*Jack Maguire*

*Candidate Number: 0109*

*Candidate Centre: 51518*

*apex turrets*

# Table Of Contents

[Table Of Contents 1](#_Toc33262904)

[Research 3](#_Toc33262905)

[Background and Problem Definition 3](#_Toc33262906)

[Analysis of Similar Systems 3](#_Toc33262907)

[Kingdom Rush 3](#_Toc33262908)

[Bloons TD5 5](#_Toc33262909)

[Brackeys Tower Defence Game 6](#_Toc33262910)

[Identification of Users, their needs and acceptable limitations 7](#_Toc33262911)

[Identification of End User: 7](#_Toc33262912)

[Student group results 7](#_Toc33262913)

[Individual pupil interview: 8](#_Toc33262914)

[Objectives 9](#_Toc33262915)

[Proposed Solution with Justification of Language 9](#_Toc33262916)

[Use of Java 10](#_Toc33262917)

[Design 11](#_Toc33262918)

[Textual Overview 11](#_Toc33262919)

[Flowchart 12](#_Toc33262920)

[Class Diagrams and Algorithm Explanations – Utility Classes 13](#_Toc33262921)

[Config File Reader 14](#_Toc33262922)

[Custom Change Listeners 23](#_Toc33262923)

[Resource Manager 24](#_Toc33262924)

[Save System 25](#_Toc33262925)

[Class Diagrams and Algorithm Explanations – Rendered Classes 26](#_Toc33262926)

[Square Class 26](#_Toc33262927)

[Base Entity Class 30](#_Toc33262928)

[Turret Class 33](#_Toc33262929)

[Enemy Class 38](#_Toc33262930)

[Canvas Class 41](#_Toc33262931)

[Class Diagrams and Algorithm Explanations – Gameplay Classes 45](#_Toc33262932)

[Player Manager Class 45](#_Toc33262933)

[TurretFrame and TurretManager 46](#_Toc33262934)

[Waves 48](#_Toc33262935)

[Class Diagrams and Algorithm Explanations – Main Package 50](#_Toc33262936)

[54](#_Toc33262937)

[UI Design 55](#_Toc33262938)

[Technical Solution 56](#_Toc33262939)

[Testing 57](#_Toc33262940)

[Evaluation 59](#_Toc33262941)

[General Evaluation 59](#_Toc33262942)

[Things I would change if I did it again. 59](#_Toc33262943)

[Completion of Objectives 59](#_Toc33262944)

# Analysis

## Research

### Background and Problem Definition

Currently, at my school, 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 (but homework was forbidden in the day). In the Senior School, however there are vastly more pupils, and there are only spaces in which to talk with friends, 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. This is because the IT Staff allow self-coded Games, but no previous pupils have made large scale games - pupils are taught only python at a basic level, and most students do not continue past that. This is largely because my school does not let you do Computer Science at either A-Level or GCSE.

## Analysis of Similar Systems

### Kingdom Rush

One good example of a Tower Defence Game is Kingdom Rush, a popular game that has 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). There is no aggression in the micro-transactions.

However, whilst this game may be available on Steam and Mobile, 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, and to steam:

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

<https://store.steampowered.com/app/246420/>

Set out below are the pros and cons of the way in which Kingdom Rush works.

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

### Bloons TD5

A screen shot of a computer

Description automatically generatedAnother good example of a Tower Defence Game is Bloons TD5. In this game enemies are balloons (bloons), of different speed and popping strength. In this game instead of set turret spaces there are areas to put down balloon poppers. Unlike in Kingdom Rush, you have areas in which you can place down turrets rather than small specific points. In the level shown below, you can only put down turrets on land on the planets. The strategy of the positioning with limited money makes it an interesting and challenging game to play. Many types of turrets exist, and keyboard shortcuts can be used to quickly grab types of turrets to place. 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>

Set out below are the pros and cons for how Bloon TD5 works.

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

### Brackeys Tower Defence Game

Another example of a Tower Defence Game is the Brackeys Tower Defence Game. This game can be downloaded locally for free on a Mac, Windows and Linux. However a significant limitation is that you must compile the Unity Part using the code in the Github Repository. Many people don’t know how to do this and so it automatically limits how many people can play it. For people who are able to do this, the game has capacity for many different levels, tower upgrades, different enemies, and most importantly, the easy way of generating 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.

Below is an screenshot of the main level of the game and the level select screen.

A screenshot of a cell phone

Description automatically generatedA picture containing crossword puzzle, text

Description automatically generated

As shown by the screenshot above, the nodes are made of cubes and, other than the enemy path, take up the whole screen. Turrets can be placed on any node – after the user has bought the turret, a click is all that is needed to place it. In addition, all of the turrets can rotate fully. The UI is not exceptional but is clear: the number of lives remaining is shown at the top of the screen and the money, time remaining, and turrets available to buy are all at the bottom of the screen. In each level, there are waves of enemies which are predetermined. The user has the option however to select different levels of the game which give different waves of enemies.

Link to the github repository below:

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

|  |  |
| --- | --- |
| pros | cons |
| * free * desktop * no wifi needed * clear ui | * **have to compile unity project to use as a standalone.** * **simplistic** * **limited assets** * **simplistic enemies / ground** |

## Identification of Users, their needs and acceptable limitations

### Identification of End User:

The end user here will be the students at my school at break and lunch times. I did not interview people from other schools because other schools do not have the same restrictions my school has on the use of computers and furthermore tend to have more computer literate students by virtue of the fact that they typically offer Computer Science at GCSE and A-Level, and therefore other options for computer games are available to those students at those schools.

I discussed this with a number of different student groups and got their feedback. I also carried out a formal interview with a very diligent and computer-game loving student once I had just started the project, so that I could obtain more specific feedback on what the users wanted. I set out below the result of that market research.

### Student group results

These student groups asked for a system that is:

* Easy to use
* Short and sweet – easy to play for just 15 minutes of break.
* Free
* Advertising Free
* Replayable – fun and interesting, ie. Not just the novelty factor.
* Clean interface.

### Individual pupil interview:

**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: 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 hopes to run 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)

## Objectives

1. The game should not be repetitive.
   1. There should be different turrets.
      1. There should be at least two different turret templates.
      2. There should be upgrades available for each turret (at least one, but more should be able to come.)
         1. An upgrade should be a positive development to a turret property. For example, the turret could be able to shoot more often, or inflict more damage.
   2. There should be multiple different enemies
2. The game should be as efficient as possible.
   1. The storage space should be as efficient as possible to increase performance on the low-spec machines of my end-user.
3. Turret Features:
   1. You should be able to sell turrets for if the player makes a mistake.
   2. You should be able to upgrade turrets (mentioned earlier – see para 1.1.2).
   3. It should be clear where the bullets are and how the long the turret needs to wait for its cooldown period.
4. Bullet features:
   1. You should be able to see bullets – not just lines drawn.
   2. The bullets should have a limited range.
   3. The bullets should show which enemy their target is.
5. General Features:
   1. You should be able to choose the level and to play again.
   2. You should be able to see the whole map at once.
   3. The UI should be clear and reasonable – no menus with excessive submenus.
   4. There should be a leaderboard.
   5. You should be able to save a save file with details of scores etc., and view your score on a leaderboard.

## Proposed Solution with Justification of Language

After analysing similar systems, I have decided to use Java (JDK 13 (experimental)) as the language. Java is a language famous 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. It is widely used, with famous examples including Minecraft. 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.

|  |  |
| --- | --- |
| 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 or unreal)** |
|  |  |

I also used a popular IDE, IntelliJ IDEA.. I 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. 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.

The other main option I considered was Eclipse, famous for its Class Wizard, and external library manager. Ultimately, I decided not use Eclipse as it has little to no auto-code completion and compared to IntelliJ, for me, it seemed significantly less efficient.

# Design

## Textual Overview

1. The game starts with the game double checking whether the user wishes to play.
2. If the user says no, the program ends.
3. If they say yes it launches into the program.
4. A level select option is given and the user can select a level to play.
5. The UI is shown for the main game at the selected level – 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.
   1. The left hand side – comprises 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 the path. The start is the square that looks red and black, and the end is the bluer square. The enemies come in waves – a few enemies with a set gap between their spawning, 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.
   2. The right hand side – comprises 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. 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. There is also an upgrade button, which randomly gives a positive effect to the turret selected.
   3. The status window in the bottom right corner – this shows the player how much money and how many lives he or she has left, as well as all of their turrets (name, position, sell value, level).

## 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, on the Github Respository with the proposed solution – in the UML Diagrams folder.

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)

All diagrams were generated by pressing a few buttons in IntelliJ IDEA Ultimate as part of the built in UML Class diagram maker.

### Config File Reader

A screenshot of a cell phone

Description automatically generated

NB: This class’s format for .cfg files is loosely based off of the part files from Kerbal Space Program. Eg. (This is an actual part file)

PART

{

name = ubstrin

module = Part

author = AlexanderM

mesh = aerodynamicNoseCone.mu

scale = 1.0

rescaleFactor = 1

node\_stack\_bottom01 = 0.0, 0.0, 0.0, 0.0, -1.0, 0.0

CenterOfDisplacement = 0.0, 0.2, 0.0

TechRequired = stability

entryCost = 2000

cost = 240

category = Aero

subcategory = 0

title = #autoLOC\_500109 //#autoLOC\_500109 = Aerodynamic Nose Cone

manufacturer = #autoLOC\_501623 //#autoLOC\_501623 = Goliath National Products

description = #autoLOC\_500110 //#autoLOC\_500110 = Aerodynamic, lightweight and mostly non-explosive. As a reminder to all personnel operating nearby, this part is really sharp and it’d probably hurt if you fell on it.

attachRules = 1,0,1,1,0

stackSymmetry = 2

mass = 0.03

dragModelType = default

maximum\_drag = 0.1

minimum\_drag = 0.1

angularDrag = 0.5

crashTolerance = 10

maxTemp = 2400 // = 3400

emissiveConstant = 0.7

thermalMassModifier = 6.0

fuelCrossFeed = False

bulkheadProfiles = size1

tags = #autoLOC\_500111 //#autoLOC\_500111 = aero aircraft booster )cap drag fligh plane rocket speed stab stream

MODULE

{

name = ModulePartVariants

baseVariant = BlackAndWhite

useMultipleDragCubes = false

VARIANT

{

name = BlackAndWhite

displayName = #autoLOC\_8007122

themeName = BlackAndWhite

primaryColor = #ffffff

secondaryColor = #000000

TEXTURE

{

mainTextureURL = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_diffuse\_O

\_BumpMap = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_normal\_O

}

}

VARIANT

{

name = Dark

displayName = #autoLOC\_8007117

themeName = Dark

primaryColor = #4c4f47

TEXTURE

{

mainTextureURL = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_diffuse

\_BumpMap = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_normal

}

}

VARIANT

{

name = White

displayName = #autoLOC\_8007119

themeName = White

primaryColor = #ffffff

secondaryColor = #ffffff

TEXTURE

{

mainTextureURL = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_diffuse\_W

\_BumpMap = Squad/Parts/Utility/rockomaxAdapters/Assets/Rockomax\_Adapters\_normal

}

}

}

}

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 <- modules.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)

modules.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, each config file is made up of several modules, each of which has lots of properties and values and 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 therefore the indices of moduleNames to Strings of names are the same. This means that we can just get the index of the name and 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 parsed again at the other end. It works by using the getModule method to get a HashMap and passing the module and then uses that HashMap to get 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 Strings), 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. I have the checker to avoid blank lines, as the files care easier to read with blank lines.

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, there is a series of if statements. The first if statement is whether it is an integer.If so, then 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.

BEGIN 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

This coordinate class is a class to handle coordinates, with x and y coordinates.

BEGIN directionTo (other)

equal <- this.equals(other)

BEGIN IF equal

RETURN dir.Nothing

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 Nothing (though not null: there is a nothing value in the enumerator for direction). Then, it gets the distance, to their x and their y coordinates. It uses that to work out if the distance between the x coordinates is greater than the distance between the y coordinates. If the y distance is greater, it checks if the distance is positive or negative. If the y coordinate of the coordinate calling the method minus the y coordinate of the argument passed as an argument is positive, the coordinat calling the method must be above the argument coordinate, 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. Firstly, 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 2 lines, 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 is not 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 generated

The 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 has already had a request for that image. If it has already had that request before – then it just returns a cloned version of the image from a hashMap. If not, then it gets that image from the internet, puts it in the hashMap and then returns getImg with the URL already given.

NB: This was not entirely my idea – the idea was given by:

<https://www.youtube.com/channel/UCeQhZOvNKSBRU0Mdg7V44wA>

However, only the mention of a ResourceManager was given and I saw no code or details.

#### 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.*

### Save System

A screenshot of a cell phone

Description automatically generated

The save system is used to save high scores for each level.

BEGIN getHighScore (lvl)

fn <- fns[lvl – 1]

BEGIN TRY

reader <- NEW BufferedReader(NEW FileReader(fn))

score <- reader.readLine()

reader.close()

RETURN score

END TRY

BEGIN CATCH e

RETURN “0”

END CATCH

END getHighScore

BEGIN setHighScore (lvl, score)

fn <- fns[lvl – 1]

BEGIN TRY

writer <- NEW BufferedWrite(new FileWriter(fn))

writer.write(NEW TimeStamp(System.currentTimeMillis()) + “: “ + score))

writer.close()

END TRY

BEGIN CATCH e

e.printStackTrace()

END CATCH

END setHighScore

#### getHighScore (int lvl)

This method returns the high score of a given level using a BufferedReader. Firstly, it gets the file name from an array, but we subtract one from the level given because the levels start at 1 but java arrays start at 0. It tries to get the score, and to return it, but if there were any Exceptions, eg. if the file did not exist, it returns 0, implying that this level doe yets not have a high score.

#### setHighScore(int lvl, int score)

This method sets a high score, and starts with getting the file name the same way as the getHighScore, before creating a BufferedWriter and writing the time and score to that file. If there any exceptions it prints the stack trace.

## 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 computer 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.getI

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] »*

*\*/*

### Base Entity Class

A screenshot of a cell phone

Description automatically generated

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 the new Coordinate with the combined x and y coordinates.

changeX (int dst):

This method is used to change the x coordinate of an object. NB: The method 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.

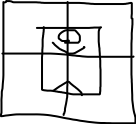
Then we use the xInArr and xInTile, as well as the y variables from the start to setXYInArr, and XYInTile.

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 its 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 (() -> {})

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), it checks for new enemies and updates 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 (() -> {})

This thread runs the bullet, checking if it has hit one of the death conditions, and if so removing the bullet from the render list, and if not it moves 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 the range of the bullet, 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 computer Description automatically generatedEnemy Class

This is the enemyActual class, and it is created similar to the turret, 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

})

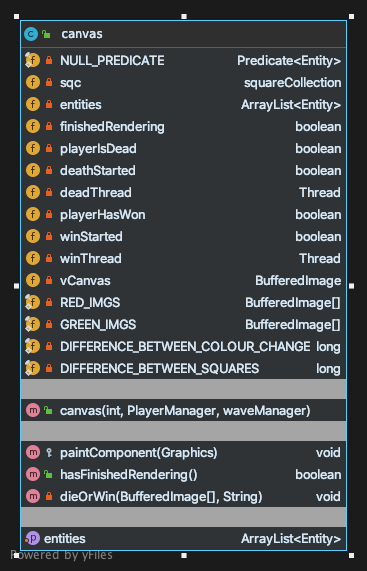
#### runThread = new Thread(() -> {})

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 next target has been reached, the step variable 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 the enemy is dead, and the enemy exits out of the loop. If none of those are true the enemy continues.

The rest of the loop begins with getting the difference between last time the enemy went through the loop and this time, getting the time it took to process the movement, and the enemy creates a difference variable to store this. The enemy checks it is positive and in acceptable range (so it does not wait for too long or for a negative time), and then it waits for the move gap minus that difference variable. It then gets the current coordinate (in inArr format), from the list of enemy squares, and turns it to onScrn format, and adds tolerances for its hitbox. Then the movement is identical to the bullet, in that the enemy gets direction and distance, then checks if the distance it can go is greater than or equal to the distance to go, and if so increments the current step variable, and teleports to the target. If not the exact same switch statement is used as from the bullet.

### Canvas 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. 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 / 2 and 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[][].

Then 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 it clones 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.

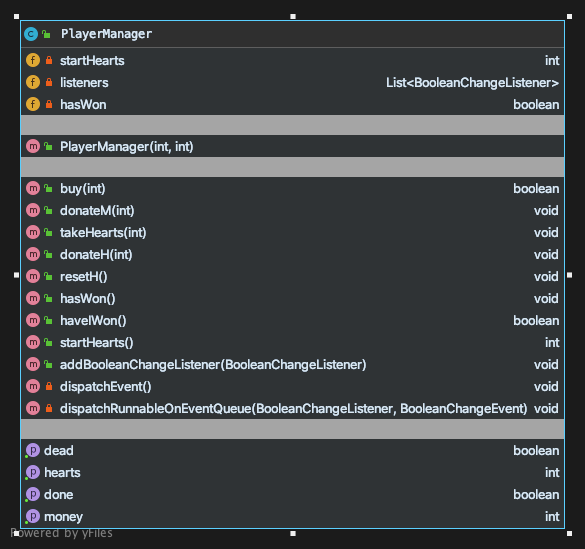
It then checks to see if the list is still greater than 0, and if so, it opens a try statement and inside a switch statement with the Entity type, as it does not want to render any unspawned enemies (players may wonder why an enemy is not moving, and then get angry when they place turrets to shoot at it, but they can’t shoot it due to checks in the turret). If it is unspawned the image remains null, but if the enemy has been spawned, we get the enemy image, and in the default part (for bullets and turrets), we just get the image. Then it double checks for null images, and if not – we get the position of the entity using getXYOnScrn and render their image there.

It then ends the try statement, and if it caught any errors, then it prints “ENTITY NOT FOUND”, and then prints the exception’s stack trace. Then, if neither of the death or win threads have been started, it checks for player win or death, and if either apply, it starts the relevant thread.

## 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



The 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

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. I am covering the buy button action listener of the turretFrame, and the buy function of the turretManager.

Important local variables for buyBtn – tt -> current turret template, name -> tt.getName(), btn -> the button with the listener.

Btn.addActionListener(e -> {

BEGIN IF freeSquares.size() == 0

JoptionPane.showMessageDialog(panel, “Unfortunately, there are no turret spaces left. Good luck!”, “No free space.”, JoptionPane.ERROR\_MESSAGE)

RETURN

END IF

iconUrl <- none

BEGIN TRY

iconUrl <- NEW URL(tt.getFn())

END TRY

BEGIN CATCH e

PRINT “Ex”

END CATCH

result <- JoptionPane.showConfirmDialog(panel, tt.toString(),

“Confirm buy Turret: “ + tt.getName(),

JoptionPane.OK\_CANCEL\_OPTION,

JoptionPane.QUESTION\_MESSAGE,

ResourceManager.getIcon(iconUrl))

BEGIN IF result == JoptionPane.YES\_OPTION

main.quickEntity(turretActuals)

main.quickCoord(usedSquares)

main.quickCoord(freeSquares)

location <- JoptionPane.showInputDialog(panel, “Please enter a location”,

“Where would you like your tower?”,

JoptionPane.QUESTION\_MESSAGE,

ResourceManager.getIcon(iconUrl),

freeSquares.toArray(), 0)

BEGIN IF location == none

RETURN

END IF

resultInStr <- location.toString()

type <- btn.getText().subString(4)

loc <- Coordinate.parseFromTS(resultInStr)

BEGIN IF loc == Coordinate.NULL\_COORD

RETURN

END IF

tm.buyTurret(loc, type)

main.quickEntity(turretActuals)

main.quickCoord(usedSquares)

main.quickCoord(freeSquares)

END IF

})

BEGIN buyTurret (where, type)

BEGIN IF !turretSquaresFree.contains(where)

RETURN

END IF

temp <- NEW turretActual(where, dictionary.getTurret(type), pm)

BEGIN IF !pm.buy(temp.getTurret().getCost())

temp.noLongerWorking()

JoptionPane.showMessageDialog(tf,

“You do not have enough remaining money to do that”,

“No money.”,

JoptionPane.ERROR\_MESSAGE)

RETURN

END IF

turretSquaresUsed.add(temp)

turretSquaresFree.add(temp)

turrets.add(temp)

END buyTurret

#### btn.addActionListener(e -> {})

This listener is for when the user clicks on one of the turret buttons – at the start, the turretFrame gets the names of each of the turret types from the turretDictionary and creates a button with “buy ” + its name, and an action listener is added to it – this action listener.

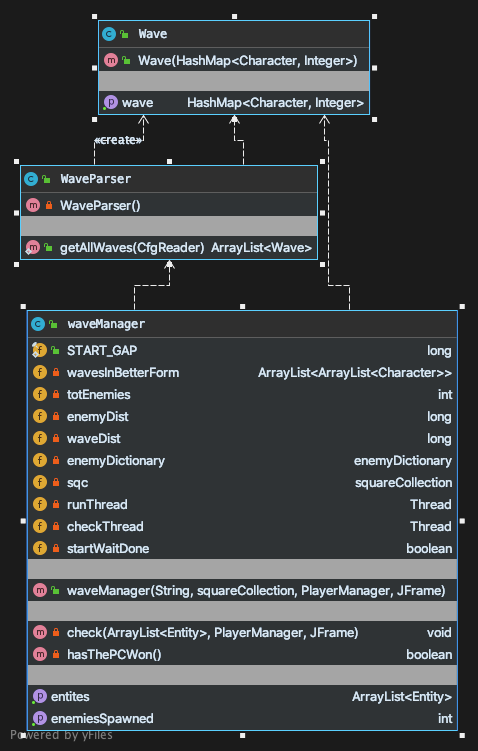
Firstly, when this button is clicked it checks if the user has any turret squares free, and if there are not any squares free it gives an error message, and exits out of that actionListener.

If there are turret squares free it continues, and gets the icon for the turret (the image of that turret), via the resourceManager, in a URL in a try/catch. Then it double checks with the user that they want the turret, and gives the user information about it. If they click yes, it sorts the lists of coordinates, and of turrets, as it the turrets are sorted by coordinate, and so the turrets indicies will match up with the other indices. The location of where to have the turret is asked with a JoptionPane, and if the user backs out, it returns null, and the program checks for that – and if it is null, it leaves the listener, if not, then it parses the coordinate, and gets the type as a string, before using the turretManager’s buyTurret method.

#### buyTurret (Coordinate where, String type)

This method is used to buy turrets in the turret manager. Firstly it checks if there is space in the freeSquares where where is for that turret, and if not, then an error message (in a JoptionPane) is thrown to the user, and the program exits from that method. Then, it creates a temporary turretActual variable, and checks if there is enough money to buy it (Gameplay.player.PlayerManager.buy() returns false if you cannot buy), and if there is it buys the turret, and adds it to the render list. It also updates the free and used squares. If there is not enough money then it gives an error message (JoptionPane) to the user.

### Waves



The WaveManager creates the enemies, and keeps track of them – the waveParser gets the waves, and the Wave is a temporary storage for those waves. NB: wavesInBetterForm, is an ArrayList of ArrayLists of Characters, with each inner ArrayList representing a wave, and each character in there representing an enemy.

BEGIN getAllWaves (reader)

noOfWavesStr <- reader.get(“enemyGaps”, “wavesNo”).toString()

noOfWaves <- 0

BEGIN IF main.INT\_REGEX.matcher(noOfWavesStr).matches()

noOfWaves <- Integer.parseInt(noOfWavesStr)

END IF

wavesRaw <- NEW ArrayList<String>()

BEGIN FOR I <- 1; I <= noOfWaves; i++

m <- “enemyGaps”

p <- “wave” + i

rawWave <- reader.get(m, p).toString()

wavesRaw.add(rawWave)

END FOR

enemiesBetweenFinal <- NEW ArrayList<Wave>()

BEGIN FOR enemies : wavesRaw

chars <- enemies.toCharArray()

hashMap <- NEW hashMap<Character, Integer>()

BEGIN FOR c : chars

current <- 1

BEGIN IF hashMap.containsKeyI

current += hashMap.removeI

END IF

hashMap.put(c, current)

END FOR

w <- NEW Wave(hashMap)

enemiesBetweenFinal.add(w)

END FOR

RETURN enemiesBetweenFinal

END getAllWaves

checkThread <- new Thread(() -> {

PRINT “Running”

BEGIN WHILE !startWaitDone

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(200)

END TRY

BEGIN CATCH e

PRINT “Enemy start wait sleeper interrupted”

END CATCH

END WHILE

checkThread.start()

BEGIN FOR thisWave : wavesInBetterForm

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(waveDist);

END TRY

BEGIN CATCH e

PRINT “WAVE WAIT INTERRUPTED”

END CATCH

BEGIN FOR c : thisWave

BEGIN TRY

TimeUnit.MILLISECONDS.sleep(enemyDist);

END TRY

BEGIN CATCH e

PRINT “WAVE WAIT INTERRUPTED”

END CATCH

eT <- enemyDictionary.getI

eA <- NEW enemyActual(eT, sqc.clone(), pm)

enemyActuals.add(eA)

enemiesSpawned++

BEGIN IF pm.isDead()

RETURN

END IF

END FOR

END FOR

})

#### getAllWaves (CfgReader reader)

This method gets all of the waves using a Config File Reader and starts by getting the number of waves, and then for each in the number, getting the raw wave from the reader (eg. “ABCABDEFGAAB”, “ABBDGAVSNS”, “HBHHBSD”).Then it initializes a new ArrayList of Waves. For each raw wave, it gets all of the characters, and for each of the characters, it creates an integer called currentNumber at 1 (because there will definitely be one of the current character), and if the hashMap already has that key, that character, then it removes it, and gets the value and adds that value to currentNumber. currentNumber now has the number of instances of all before, and now, plus 1. Then, we add back that number, with the character as the key. Then we create a new Wave, using that HashMap, and add it to the toBeReturned list (here called enemiesBetweenFinal), and then we return that list.

#### checkThread = new Thread(() -> {})

Firstly, this thread prints “running” – this is to signify that the waveManager has started. Then it waits for the start wait to be finished (handled in a separate thread), by waiting for blocks of 0.2 seconds, and checking again. Then it starts the checker thread, which checks for player death, player win, and when the enemies are done. Then it goes for each ArrayList<Character> in wavesInBetterForm – an easier readable format for the waveManager. At the start of that it waits for the wave gap and before every wave it waits for the waveGap. Then for each character of that wave, it waits for the enemy spawn gap. Then it creates an enemyTemplate (eT) using the dictionary, and then an enemyActual using that enemyTemplate, the squares, and the playerManager. Then it adds the enemy to the spawn list, and increments the tracker for how many enemies have been spawned. Finally, it checks if the player has died, and if so, it exits. All of the curly brackets close.

## Class Diagrams and Algorithm Explanations – Main Package



There are two packages in the main class – main.java and ApexTurrets.java.

The main class contains lots of helper variables for lots of classes, as well as a method to run level 1.

The ApexTurrets class only runs lvl1 from main.main, in its main method, and other than that, it just handles whether users accidentally ran this by accident and whether they want to play.

BEGIN lvl1 ()

moneyStr <- level.get(“playerGets”, “money”).toString()

heartsStr <- level.get(“playerGets”, “hp”).toString()

money <- 1

hearts <- 1

BEGIN IF INT\_REGEX.matcher(moneyStr).matches() && INT\_REGEX.matcher(heartsStr).matches()

money <- Integer.parseInt(moneyStr)

hearts <- Integer.parseInt(heartsStr)

END IF

window <- NEW Jframe(“Apex Turrets”)

window.setDefaultCloseOperation(WindowConstants.DISPOSE\_ON\_CLOSE)

window.setLayout(NEW GridLayout(1, 2))

pm <- NEW PlayerManager(money, hearts)

sqc <- NEW squareCollection(NEW squareParser(NEW CfgReader(main.MAPS\_LOC + “stg1.cfg”)))

waves <- NEW waveManager(“lvl1.cfg”, sqc, pm, window)

c <- NEW canvas(CURRENT\_LEVEL, pm, waves)

window.addI

tm <- NEW TurretManager(sqc, pm, window)

window.pack()

window.setPreferredSize(Toolkit.getDefaultToolkit().getScreenSize())

window.setVisible(TRUE)

runThread <- NEW Thread (() -> {

current = System.currentTimeMillis()

delay = 10

windowHasStarted = TRUE

BEGIN WHILE window.isVisible()

c.paint(c.getGraphics())

BEGIN WHILE !c.hasFinishedRendering()

CONTINUE

END WHILE

BEGIN IF System.currentTimeMillis() – current > delay

enemyActuals <- waves.getEntites().clone()

turretActuals <- tm.setEnemiesAndGetTurretsAndBullets(enemyActuals).clone()

finalEntites <- NEW ArrayList<Entity>()

finalEntites.addAll(enemyActuals)

finalEntites.addAll(turretActuals)

c.setEntities(finalEnties)

current <- System.currentTimeMillis()

END IF

END WHILE

playAgain <- JoptionPane.showConfirmDialog(null, “Would you like to play again?”)

BEGIN IF playAgain == JoptionPane.YES\_OPTION

lvl1()

END IF

BEGIN ELSE

System.exit(0)

END ELSE

})

runThread.start()

END lvl1

#### lvl1()

The lvl1 method sets up the first level, and starts it. Firstly, it gets the money and lives from the Config Reader, and checks them before parsing them, and if the parse fails, they default to one. Then it creates the window, and sets the default close operation, as well as a grid layout – left half for the canvas, and right half for the turretFrame. Then it creates the core game objects – PlayerManager, squareCollection, waveManager, canvas, and the turretManager. The rest of the objects are created by those 5. The turretManager takes the window as an argument, and the turretFrame, adds itself to that window. Then it packs the window, sets the size to be fullscreen, and sets it visible.

Then it begins the runThread, in which (whilst the window is visible (ie. The player has not yet quit)), it manually calls the paint function on the canvas every time, and after a delay each time it updates the canvas’s render list. Then, after the window goes invisible (the player closes the window), it asks whether they want to play again, and if so, it calls itself, and if not, it calls System.exit(0), ending the program.

### Table Of Definitions For Main Static Final Fields:

|  |  |
| --- | --- |
| NAME | MEANING |
| windowHasStarted | Has the window been created yet? |
| INT\_REGEX | A regular expression for integers – it just checks for numbers from 0-9, with no other character |
| DBL\_REGEX | As above, for doubles – the regex is the integer regex, a decimal point (escaped to mean that only a decimal point can be used rather than any character), and then the integer regex again. |
| BASE\_LOCATION | The base URL for all URLs relating to NEAAssets. |
| IMAGES\_LOC | The base URL for all URLs relating to NEAAssets images. |
| ENEMIES\_LOC | Location of enemy config (configuration) files |
| TURRETS\_LOC | Location of turret config files |
| WAVES\_LOC | Location of wave config files |
| MAPS\_LOC | Location of stage config files |
| ENEMY\_IMAGES\_LOC | Location of enemy images |
| TURRET\_IMAGES\_LOC | Location of turret images |
| BULLET\_IMAGE\_LOC | Location of bullet images |
| PATHS\_IMAGES\_LOC | Location of path images |
| ICON\_LOCATIONS | Location of icons |
| AUDIO\_LOCATIONS | Location of audio files |
| CURRENT\_LEVEL | Current level |
| stage | CfgReader for the map of current level |
| NUM\_OF\_TILES\_WIDTH | The width of the tiles (NB: Not combined width, but the number of squares on the x axis) |
| NUM\_OF\_TILES\_HEIGHT | As above, but for height (and the y axis) |
| WINDOW\_WIDTH | Overall window height (pixels) |
| WINDOW\_HEIGHT | Overall window height (pixels) |
| size | A dimension with the above for width and height |
| TILE\_HEIGHT | The height of each individual tile (pixels) |
| TILE\_WIDTH | Above for width |
| BOUND | Half of the average of the tile\_width and tile\_height – used for measuring if things are close enough to their targets. |
| TURRET\_WIDTH | Desired image width of turrets (pixels) |
| TURRET\_HEIGHT | Above for height |
| BULLET\_WIDTH | Above but for bullets and width |
| BULLET\_HEIGHT | Above for height |
| ENEMY\_WIDTH | Above but for enemies and width |
| ENEMY\_HEIGHT | Above for height |
| TURRET\_X\_ON\_TILE | The x to render a turret on in a tile |
| TURRET\_Y\_ON\_TILE | Above for y |
| TURRET\_FNS | File names for turret config files (NB: just the name – not a url) |
| ENEMY\_FNS | Above for enemy config files |
| ENEMY\_IMG\_FNS | Above for enemy images |
| SOUNDS | HashMap for sounds – key is the file name, and value is a clip.  NB: The lineListener attribute on the clips is to enable the clip to be played multiple times, rather than just once, as clips have no clone method. |

A circuit board

Description automatically generated



## UI Design

My program is relatively simplistic, and so only has one main phase of UI, with the remaining UI, being JoptionPanes, due to their simplicity and ease of use.

1. The canvas – This renders all of the components, and takes no part in UI design other than it draws what it has been given.
2. The turretFrame – On the other side – the turret frame is the UI that the player uses to interact with the canvas.
3. The tiles of the canvas – On the left hand side – there is a grid of tiles, or squares, and some the user can use for turrets, some are for the enemy path, and the rest are nothing.
4. The path on the canvas – This is the path the enemies follow – starting at the red, satanic ritual-esque square, and ending on the blue one.
5. The buy buttons on the turretFrame – these buttons are used for buying turrets – and one appears for each turret.
6. The sell and upgrade buttons on the turretFrame – these buttons are used for selling and upgrading the turrets. The selling mechanic, is that the turrets are worth less used, and so sell for less, and the upgrade mechanic works randomly, giving an element of chance to the game.
7. The status window of the turretFrame – this is the lower left part of the screen, which details, money, lives and all of the turrets (using the toString of those turrets).
8. The green bar above the enemy – health bar, and whilst some turrets can use only one or two shots to kill these enemies – there are harder enemies to kill, which can need more bullets.

# Technical Solution

For the technical solution: please see the master branch of: <https://github.com/Epacnoss/NEA-Code>

The class with the main method is at

JackMaguireNEA/src/main/ApexTurrets.java

# Testing

In order to test how well my program has met the objective of providing a user-friendly computer game for bored students to occupy themselves in a fun way at break and lunch times (see Analysis section for further details), and how rigorous and successful my coding has been, I conducted a series of different intricate procedures to test the various different functions, mechanics and features used in my solution. I have set these out below.

A close up of text on a white background

Description automatically generatedI have also recorded the evidence of these tests into a single video which I have uploaded on Youtube. To see that video, please use the following link: <https://youtu.be/UayhmBDIvds>.

A close up of text on a white background

Description automatically generated

# Evaluation

## General Evaluation

Overall, I believe that the project went very well in that all of the core game mechanics were implemented, and that it works well consistently. It meets my objective of providing a user-friendly game for teenage students to occupy themselves in break and lunch times, and does not require them to download anything from the internet. However, if I could implement a system for logging in users and keeping track of their data, eg. Their names and high scores, it would have been better.

## Things I would change if I did it again.

* I would add a system for users logging in.
* I would add keyboard shortcuts for quickly buying turrets.
* I would introduce a leaderboard.
* I would introduce an endless mode (currently partially done in level 3).
* I would consider adding in buttons on the canvas, rather than in a separate part of the window because this might be more immersive for the end user. This so far has not been an option due to limitations with JComponent rendering and buttons.

## Completion of Objectives

I set out below a table of my objectives when doing this game together with details of whether I think the objective was met; if it was, how well it was, and any particular improvements I think I could implement in a future version.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Objective | How was it met? | How well was it met? | Improvements? |
| 1 | The game should not be repetitive | Due to the randomness of the upgrades, and the enemies, as well as lots of factors, the game did not feel a slog or repetitive. | Very well. | Possibly a button to start the enemies – sometimes the 30s was too much or not enough |
| 1.1 | There should be different turrets. | There are indeed. | Perfectly. | More turrets. |
| 1.1.1 | There should be at least two different turret templates. | There are indeed. | Perfectly. | More Turrets |
| 1.1.2 | There should be upgrades available for each turret. | There are upgrades. | Perfectly. | Possibly less random – or a choice of damage or time upgrades. |
| 1.1.2.1 | An upgrade is an increase to a statistic of a turret, in a way that positively effects gameplay for the player. | Upgrades are applicable to increase damage or to decrease cooldown time. | Perfectly. | N/A |
| 1.2 | There should be different enemies. | There are. | Perfectly. | More enemies |
| 2 | The game should be as efficient as possible. | It is relatively efficient. | Relatively well. | N/A |
| 2.1 | The storage space should be as efficient as possible to increase performance on the low-spec machines of my end-user. | Due to the ResourceManager, space in memory is saved, from keeping lots of images in one place. | Perfectly. | Use a spritesheet for all of the sprites. |
| 3 | Turret Features |  |  |  |
| 3.1 | You should be able to sell turrets – if the player makes a mistake. | That is implemented, though mistakes aren’t encouraged, through a slight money penalty. | Perfectly. | N/A |
| 3.2 | You should be able to upgrade turrets (mentioned earlier in 1.1.2.1) | See 1.1.2.1 | | |
| 3.3 | It should be clear where the bullets are and how long the turret needs to wait for. | When the player mouse hovers close enough to a bullet, an aura shows around it’s target.  It also shows how long it needs to wait for in the turret status. | Perfectly | N/A |
| 4 | Bullet Features |  |  |  |
| 4.1 | You should be able to see bullets – not just lines drawn. | Achieved, they pathfind as well. | Perfectly | N/A |
| 4.2 | The bullets have a limited range | They do – they can’t go the whole way across the screen, unless their range says they can. | Perfectly. | N/A |
| 4.3 | The bullets should show which enemy their target is. | As mentioned earlier in the turret features – if the player hovers the mouse close enough to a bullet, an aura appears round the bullet’s enemy. | Perfectly | N/A |
| 5 | General Features |  |  |  |
| 5.1 | You should be able to choose the level and play again. | There are multiple levels to choose from, and all can be replayed. | Perfectly | More levels |
| 5.2 | You should be able to see the whole map at once. | You can. | Perfectly | Add in coordinates to help players choose a position for buying turrets |
| 5.3 | The UI should be clear and reasonable – no menus in menus in other menus | All UI is clear, and the buttons are simple. | Perfectly | N/A |
| 5.4 | There should be a leaderboard. | In each level, you can view the top scorer, and when they did it. | Well. | A centralised leaderboard could be added. |
| 5.5 | You should be able to save a save file with details of scores etc., and view your score on a leaderboard. | Unfortunately, due to complexity and time, I couldn’t do the save file, and the leaderboard is stated above. | OK. | A save file system is preferable. |

## End User Feedback

It is important to get feedback from the end users on any game. I set out below a summary of the feedback I received.

* The initial menu screen is quite basic but in many ways that makes it easy to play and to use which is quite useful given its purpose.
* Textures:
  + There are white spots around corner pieces of path
  + Might be good to add in another sort of grass texture to make it more visually appealing.
  + The enemies only teleport across the screen, and so it may be better if they had a walking animation.
* Help the players to visualise the coordinates of squares.
  + Perhaps a GUI overlay would be better for placing pieces.
* A cooldown stat should be in seconds, not milliseconds.
* The end screen score and “you won” text should be a lot bigger.

## Analysis of End User Feedback

* The initial screen is quite basic (but deliberately so) – it was easier for me to use JOptionPanes, both for simplicity, and for keeping a clear UI theme.
* Textures:
  + There are white spots – so that it is clear which way the corner is turning – in some areas the corners are quite close together, and I wanted to keep it clear to the end user how the enemies would move. This is to make it easier for optimal turret placement.
  + Grass Textures - again as before, I want to keep it simple.
  + Enemies do not walk – as walking is incredibly complicated, I decided to stick with simple movement across the screen. With more time, I could program a simple thread to change the image to a different sprite every so often.
* It is hard to visualise placing of turrets – This is a limitation, but unfortunately, Javax.swing JComponents do not have room for buttons. If I was doing this again, would consider and seek further feedback on whether I should use a different system for users to select turrets and turret positions. It could be made more user-friendly.
* Cooldown stat – I felt that if the user had 3 decimal places, it may appear to complicated, and so I have kept with milliseconds. If I was doing this again, I would seek further feedback on this point from a larger group to get proper consensus.
* You won text – This is a problem, yet the way I have rendered, there is no option to change the font, even using vCanvas.getGraphics().setFont(Font), as it does not appear to work.