System design document for Super duper Omega tower Defense

Samuel, Oskar, Erik, Behroz, Sebastian

October 23, 2020
1.0.1

Contents

1 Introduction

Super duper Omega tower Defense is a Tower defence game highly inspired by Bloons Tower Defense. This document aims to explain the core structure and the technical solutions we found during the project.

1.1 Definitions, acronyms, and abbreviations

- **MVC,** Model-View-Controller, it is a way to construct an application where the model is not allowed to access the view or the controller.
- **MVVM,** Model-View-View Model, another way to construct an application where the view model acts as a middle layer between the model and the view.
- **Java,** a platform independent programming language.
- **GUI,** Graphical User Interface.
- **Swing,** a java GUI toolkit that enables a graphical user interface.
- **Persistent data management,** information that is stored when the application is shut off.
- **Junit5,** a framework for developer-side testing.
- **Travis,** a service for building and testing software projects.
- **Scrum Board,** a way to track current tasks and progress of the project.
- **HashMap,** a way of storing data with a constant lookup time.
- **FPS,** Frames Per Second.
- **UPS,** Updates per second, how often the model updates per second.
- **DSL,** Domain Specific Language.
- **TDD,** Test Driven Development.
- **CQSP,** Command Query Separation Principle.

2 System architecture

2.1 MVC

Using some kind of Architectural pattern for the structure of an application of this size is preferred. As the team has worked with the MVC pattern before that became what is used for the project. This is done by separating the game and its logic (model) from the graphical representation of it (view) and user input handling (controller).

By separating the logic from the graphical representation it makes it easier to, for example, change the library used to create the GUI. It also makes it easier for multiple people to work on the same thing but different parts of the MVC structure, which is important considering we are five people working on this project.

2.2 Application and ApplicationLoop

When the application is started, the Application class creates an instance of each Model, View and Controller and sets up the references between them. An ApplicationLoop is also created and started, with the purpose of calling update on the model 60 times a second and draw on view. If, for some reason, the computer cannot keep up with 60 fps and 60 ups, the loop will favor updating the model over drawing the graphics. If the application window gets shut down, view (handling the window) tells the loop to stop.

2.3 Swing as View

Swing is one of Oracle's "Java Foundation Classes" and is an API for creating GUI:s in Java development. Swing is also connected to another java library called AWT (abstract window toolkit), which in this paper will be used interchangeably.

In our application, swing is what is creating our application window, and is responsible to display everything from the model. This means that swing is responsible for the view elements of our application. Swing is also responsible about how the application takes in mouse inputs from the user of the application with its inbuilt mouse listener interfaces. This also means that controller is dependent on swing and AWT as well.

Because a swing window component has a button for closing the window component, the application loop is dependent to know when the window is closed, so the application loop can also stop.

3 System design

3.1 Domain model and Design model

The projects domain model, see figure 1, describes that the game/player should always only be one and be directly dependent on multiples of the other major parts of the model, such as the structures and collections of Enemies, Projectiles and Towers. The game/player is, according to the domain model, also be dependent on the tiles that divide the map into a grid. Each of the many possible enemies, projectiles and towers should also be dependent on their position on the tile map.

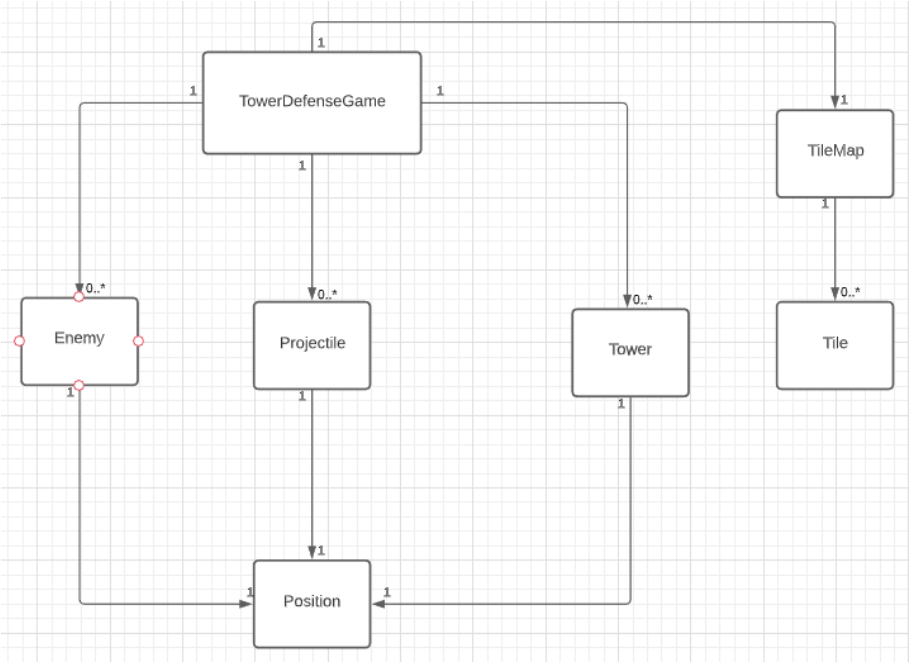

Figure 1: Picture of projects domain model

The current design model, see figure 8 and 9, is highly reflective of the domain model. There is a game class that is dependent on the lists of enemies, projectiles and towers. This class tells each of these separate entities to to update their state in the game. Each of those entities also have attributes that describe their individual position on the tile map. Speaking of, the game also, as the domain model describes it should, has a reference to the tile map which gives the game access to the states of each tile on the map grid. See class diagrams under references.

3.2 General Game Loop Sequence Diagram

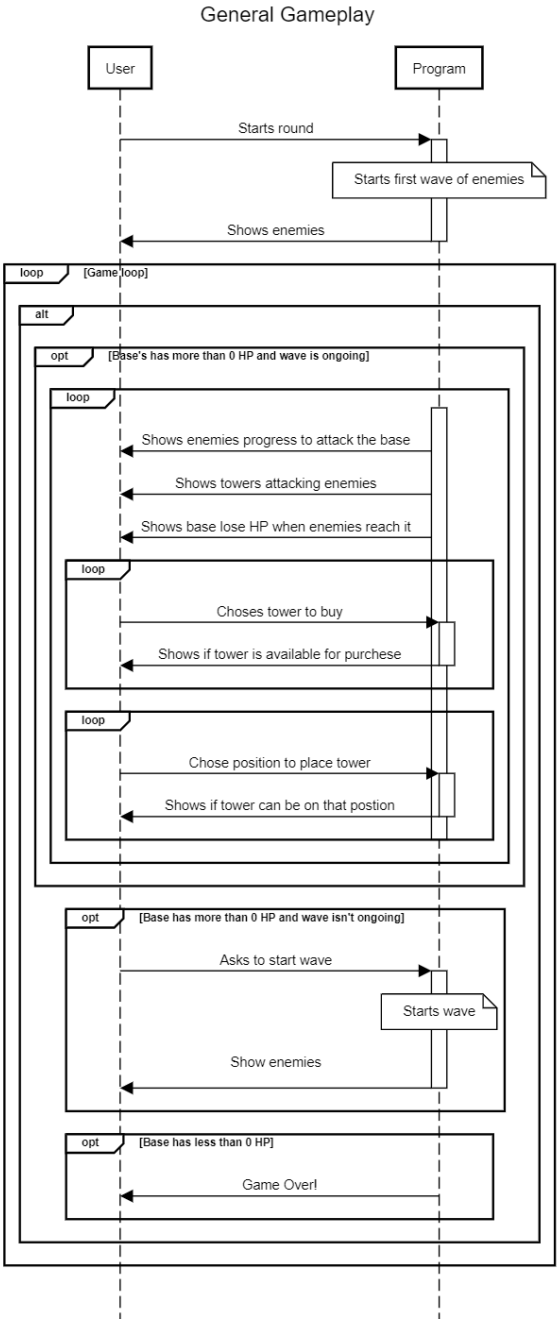

Figure 2: Sequence diagram of general game play from the user's perspective

3.3 Mouse Hover Sequence Diagram

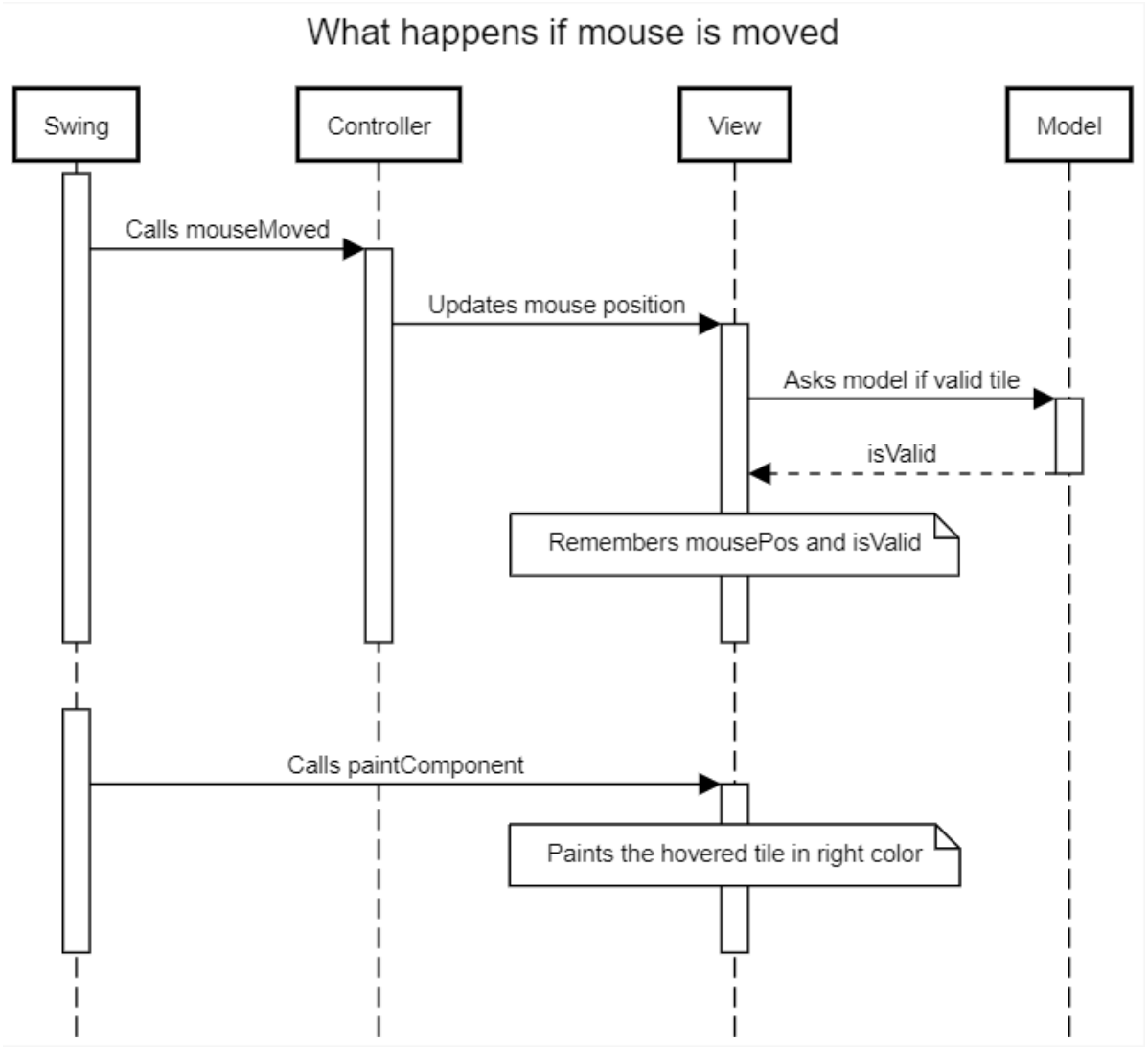

Figure 3: Sequence diagram of when the mouse moves

3.4 Towers

Towers are created via a TowerFactory. Each tower has different functionality when they fire, some buff other towers and some shoots projectiles, therefore we used Template Method pattern. To avoid code duplication all attacking towers have a common fire method. To then have unique attacks we used another Template Method for attack. For how each tower charges their attack or buff we use a Strategy pattern.

3.4.1 Buffs

Some towers in this application can buff/debuff towers. Therefore every tower has a buff manager that takes care of all buffs, and also a method for applying buff. See figure 4. When Tower Handler updates one of these buffing towers, the buffing tower calls towerFinder for finding towers in radius and add a towerModifier and duration to the tower via applyBuff. The buff manager in the tower to be buffed saves the modifier and the duration for being used on the next update cycle. When this update cycle is called, the tower that is buffed asks for a tower multiplier, from buffmanager that is based on all towerModifiers, when checking any of its stats. Damage or range for example. One problem with how it works now, is that buff towers can buff other buff towers, which means that scaling on these buffs (when two buff towers are close to each other), is very high. To combat this problem, we have put in a max value, for total tower multiplier, at +200 percent. This means the power of these buffs do not scale infinitely. Another solution to this problem is: not being able to buff ”buff towers”. If we had more time we would have implemented different kinds of buffManagers that we could decide in exactly what way buffTowers can be buffed.

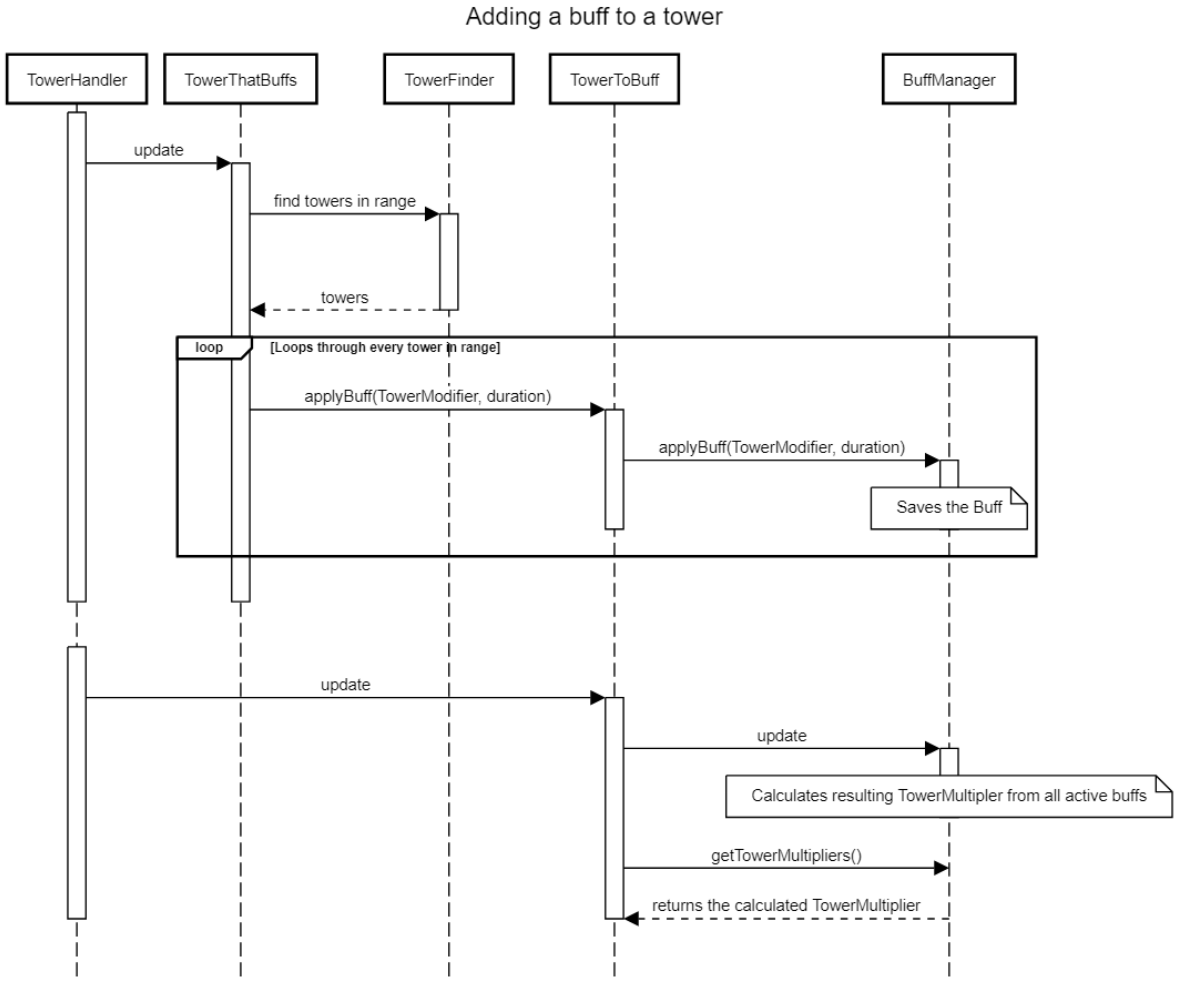

Figure 4: Sequence diagram for when a buff is applied

3.4.2 Possible improvements by removing inheritance

When we first created towers we thought they would have had a lot more differences. Therefore we decided to have a separate class for every tower. Because of this we decided that we could avoid using a towerType enum and just use the class as the type instead. We did the same thing with enemies and with projectiles so there are many places in the code that has classes as variables

Later on in the project we realized that many towers are very similar so using delegation for everything would have been a lot better. For example, we use a template method pattern for every attacking tower called attack(Enemy). If this was a strategy instead, and if there was an enum for towerTypes, we could remove all concrete attacking towers and just let the factory build them with the correct strategies. This would remove a lot of boilerplate code. Many of the attacking towers do the same

thing in the attack(Enemy) method; they spawn a projectile at the enemy and send an event that they have attacked. It should be abstracted away with a strategy.

In Projectile there is a similar problem. Bombarda charm and Rocket are both projectiles that explode, they just differ in some constants like radius, damage, etc. Currently, the algorithm for damaging enemies in a radius is copy pasted. If it delegated to some "OnHitStrategy" we wouldn't have that code duplication and we could remove all classes for concrete projectiles (except for BombardaCharm because it is heat seeking).

We realized this too late in the development process and decided that changing all class references to enums would take too much time. If we had more time we would try to remove all inheritance from towers, projectiles and enemies.

3.4.3 EnemyTargetter

Towers use an EnemyTargetter to delegate functionality for finding enemies so that Towers can focus on what makes them unique instead. Later we added exploding projectiles which should use the same functionality (getEnemiesInRange() to then damage them inside explosion radius). But the projectile should never use the getEnemyToAttack() that is also defined in EnemyTargetter. We thought about separating them into EnemyTargetter and EnemyFinder (as we already have TowerFinder) but never got around to it because we were not sure it would be better just by adding more classes. Another questionable thing about EnemyTargetter is that it is stateless, but each exploding projectile and tower has its own EnemyTargetter.

3.4.4 Tower proxies

In this project, there are tower proxies, which represent wanted data from the concrete towers in controller and view. It is used for having a "selected tower", which means that the user has a tower that can be placed when the user clicked on a tile. When there is a selected tower, there will be a ghost image that shows the range of the tower and what the tower looks like. This is the main reason for why the project uses a proxy, but also to display how much the cost is for all the towers are in the tower panel. The proxy also saves a factory method for creating the tower the proxy is supposed to be a copy of, for sending it to model when the user click on a tile.

3.5 Enemy

Enemies have an Enemy interface which others depend on in order to follow DIP. We then have an AbstractEnemy which implements Enemy. AbstractEnemies are able to follow the path, damage the base once they reach it and take damage themselves. All the concrete enemies are subtypes to AbstractEnemy.

AbstractEnemy uses a PathIterator to follow their path. The iterator gives it a position on the map to go to and once it has reached that position it asks the iterator for the next position. The PathIterator also allows navigation backwards which allows enemies to walk backwards because of some status effect.

3.6 Wave

A wave is an asynchronous operation in essence. It needs to describe which enemies to spawn and when to spawn them. Our stories also had some other requirements, like being able to check which types of enemies will appear on a certain wave, checking the remaining health of a wave, and being able to start multiple waves at the same time.

3.6.1 How wave was implemented

The implementation of wave is inspired by RXjava. In RXjava you have a DSL for describing how streams of data should be modified using operators. It is similar to a builder pattern but the order of the operators matter. We thought of building something similar for waves where you can repeatedly call .spawn(EnemyType) and .delay(numberOfUpdates) to eventually build up a wave. See figure 5 for the full interface.

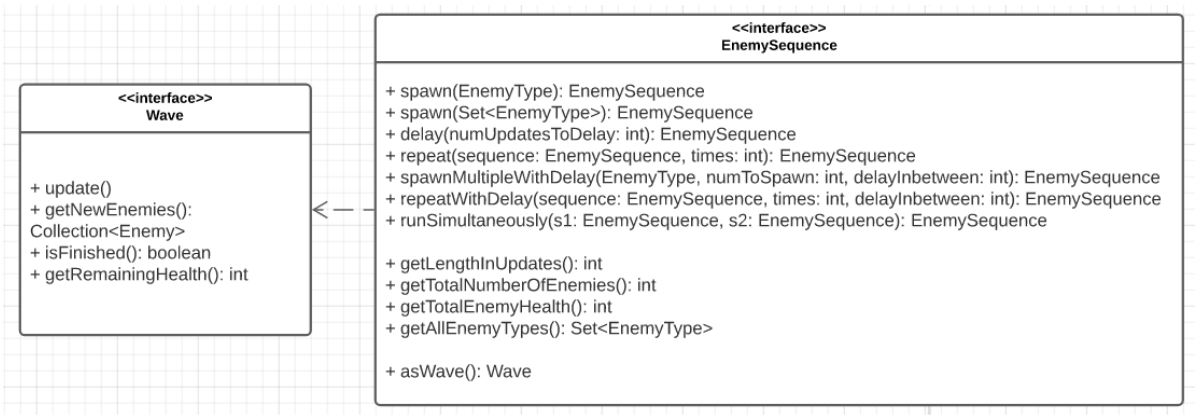

Figure 5: Early idea for creating waves

This ended up being very difficult to implement so we had to scrap some of the methods, most significantly runSimultaneously(). We used interfaces for both Wave and EnemySequence because we didn't know how to implement it in a good way. That way we could easily change implementation later on if we had to. TDD was very useful here because we could make sure that all edge cases we're covered while changing the algorithm. Our supervisor suggested that we could make wave.update() and

wave.getNewEnemies() the same method which makes Wave an iterator. We had tried to avoid that because it breaks CQSP but after we changed it the code became a lot simpler. We made Wave extend a Iterator of a Collection of enemies which means that the client is supposed to call .next() once every game update.

One major difference between this wave implementation and RXjava is that EnemySequences are not immutable. If you call .toWave() on an EnemySequence and then modify the EnemySequence afterwards this will change the created wave in undefined ways. This could be fixed by using a Decorator pattern when changing an EnemySequence instead of modifying a list directly.

3.6.2 WaveData

WaveData is the class which defines all the waves in the game. Similar to Bloons Tower Defense there is a number of manually defined waves first, followed by an infinite number of algorithmically generated waves. WaveData has a method called getWave(int level) which is used to get the wave for a level. WaveData is reponsible for checking if it has a predefined wave for that level.

One issue with this implementation is that predefined waves are stored in a list. This lets you easily add a new wave between two old ones. However it does make it harder to balance since if you fail on wave 15 and want to make that specific wave slightly easier you have to count until the 15:th definedWaves.add() command.

3.6.3 WaveHandler

We started off having a WaveHandler and an EnemyHandler where EnemyHandler handles the enemies that have already spawned and that towers can shoot and where WaveHandler which enemies will spawn. However one of our stories required an ”enemy health bar” which shows the remaining health and the total health of all waves that have been started since the last time no waves were active. This meant that we something that could access when waves were started, the remaining health of all waves, and the remaining health of all spawned enemies. We decided that the simplest way to do this would be to combine WaveHandler and EnemyHandler.

We have a variable called originalEnemyAttackHealth inside WaveHandler which remembers the sum of the total health of all waves that have been started. It gets reset when all waves and enemies are cleared. This variable is very confusing and could be removed if we instead kept all waves inside activeWaves until all of them are cleared. We currently remove waves from activeWaves when they are cleared which means that we can no longer check the total health of them. This implementation would be a lot worse for performance though since we would loop through finished waves and we would calculate the total health every tick instead of remembering it in a variable.

Although the code quality would be much better. Some of the performance could be saved by having Waves cache their total health once it has been calculated.

There is also a method called getEnemyTypesInNextWave(). This is needed because View shows information about new enemies that will appear. This was added late in the development process so we didn't have much time to implement it correctly. It calls WaveData and gets the next Wave. The problem is that WaveData then has to create that wave which means that all the enemies in that wave gets created. This means that all the enemies gets created once when that method is called, and then once again then the waves is actually started. This could be solved by either caching the next wave in WaveHandler, caching it in WaveData and making WaveData keep track of the current wave (i.e. turning it into a subtype of an iterator).

3.7 TileMap

The game's maps are loaded in from files and read as Lists of Strings and decoded to Tile matrices (read more in section 3.11). From there TileMaps are instantiated and the map's path is calculated (the path meaning which route the enemies take from their starting point to the base). If the loaded Tile matrix is invalid, meaning the programmed map layout is invalid, it will throw a RuntimeException and the program will not start. This is because the user is not meant to change the maps. The (simplified) algorithm for determining and calculating the path and if it is valid goes like this: First it checks if the map has exactly one marked start position and base position. Then, starting from the marked start tile, it checks a set of deltas (unit vectors in each direction), and sees if that position is a path tile that has not yet added in the path. If it is a path tile it is added and the process continues until it finds the base tile. If it cannot find a path tile it is an invalid map. The path that is calculated is stored as a List of Vectors and the whole class is obviously immutable.

3.7.1 ConnectedSequence

The path that is calculated when creating a new TileMap was previously stored in a different format, in a class called ConnectedSequence. Enemies will move to a Vector position and then ask for the next Vector position to go to until it reaches the base. Therefore we needed functionality that could give the next Vector in the path just by giving the current one as an argument. This is achieved by storing the sequence in a HashMap internally. Also, the first and last element must have separate variables. When the second element is added to the ConnectedSequence it puts the first element as key to the second element as value to the internal HashMap. When the third element is added it will be put as the value for the second element as key, and so on (see figure 6). This structure makes it easy to create functionality that can give the next element in a sequence by giving the current element as argument.

3.7.2 PathIterator

But we realized that we had user stories that demanded the ability for enemies to walk backwards, which the current way of storing the path could not support. Therefore we redesigned it. This time we used the idea of an iterator as a base, but added functionality for the methods hasPrevious and previous. This way we also wanted the enemies to store their own path instead of asking for the next vector to go to. We never had the time to implement our user story that would make enemies walk backwards, though. But even though ConnectedSequence was a good solution (at least we think so), PathIterator was not a worse one (again, at least we think so).

The class ConnectedSequence is still in the code base, even though it is not used. And it also has a custom exception, which we recently learned is not how you should do it.

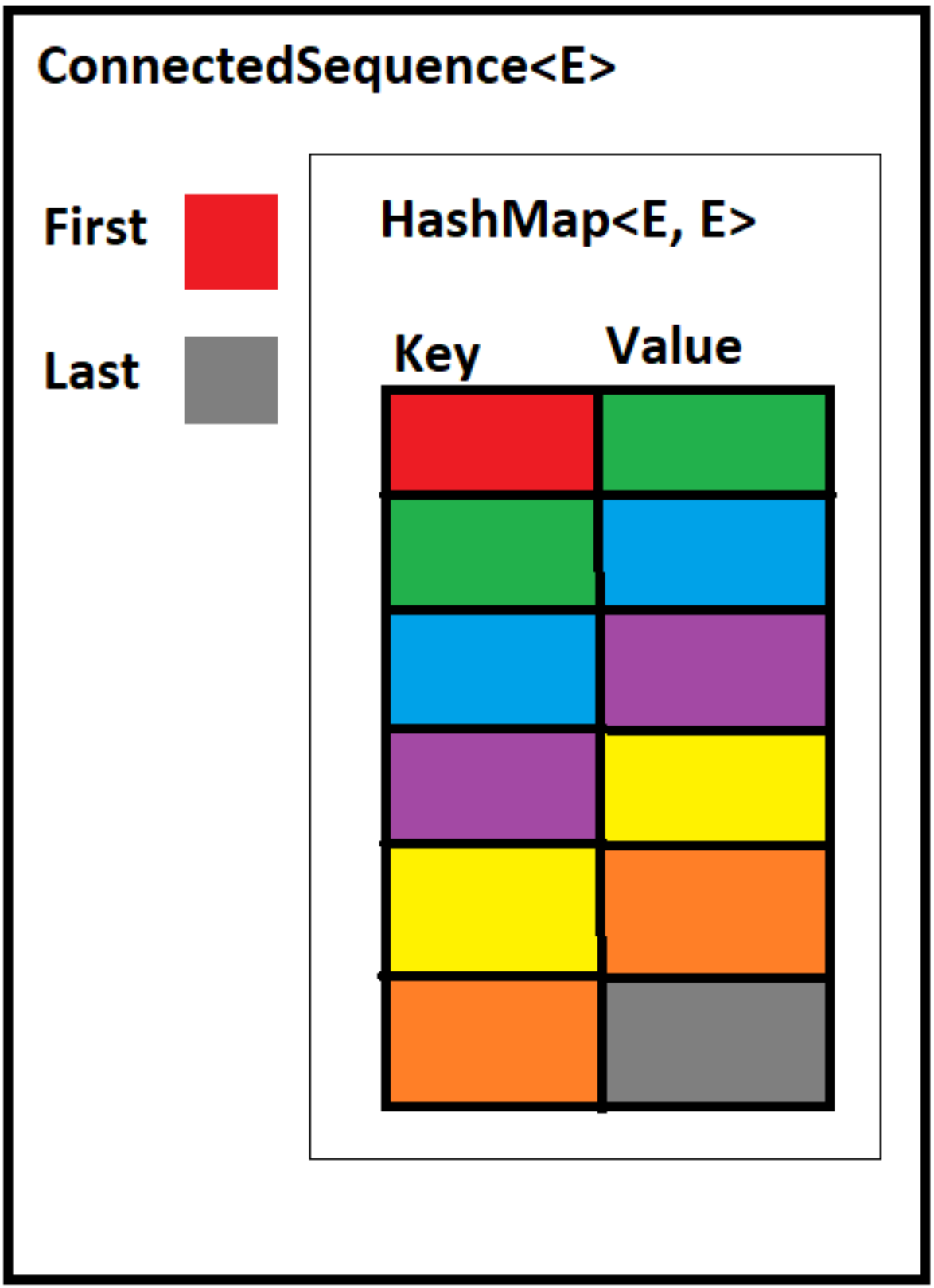

Figure 6: How the class ConnectedSequence is built internally

3.8 Config file and class

In this project we use a config.properties file that take contains some values that can be changed without having to recompile the program. This config class has had three iterations, one with ”fake” singleton where we used a private constructor and a get instance. Another iteration with a correct singleton using an enum instead. But in the end, we decided to use a lot of static final variables instead, because this means less code to implement new properties items. In all iterations we have group all the config values in categories by using inner classes. For example one can get the price of GrizzlyBear by: *Config.GrizzlyBear.COST*.

3.9 Main menu and multiple screens

In this application, there are two panels: Main menu screen and Game panel. The transition between these two panels, are not that bad. But if the application would have more than two, the system would get exponentially more complicated. The way it is now, there has to be hooks that get send around for giving a method to do when a new panel is supposed to be shown, and also another method that removes the previous panel and creates the wanted screen. An improvement for this is probably following what Android application is doing. They have something called activities that are “screens“ stored in a stack. This means that there is a MainActivity that starts other activities that are laid on top of the MainActivity. When the back button is pressed it is discarded and the activity under is shown.

3.10 Image cache

In Swing, there is no simple way to draw rotated images. The way to do that is to create a new Image which is a rotated version of the other. This is fairly performance heavy which means that we want to minimize the amount of rotations we want to do. Therefore we created an ImageHandler class which stores rotated images in a HashMap. When some client code needs an image with a certain rotation it requests it from the ImageHandler which will return it directly if it already exists in the HashMap. If the image does not exist in the HashMap it will be rotated from the baseImage and then stored in the HashMap for future use. This is very useful for Towers because they only change their rotation every time they shoot, meaning that the rotation method only has to be run when their rotation changes as opposed to every frame.

What we also did is that we have a finite number of possible rotations to limit the amount of rotations being stored. What this means is that if an image rotates, and another image that would have done a similar rotation (depending on number of possible rotations)it will instead use the same image. In practice the difference in angle to what it is supposed to be is almost impossible to see for the naked eye as

long as the number of possible rotations is high enough (we use 100 possible rotations which equals to a new image every 3.6 degrees). This effectively creates a scenario where the fewer possible rotations, the better the performance is and we can adjust it to what seems to fit our needs.

3.11 Concurrency

The application has four threads. One main thread that start the whole application and starts the application loop. One thread for application loop that calls updates on the application's model and application's view. One thread for all the swing components, that updates these components via a built in event queue in AWT. It also starts a new thread every time there is a mouse listener event, that updates the controller (who then manipulates model, which causes problems).

Problems that can comes up from having multiple threads like this is concurrent modification exception. This is because different threads modify and read collections at the same time. For example, View wants to display enemies, but at the same time it might die and be removed in the Model thread. We fixed all View/Model thread concurrency issues by adding synchronize blocks around parts where the collection is modified or read.

For the Controller thread we had to use another solution. We implemented the Command pattern, which means that instead of the Controller thread manipulating Model directly it adds a command to a list in the model, where the command is just a lambda of the thing Controller wanted to manipulate. Later when the Model gets updated it first goes through all commands and executes them. By doing this we only need to synchronize over the list containing all commands and all Concurrency issues are removed.

4 Persistent data management

4.1 Not saving user data

The program those not in it is final version persistently save any user data. During the development of the project we discussed different user inputs that would be useful to save, the following are examples that we either decided against or did not have time to implement:

- Saving the furthest wave the user completed and to show the wave number, next to the respective map, on the main menu. The idea was scraped, since there can be multiple waves ongoing at the same time. This would have made it a hassle to know which wave had been completed and which one had not been.

- Adding a map creator that would let the user make, save, and play their own maps. This idea would not have been especially hard to implement, because of the way the program reads .map files and converts them into maps for the game. However due to time constraints we decided that we had more important features to add.

4.2 Persistently saved files

Pictures, maps, and the properties file are all located in the ”resources” directory. Pictures and maps have their own sub directories here. The program reads these files by getting their paths from the ”config.properties”. The map files are stored in .map format and interpreted as text files. There is a map file loader service decodes each characters into specific tiles and can also handle commented lines (using double slashes right now). A map loader uses this service to read the maps into Tile matrices and then instantiates the TileMaps.

5 Quality

5.1 Testing with Junit5/Travis

Testing is done with JUnit5.4. We chose that version because that allows the assertThrows() method which means that we can check multiple exceptions in the same test. In versions prior to 4.13 you could only do one exception-check per test. Because we are using MVC, testing the View is difficult. If testing the View was a big priority we could have used a Architectural pattern which has a layer in between the Model and the View. For example, if we would have used MVVM we could have tested the output from the ViewModel which might have been useful, although we still decided to use MVC.

All group members are encouraged to run all the tests often; especially before committing anything. Since people might forget, or there could exist bugs that don't appear on their machine, we decided to setup an automated build. We set up Travis to automatically run all the tests every time something gets pushed to GitHub and notify us if something goes wrong.

A problem that slipped trough us for a long time was that Travis was not actually set up correctly. It would give us green check-marks in GitHub but that was because it ran zero tests successfully. Once we found out that Travis did not actually run our tests we investigated and found out that it was because of our current file structure. When using maven it is suggested to use their file structure to make sure maven runs correctly, and as Travis is dependent on maven, when maven runs correctly Travis works correctly.

5.2 Reviews

Trello is used as our scrum board to track tasks. Whenever someone has implemented a new feature that task goes to Review instead of Done, meaning that someone else has to review it before the task is fully finished. This means that every feature that has been implemented has been looked over by at least two people, which increases both the code quality, and the confidence that the code is bug-free. It also gives the team a better understanding about the whole codebase.

5.3 Known issues

5.3.1 Labels flickering

A limitation to swing is that the labels start to flicker when the game starts to slow down, because of very many enemies, towers and particles. The labels that flicker are either transparent, or invisible. As far as we know, there are no solution to this problem, or the solution is to have opaque labels, but that is not wanted in this application.

5.3.2 Do not know which wave the player died on

When the player dies, he/she does not know which wave has been completed. This is a problem in this application, because one can start many waves at the same time, and therefore the player do not know which wave was completed, and only the current wave that is spewing out enemies. This is more of a missing feature than an issue.

5.3.3 Performance issues with collision detection

In the current state of the application, the collision detection is checked every frame on every projectile, and each projectile loop over every enemy. This is very inefficient and would be better if the application used spacial subdivision instead. This means that the map is subdivided in many smaller squares, and every projectile only look for enemies in that square if there is a collision. This is much more efficient, because every projectile will loop through many less enemies for checking on collision.

5.4 Overall problems with Swing

There are some problems that has come up with using Swing. One of the problems is that how the repaint method works in Swing. Whenever repaint is called, it adds it to a queue, that will be called sometime in the future. This means we do not know exactly when it will be repainted, and we therefore do not know exactly what the frames per second is, and our loop is not working fully as it should, because it is

depended on knowing the frames per second.
This also means, because repaint happens on a new thread, that we have to take in consideration about Concurrent Modification Exception (CME). This means a lot of extra work for not getting CME, and no certainty that every point there can be a CME is fixed either.
As far as we know, rotating images is a very time consuming action in swing, and there are a lot of image rotation in this application. This means that it is bad for performance using swing for this specific situation.

6 References

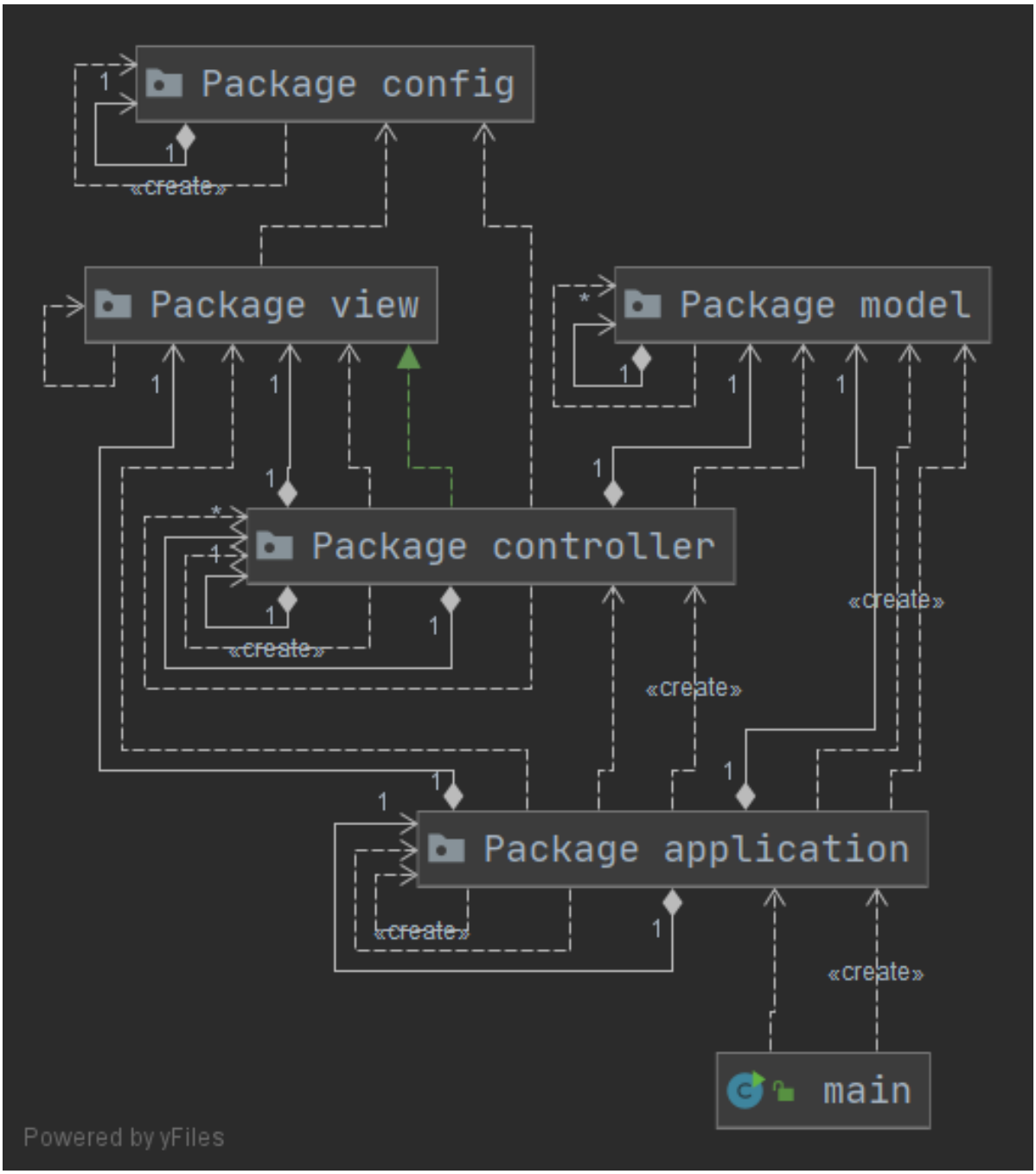

Figure 7: Class diagram for Source

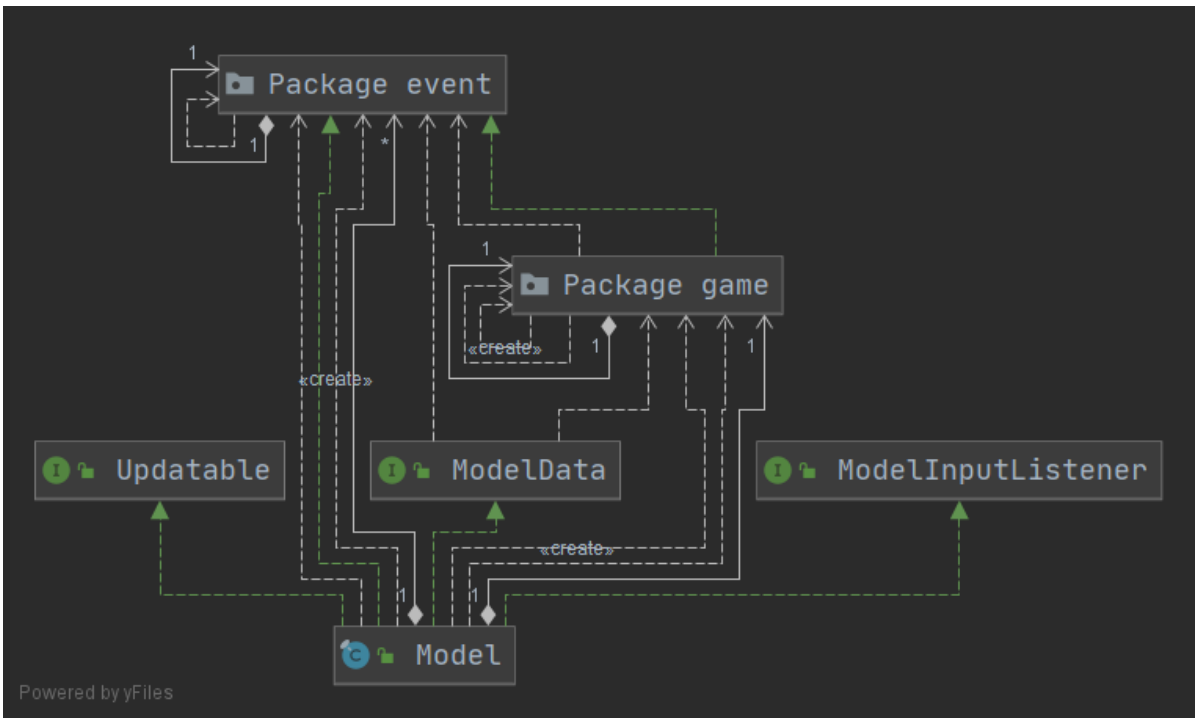

Figure 8: Class diagram for Model

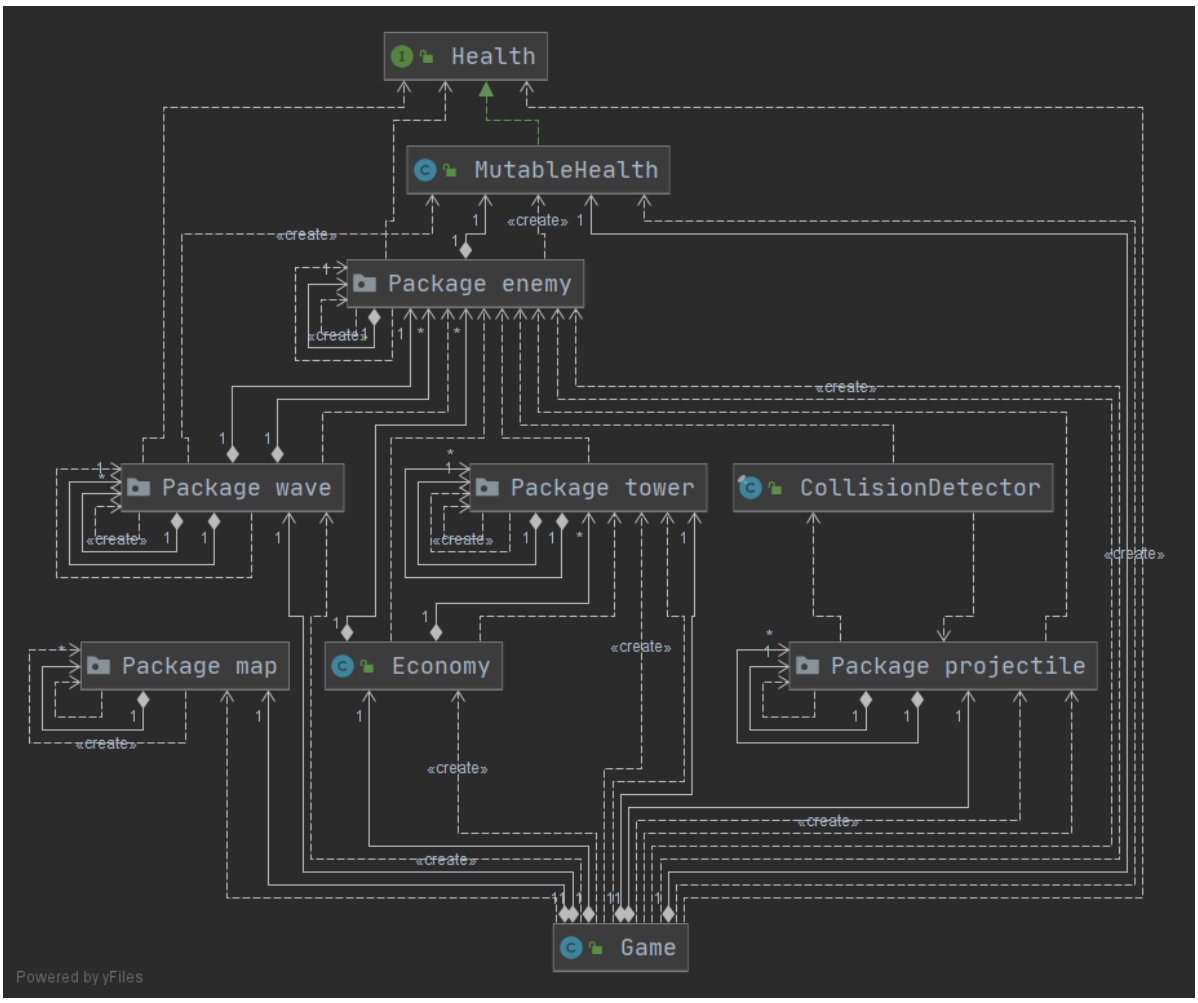

Figure 9: Class diagram for Game

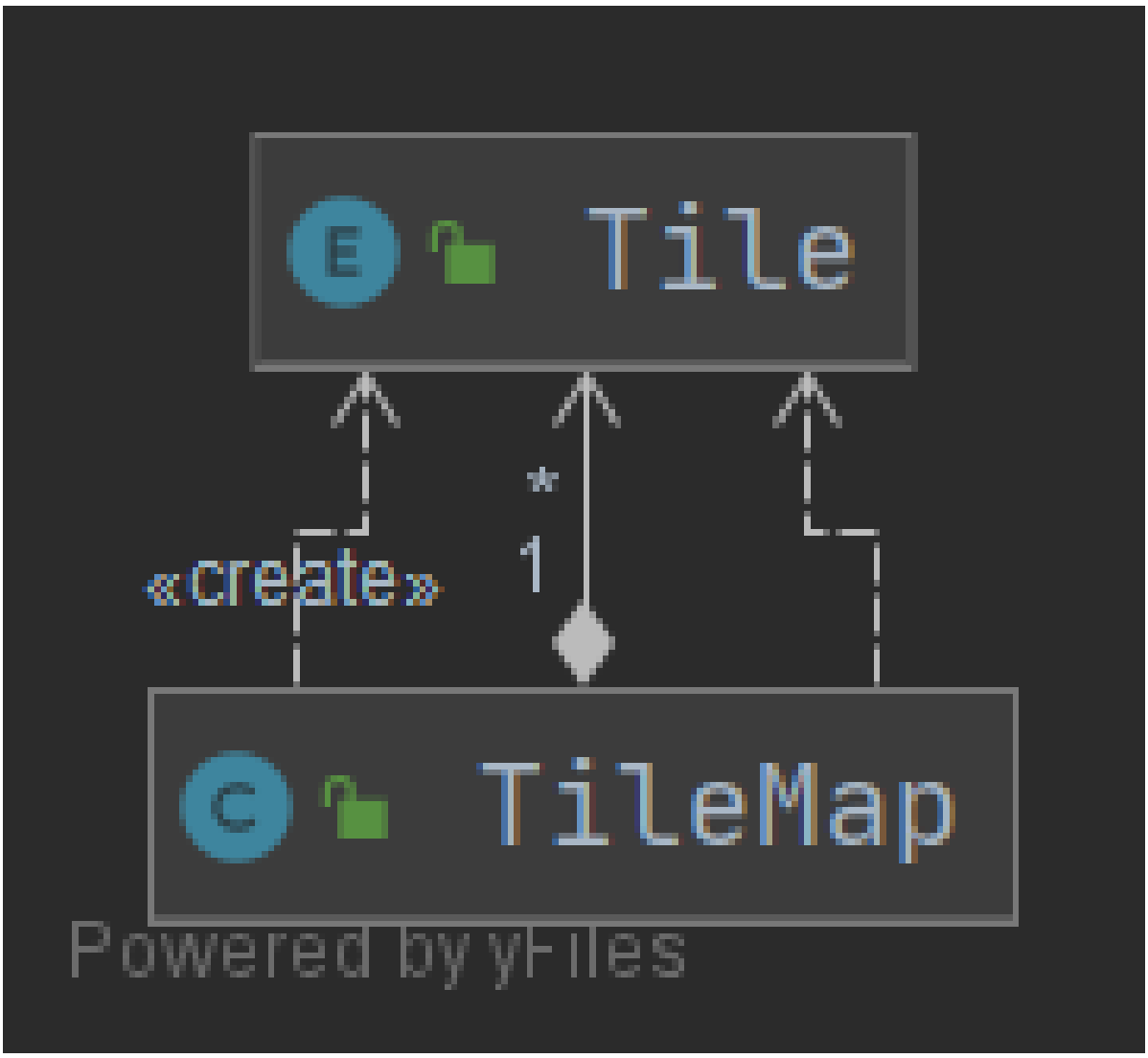

Figure 10: Class diagram for Map

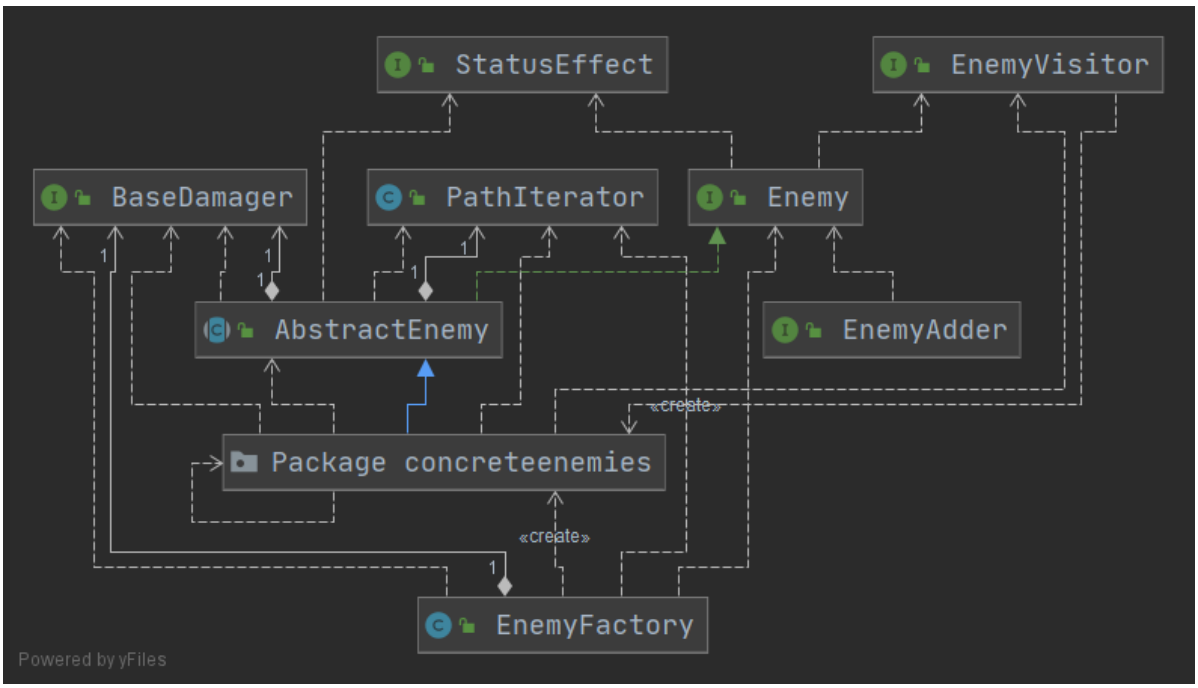

Figure 11: Class diagram for Enemy

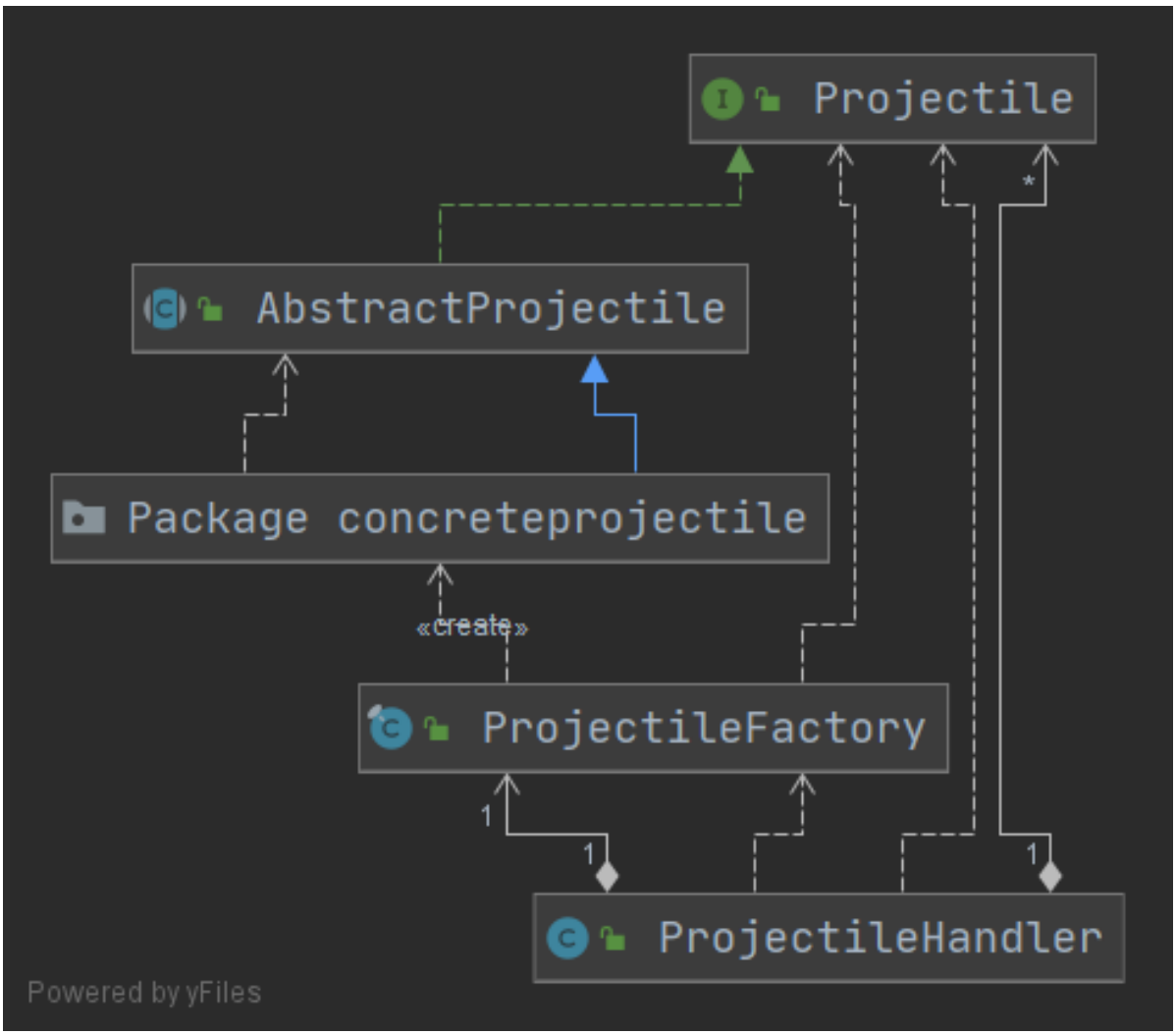

Figure 12: Class diagram for Projectile

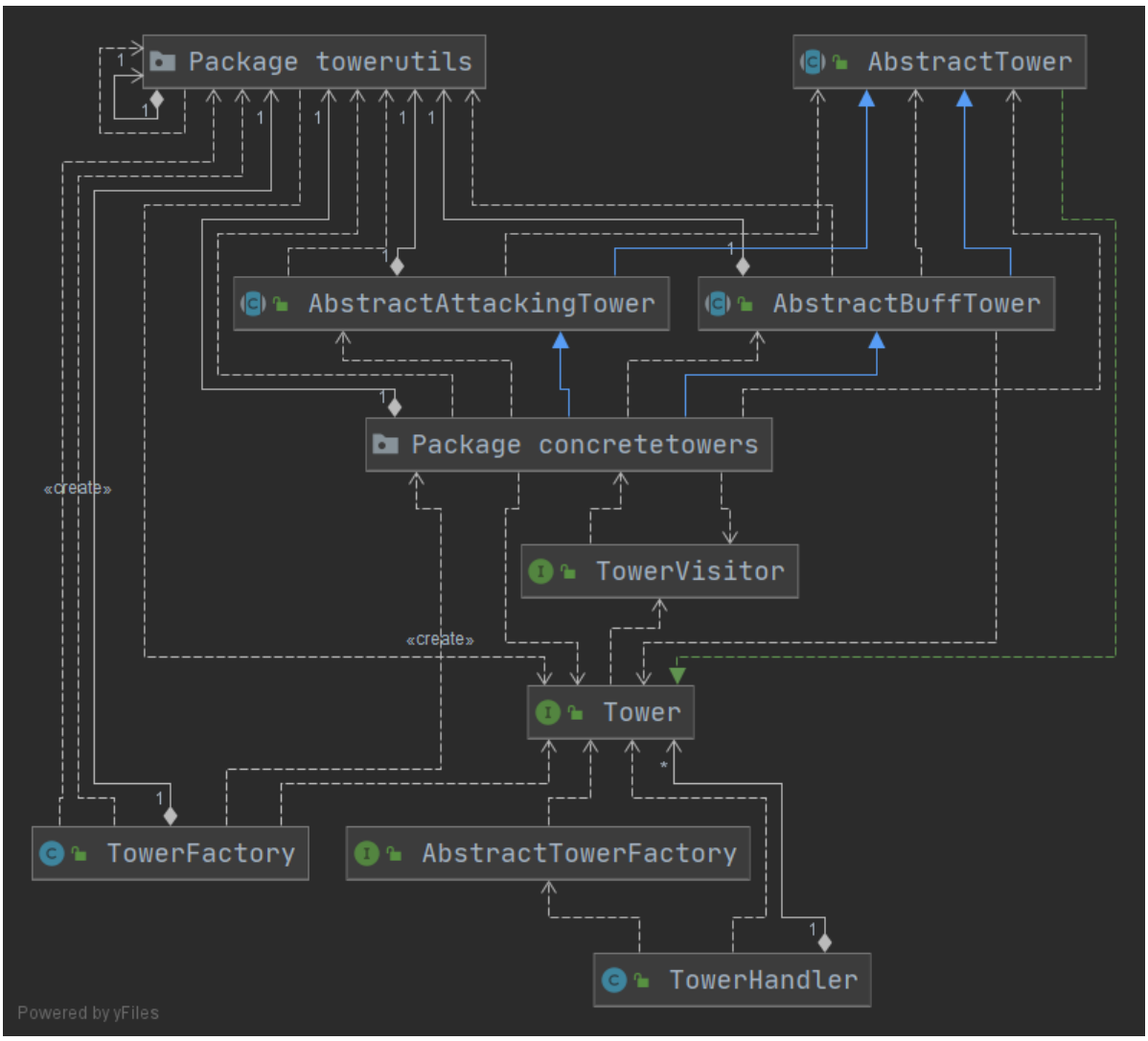

Figure 13: Class diagram for Tower

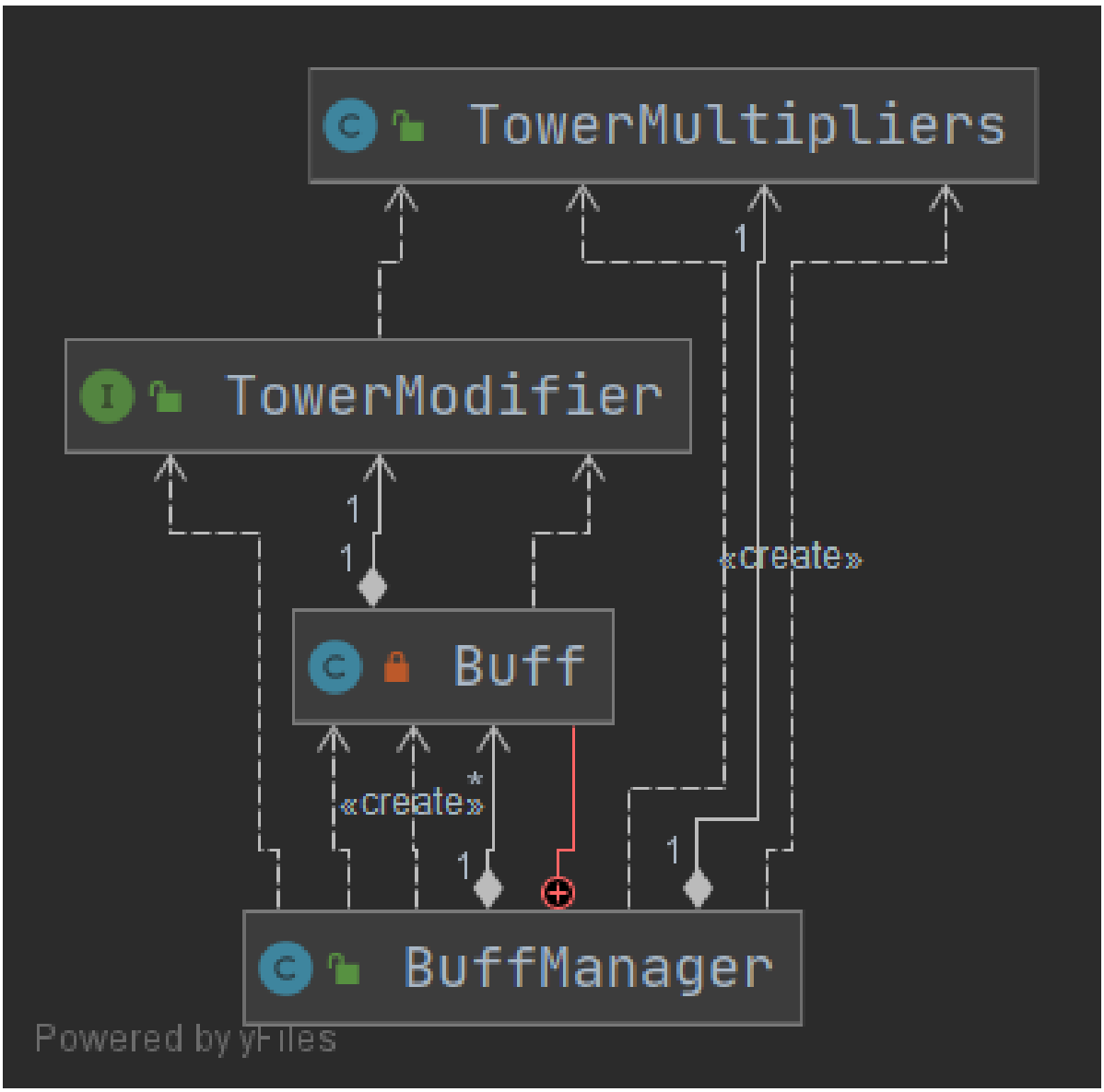

Figure 14: Class diagram for buff

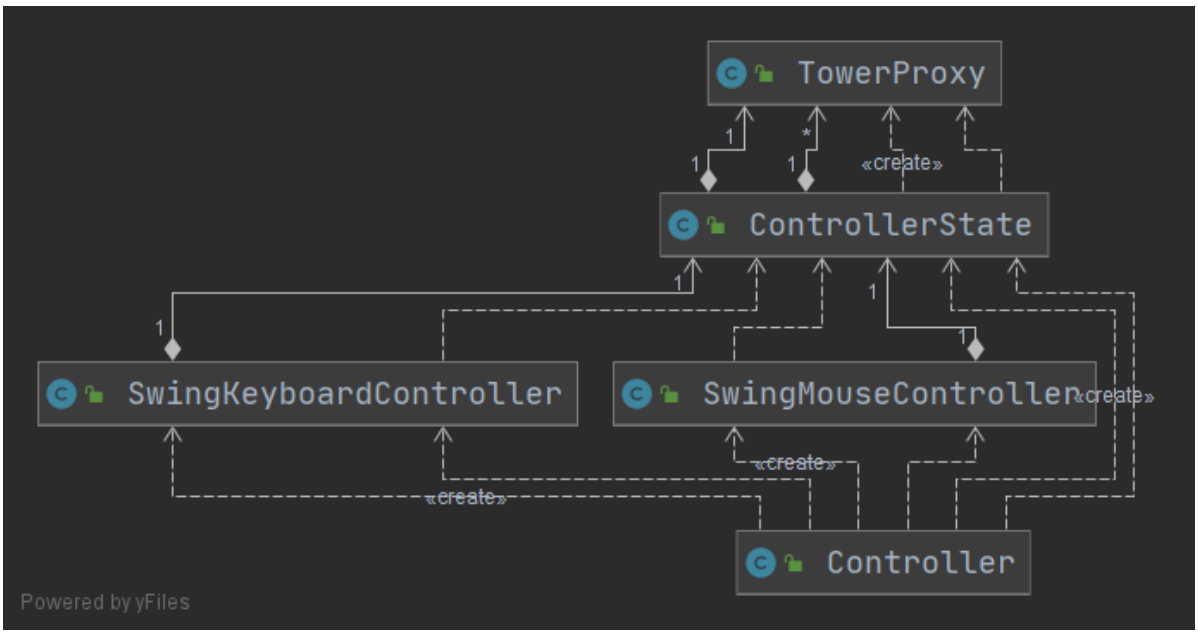

Figure 15: Class diagram for Controller

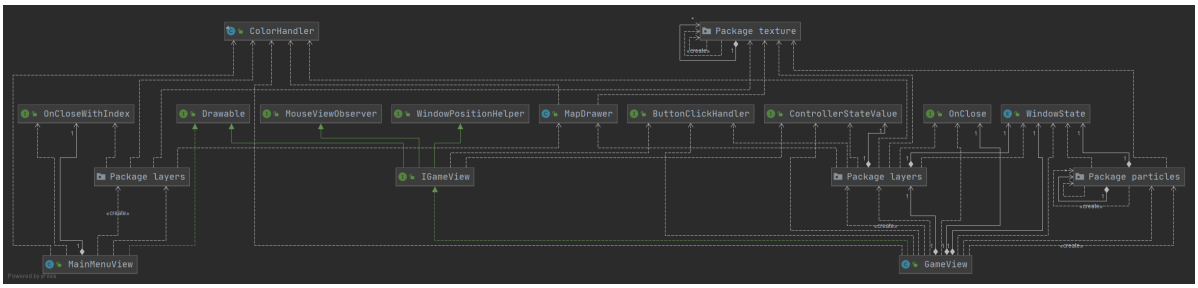

Figure 16: Class diagram for View with Layers

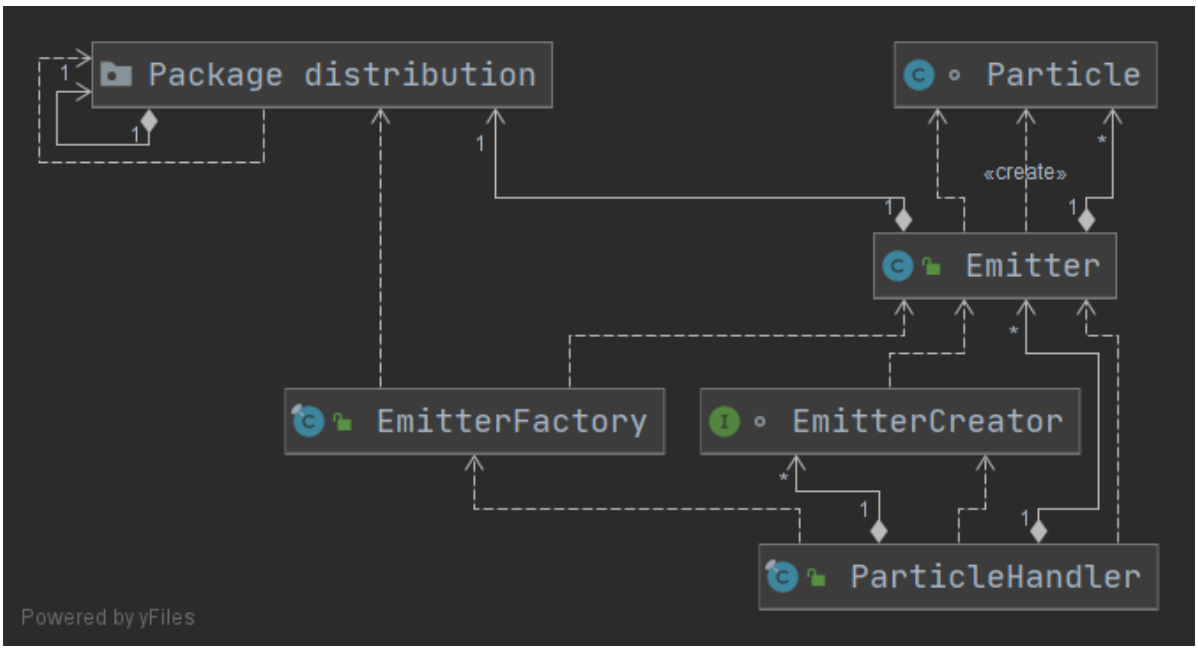

Figure 17: Class diagram for Particles

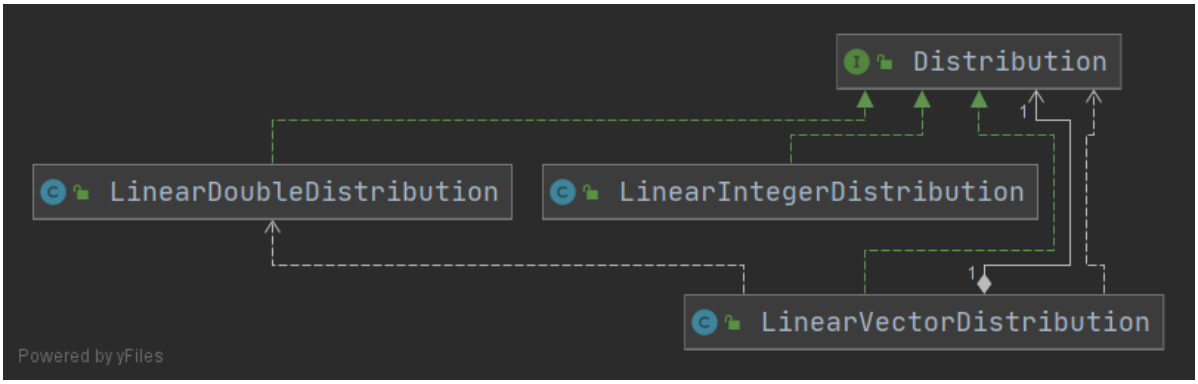

Figure 18: Class diagram for Distribution