to use, similar to C# and C, and is widely used, with famous examples including Minecraft. I have also decided to use a popular IDE, IntelliJ IDE. I have submitted an application for the free full student version, and I have succeeded, and so am using IntelliJ IDEA Ultimate. It has auto-code-completion, based on previous code, and it has a dark mode. The other main option was Eclipse, famous for its Class Wizard, and external library manager, but before moving to Java, I coded in C# for a few months, and got used to the auto-fill of Visual Studio (also the Community Edition), and so typing with no code completion felt strange.

IntelliJ also has syntax highlighting, and an automatic constructor for the constructor, getter, setter, override, and toString methods. I have now been coding using Java for more than a year, but with C# for about 2 years, and experience can be transferred over, with little difference.

Due to its popularity, Java also has many questions and answers on online forums such as stack overflow, which helps if the documentation is unclear or vague (which can happen sometimes with Oracle).

NB: I have used the very latest version of Java (at time of writing) – JDK 13 (experimental)

|  |  |
| --- | --- |
| pros of using java | cons of using java |
| * lots of community questions and answers. * widely used. * versatile. * very good ide. | * **JDK must be installed.** * **for games – no built-in entity/rendering system (like unity)** |
|  |  |

# Design

## Textual Overview

In the game – it starts with the game double checking whether the user wishes to play. If they say no, the program ends. If they say yes however, it launches into the program. Then, the UI is shown for the main game – a full screen window, with a collection of squares and a path to the left, and buttons and a status text area to the right.

First, the left hand side – a collection of squares, forming a path, and after an initial grace period (30 seconds), enemies start spawning going from the start of the path to the end of that path. The start is the square that looks like a satanic ritual, and the end is the bluer square. These enemies come in waves – a few enemies with a set gap between them, and then another set gap, before another wave and so on. Each enemy has a value depending on their strength, and you have a certain amount of lives. If an enemy gets to the end – then it takes away however much it is worth from the player’s lives.

Now, for the right hand side – a few buttons and a status area. There is one button for each type of tower, and when that button is clicked – information appears about the tower. An option is then given to select a location for that tower. If there is no space left, an error message is given to the user. If there is not enough money left, then an error message is given to the user. If they select a location, money is subtracted from the player’s account and the turret appears where the user specified. Under is also a sell button – and the player has the choice to sell a turret – NB: Each turret has a sale value, that is **less** than their buying value to ensure that each buy is worthy and there are no unlimited money schemes.

Finally, the status window – this shows the player how much money and how many lives they have left, as well as all of their turrets (name, position, initial investment, resale value).

## System flow diagram - listen to the code.Flowchart

## Class Diagrams and Algorithm Explanations – Utility Classes

For a class diagram showing everything in one place see All.png, or All.uml, on the Github Respository with the code.

NB: Key for diagrams:

* F in an orange circle - field (ie. variable without getter)
* M in a pink circle – method or constructor
* P in a purple circle – property (method with a return value or a variable with a getter)
* Red locked padlock – private
* Green unlocked padlock – public
* Key – protected
* Diamond in bottom left of circle, static.
* Full line with -> arrow – association (use of one class in another)
* Full line with -|> arrow – inheritance (one class extending from another)
* Dashed line with -|> arrow – implementation (one class implementing another)
* Dashed line with -> arrow – dependency (change one thing in one place, it changes elsewhere)
* Full line with hollow diamond – aggregation (eg. A car has an engine type, or the enemyActual has an enemyTemplate)
* Full line with filled diamond – composition (eg. An engine is a part of a car, or an enemyActual is a part of the waveSpawner)

NB: All diagrams were generated by pressing a few buttons in IntelliJ IDEA Ultimate as part of the built in UML Class diagram maker – I simply selected the options I needed.

### Config File Reader

A screenshot of a cell phone

Description automatically generated

Here, is the class diagram for the Config File Reader. Firstly, it has two regular expressions – one for a property and one for the start of a module in order to check we have the correct input. It has an ArrayList of Strings, with all of the names of the modules. We have all of the HashMaps, stored in an ArrayList, and they are the modules themselves. We have a BufferedReader to get through the file, and fn stands for file name, which is the file that the BufferedReader reads.

Pseudocode for algorithms:

BEGIN getModule (module)

IF !moduleNames.contains(module)

RETURN none

END IF

moduleIndex <- moduleNames.indexOf(module)

moduleToGet <- hashMaps.get(moduleIndex)

RETURN moduleToGet.clone()

END getModule

BEGIN get (module, property)

IF !moduleNames.contains(module)

RETURN none

END IF

Value <- getModule(module).get(property)

RETURN Value

END get

BEGIN read ()

currentLine <- “”

thisModuleName <- “”

currentModule <- NEW HashMap<>()

propertyName <- “”

propertyValue <- “”

equalsLocation <- 0

BEGIN TRY

BEGIN WHILE (currentLine <- reader.readLine()) != none

BEGIN IF currentLine <- “”

CONTINUE

END IF

chars <- currentLine.toCharArray()

BEGIN IF MODULE\_START\_REGEX.matcher(currentLine).matches()

BEGIN FOR i <- 0; i < chars.length; i++

thisModuleName += chars[i] + “”

END FOR

END IF

BEGIN ELIF currentLine.contains(“}”)

moduleNames.add(thisModuleName)

hashMaps.add(currentModule.clone())

thisModuleName <- “”

currentModule.clear()

END ELIF

BEGIN ELIF PROPERTY\_REGEX.matcher(currentLine).matches()

equalsLocation <- currentLine.indexOf(“=”)

isInt <- FALSE

isStr <- FALSE

isDbl <- FALSE

BEGIN FOR i <- 4; i < chars.length; i++

BEGIN IF i == equalsLocation

CONTINUE

END IF

BEGIN ELIF i = equalsLocation + 1

BEGIN IF chars[i] == ‘s’

isStr <- TRUE

END IF

BEGIN ELIF chars[i] == ‘i’

isInt <- TRUE

END ELIF

BEGIN ELIF chars[i] == ‘d’

isDbl <- TRUE

END ELIF

END ELIF

BEGIN ELIF i < equalsLocation

propertyName += chars[i]

END ELIF

BEGIN ELIF i > equalsLocation

propertyValue += chars[i]

END ELIF

END FOR

BEGIN IF isInt

BEGIN IF !main.INT\_REGEX.matcher(propertyValue).matches()

RETURN

END IF

value <- Integer.parseInt(propertyValue)

currentModule.put(propertyName, value)

END IF

BEGIN ELIF isStr

currentModule.put(propertyName, propertyValue)

END ELIF

BEGIN ELIF isDbl

BEGIN IF !main.DBL\_REGEX.matcher(propertyValue).matches()

RETURN

END IF

value <- Double.parseDouble(propertyValue)

currentModule.put(propertyName, value)

END ELIF

END WHILE

reader.close()

END TRY

BEGIN CATCH e

e.printStackTrace()

END CATCH

END read

Algorithm explanation:

getModule(String module). This returns a HashMap<String, Object>, and here is how it works. Firstly, is that each config file is made up of several modules, each of which has lots of properties, and values, and each is either a string, an integer, or a double. getModule, returns a cloned version of one of those modules. It does this by having a separate list, of module names. Both of these lists are changed at the same time, and so the indices of hashMaps to Strings of names are the same. This means that we can just get the index of the name, and then use that index to get the correct hashMap.

get(String module, String property). This returns an Object, of either, String, Double, or Integer, although it has to be reparsed on the other end. It works by using the getModule method to get a HashMap, and passing the module, and then using that HashMap, and getting the value using the property as the key.

read(). This method is used to read the entire file. Firstly, it sets up some temporary variables - currentLine (stores the current line as a String), thisModuleName (stores the name of the current module as a String), currentModule (which is the currentModule in a HashMap<String, Object>), propertyName & propertyValue (which store the name and value of the property respectively, in a String), and equalsLocation (which stores the location of the equals sign, in an integer). Firstly, the method opens up a try, to avoid any IOExceptions. Then it sets up a while loop, for each line of the BufferedReader’s file.

For each iteration of the loop: if the current line is a blank line – just go past it, to avoid checking a line with no information. Then create a character array with the currentLine.

Then, a series of if statements – if the line is the start of a module, then get the name. Skip the first 7 characters as they are just “module ”, and the last 2 as they are “ {”. If not, check if it is the end of a module, and if so, add the module name to the list, add the hashMap to the list, and reset both the name and the hashMap. If not, then is it a property? If so – find the equalsLocation, and go from the 5th character to the end – missing the first 4 as they are whitespace. Check if i is equal to the equalsLocation, if so continue. Else if it is 1 character after, check the character. If it is an s, isStr = true, an i, isInt = true, and an d means that isDbl = true. But if i is after the equals location, by more than 1 character, then add that character to the property name. If however, it was before the equals sign, then add the character to the property value.

Then, after the for loop, a series of if statements. Firstly, is whether it is an int. It checks for an INT\_REGEX, and if it succeeds, it parses the propertyValue, and puts that in the currentModule. If not, and isStr, then it places the value straight into the currentModule. Else, if isDbl, then it follows the same as isInt, only this time it checks with a different regex, and a different parser.

Other methods: the only other methods are a simple constructor, and a clone method.

Example Data:

FILE: test.cfg:

Module test1 {

Data1=sValue1

Data2=i2

Data3=d3.3

}

END FILE: test.cfg

CfgReader r = new CfgReader(test.cfg);

String Data1 = r.get(“test1”, “Data1”).toString(); *//returns “Value1”*

int Data2 = Integer.parseInt((“test1”, “Data2”).toString()); *//returns 2*

double Data3 = Double.parseDouble((““test1”, Data3”).toString()); *//returns 3.3*

HashMap<String, Object> = r.getModule(“test1”); *// returns HashMap<String, Object>:*

*“Data1”->”Value1”,*

*“Data2”->2,*

*“Data3”->3.3;*

*//these would be objects, but in the debugger, they are their correct types.*Coordinate Class

A screenshot of a cell phone

Description automatically generated

BEGIN directionTo (other)

equal <- this.equals(other)

BEGIN IF equal

RETURN dir.N

END IF

xDist <- x - other.x

yDist <- y - other.y

xGreater <- Math.abs(xDist) > Math.abs(yDist)

BEGIN IF !xGreater

BEGIN IF yDist < 0

RETURN dir.S

END IF

BEGIN ELSE

RETURN dir.N

END ELSE

END IF

BEGIN ELSE

BEGIN IF xDist < 0

RETURN dir.E

END IF

BEGIN ELSE

RETURN dir.W

END ELSE

END ELSE

END directionTo

BEGIN parseFromTS (tbp)

BEGIN TRY

BEGIN IF !COORDINATE\_REGEX.matcher(tbp).matches()

RETURN TurretFrame.NULL\_COORD

END IF

xIndexStart <- tbp.indexOf('x') + 2

xIndexEnd <- tbp.indexOf('y') - 2

yIndexStart <- tbp.indexOf('y') + 2;

yIndexEnd <- tbp.length() - 1

xStr <- tbp.substring(xIndexStart, xIndexEnd)

yStr <- tbp.substring(yIndexStart, yIndexEnd)

BEGIN IF !main.INT\_REGEX.matcher(xStr).matches() OR !main.INT\_REGEX.matcher(yStr).matches()

RETURN TurretFrame.NULL\_COORD

END IF

x <- Integer.parseInt(xStr)

y <- Integer.parseInt(yStr)

RETURN NEW Coordinate(x, y)

END parseFromTS

BEGIN compareTo (other)

BEGIN IF this.equals(other)

RETURN 0

END IF

xComparison <- Integer.compare(x, other.x)

yComparison <- Integer.compare(y, other.y)

BEGIN IF yComparison != 0

RETURN yComparison

END IF

BEGIN ELIF xComparison != 0

RETURN xComparison

END ELIF

BEGIN ELSE

RETURN 0

END ELSE

END compareTo

The coordinate class is used to keep track of where things are.

directionTo (Coordinate o): This method gives the direction to another coordinate. Firstly, it checks if the two coordinates were equal, and if so, it just returns N (meaning North). Then, it gets the distance, to their x, and their y. It uses that to work out, if the x distance is greater or the yDistance. Then, if the y distance is greater, then, it checks if the distance was positive or negative. If ours minus theirs is positive, we must be above them, so it returns S (meaning South). If negative it returns N (meaning North). It then runs the same check, replacing y with x, and returning either E (meaning East) or W (meaning West).

parseFromTS (String tbp): This method is static, and parses a String to a Coordinate, similar to Integer.parseInt or Double.parseDouble. Firstly, it checks if the string matches a regular expression, which is defined earlier. If not, it returns NULL\_COORD, a value with Integer.MAX\_VALUE for x and y. Then it uses indexOf and substring, to get the string values of the x and y. It then checks them against another regex (just in case), and if either fail, it returns NULL\_COORD. Else, it just parses the values, and returns a new Coordinate with them.

compareTo (Coordinate o): This overrides the compareTo method in comparable, which Coordinate implements. First it checks if they are equal, and if so returns a 0. If not, it makes two variables, one for the comparison of x, and one for the comparison of y, and it does this in 1 line, using Integer.compare(x, o.x), and Integer.compare(y, o.y). Then, to give the appearance to the user (this is used for sorting algorithms), that each row is given, part by part, it first returns the yComparison (assuming the ‘y’s aren’t equal), and then if the ‘y’s are equal(assuming the ‘x’s aren’t equal), it returns the xComparison. Then, as a just in case, it returns a 0.

Coordinate a = new Coordinate(10, 11);

Coordinate b = new Coordinate(0, 1);

String direction = a.directionTo(b).toString(); *//returns “North”*

int comparison = a.compareTo(b); *//returns 1*

Coordinate c = Coordinate.parseFromTS(“Coordinate[x=245, y=0]”);

*// is a new Coordinate, where x = 245, and y = 0*

### Custom Change Listeners

A screenshot of a cell phone

Description automatically generated

NB: Whilst only these methods are in the diagram, some are in the PlayerManager, and so I will also detail them here.

These classes, are used as action listeners, to update several classes on when the player dies, wins, or their money or lives change. Also, due to the fact that this is an action listener, addBooleanChangeListener, and stateChanged are methods that change as they must be overridden. BooleanChangeEvent, only has a constructor.

BEGIN dispatchEvent ()

FINAL event <- NEW BooleanChangeEvent(this)

BEGIN FOR listener : listeners

dispatchRunnableOnEventQueue(l, event)

ENDD FOR

END dispatchEvent

BEGIN dispatchRunnableOnEventQueue(l, event)

EventQueue.invokeLater(() -> listener.stateChanged(event))

stateChanged (BooleanChangeEvent e): This method is the one that is overridden by the class that needs to be notified (though not relating to the notify() method). This however isn’t seen in my code, as I have mainly used lambda expressions.

addBooleanChangeListsener (BooleanChangeListener l): This method is on the playerManager, and adds a listener to the list that must be notified of the change.

dispatchEvent (): This event gives notice to every listener, by going through with a for loop, and invoking later the stateChanged method, on the listener with the event.

instanceThatExtendsFromDispatcher.addBooleanChangeListsener (e -> {

System.out.println(“Change Triggered”)

});

System.out.println(“Wait starting”);

TimeUnit.SECONDS.sleep(10);

instanceThatExtendsFromDispatcher.doEvent();

*//prints the wait starting message, waits 10 seconds, and prints change triggered.*

### Resource Manager

A screenshot of a cell phone

Description automatically generatedThe ResourceManager is used to get images and icons from the internet. This is used rather than ImageIO.read() in every class, as with this – there is no need to get the same image multiple times.

BEGIN getImg (url)

BEGIN IF ALL\_IMAGES.containsKey(url)

RETURN clone(ALL\_IMAGES.get(url))

END IF

img <- null

BEGIN TRY

img <- ImageIO.read(url)

ALL\_IMAGES.put(url, img)

END TRY

BEGIN CATCH e

RETURN none

END CATCH

RETURN getImg(url)

END getImg

BEGIN clone (original)

newOne <- NEW BufferedImage(original.getWidth(none), original.getHeight(none), BufferedImage.TYPE\_INT\_ARGB)

newOne.getGraphics().drawImg(original, 0, 0, none)

RETURN newOne

END clone

getImg(URL url): This method gets an image, either from the database or from the internet, depending on whether we already have it or not. If we already have it – then just return a cloned version of it, from a hashMap. If not, then get that image, and put it in the hashMap, and then return the function – and it should clone it and give it.

Clone(Image original): This method clones an Image, by creating a BufferedImage with the width and height of the original, and drawing on the original.

URL url = new URL(“https://static.aqa.org.uk/assets/image/0015/51135/aqa\_og\_logo.png”);

Image img = ResourceManager.getImg(url);

*//img is the AQA logo, with transparency.*

## Class Diagrams and Algorithm Explanations – Rendered Classes

**NB for the Entity classes and the canvas**: Unfortunately, I cannot provide sample code, as lots of the code depends on runtime generated data, such as enemies to hit.

### A screenshot of a video game Description automatically generatedSquare Class

The Square class is used for rendering, and the base class extends from BufferedImage, and has many child classes. There is also a squareCollection, and a squareParser. The squareCollection keeps track of squares, and squares for turrets, as well as the enemy path. The squareParser gives that information to the squareCollection, and creates a 2D Array of Squares. The squareParser has two constructors, one private for cloning.

BEGIN squareParser.setUpHashMap ()

module <- r.getModule("refs")

chars <- module.keySet().toArray()

fnsFromR <- module.values().toArray()

BEGIN FOR i <- 0; i < chars.length; i++

c <- chars[i].toString().charAt(0)

fn <- fnsFromR[i].toString()

fns.put(c, fn)

END FOR

END squareParser.setUpHashMap

BEGIN squareParser.squareParser(reader)

this.r <- reader

w <- main.NUM\_OF\_TILES\_WIDTH

h <- main.NUM\_OF\_TILES\_HEIGHT

fns <- new HashMap<>()

setUpHashMap()

tbp <- r.get("mapDeets", "map").toString().toCharArray()

squares <- new Square[w][h]

x <- 0

y <- 0

BEGIN FOR c : tbp

fn <- fns.get(c)

newSquare <- none

ici <- new Coordinate(x, y)

BEGIN SWITCH fn

BEGIN CASE "general\_big.png"

newOne <- new pathSquare()

BREAK

END CASE

BEGIN CASE "happy\_big.png"

newOne <- new homeBase()

BREAK

END CASE

BEGIN CASE "satan\_big.png"

newOne <- new PathSquare()

satan <- ici

BREAK

END CASE

BEGIN CASE "turret\_base\_big.png"

newOne <- new turretSquare()

BREAK

END CASE

BEGIN CASE "nothing\_big.png"

newOne <- new nothingSquare()

BREAK

END CASE

BEGIN CASE DEFAULT

newOne <- new nothingSquare()

BREAK

END CASE

END SWITCH

squares[x][y] <- newOne

x++

BEGIN IF x == w

x <- 0

y++

END IF

END FOR

END squareParser.squareParser

BEGIN squareCollection.getCoordinates (tbp)

path = new ArrayList<Coordinate>()

chars = tbp.toCharArray()

BEGIN FOR i <- 0; i < chars.length; i+=4

xStr <- chars[i] + "" + chars[i + 1]

yStr <- chars[i + 2] + "" + chars[i + 3]

BEGIN IF !main.INT\_REGEX.matcher(xStr).matches() OR !main.INT\_REGEX.matcher(yStr).matches()

CONTINUE

END IF

x <- Integer.parseInt(xStr)

y <- Integer.parseInt(yStr)

newOne <- new Coordinate(x, y)

path.add(newOne)

END FOR

RETURN path

END squareCollection.getCoordinates

setUpHashMap (): Firstly, the algorithm gets the refs module from the CfgReader. This module contains references from codes used in the map section to file names of the corresponding images. Then it gets all of the characters and the file names. Then for each of the characters, it gets the character, and the file name and converts them to their proper type (char, and String), and adds them to the new HashMap.

squareParser (): Firstly, it sets the field of the CfgReader r, and gets the width of tiles and the height of tiles, and creates a 2D array with those widths and heights. It also gets a character[] of the map, calling it toBeParsed. It initialises the hashMap of fileNames, and sets up the hashMap using the above method. It initialises two integers, called x and y. Then, for each character in toBeParsed (or tbp), it gets the file name from the array, and initialises a temporary variable of type square. Then it goes through a switch statement, and dependent on file name, an object extending from Square is created. The 2D array at x and y, is set to that new object. X is incremented. If x is equal to the width, then x is set back to 0, and y incremented.

getCoordinates (String toBeParsed): This gets a list of coordinates from a string of numbers. Firstly, it initialises an ArrayList of Coordinates called path, and creates a character array from the toBeParsed.toCharArray() called chars. Then, it goes through a for loop with i = 0, while less than chars.length, but i += 4. For each iteration, it gets the characters at (i)-> (i + 1) and (i + 2) -> (i + 3). Theseare the x and y. It checks for a regex, and if it - fails it has a continue statement, in that the for loop should move to the next iteration. If it succeeds, it parses the x and y, calls them x and y, creates a new Coordinate with them and adds it to the list. At the end it returns the path.

*//for purposes of testing, currentLevel = 1*

CfgReader sqpCfg = new CfgReader(main.MAPS\_LOC + "stg" + currentLevel + ".cfg");

squareParser sqp = new squareParser(sqpCfg);

squareCollection sqc = new squareCollection(sqp);

sqc.getEnemyPath()

sqc.getAvailableTurretSquares()

*//getEnemyPath returns:*

*/\**

*result = {ArrayList@2829} size = 13*

*0 = {Coordinate@2831} "Coordinate[x=9, y=1]"*

*1 = {Coordinate@2832} "Coordinate[x=9, y=8]"*

*2 = {Coordinate@2833} "Coordinate[x=2, y=8]"*

*3 = {Coordinate@2834} "Coordinate[x=2, y=7]"*

*4 = {Coordinate@2835} "Coordinate[x=1, y=7]"*

*5 = {Coordinate@2836} "Coordinate[x=1, y=5]"*

*6 = {Coordinate@2837} "Coordinate[x=7, y=5]"*

*7 = {Coordinate@2838} "Coordinate[x=7, y=3]"*

*8 = {Coordinate@2839} "Coordinate[x=2, y=3]"*

*9 = {Coordinate@2840} "Coordinate[x=2, y=2]"*

*10 = {Coordinate@2841} "Coordinate[x=3, y=2]"*

*11 = {Coordinate@2842} "Coordinate[x=3, y=1]"*

*12 = {Coordinate@2843} "Coordinate[x=1, y=1]"*

*\*/*

*//getAvailableTurretSquares returns :*

*/\**

*result = {ArrayList@2858} size = 14*

*0 = {Coordinate@2860} "Coordinate[x=2, y=0]"*

*1 = {Coordinate@2861} "Coordinate[x=3, y=4]"*

*2 = {Coordinate@2862} "Coordinate[x=4, y=4]"*

*3 = {Coordinate@2863} "Coordinate[x=6, y=4]"*

*4 = {Coordinate@2864} "Coordinate[x=2, y=6]"*

*5 = {Coordinate@2865} "Coordinate[x=3, y=6]"*

*6 = {Coordinate@2866} "Coordinate[x=6, y=7]"*

*7 = {Coordinate@2867} "Coordinate[x=7, y=7]"*

*8 = {Coordinate@2868} "Coordinate[x=2, y=9]"*

*9 = {Coordinate@2869} "Coordinate[x=3, y=9]"*

*10 = {Coordinate@2870} "Coordinate[x=5, y=9]"*

*11 = {Coordinate@2871} "Coordinate[x=6, y=9]"*

*12 = {Coordinate@2872} "Coordinate[x=8, y=9]"*

*13 = {Coordinate@2873} "Coordinate[x=9, y=9]"*

*\*/*

### A screenshot of a cell phone Description automatically generatedBase Entity Class

This entity class is used for rendering, and also for utility functions to do with movement, position, and other features useful for objects that need to be rendered.

BEGIN turnFromArrToScrnPlusHalfTile (original, overrideHalf)

bigX <- original.getX() \* main.TILE\_WIDTH

bigY <- original.getY() \* main.TILE\_HEIGHT

smallX <- overrideHalf.getX() / 2

smallY <- overrideHalf.getY() / 2

x <- bigX + smallX

y <- bigY + smallY

fin <- new Coordinate(x, y)

RETURN fin

END turnFromArrToScrnPlusHalfTile

BEGIN changeX (dst)

newXInTile <- getXYInTile().getX() + dst

newXInArr <- getXYInArr().getX()

oldYTile <- getXYInTile().getY()

oldYArr <- getXYInArr.getY()

BEGIN IF newXInTile < 0

newXInTile += main.TILE\_WIDTH

newXInArr--

BEGIN IF newXInTile < 0

newXInTile <- 0

END IF

END IF

BEGIN ELIF newXInTile > main.TILE\_WIDTH

newXInTile -= main.TILE\_WIDTH

newXInArr++

BEGIN IF newXInTile >= main.TILE\_WIDTH

newXInTile <- main.TILE\_WIDTH - 1

END IF

END ELIF

BEGIN IF newXInArr < 0

newXInArr <- 0

END IF

BEGIN ELIF newXInArr >= main.NUM\_OF\_TILES\_WIDTH

newXInArr <- main.NUM\_OF\_TILES\_WIDTH - 1

END ELIF

XYInArr -> new Coordinate(newXInArr, oldYArr)

XYInTile -> new Coordinate(newXInTile, oldYTile)

END changeX

BEGIN addHitBoxTolerances (onScrn, HITBOX)

currentX <- onScrn.getX()

currentY <- onScrn.getY()

addedX <- -HITBOX.getX()

addedY <- -HITBOX.getY()

onScrn.setX(currentX + addedX)

onScrn.setY(currentY + addedY)

RETURN onScrn

END addHitBoxTolerances

turnFromArrToScrnPlusHalfTile (Coordinate original, Coordinate override): This method is used to turn Coordinates from XYInArr (Arr meaning the Square[][] in the squareParser) to XYOnScrn (coordinates that can be given to the canvas), and XYInTile is the x and y in a square. NB: The version of this without an override, uses a Coordinate with main.TILE\_WIDTH and main.TILE\_HEIGHT. This method works by getting a bigX, or the TILE\_WIDTH \* the original.getX(). This works as if the x was 0, it would still return 0. It then also does this for the y, and then it gets the smallX, which is half of the getX from the override. It then returns this.

changeX (int dst): This method is used to change the x coordinate of an object. NB: The changeY is the same, but all of the widths are swapped with heights, and the ‘x’s swapped with the ‘y’s. Firstly, it initialises 4 variables, with the x and y of the XYInArr and the XYInTile, but the x of XYInTile has the dst argument added to it. Then it checks if the new x in the tile is less than 0 (this could happen, as to move East, the dst (meaning distance) is negative). If so, then it adds the TILE\_WIDTH to the negative x, and moves it one tile to the left in the Square[][] x. Then, if the inTile x value is still negative, we assume that there has been an error, and that we just reset it to 0. The first if statement is done.

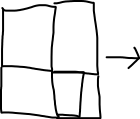
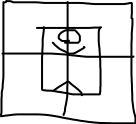
Then we check if the x in the tile is too big (ie. Bigger than the TILE\_WIDTH), in which case we do the reverse than if it was negative, in that we minus the TILE\_WIDTH from the x in tile value, and add one to the x in arr value. Then, we check (still inside the second if statement), if the value is still too big, in which case we set it to just below TILE\_HEIGHT.

Then, after all of the if statements, we check if the x in arr is too low, in which case it reverts to 0, and then if it is too high, in which case it reverts to NUM\_OF\_TILES\_WIDTH – 1.

Then we use the xInArr and xInTile, as well as the y variables from the start to setXYInArr, and XYInTile.

NB: XYInArr is the x and y coordinate the keeps track of where the Entity is using the tile system. For a visual representation, each square on the canvas is a tile. The XYInTile keeps track of where the Entity is on that tile. The method below is used in combination with getXYOnScrn to render and get targets correctly.

addHitBoxTolerances (Coordinate onScrn, Coordinate hitBox): Firstly, it gets the current x and y from the onScrn coordinate. Then it gets the negative of the hitbox’s x and y coordinates. It then adds these coordinates onto the onScrn’s x and y, and returns that. This works, because of the way that graphics work in Java, in that all is rendered from the top left corner. See below diagram for if we want to render in the centre of the grid. At first, it renders from the top left, and we gave it the centre, so it rendered incorrectly, but if we minus, we move North East, and so can render in the correct place



### Turret Class

A screenshot of a cell phone

Description automatically generated

The turretActual class is used to find enemies, and wait for a cooldown, and once they can fire, they create instances of the bulletActual class. In the runThread of turretActual, this is what it does. Turrets are created by the turretManager, which is covered later. Part of creating a turretActual, is a turretTemplate, a dictionary of which is kept in the turretDictionary, which in it’s main method uses a set of file names (given in a String[]), to create config file readers. Then, it gets information and creates turretTemplates, and puts them into a HashMap, and their names into another ArrayList. Due to the fact that most of the methods here are simple or in Runnables and Threads, rather than methods, I will cover them. The only methods in enemyFilter are filters for enemies, which are if statements to check if the enemy is done to make sure it doesn’t aim for an enemy which will soon (1 – 2 frames) be off the rendering list, or is out of range.

turretActual.runThread <- NEW Thread(() -> {

BEGIN WHILE !pm.isDone() AND stillWorking

BEGIN TRY

BEGIN IF hasToWait

TimeUnit.MILLISECONDS.sleep(differenceMs)

END IF

BEGIN ELSE

hasToWait <- TRUE

END ELSE

END TRY

BEGIN CATCH e

PRINT "Cooldown violated"

END CATCH

enemies <- enemyFilter.filterEnemies(entities.clone(), turret.getRangeInt(), getXYInArr().clone())

BEGIN IF enemies.size() != 0

enemyToHit <- enemies.get(0)

bulletToHitIt <- new bulletActual(getXYInArr.clone(), turret.getBulletFn(), turret.getBulletSpd(), turret.getRangeInt())

shotsFired.add(bulletToHitIt)

END IF

BEGIN ELSE

hasToWait <- FALSE

END ELSE

BEGIN FOR entity : shotsFired.clone()

bullet <- (bulletActual) entity

BEGIN IF ba.isDone()

shotsFired.remove(entity)

PRINT "Bullet removed"

END IF

END FOR

})

bulletActual.runThread <- NEW Thread(() -> {

av <- main.BOUND \* 2

distInPx <- (spd \* av) / SPEED\_DIVISOR

BEGIN IF distInPx < 0

distInPx <- spd

END IF

BEGIN WHILE !isDone()

BEGIN IF !enemyToHit.hasBeenSpawnedYet()

dead <- TRUE

PRINT "?????"

BREAK

END IF

BEGIN IF getXYInArr().equals(enemyToHit.getXYInArr()) OR hit

BEGIN IF getXYInTile().distTo(enemyToHit.getXYInTile()) < main.BOUND OR hit

enemyToHit.damage(dmg)

hit <- TRUE

PRINT "Enemy killed - Sincerely bullet"

RETURN

END IF

END IF

BEGIN IF enemyToHit.isDone()

dead <- TRUE

PRINT "Enemy already done"

RETURN

END IF

BEGIN IF getXYInArr.distTo(enemyToHit.getXYInArr()) > range + 1

dead <- TRUE

PRINT "he got away"

RETURN

END IF

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(moveGap)

END TRY

BEGIN CATCH e

e.printStackTrace()

END CATCH

onScrnTarget <- Entity.turnFromArrToScrnPlusHalfTile(enemyToHit.getXYInArr(), Coordinate.ZERO)

onScrnTarget <- Entity.addHitBoxTolerances(onScrnTarget, CENTRE\_OF\_HITBOX)

direction <- getXYOnScrn().directionTo(onScrnTarget)

dist <- getXYOnScrn().distTo(onScrnTarget)

BEGIN IF distInPx >= dist

setXYInArr(enemyToHit.getXYInArr())

setXYInTile(enemyToHit.getCentreOfHitbox())

hit <- TRUE

CONTINUE

END IF

BEGIN SWITCH direction

BEGIN CASE N

changeY(-distInPx)

END CASE

BEGIN CASE S

changeY(distInPx)

END CASE

BEGIN CASE E

changeX(distInPx)

END CASE

BEGIN CASE W

changeX(-distInPx)

END CASE

END SWITCH

END WHILE

})

turretActual.runThread = new Thread (() -> {}): The idea this thread, is to, while the playerManager is still alive, and the turret is still on the render list (dictated by turretManager, as when it is sold, it is told to stop), to check for new enemies, and to update all of the bullets. Firstly, it checks if it has to wait, and if it does, then it waits. If not, then it sets hasToWait as true, so it will wait next time. Then it filters the enemies using the filterEnemies method from the enemyFilter. Then it checks is there even are any enemies to hit, and if so – it gets the first one, creates a bullet for it, and adds that bullet to the render list. If not, then it sets hasToWait to false, as it doesn’t need to wait, if there was no firing. Then it goes through all of the bullets, and if they are done, it removes them from the rendering list.

bulletActual.runThread = new Thread (() -> {}): Then this thread runs the bullet, checking if it has hit one of the death conditions, and if so removing it else, running closer to the enemy. Firstly, it checks if the enemy has been spawned, and if not, then it sets dead = true, and restarts the loop, rechecking the condition, and so ending the loop.

Then, it checks if it is close enough to the enemy, and if it is, it damages the enemy, sets hit = true, and gets out of the loop.

Then it checks if the enemy is done, and if so it sets dead = true, and exits out.

Finally in the checks, is if the enemy leaves range, and if so, the bullet dies and exits out of the loop.

If it makes it past all of the checks, then it waits for the move gap (as the pixel distances aren’t large, but the bullets and enemies need to be slow, a move gap was introduced, so that they move less often and less distance, slowing them down), and then finds a direction to the enemy (accounting for hitbox), and then makes another check. If the distance we can go is greater than or equal to the distance we need to go, then the bullet teleports there by use of setXYInArr, and setXYInTile, sets hit to true, and exits the loop. If not it opens a switch statement, and moves differently dependent on direction (either North, South, East, or West).

### A screenshot of a video game Description automatically generatedEnemy Class

This is the enemyActual class, and it is created similar to the turrer, in that there is an enemyTemplate, and an enemyDictionary, only here there is no bullet, and there is no filter.

runThread <- NEW Thread (() -> {

BEGIN IF pm.isDone()

RETURN

END IF

current <- System.CurrentTimeMillis

hasBeenSpawned <- TRUE

main.SOUNDS.get("spawn.wav").start()

BEGIN WHILE !isDone() AND !pm.isDone()

BEGIN IF currentStep == squares.getEnemyPath().size()

hasHit <- TRUE

RETURN

END IF

BEGIN IF currentHP <= 0

isDead <- TRUE

RETURN

END IF

diff <- System.CurrentTimeMillis() - current

current <- System.CurrentTimeMillis()

BEGIN IF diff < 0

diff <- 0

END IF

BEGIN IF diff > MOVE\_GAP

diff <- MOVE\_GAP

END IF

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(MOVE\_GAP - diff)

END TRY

BEGIN CATCH e

PRINT "Enemy move gap changed"

END CATCH

currentCoord <- squares.getEnemyPath().get(currentStep)

onScrnTarget <- Entity.turnFromArrToScrnPlusHalfTile(currentCoord)

onScrnTarget <- Entity.addHitBoxTolerances(onScrnTarget, CENTRE\_OF\_HITBOX)

direction = getXYOnScrn().directionTo(onScrnTarget)

dist = getXYOnScrn().distTo(onScrnTarget)

BEGIN IF distInPx >= dist

setXYInArr(currentCoord)

setXYInTile(IN\_TILE\_TARGET)

currentStep++

CONTINUE

END IF

BEGIN SWITCH direction

BEGIN CASE N

changeY(-distInPx)

END CASE

BEGIN CASE S

changeY(distInPx)

END CASE

BEGIN CASE E

changeX(distInPx)

END CASE

BEGIN CASE W

changeX(-distInPx)

END CASE

END SWITCH

END WHILE

})

runThread = new Thread(() -> {}): The enemy’s runThread is designed to move the enemy, and to check for deaths and win conditions, similar to the bullet runThread.

Firstly, it double checks that the player has not died since the enemySpawner spawned the enemy. If so, then it exits from the thread. Then it gets the current time, and once again checks for player death, with the same result on true as last time.

Then it begins the main while loop, while the enemy isn’t done and neither is the player. Firstly it checks if it has reached the end – each time the step is reached, step is incremented, so if the currentStep is greater than or equal to the length of the list, then we have reached the end (index: length - 1), and incremented to get an index of the length. If so, then it sets hasHit to true, and returns out of the loop.

Then it checks if our current health is less than or equal to 0. If so, then we are dead, and we exit out of the loop.

If none of those are true – we continue.

This begins with getting the difference between last time we went through this and this time, getting the time it took to process the movement, and we create a difference variable to store this. We check it is positive and in acceptable range (so we don’t wait for too long or a negative time), and then we wait for the move gap minus that difference variable. We then get the current coordinate (in inArr format), from the list of enemy squares, and turn it to onScrn format, and add tolerances for our hitbox. Then the movement is identical to the bullet, in that we get direction and distance, check if the distance we can go is greater than or equal to the distance to go, and if so increment current step, and teleport to the place. If not the exact same switch statement is used.

### A circuit board Description automatically generatedCanvas Class

The canvas class extends from JComponent, and is used to render all of the Entites and squares. The vCanvas is a canvas that is drawn onto, and then the actual graphics object in paintComponent, is set to that of the vCanvas, to avoid tearing.

BEGIN dieOrWin (imgs, txt)

vCanvas.getGraphics().drawString(txt, main.TILE\_WIDTH, main.TILE\_HEIGHT)

BEGIN FOR y <- 0; y < main.NUM\_OF\_TILES\_HEIGHT; y++

BEGIN FOR x <- 0; x < main.NUM\_OF\_TILES\_WIDTH; x++

vCanvas.getGraphics().drawImage(imgs[i], x \* main.TILE\_WIDTH, y \* main.TILE\_HEIGHT, none)

vCanvas.getGraphics().drawImage(txt, main.TILE\_WIDTH, main.TILE\_HEIGHT)

repaint()

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(DIFFERENCE\_BETWEEN\_COLOUR\_CHANGE)

END TRY

BEGIN CATCH e

PRINT "DIFF BETWEEN COLOUR CHANGE INTERRUPTED"

END CATCH

END FOR

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(DIFFERENCE\_BETWEEN\_SQUARES)

END TRY

BEGIN CATCH e

PRINT "DIFF BETWEEN SQUARES CHANGE INTERRUPTED"

END CATCH

END FOR

END dieOrWin

BEGIN paintComponent (g)

super.paintComponent(g)

finishedRendering <- FALSE

vCanvas <- new BufferedImage(

(getWidth() == 0 ? main.WINDOW\_WIDTH : getWidth()),

(getHeight() == 0 ? main.WINDOW\_HEIGHT : getHeight()),

BufferedImage.TYPE\_INT\_ARGB)

xOnScrn <- 0

yOnScrn <- 0

w <- main.TILE\_WIDTH

h <- main.TILE\_HEIGHT

squares <- sqc.getSquares()

BEGIN FOR yInArr <- 0; yInArr < squares[0].length; yInArr++

BEGIN FOR xInArr <- 0; xInArr < squares[0].length; xInArr++

img <- squares[xInArr][yInArr]

vCanvas.getGraphics().drawImage(img, xOnScrn, yOnScrn, none)

xOnScrn += w

END FOR

xOnScrn <- 0

yOnScrn += h

END FOR

temp <- entities.clone()

temp.removeIf(NULL\_PREDICATE)

BEGIN IF temp.size() != 0

BEGIN FOR e : temp

BEGIN TRY

img <- none

BEGIN SWITCH e.getType()

BEGIN CASE enemy

casted <- (enemyActual) e

BEGIN IF casted.haveIBeenSpawned()

img <- casted.getImg()

BREAK

END CASE

BEGIN CASE DEFAULT

img <- e.getImg()

BREAK

END CASE

END SWITCH

BEGIN IF img != none

x <- e.getXYOnScrn().getX()

y <- e.getXYOnScrn().getY()

vCanvas.getGraphics().drawImage(img, x, y, null)

END IF

END TRY

BEGIN CATCH e

PRINT "ENTITY NOT FOUND"

e.printStackTrace()

END CATCH

END FOR

END IF

BEGIN ELSE

BEGIN IF !deathStarted && !winStarted

BEGIN IF playerIsDead

deathStarted <- TRUE

deadThread.start()

END IF

BEGIN ELIF playerHasWon

winStarted <- TRUE

winThread.start()

END ELIF

END IF

END ELSE

BEGIN IF g != none

g.drawImage(vCanvas, 0, 0, none)

repaint()

finishedRendering <- TRUE

END paintComponent

These two algorithms are key to the canvas, and one is a winThread, and the other is the main paintComponent method, extending from the JComponent class. The only other parts are the win and deadThreads, which create a score, and a StringJoiner, and pass images to the

dieOrWin (Buffered[] imgs, String txt): This method slowly draws squares onto the screen dependent on win or loss. Firstly, it draws on the text at x=TILE\_WIDTH, and y=TILE\_HEIGHT. Then it goes through each of the tiles, row, by row, and every image of the img array (incrementing x by the TILE\_WIDTH every after every full square, and the y by TILE\_HEIGHT after every full row, also setting x to 0 after every row), and draws on the image. It also redraws the text every time.

paintComponent (Graphics g): This method paints onto the JComponent. Firstly it calls the super, as this is an overriding method and it has to call the super. Then it recreates the vCanvas object, using the widths and heights of the component, if they are not 0 – if they are, then it uses main.WINDOW\_HEIGHT and main.WINDOW\_WIDTH.

Then it initialises 4 variables – xOnScrn (at 0, meaning the current x position on the canvas), yOnScrn (at 0, meaning the current y on the canvas), w (at main.TILE\_WIDTH, meaning the width of one tile), and h (at main.TILE\_HEIGHT, meaning the height of one tile), and gets the squares[][]. Firstly, it renders the squares (to ensure that everything appears above the squares rather than the squares appearing on top and nothing else showing), using a for loop for x in arr, and for y in arr, with arr referring the to the 2D Squares array, and the xOnScrn and yOnScrn are used to keep track of x and y positions, and are incremented similar to the dieOrWin rendering, only these only have one image. The base Square class already extends from BufferedImage, so there is no need for a getImg method.

Then we clone the list of entities (to avoid concurrentModificationExceptions due to multiple objects accessing the list). It then runs removeIf on the list with an earlier created NULL\_PREDICATE which returns if the Entity object is null. So all null objects from the list have been removed.

Then we check if the list is still greater than 0, and if so, we open a try statement and inside a switch statement with the Entity type, as we don’t want to render any unspawned enemies (players may wonder why an enemy isn’t moving, and then get angry when they place turrets to shoot at it, but they can’t shoot it due to checks in the turret). If it is unspawned the image remains null, but if it is spawned, we get the enemy image, and in the default part (for bullets and turrets), we just get the image.

Then we double check for null images, and if not – we get the position of the entity using getXYOnScrn and render their image there.

We then end the try, and if we caught any errors, then we print “ENTITY NOT FOUND”, and then print the exception’s stack trace.

After that, we check if the both the playerDeathThread hasn’t started nor the playerWinThread, and if they haven’t started, then

## Class Diagrams and Algorithm Explanations – Gameplay Classes

**NB: For Gameplay classes except for PlayerManager:** Due to runtime generated data required, I cannot give code examples for these classes, similar to the Entity classes.

### Player Manager Class

A screenshot of a computer screen

Description automatically generatedThe PlayerManager class is designed to keep track of the statistics of the player, such as their lives left, their money and whether they have lost or won. It also implements the BooleanChangeDispatcher, and so can can have action events attached to it, and a list of the listeners is in the listeners list. As mentioned with the CustomActionListeners package earlier, all methods relating to them have been described there rather than here.

BEGIN buy (amt)

BEGIN IF amt > money

RETURN FALSE

END IF

money -= amt

dispatchEvent()

RETURN TRUE

END buy

Buy(int amnt): This method is used for buying money, and returns whether the transaction went through. It starts with an if statement: If the amount to buy is greater than the money we have – return false as we cannot buy it, but if we can, continue, buy it, dispatchEvent, and return true. dispatchEvent tells the listeners of a change, and we return true to say that we have bought it.

PlayerManager pm = new PlayerManager(100, 10);

System.out.println(pm.buy(90)); //prints true

System.out.println(pm.getMoney()); //prints 10

System.out.println(pm.buy(90)); //prints false

System.out.println(pm.getMoney()); //prints 10

pm.takeHearts(5);

System.out.println(pm.getHearts()); //prints 5

System.out.println(pm.isDead()); //prints false

pm.takeHearts(5);

System.out.println(pm.getHearts()); //prints 0

System.out.println(pm.isDead()); //prints true

System.out.println(pm.haveIWon()); //prints false

pm.hasWon();

System.out.println(pm.haveIWon()); //prints false

pm.addBooleanChangeListener(e -> {

System.out.println("Change Triggered, " + e.toString());

});

pm.resetH(); //prints "Change Triggered" + event.toString() with the Action

//Listener

pm.hasWon(); //print as above

System.out.println(pm.haveIWon()); //prints true, but before the action listeners,

//due to EventQueue.invokeLater

### TurretFrame and TurretManager

A screenshot of a cell phone

Description automatically generated

The turretFrame and turretManager manage the turrets, and display them on the canvas, and the turretFrame is on the right hand side of the window.

### Waves

## Class Diagrams and Algorithm Explanations – Main Class

### Main Class

### Apex Turrets Class

# Technical Solution

# Testing

# Evaluation