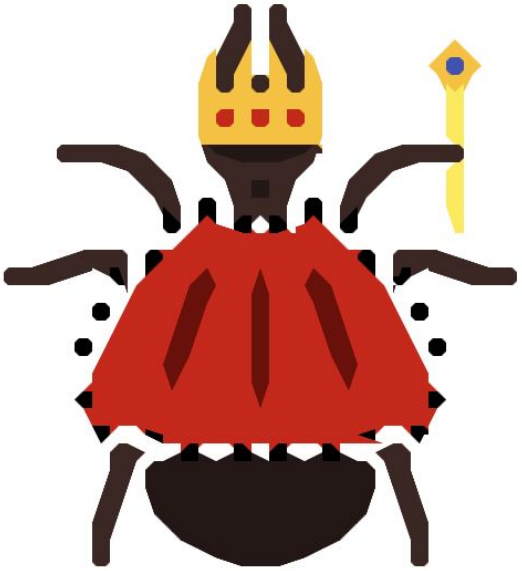
BugTD

- A buggy game



Mads Due Kristensen (vejleder) madkr17@student.sdu.dk

Gruppenummer: 1

Afleveringsdato: 27/05-2020

Jakob S. Winkel *jwink18@student.sdu.dk*

Marcus M. Pedersen *marpe18@student.sdu.dk*

Morten K. Jensen *mortj18@student.sdu.dk*

Nicolai C. G. Rasmussen *nrasm11@student.sdu.dk*

Oliver M. van Komen *olvan18@student.sdu.dk*

Thomas S. Laursen *tlaur18@student.sdu.dk*

Table of contents

1 Editorial

Chapter	Responsible	Contribution by	Controlled by
Title page	Morten		All
Abstract	Jakob, Marcus, Thomas		All
Introduction	Jakob		All
Requirements	Jakob		All
Analysis			
Use cases/ gameplay	Marcus		All
Object model	Marcus		All
Sequence diagrams	Marcus		All
Algorithms and data structures	Oliver		All
AI analysis	Thomas		All
Design			
Components	Morten, Nicolai		All
Different elements of the game and connections	Oliver	Nicolai	All
AI design	Thomas		All
Implementation			
Map	Nicolai	Morten	All
AI implementation	Thomas		All
Player implementation	Jakob		All
Weapon implementation	Marcus		All
Enemy implementation	Morten		All
Algorithms and datastructures	Oliver		All
OSGi implementation details	Thomas		All
Known bugs	Morten	Thomas	All
Test			
Testing	Nicolai		All
Discussion			
Discussion	Oliver		All
Conclusion	Morten, Nicolai, Oliver		All

2 Code editorial

Please note that every team member has contributed to all the components, this is just to indicate the focus of each team member.

Component	Responsible	Contribution by
Common	All	All
CommonAI	Oliver	
CommonEnemy	Oliver	Marcus
CommonTower	Jakob, Thomas	
CommonPlayer	Jakob	Thomas
CommonWeapon	Marcus	Morten
CommonMap	Nicolai	Oliver
AI	Oliver, Thomas	
Enemy	Oliver, Morten	Marcus
Tower	Jakob, Thomas	
Player	Jakob	Thomas
Weapon	Marcus	Morten
Map	Nicolai	
EnemySpawner	Morten	Nicolai
GUI	Nicolai	
Queen	Thomas	

3 Abstract

The main issue is to build a component based tower defense game. The application must be able to handle dynamic loading and unloading of components during runtime. This requires a component framework to be applied. Building a component oriented system introduces many advantages in software systems such as *Robustness*, *Ease of Maintenance*, *Flexibility*, *Extensibility* and *Simple application design* **[4]**. Furthermore, our knowledge from both Algorithms and Data Structures and Artificial Intelligence had to be put into practice throughout this project.

To satisfy these requirements, the group decided to build BugTD - a bug themed tower defense game. Wasps (enemies) spawn at the top of the map and make their way through to the end where the queen ant is placed, and tries to kill her. The user has to prevent this by placing tower ants in their way to shoot the enemies. The game is won if the queen survives three waves of fierce swarms of wasps.
The game is built with Apache Felix OSGi with the use of declarative services to load components by dependency injection. Each type of entity inside the game is its own component (Enemy, Queen, Player, Weapon, Map etc.). Every component in BugTD is constructed to be consistent with the Common library. This library contains information shared across all components and contracts they can satisfy or use as a way to access other components that satisfy these contracts.
Artificial intelligence exists in the form of a pathfinding algorithm used by the enemies to navigate through the map and around obstacles to kill the queen. Here the A* algorithm is used with a heuristic function equalling the distance to the queen following a straight line. By itself the A* algorithms can be quite demanding and time consuming. Therefore, a multiplier was added to the algorithm, making it a weighted A*. This change sacrifices a little optimality but increases performance. Other parts of the system also underwent changes to optimise their run times.

The result of this project is a fully functional component oriented 2D tower defense game with working AI. By using the OSGi component framework, the game is able to dynamically unload and load components during runtime. The game itself works mostly as intended with enemies dynamically calculating routes through the map. A lot of functionality that was initially planned for the game has not been realized due to a shortage in time. However, adding this in the future should be unproblematic as the component based architecture allows for easy extension and modification of the system. With the help of Algorithms and Data Structures, some parts of the game have been optimized to decrease their time complexities. However, the user can still experience occasional stuttering if too many entities populate the game.

4 Introduction

BugTD (e.g. Bug Tower Defense) is a component based 2D game. The game implements elements of the following courses: Component based software development, Algorithms and data structures, and Artificial intelligence to create a tower defence game.

The game takes inspiration from the strategy game genre with a focus on tower defence. Furthermore the game draws heavy inspiration from Warcraft 3 tower defence custom games. A tower defence game is a game where towers can be built in order to block, impede, attack, or destroy enemies, from reaching a specific goal.

The essential part of this project is to develop the game with a modular approach. This means that the game development has to satisfy a series of requirements on how it is structured, with the ultimate goal of being able to load and unload these modules during runtime. The game is therefore built from several individual modules. Each module makes up a seperate part of the game, and together all of the modules make up the game as a whole. Each of these modules are provided with an interface that allows the components to implement a specific behavior that is needed to run the game. With the provided interface implementation, tasks such as maintenance is made significantly easier as the specific part of the game is already isolated in its own module which reduces the rippling effect on the system, and therefore reduces errors and other unfortunate occurrences. Making the game this way also helps out with swapping modules all together, as any implementation that satisfies the provider interface would work with the game.

Another important part of the project is the use of artificial intelligence and the use of data structures. As mentioned earlier, the game implements several components that each are responsible for a specific implementation of behavior. For the game to be interesting and playable, it is essential to not just have some predefined actions for the enemy, but rather a dynamic force to play against which can make intelligent decisions to make the gameplay interesting, and challenging. Therefore the use of an artificial intelligence is necessary to make this goal a reality.

With the essential core problem described, a research question has been proposed, that aims to incorporate all issues of the project.

With the use of a component framework, how can a 2D tower defence game be developed that incorporates the key components: Player, Enemy, Weapon, GameEngine and Map, that implements data structures and algorithms, as well as an Artificial intelligence?

4.1 High level features

The objective of the game is to protect the queen from incoming enemies. These spawn at the top of the level and make their way through the map. The game includes a player who can be controlled with the arrow keys. He has a small weapon that shoots enemies when nearby, but more importantly has the ability to place towers throughout the map by clicking on certain tiles. Towers block the way for enemies and deal damage to them when within range. One tower alone will not be strong enough to destroy all enemies, but rather depend on a strategic placement of multiple towers in order to succeed. The enemies will take the shortest/ most optimal path to the queen, unless the

path is occupied by towers. In this case the enemies have to find a new route around them to reach the queen. By randomly placing towers on the map, the enemies will reach the queen and destroy her and the game will be lost. Instead, towers should be placed in a maze-like orientation to maximize the travel time for an enemy as well as maximize the potential damage inflicted on an enemy. If the path to the queen is blocked, the enemies will destroy towers, forming a new path to the queen.
The game has several waves that spawn a set amount of enemies with increasing difficulty for each new wave. All waves need to be cleared in order to win the game. If the enemies manage to reach and destroy the queen, the game will be lost. Otherwise, if the player can manage to protect the queen for the duration of the waves, the game will be won.

5 Requirements

5.1 Functional Requirements

The Functional requirements are largely structured around individual components in the project. The requirements list is subdivided, so that each component has their own requirements list. For this game the main focus is the following components headlines found in the functional requirements list.
All requirements have been implemented according to the MoSCoW method, where requirements of most importance are listed as the highest priority and least important with the lowest priority. There are four categories of priorities that can be assigned, which are: M, for Must have, S for Should have, C for Could have, and W for Won't have. In addition not all requirements have made it into the final project as they were deemed to have a lower priority. Due to this fact there was not enough time to implement these and therefore they are left out in the current version.

Functional requirements		
ID	**Modules**	**Priority**
F0	The game has to include Player, AI, Enemy, Tower, Queen, Weapon, and Map components.	M
Player		
F1.1	The Player can buy and place, different towers with different attributes, on the Map's path	M
F1.2	The Player can sell/remove Towers from the Map	C
F1.3	The Player can pause/resume the game at any time	C
F1.4	The Player can buy upgrades for Towers and Queen	W
F1.5	The Player can complete multiple maps in order to complete the campaign	W
F1.6	The Player can earn gold in multiple ways	S
Game		
F2.1	The Game saves the player's progress when they complete a map	W
F2.2	The Game saves current game state when a player exits a map without completing it	W
F2.3	The Game is lost when the Queen dies	M
Enemy		
F3.1	Enemies can inflict damage to other entities	M
F3.2	Enemies can take damage and be destroyed	M
F3.3	Enemies can walk on the path and move ideally through the maze	M
F3.4	If there is no path through the maze, enemies will destroy towers	M

F3.5	The game has multiple difficulty levels	W
Tower		
F4.1	Tower can inflict damage to enemies	M
F4.2	Towers are upgradable	C
F4.3	Towers can take damage and be destroyed	M
F4.4	The Player can complete a map by surviving all waves of Enemies	M
Queen		
F5.1	Queen can damage Enemies	S
F5.2	Queen can take damage and be destroyed	M
F5.3	The Queen is upgradable	C
Map		
F6.1	The game can load a Map	M
F6.2	The game has waves of enemies	C
F6.3	Enemies spawn increasingly difficult in waves	C

5.2 Non-Functional Requirements and design constraints

The non-functional requirements to an extent describe the constraints and quality attributes of the system. These requirements are mostly concerned with regulatory constraints that have to be met.

Non-functional Requirements		
ID	**Description**	**Priority**
NF1	Enemy AI performs in real-time without calculation delays	S
NF2	The game consists of components that can be updated and removed dynamically at run-time	M
NF3	A component framework has to be applied. The component framework has to support multiple classloaders per component and component versioning.	M
NF4	The game should look good with cool animations	W
NF5	A max of 500 milliseconds delay when the human player interacts with the game	C
NF6	No components can have dependencies on other components	M

Design constraints describe boundaries given for the development of the game.

Design Constraints	
ID	**Description**
C1	The game must be made with Java
C2	The game must use LibGDX game engine

6 Analysis

6.1 Use Cases / Game-play

Below, five essential use cases are described in detail. These have been made to improve the understanding of what actions the actors can take. The actors in this system are:

- Human Player: The user of the system.
- Enemy: The ingame enemies that have to reach the Queen.
- Queen: The endpoint of the map that the human player has to protect and the enemies have to reach.
- Tower: The obstacles that the human player can utilize to prevent/slow down the enemies from reaching the queen.

01. Choose Difficulty
    - **Actors**: Human-Player
    - **Precondition**: Must be in main menu
    - **Postcondition**: The game starts with the selected difficulty
    - **Main flow**: A presentation of several difficulties is given and the user is prompted to select one, starting the game at the chosen difficulty
02. Walk
    - **Actors**: Player, Enemy
    - **Precondition**: Must be in game
    - **Postcondition**: The entity will be moved depending on the movement indication given
    - **Main flow**: A movement indicator is given and the entity moves
03. Place Tower
    - **Actor:** Player
    - **Precondition:** Must be in game
    - **Postcondition:** The tower will be placed on the map
    - **Main flow:** Tower placement is indicated on the map, then the character moves to the indicated marker and places a tower
04. Sell Tower
    - **Actor:** Player
    - **Precondition:** Must be in game
    - **Postcondition:** The tower is removed and money is given back from the sale
    - **Main flow:** A tower is selected and a sell indicator appears that can be interacted with to remove and give money back from the sale
05. Attack
    - **Actor:** Player, Queen, Enemy
    - **Precondition:** Entity must have weapon
    - **Postcondition:** The attacked entity loses health and the weapon is set on cooldown
    - **Main flow:** The entity attacks another entity with its weapon, damaging the entity depending on how much damage the weapon does. After the attack, the weapon is met with a cooldown, preventing the weapon from being used until the cooldown timer is done

6.2 Object Model

Figure **6a** displays how the system ties together through the use of an object model. The boxes in this diagram represent concepts in the game and the lines determine their relationships between each other. The use of this diagram is to get a better understanding of how the system will function and to outline the interactions between different concepts.

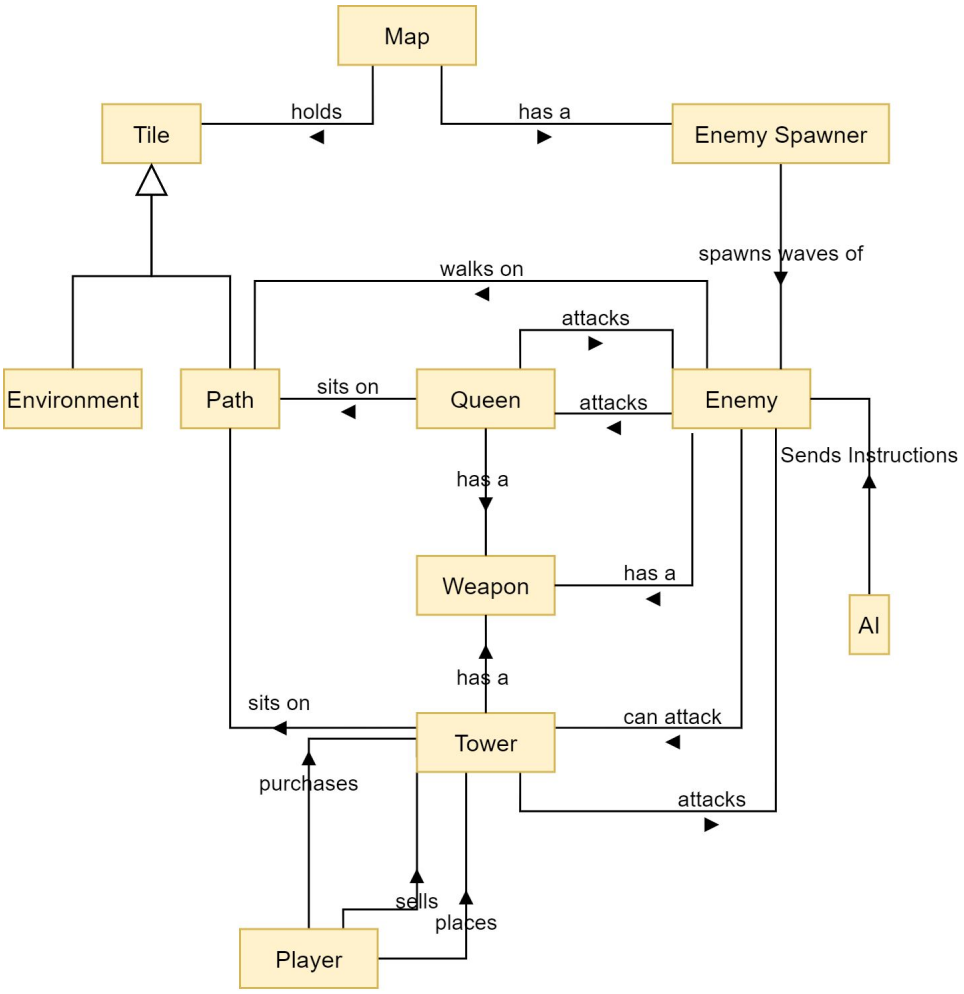


Figure 6a: *The Object Model of the project.*

6.3 Sequence diagram (communication between components)

To get a better understanding of what the system should do when the user interacts with it, sequence diagrams have been created. These sequence diagrams show how the user interacts with the system and how the different modules in the system react with each other for every use case.

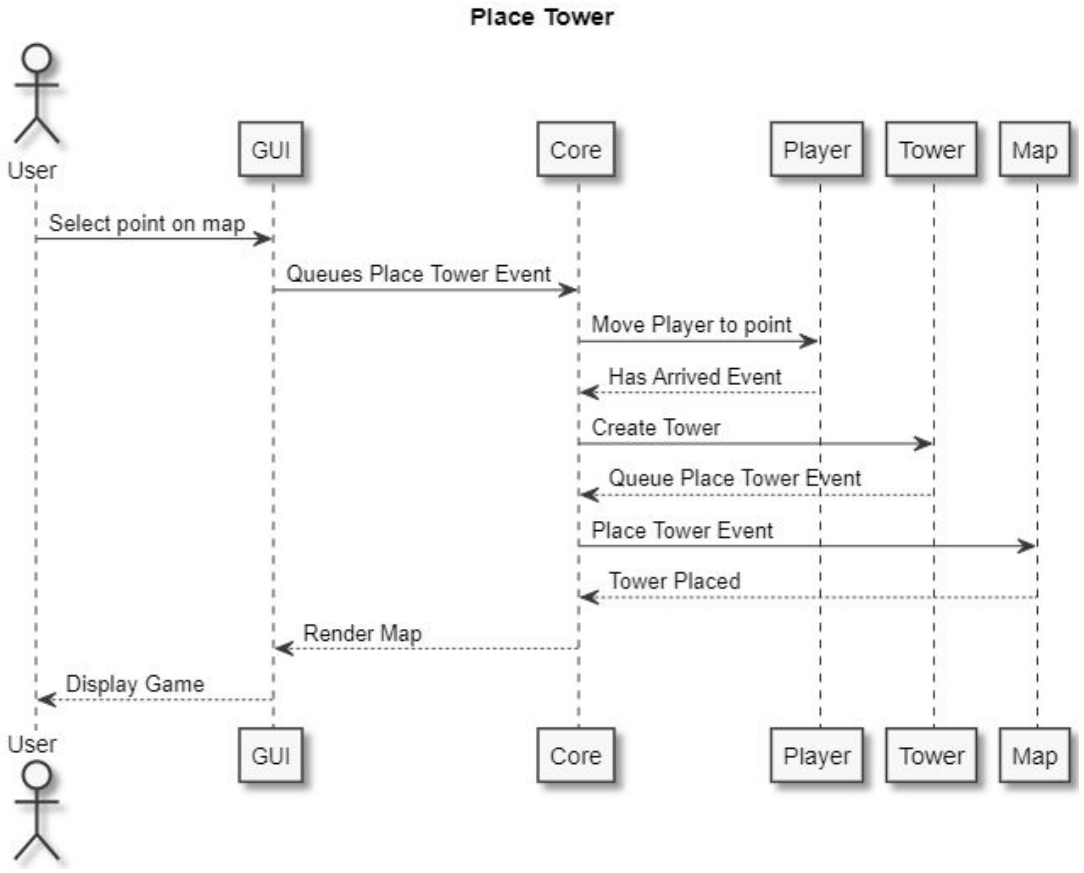


Figure 6b: *Sequence Diagram of Place Tower Use Case.*

In Figure **6b**, the sequence diagram explains the interaction between the modules in question. The modules in the place tower diagram rely on the use of events to communicate what needs to be happening in the game, the core being where queued up events are stored and the modules are triggered by said events.

6.4 Algorithms and datastructures

Run time analysis is an analysis meant to determine the performance of a piece of code. With better performance, developers have more opportunities to add more functionality to the program. With bad performance all other aspects (user experience, maintainability, modularity, security etc.) will suffer. Asymptotic analysis is a way of determining rough runtimes of algorithms based on input size (n). It does not consider the CPU the machine is using, the language the program is written in, the machine operating system or any other variables. It focuses on raw run times, in three different ways Worst, Average and Best case scenarios. The Worst case scenario is often the most important scenario to consider as statistically this is the case that is most often run. [1]

There are many different ways to notate this but the most frequent are the following three:

Theta Notation (Θ-notation)

This notation considers both upper and lower bounds running time of an algorithm and is therefore used to analyse the average case of an algorithm.

Big O Notation (O-notation)

This notation considers only the upper bounds running time of an algorithm and is therefore used to represent the worst case. It is the most used notation as the worst case scenario is the most frequent.

Omega Notation (Ω-notation)

This notation is the least used notation of the three. This is because it considers the lower bounds running time of an algorithm, which is almost never relevant as it very rarely happens. [2]

In the rest of this report Big O notation (e.g. worst case) will be used.

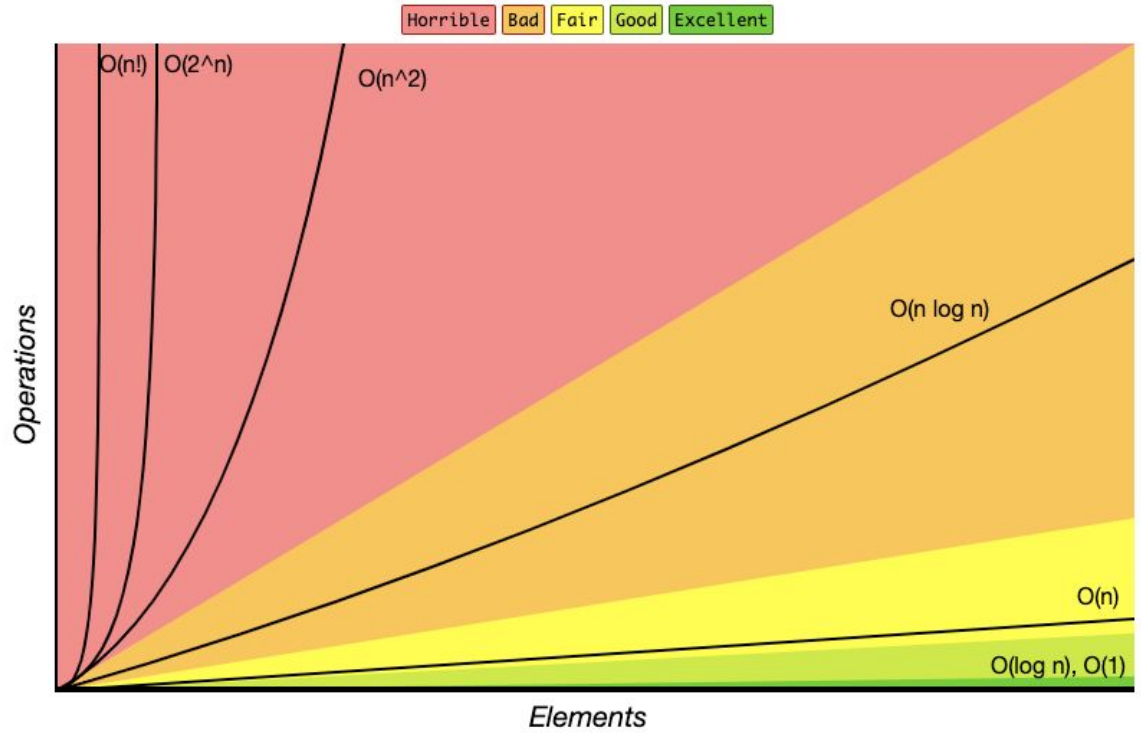


Figure 6c: *Overview of common run times divided into "Horrible", "Bad", "Fair", "Good" and "Excellent"*

As seen on figure 6c, There are some algorithms which scale horrible, bad, fair, good and excellent. The goal for this project is to be in the "Fair" category.

6.5 AI analysis

Throughout this project, we were required to implement some of the artificial intelligence knowledge we had acquired in the AI course into our game. In a game like BugTD, where enemies navigate through a map avoiding obstacles, some sort of pathfinding AI is needed. Choosing the right AI-implementation is important in order to ensure a balance between optimality and performance.

Implementing a pathfinding AI to the enemies in BugTD should not be problematic as the environment it operates in is quite friendly. The table below describes all aspects of the environment.

Fully/ partial observability	Deterministic/ stochastic	Episodic/ sequential	Discrete/ continuous	Single-agent/ multi-agent	Static/ dynamic
Fully	Deterministic	Episodic	Discrete	Multi-agent	Dynamic

Fully observable: The pathfinding AI in BugTD is able to fully observe its environment. Every obstacle can be seen from anywhere on the map. This makes it easier for the AI to determine the optimal path.

Deterministic: When the AI takes action, it will be able to fully predict its next state. For example, if it is planning to move in a certain direction, it will know exactly where it will be positioned next. No element of randomness is present. Also, the position of obstacles is not affected by the AI taking action. This allows it to predict its route with 100 percent accuracy.

Episodic: The AI is able to fully determine what to do based on the current state of the environment, and does not care about its earlier actions. It does not matter where the AI was positioned before unlike in an stochastic environment where it needs to keep memory of its earlier states.

Discrete: The environment contains a finite number of states and actions. The AI can only move up, down, left, and right, and the map contains a certain number of pixels and tiles it can move through. This makes it easier for the AI to predict the optimal route.

Multi-agent: The state of the environment is affected by the player when placing new towers. This introduces a certain amount of unpredictability to the AI's route, as it has the possibility to change at any time. The environment is made even more multi-agent because the game contains multiple enemies at certain times. These enemies can also change the environment when, for example, destroying towers in their way.

Dynamic: When the AI is performing its calculations, the environment is subject to change. This is due to the multiple agents described above that change the environment. This makes the AI more complicated as it will need to constantly monitor the environment for any changes. If a change is made, it will need to recalculate its actions.

Overall the environment our pathfinding AI will operate in is pretty favorable as only a few aspects make it unpredictable. For this sort of AI, the goal-based agent structure will work very well. This type of agent has a goal (for example a certain location on the map) that it tries to reach. The path of actions it takes to reach the goal can be long and complicated, and therefore the agent needs to constantly evaluate what will happen if it does different things and where it can go from there. Here it can make use of different search methods (informed/ uninformed) to make sure it reaches the goal in the best way possible. **Figure 6d** shows how the goal based agent operates (**[7]** page 52).

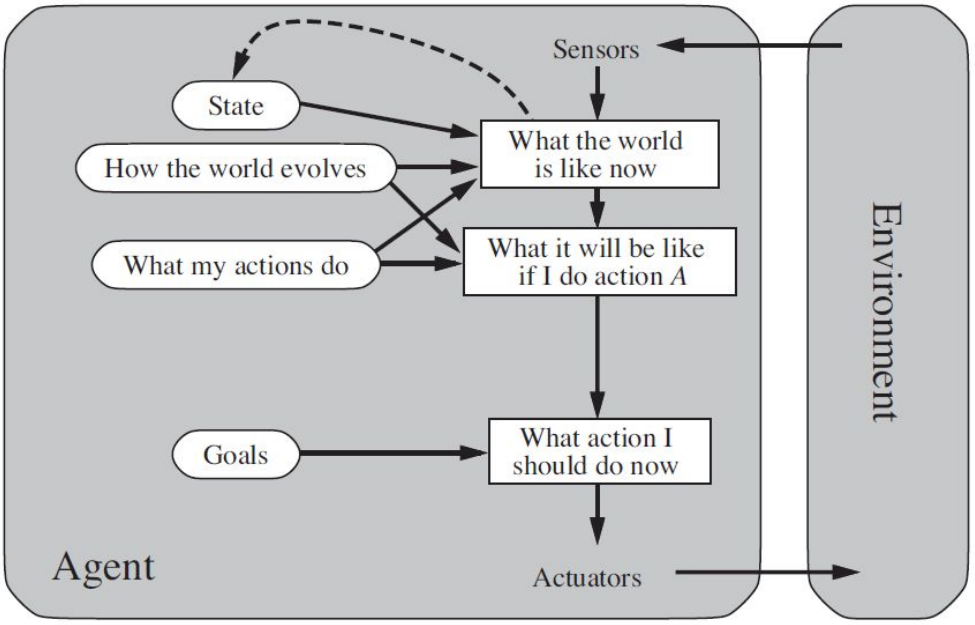


Figure 6d: *The inner workings of a goal-based agent getting input from an environment*

It takes the current environment state as input and keeps track of what will happen if it makes certain actions. Then it chooses the action that leads down the most desired path (based on parameters like length or time). In this case, this agent structure is more suitable than the simple reflex agent, because it is more flexible when you for example need to change the goal. A utility-based agent may also be possible to use. However, they are more suitable for environments with high uncertainty where probability plays a big role.

When the agent needs to search for a path to the goal, it makes sense to use an informed search algorithm. This uses a heuristic function to determine what path down the search tree is more favorable to explore. This brings an advantage over uninformed search algorithms as the agent is more likely to find the most optimal path sooner. The A* algorithm is an informed search algorithm where each stop along the path is given a cost consisting of the sum of its heuristic value (for example its straight line distance to the goal) and the total cost of travelling from start to it. Throughout the search, the agent always chooses to explore the action with the lowest total cost.

The map in BugTD will consist of several tiles. When enemies navigate through the map, they will follow these tiles by walking either up, down, left, or right. By introducing larger tiles instead of allowing the AI to navigate through individual pixels, the running time of the algorithm will be improved. For example, when the map is divided into tiles with a size of 16x16 pixels, there are 256 times less positions to consider when calculating a route than if each pixel was considered a tile.

The A* algorithm will be used in this project with the heuristic function equaling the distance along a straight line from the particular tile to the goal. A* has a running time of $O(b^d)$ where b is the branching factor (in this case 4 as the AI can move up, down, left, or right), and d is the depth of the search tree (**[7]** page 93). In this case, the depth depends on the distance the AI has to travel to reach its goal. The time complexity is exponential which lies in the "horrible"-section of **figure 6c**. This is part of the reason why the map needs to be divided into larger tiles. It is possible to make the

running time of the algorithm even faster but at the cost of some optimality. By multiplying a constant $k > 1$ on the heuristic function you get a weighted A* algorithm. For the AI this means that getting closer to the goal is more significant than taking a longer path. This results in the AI finding a path to the goal faster, but it has the risk of not being entirely optimal. Due to the improvement in time complexity, the weighted A* algorithm will be used in BugTD.

Another search algorithm that can be used to find the shortest path between an enemy and its goal is Dijkstra's algorithm. However, this algorithm is not faster than A*. It can be described as an uninformed version of A* because it does not use any heuristic function which is what makes A* search for a path in the direction it thinks is best. Not having a heuristic function makes it simpler and appropriate to use in lighter situations where time complexity is insignificant. However, because responsiveness is so important in this game, and the group members have more experience with A*, Dijkstra's algorithm will not be used.

In this game it is important that the time it takes for the AI to operate is as low as possible. Otherwise, the game would have frequent stuttering due to the amount of enemies in it. When an enemy spawns inside the game, it will need to calculate a route to the goal (the queen). After the route is calculated, the enemy remembers the path and can continue along it until it reaches the queen. For every new enemy that spawns, a route needs to be calculatet. However, if the map changes when for example the user places a new tower all the routes have to be recalculated in case the tower stands in the way of the enemies' already calculated routes.

7 Design

7.1 Components

In this section, the game's component architecture and the usage of OSGi is explained.

7.1.1 Component Architecture

The application has been designed using the *Entity Component System architecture* **[6]**, which aims to reduce the logic in the core game loop, and allow components to easily hook into the loop using a few simple common contracts. This type of architecture is good for encapsulating logic within each component, which is necessary for the requirement of loading and unloading components.

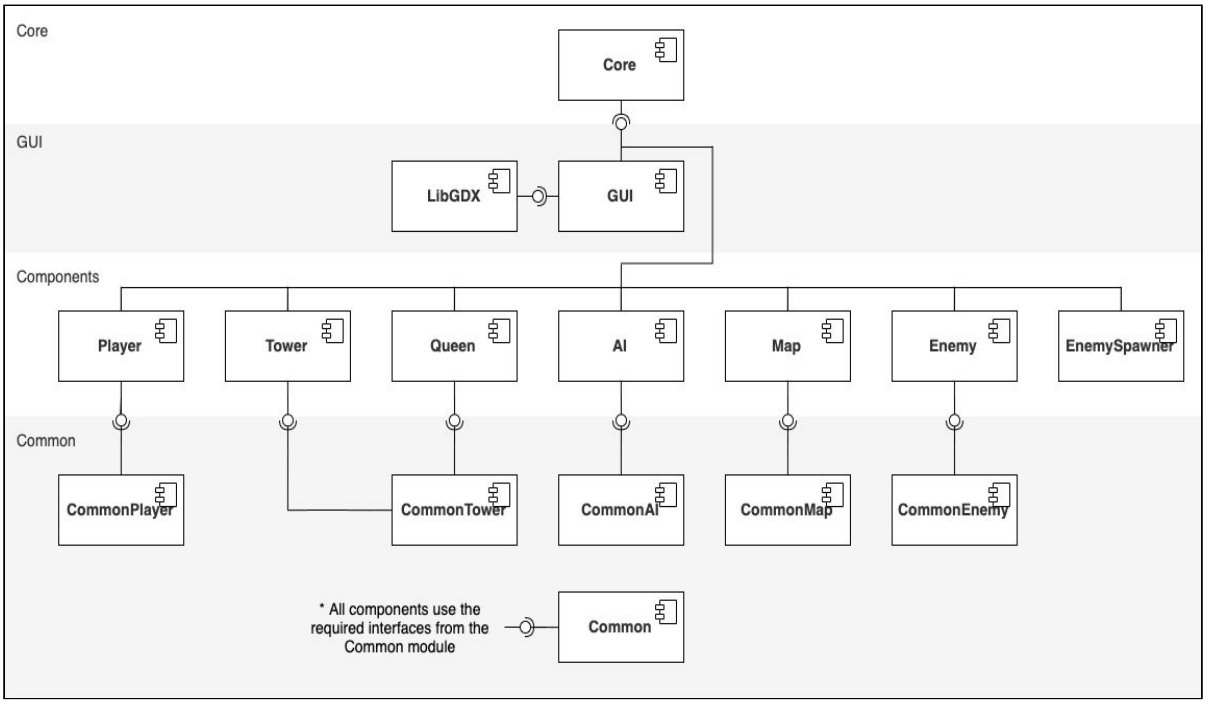


Figure 7a: *Component Diagram*

Core

At the top of the application is the *Core* component. It is the one responsible for loading and processing all registered services within the application.

GUI

The *GUI* level of the application is responsible for all drawing and user interaction with the game. It utilizes a bundled version of the game library *LibGDX*.

The *GUI* level consists of multiple screens (menu, game), which each have their own separate game loop that decides what to process while the screen is active.

Components

These are the components that need to be registered as services and loaded by the core component. They can also be seen as the actual content of the game. These components contain the implementation of each component.

Common
Common components consist of contracts. These contracts make it possible for components to depend on each other. It is used when for example the AI needs to process all enemies. Then the AI components can reference all objects that implement the enemy contract, which can be enemies of many different types.

7.1.2 Component framework choice

A component framework helps the developers create an abstract software implementation. Abstraction in the software eases some of the key advantages in software designs such as *Robustness*, *Ease of Maintenance*, *Flexibility*, *Extensibility* and *Simple application design*. All of these advantages help create software which gives a better prediction of the behaviour in the code while making it easier for the developer to isolate problematic parts and thereby reduce cost of bug fixing. Different components can be reused across multiple applications and the components ability to be loaded and unloaded during runtime, is an important advantage when the developer has to either update or bug fix code that needs to have the rest of the modules active, when compared to other architectures this feature can help keep an great uptime on the application. **[4]**

One of the reasons to use component framework is because of the "jar hell" problem. This is a versioning problem, where different libraries can depend on different versions of the jar. In the component framework, each component gets its own classloader which allows each library to get the right version for that library.

OSGi is a framework that lays on top of the Java platform, and it creates modularity which handles load, unload, the order of the loading of the modules, and much more. OSGi has a service layer which supplies a service registry along with a lookup model. This is relevant when a whiteboard model is implemented in the application. Most important for this project, is the declarative service layer of OSGi, which is the one used in this project to declare services and used in Dependency injection.

The modules in this project are connected with dependency injection. Alternatively the whiteboard model through the bundlecontext API with activators could have been used. The main difference between those two models is the way the target object is accessed. In the dependency injection model, the object inject itself into the target object, where in the whiteboard model there is a service locator, which the target object uses to access the dependencies. **[5]**

Netbeans has its own module system, but it can also depend on the OSGi module system. Using a runtime container such as Apache Felix. This opens the opportunity to run both the runtime container from the netbeans module system, but also in parallel to this, running OSGi module through the Apache Felix runtime container.

OSGi is the component framework chosen for this project, and Apache Felix is the runtime container implemented in this project. OSGi was also chosen because of recommendation from the supervisor (Mads) and its support for dependency injection along with the ease of console command for loading and unloading.

7.2 Different elements of the game and connections

Common component

Every component in our system will have a dependency on Common. This component consists of some very important classes which will tie the whole game together. These are classes such as World, Entity, Event, Parts. They will be used in all other packages. In the table below Is a run through of all the classes and what function they will have in our project.

Class/Interface Name	Package	Function
Entity	data	Will be the class which all physical entities in our game extends from. So Enemy, Tower, Queen etc. will all extend from Entity.
GameData	data	Will contain important data for all parts of our game. It contains events, Pixel width and height, Player's mouse placement, and time past since last frame (delta)
GameKeys	data	Will contain all the keys our game listens for
World	data	Will contain all active entities placed in the game.
EntityPart	data/ entityparts	An interface contract which other parts will implement.
AnimationPart	data/ entityparts	Will contain relevant data and logic in order to make animations for entities. We will be using atlas files for this purpose.
CollisionPart	data/ entityparts	Will give entities a width and height on which they can collide with other entities.
LifePart	data/ entityparts	Will be responsible for checking if entities are dead and to remove health when an entity is hit.
PreciseMovingPart	data/ entityparts	Will be used for precise movement when moving from Tile to Tile. This will ensure that entities only moves from Tile to Tile
PositionPart	data/ entityparts	Will be used to figure out where entities currently are placed on the game screen.
SpritePart	data/ entityparts	Will be used by entities who do not require an animation and is just a static picture. It will be responsible for drawing a picture.
WeaponPart	data/ entityparts	Will be used by entities who attack other entities. This will be able to reduce other entities' life and create an animation to indicate an attack.
Event	events	Will be the class which all other events will extend from.
ClickEvent	events	When the user clicks, this event will be triggered.
GameOverEvent	events	When the Queen is killed this event will be triggered.
GameWonEvent	events	When the Queen has been protected and all enemies are dead. This event is triggered.
PlayerArrivedEvent	events	Will be triggered when the user places a tower and the player entity starts walking towards the placement and reaches the correct position.
EnemySpawnedEvent	events	Will be triggered when an enemy is created and placed inside the world. The AI will listen for this event and calculate a route for

		the enemy spawned.
RouteCalculated-Event	events	Will be triggered when the AI is done with the calculation of a route for an Enemy. The AI will place instructions for Enemy inside this Event. Enemy will listen for this event and start moving each Enemy according to the instructions related to it.
MapChanged-DuringRoundevent	events	This event will be used in our AI processing service and our Tower processing service. When a new tower is placed this event is triggered. AI will then recalibrate the optimal path towards the Queen for all enemies. This is to both ensure that the enemies don't walk over newly placed towers but also to find the best path.
IEntityProcessing-Service		Is used in every module, and will be explained below
IGamePluginService		Is used in every module, and will be explained below

The game also consists of other common Components, such as CommonAI, CommonEnemy etc. These common components serve the same purpose of the Common component, it is these components other components can depend upon. For example CommonEnemy has an Enemy class, so that other components can search after entities which are Enemies.

7.2.1 Common Interfaces

This section explains the provided interfaces from the Common component that are used to load and process external components in the Entity Component System architecture. These interfaces are executed from the Core component.

IGamePluginService

This interface allows components to execute code while being loaded (start method) and unloaded (stop method). This is important to achieve dynamic loading, as the component must clean up after itself when unloading.

```java
public interface IGamePluginService {

    /**
     * Pre-condition:   Entities have not been added to the world
     * Post-condition:  Entities have been added to the owrld
     */
    void start(GameData gameData, World world);


    /**
     * Pre-condition:   There are entities in the world
     * Post-condition:  Entites are removed from the world
     */
    void stop(GameData gameData, World world);
}
```

Figure 7b: *IGamePluginService.java*

IEntityProcessingService

This interface allows to execute component logic once per iteration in the game loop. It is meant to be used by looking up all spawned entities using the *world.getEntities(Class<E>... entityTypes)* method, and then execute logic on them like moving or checking for conditions.

```
public interface IEntityProcessingService {

    /**
     * Pre-condition:   There are entities to process in the world
     * Post-condition:  Entities have been processed
     */
    void process(GameData gameData, World world);
}
```

Figure 7c: *IEntityProcessingService.java*

7.2.2 Events

From the very beginning we wanted to use an event driven architecture. For example with our AI, we considered two different models.

1. Every Enemy should have an instance of an AI which then would control it.
2. One central AI processing service which will send out instructions to all enemies.

The second model was chosen and for that we would need to use an Observer Pattern for events. On **Figure 7d** it is seen how we wanted to implement this.

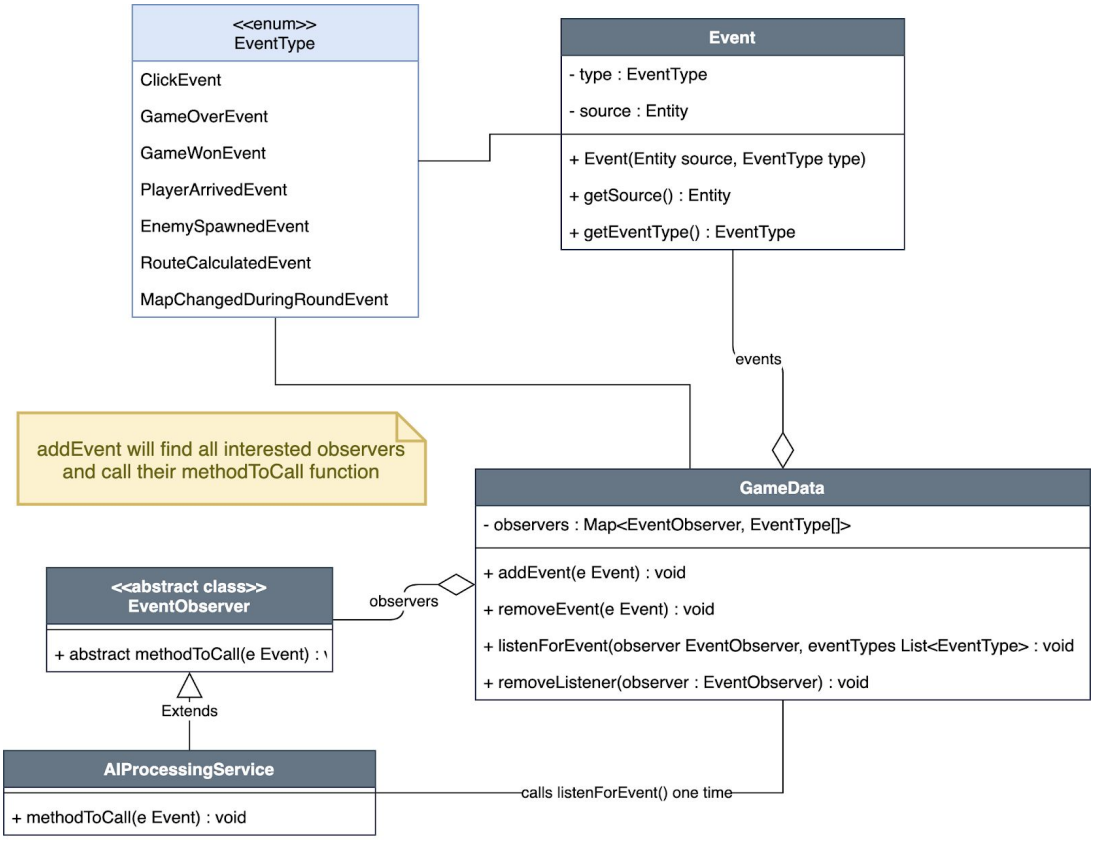


Figure 7d*: Event Observer Pattern. GameData holds a map of observers, with the observer as key and a value of all the events the observer is interested in. AIProcessingService is an example of an observer which calls* listenForEvent() *on GameData. GameData will then call AIProcessingService* methodToCall *when an event they are interested in is added.*

The alternative to the observer pattern was for each process cycle in our IEntityProcessingService interfaces to check if there are any events that the service is interested in. However this would lead to some unnecessary overhead, and we would like to keep the process cycle as quick as possible in order to increase frames per. second. The observer pattern was first implemented late in the project, as other things had priority.

7.3 AI design

Earlier in the analysis-section, we established that the A* algorithm is suitable for the pathfinding AI needed to control the enemies in BugTD. In this section, we will go into detail on how the AI works and is put together. To get an understanding of how the AI component in BugTD works, one might want to look at the class diagram on **figure 7e**. This diagram shows how the internal classes of the AI are connected.

At the top lies *AIPlugin*. This has access to an instance of *MapSPI* that can be used by *AIProcessingService*. *AIProcessingService* keeps track of any changes done to the map throughout the game (for example when a new tower is placed) in a manner that follows the observer pattern discussed earlier. If this is the case, it uses the *RouteFinder* to find a new route for each enemy in the game. In here, the actual A* calculation is performed. Its method *findRoute* shall iterate the tiles in the map and their connections to each other using the A* algorithm to find the most optimal route to the goal. For A* to work, each tile needs a heuristic score, and each connection between two tiles needs a cost. *RouteFinder* uses *QueenHeuristicScorer* and *TilePathCostScorer* respectively for this purpose, where the heuristic score is based on the distance to the queen, and the path cost depends on which tiles it travels from and to. When constructing the path, RouteFinder uses the RouteNodes as they hold the needed scores and a Tile “previous” allowing to traverse the route backwards when the goal is reached.

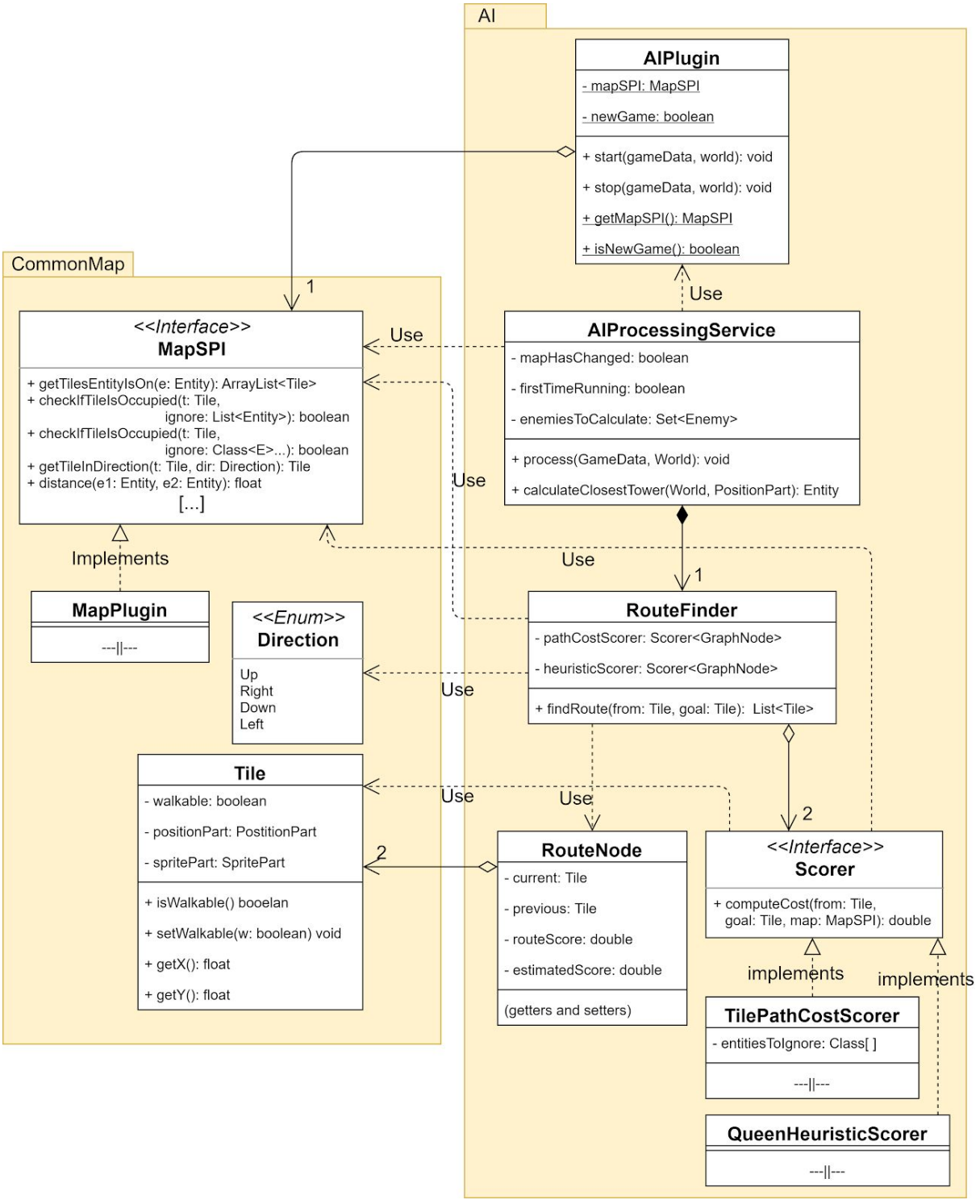


Figure 7e: *A design class diagram of the AI component and the parts of the CommonMap component it uses. Some of the less relevant methods and connections have been left out to keep the diagram more tangible.*

8 Implementation

8.1 Map

The *Map* component contains information about the current level in the game. At the start of any level, a map is loaded from a file (with file extension *.buggydata)*, which describes the tile pattern, how many enemies there are and how strong they are etc.

This file is a simple file with notations that define each part of the file, and the file holds data for waves, their enemy type, the amount of enemies, and their life, together with the information about the queen, her position, life, damage, range, and attack speed. Finally, the file also holds the position for the *EnemySpawner* to spawn enemies.

Other components are depending on this information, which can be accessed by referencing the *MapSPI* from the *CommonMap* component. As seen on **figure 8a**, the *MapSPI* also implements some utility methods like finding the closest tile to your position and getting the tiles any entity object is standing on, which are used by multiple components.

```java
public interface MapSPI {
    void loadFile(String filepath);

    /* MAP STATE */
    float getEnemySpawnY();
    float getEnemySpawnX();
    ArrayList<MapWave> getMapWaves();
    Tile[][] getTiles();
    void setQueenID(String queenID);
    Entity getQueen();

    /* UTILITY FUNCTIONS */
    Tile getClosestTile(float x, float y);
    ArrayList<Tile> getTilesEntityIsOn(Entity e);
    Tile getTileXAndY(Tile t);
    boolean checkIfTileIsOccupied(Tile t, List<Entity> ignoreThese);
    public <E extends Entity> boolean checkIfTileIsOccupied(Tile t, Class<E>... ignoreTheseClasses);
    Tile getTileInDirection(Tile tile, Direction direction) throws ArrayIndexOutOfBoundsException;
    void fitEntityToMap(Entity e);
    float distance(Entity e1, Entity e2);
}
```

Figure 8a: *MapSPI.java*

8.2 AI implementation

The AIProcessingService is in charge of detecting any changes made to the map and acting upon them. It uses the observer pattern to detect when events like *EnemySpawnedEvent* and

MapChangedDuringRoundEvent are created. When the map is changed, the AIProcessingService sets its boolean value *mapHasChanged* to true. In the process-method, the if-statement in **figure 8b** below is checked. Here all enemies are added to the list *enemiesToCalculate* to get their path recalculated later in the process-method.

```
1 if (mapHasChanged) {
2     mapHasChanged = false;
3     world.getEntities(Enemy.class).forEach((enemy) -> {
4         enemiesToCalculate.add((Enemy) enemy);
5     });
6 }
```

Figure 8b: *The if-statement in AIProcessingService's process-method that handles map changes*

The code snippet in **figure 8c** shows what happens later when the enemies are iterated through to have their routes recalculated. It starts by creating a new RouteFinder with the appropriate Scorers and then calls the method *findRoute* with the startTile (where the enemy is placed now), queenTile (where the enemy needs to be), and the mapSPI as arguments. This simply returns a list of tiles the enemy shall follow. For each of these tiles, a WALK-command is added to the current enemy's list of commands to let it know where to walk (line 6-8). Lastly, an ATTACK-command is added to make the enemy attack the queen when it is done walking (line 10) and it lets the enemy know its route is done by adding a new RouteCalculatedEvent (line 11).

```
1  RouteFinder routeFinder = new RouteFinder(new TilePathCostScorer(),
2                                            new QueenHeuristicScorer());
3
4  List<Tile> route = routeFinder.findRoute(startTile, queenTile, mapSPI);
5  List<EnemyCommand> enemyCommands = new ArrayList<>();
6  route.forEach((tile) -> {
7      enemyCommands.add(new EnemyCommand(tile, Command.WALK));
8  });
9
10 enemyCommands.add(new EnemyCommand(mapSPI.getQueen(), Command.ATTACK));
11 gameData.addEvent(new RouteCalculatedEvent(enemy, enemyCommands));
```

Figure 8c: *Later in the process-method where new routes are calculated for enemies.*

Inside the RouteFinder's *findRoute*-method, the code in **figure 8d** resides. This is the actual A*-algorithm. It starts by initializing a fringe as a priority queue. This way all tiles (actually RouteNodes) inside it are sorted according to their A*-cost. It initially has the starting-tile as the only element (where the enemy is currently positioned). Then it enters a loop that runs until the fringe is empty (unless a route is found (more on this later)). Here it extracts the element from the fringe that has the lowest cost and calls it *next*. Keep in mind that these elements are RouteNodes which have

cost attributes and an attribute for the tile that came before itself on the route. The method then iterates through all the neighbors of the current Tile by using the Direction enums found in the CommonMap component. If the neighbor has already been visited by the algorithm before, it is queried from the HashMap *allNodes* (line 17). If not, it creates a new RouteNode which has an initial PathCost of infinity meaning that no path to this node has yet been found. A new path cost is determined by the Scorers and saved as *newPathCost*. If this is lower than the neighbor's existing path cost, it means that we have found a better path to this tile, in which case its previous node, pathcost, and heuristic score are updated (line 26-33). Lastly, it is added to the fringe to allow it to become explored in a later iteration of the while-loop.

```
1  Queue<RouteNode> fringe = new PriorityQueue<>();
2  Map<Tile, RouteNode> allNodes = new HashMap<>();
3
4  RouteNode start = new RouteNode(from, null, 0d, heuristicScorer.computeCost(from, goal, map));
5  fringe.add(start);
6  allNodes.put(from, start);
7
8  while (!fringe.isEmpty()) {
9
10     RouteNode next = fringe.poll();
11
12     [...]   // Check for goal reached
13
14     for (Direction direction : Direction.values()) {
15         Tile neighbor = map.getTileInDirection(next.getCurrent(), direction);
16
17         RouteNode neighborRouteNode = allNodes.getOrDefault(neighbor, new RouteNode(neighbor));
18
19         double newPathScore = next.getRouteScore() +
20                 pathCostScorer.computeCost(next.getCurrent(), neighbor, map);
21
22         if (neighbor.equals(goal)) {
23             newPathScore = 0;
24         }
25
26         if (newPathScore < neighborRouteNode.getRouteScore()) {
27             neighborRouteNode.setPrevious(next.getCurrent());
28             neighborRouteNode.setRouteScore(newPathScore);
29             neighborRouteNode.setEstimatedScore(newPathScore +
30                                                 heuristicScorer.computeCost(neighbor, goal, map));
31             fringe.add(neighborRouteNode);
32             allNodes.put(neighbor, neighborRouteNode);
33         }
34     }
35 }
```

***Figure 8d**: Inside RouteFinder's findRoute-method. Here the A* algorithm resides.*

The findRoute method explores the tiles with the lowest A*-cost first. Our heuristic function returns the distance in pixels from a tile to the goal in a straight line. In addition to this, a constant of 5 is multiplied to it to make the algorithm weighted and therefore run faster. By testing with different multipliers, we found that 5 has a good balance between optimality and performance. The path cost for moving from one tile to its neighbor is initially 16 as this is the width/height of each tile in pixels. However, if the tile is not walkable or it is obstructed by other entities like Towers, the cost of travelling to the tile becomes infinity.

At one point in the findRoute-method, it will retrieve the goal node from the fringe (unless the path is blocked). This check is made in line 12 in **figure 8d** above. However we have left this part out to

make the code snippet simpler. Instead, this code can be seen in the snippet in **figure 8e**. After the node *next* is extracted from the fringe and is confirmed to be the goal, an ArrayList called route is initialized along with the RouteNode *current*. Then the do-while loop traces the route backwards and adds the tiles to the route list along the way which is then returned.

```
1 if (node.getCurrent().equals(goal)) {
2     List<T> route = new ArrayList<>();
3     RouteNode<T> current = node;
4     do {
5         route.add(0, current.getCurrent());
6         current = allNodes.get(current.getPrevious());
7     } while (current != null);
8
9     return route;
10 }
```

Figure 8e: *The if-statement inside findRoute that is executed when the goal tile is extracted from the fringe.*

The route has now been generated and delivered to AIProcessingService that constructs a list of commands for each enemy to follow. This has resulted in an AI that works just like expected. The route of the AI is not entirely optimal because of the weight added to the heuristic function, however this is made up for by the increased performance. If the weight is removed, the AI always takes the most optimal path, but there is noticeable stuttering when doing so. Even with the weight, the AI introduces a little stuttering when the game is filled with many entities.

8.3 Player implementation

The implementation of the player is found in the Player module, and the CommonPlayer module. Creation of the player is handled by the *PlayerPlugin* class, that implements the *IGamePluginService* interface. Here entity parts are given and their corresponding values. And the player is added to the world.

Control of the player is handled by *PlayerControlSystem*, which implements the *IEntityProcessing-Service.* The player can move in two different ways, either by clicking the screen, or using the arrow keys. The movement is controlled by the movePlayer method. This method consists of a series of if-statements that checks if any arrow key is pressed, and if so the player is moved in the direction. Movement in the four different directions are caused by the *moveHorizontal* and *moveVertical* methods. Each method takes an Entity object as well as a speed variable. The method is given a position part to record the current position., to move the player a new position is set with the speed variable. The player also has animations that are triggered by movement, and here the side to side and up and down animations are set. The player can also be moved by clicking on the screen. This is achieved by having a *clickEvent* that records the x and y coordinates of the map tiles. In addition, the target boolean is set to true to prompt the player to move towards the target. In movePlayer's

if-statement, the coordinates are checked to see if they are larger or smaller than the input to move the player in the given direction.

Another important part of the player's functionality is to place towers on the map. The player does not directly place towers on the map, as towers are handled by the tower module. Instead, when the player clicks on the screen and moves to the point, a *playerArrivedEvent* will trigger. This will only happen if the placement of the tower is legal as shown by the preview color of the tower. The tower module will then listen for the event, and place a tower when the player arrives at a given coordinate. The coordinates are given from the *clickEvent* and stored in the *playerArrivedEvent*.

```
public void movePlayer(GameData gameData, World world) {
    PositionPart posPart = player.getPart(PositionPart.class);

    if (isKeyPressed(gameData)) { // Movement from arrow keys
        player.setHasTarget(false);  // Cancel any current tower placement

        if (gameData.getKeys().isDown(GameKeys.RIGHT)) {
            moveHorizontal(player, speed);
        }
        if (gameData.getKeys().isDown(GameKeys.LEFT)) {
            moveHorizontal(player, -speed);
        }
        if (gameData.getKeys().isDown(GameKeys.UP)) {
            moveVertical(player, speed);
        }
        if (gameData.getKeys().isDown(GameKeys.DOWN)) {
            moveVertical(player, -speed);
        }
    } else if (player.hasTarget()) { // Automatic movement
        float dx = targetX - posPart.getX();
        float dy = targetY - posPart.getY();

        // Horizontal movement
        if (dx > 0) {
            moveHorizontal(player, dx > speed ? speed : dx);
        } else {
            moveHorizontal(player, dx < -speed ? -speed : dx);
        }
        // Vertical movement
        if (dy > 0) {
            moveVertical(player, dy > speed ? speed : dy);
        } else {
            moveVertical(player, dy < -speed ? -speed : dy);
        }
    }
}
```

Figure 8f: *implementation of movePlayer function*

8.4 Weapon implementation

For the implementation of a weapon in this project, a separate module is used to represent the weapon. The CommonWeapon module contains a weapon class, *WeaponPart*, that extends EntityPart, meaning that it is able to function the same way as other classes that extend EntityPart.

WeaponPart is created using damage, range, speed and color variable, representing weapons potential damage to the target, the range that the weapon can attack in its diameter, the speed of

how often the weapon can attack and the color of the attack line created when attacking its target. The process method in *WeaponPart* then checks if the target is valid, and if it is then it attacks the enemy and draws the visual line between the entity with the weapon and the target entity, as shown in **Figure 8g**.

```java
@Override
public void process(GameData gameData, Entity source) {
    LifePart targetLife = target.getPart(LifePart.class);
    if (targetLife != null) {
        if (targetLife.getLife() <= 0) {
            target = null;
        } else {
                // Check whether the Weapon is ready to shoot or not
            if (cooldown <= 0) {
                cooldown = speed;   // Reset cooldown
                targetLife.setLife(targetLife.getLife() - damage);    // Damage entity
                attackflash = 0.15f;
            }

            if (attackflash > 0) {
                //Get positionpart, spritepart and animationpart
                ...
                //Get position of the entities
                ...

                drawAttack(x1, y1, x2, y2);
            }
        }
    }
    cooldown -= gameData.getDelta();    // Slowly decreasing the cooldown
    attackflash -= gameData.getDelta();
}
```

Figure 8g: *WeaponParts process and use of drawAttack method*

Where other EntityParts are processed by the Entities that they are tied to, *WeaponPart* is processed separately in its own component. To process *WeaponPart*, a separate component called Weapon exists and contains *WeaponControlSystem* and *WeaponPlugin*, which are in charge of processing *WeaponPar*t and enable/disable weapons, respectively.

```java
@Override
public void process(GameData gameData, World world) {
    for(Entity e: world.getEntities()){
        WeaponPart weapon = e.getPart(WeaponPart.class);
        if(weapon != null){
            if(weapon.getTarget() != null && map.distance(e, weapon.getTarget()) <= weapon.getRange()){
                weapon.process(gameData, e);
            }
        }
    }
}
```

Figure 8h: *WeaponControlSystem's process method*

For the entities that utilize the weapon component, their job is to define targets for the weapon, then the *WeaponControlSystem* processes the weapon if it is in range to the target. This cycle is completed constantly, as the *WeaponControlSystem* is running as an *EntityProcessingService*.

8.5 Enemy implementation

The most used class of the Enemy module is the *EnemyControlSystem.* The control system uses the *IEntityProcessingService* interface which allows access to Gamedata and the game world. This makes it possible to get the part attached to each enemy instantiation and for each game loop, update the enemy health, position, and AI event.

For the movement of the Enemies, a *PreciseMovingPart* is created, and together with *PreciseMovementInstruction*, this handles the movement of the enemies according to the enemies x and y position. The movement of the enemy is calculated from the AI's A* algorithm, explained in the AI part of this report. The AI defines its position based on tiles, but the enemies movement is based on x and y positions. Since the tiles are not the same size as the acceleration of the enemy, the function *moveAndAttack()* in *EnemyControlSystem*, calculate the future position of the enemy and if that position is not the same as the tile from the AI's instruction, then it adds another movement to the queue in the PreciseMovement part. Only when the future position of the enemy is the same as the position of the destination given from the AI, then will the EnemyControlSystem begin to move the enemy towards the next tile in the AI's list for that enemy's movement. The rest of the movement is handled within the process method of the PreciseMovementPart. That Process takes the next element in the queue, and according to the direction of the movement, it changes either the x or y value and the amount the value is increased or decreased is based on the acceleration.

To create the amount of enemies needed for each map in the correct size of waves, the *EnemySpawner* module was made. This module uses the MapSPI to get the wave data such as the amount of enemies, the type of enemy, their spawn point, and the amount of life the enemies had. All the information needed about waves and entities, is located in the *.buggydata* file mentioned in the map part of this report.

All of this data is loaded into the map module. To get the wave information, a call to the method *getMapWave()* is made, and the call returns an ArrayList with all the waves. This is the wave information that the *EnemySpawner* uses to spawn its enemies.

The *EnemySpawner* keeps track of the amount of enemy spawned, and how many enemies had died to make sure that the correct amount of enemies is spawned per wave.

```
public void calculateCurrentRound(GameData gd) {
    if (deadEnemies == targetEnemyCount) {
        nextRoundTimer = nextRoundTimer + gd.getDelta();
        if (nextRoundTimer > nextRoundDelay) {
            if (currentWaveNumber < waves.size()) {
                if (waves.get(currentWaveNumber) != null) {
                    updateWaveInfo();
                    [...]
                }
            } else {
                gd.setMenuFlashMessage("Congratulations, you beat the level!");
                gd.addEvent(new GameWonEvent(null));
            }
            nextRoundTimer = 0f;
        }
    }
}
```

Figure 8i: *Method calculateCurrentRound from the EnemySpawner class.*

Figure 8i shows the implementation of the part of the *EnemySpawner* where it uses the data of enemies alive and the amount of enemies it's supposed to spawn to know when to begin the next wave.
When all enemies are dead, a timer is triggered, the timer uses an in-game delta time, to count down from when the last enemy dies, so the spawner begins to spawn enemies from the next wave.
Whenever an enemy is spawned an EnemySpawnedEvent is created which is used by the AI to know when it needs to calculate a new route for an enemy.

8.6 Algorithms and datastructures

In this section, a few code snippets will be shown. The worst case run time will be found and a few remarks on potential improvements will be made, or if improvements have been made, what they have done.

Core Game Loop

```
for (IEntityProcessingService entityProcessorService : entityProcessorList) {
  entityProcessorService.process(gameData, world);
}
```

Figure 8j: *Core Game Loop*

Figure 8j shows the core game loop. It is run constantly. If we ignore the amount of work done inside each process method call, and consider each process method call as a constant amount of work $O(1)$. Then the run time is $O(i)$, where i is the amount of *IEntityProcessingServices*, which means it is linear time. Linear time, as discussed in the analysis section, is a fair run time. We just have to be aware that for every entityProcessor we add, the run time increases by 1 amount of work. However we cannot ignore the amount of work which gets done on each process method call. So we have to analyze an *IEntityProcessingService* to get the true run time of the core game loop.

Enemy IEntityProcessingService

```
@Override
public void process(GameData gameData, World world) {
    // Looking for all Enemy Entities
    for (Entity entity : world.getEntities(Enemy.class)) {
        Enemy enemy = (Enemy) entity;
        // Check enemy lifepart
        ...

        // Move and Attack based on AI instructions
        ...
    }
}
```

Figure 8k: *Enemy* IEntityProcessingService*. An example of the logic in most IEntityProcessingServices*

Figure 8k, shows an example of a process method call inside Enemy *IEntityProcessingService*. Most of the *IEntityProcessingServices* have similar code with the same run time. First it calls *world.getEntities*.

```
public <E extends Entity> List<Entity> getEntities(Class<E>... entityTypes) {
    List<Entity> r = new ArrayList<>();
    for (Entity e : getEntities()) {
        for (Class<E> entityType : entityTypes) {
            if (entityType.equals(e.getClass())) {
                r.add(e);
            }
        }
    }
    return r;
}
```

Figure 8L: World.getEntities()

This method call has nested for-loops. The outer loop, loops over all entities (e) in the world, and the second loop, loops over the parameters entityTypes. The parameter entityTypes will in most cases only be a list of length 1, which means the nested loop in almost every scenario is a constant amount of work. So let's ignore the few cases where two or three different classes are searched after. Then the run time of this method will be $O(e)$ where e is the amount of entities and represents the first for-loop.

The amount of entities returned will be the amount of enemies in the world, (en). So the overall run time for enemy process method call will be $O(e \cdot en)$ As it first loops over all entities (e) and for each loop it loops over all enemies (en).

This means that the runtime for the core game loop will be $O(i \cdot e \cdot en)$. Here we assume that most IEntityProcessingServices' will have the same runtime as Enemy IEntityProcessingService.
Our game has five IEntityProcessingService's, which all has similar runtime analyzed in Enemy, except AI. This means that if we add another similar IEntityProcessingService we go from $5 \cdot e \cdot en$ to $6 \cdot e \cdot en$ which is a 20 percent increase in runtime. The same goes for the amount of entities in the game, if we go from 10 enemies to 20 enemies, the runtime will increase rapidly as it is a multiplier. So it's very important not to add too many IEntityProcessingServices and enemies, this however can be optimized.

Optimizing entity loop runtime

A 20 percent increase in runtime when adding one more IEntityProcessingService is not acceptable. Therefore the runtime of *world.getEntities* is optimized.

The game contains 2704 tiles because the map is 52 $\times$ 52 (width, height) tiles. Which means that when enemies want to iterate over all enemies in the world, *world.getEntities* will search through all 2704 tiles. Therefore this runtime needs to be improved as almost every *IEntityProcessingService* uses this each process cycle and unnecessarily iterates over all tiles.

Before the improvements world contained a *ConcurrentHashMap<String, Entity>* where the key was the entities ID, and the value was the *Entity* itself.

```java
private final Map<String, Entity> entityMap = new ConcurrentHashMap<>();

public String addEntity(Entity entity) {
    entityMap.put(entity.getID(), entity);
    return entity.getID();
}

public void removeEntity(String entityID) {
    entityMap.remove(entityID);
}

public Collection<Entity> getEntities() {
    return entityMap.values();
}

public <E extends Entity> List<Entity> getEntities(Class<E>... entityTypes) {
    ...
}

public Entity getEntity(String ID) {
    return entityMap.get(ID);
}
```

Figure 8m: *World entities methods.*

As seen on **figure 8m** all methods besides *getEntities(Class<E>... entityTypes)* have a constant runtime O(1). However it is the method *getEntities(Class<E>... entityTypes)* which is the most important to optimize as it is the most commonly used and has the biggest performance increase potential.

The goal was therefore to optimize this method call. This has been done by changing the entityMap to a *ConcurrentHashMap<String, HashMap<String, Entity>>*. Where the outer key is the entity class. This means for example that Tiles and Enemies entities will be separated out into each their own HashMap. So that when searching for all Enemies, it's possible to find them all in constant time.

```java
private final Map<String, HashMap<String, Entity>> entityMap = new ConcurrentHashMap<>();
public String addEntity(Entity entity) {
    HashMap<String, Entity> entities = entityMap.get(entity.getClass().toString());
    if (entities == null) { // first time an entity of that type has been added to the world
        entities = new HashMap<>();
        entities.put(entity.getID(), entity);
        entityMap.put(entity.getClass().toString(),entities);
    } else {
        entities.put(entity.getID(), entity);
    }
    return entity.getID();
}

public void removeEntity(String entityID) {
    for (HashMap<String, Entity> le : entityMap.values()) {
        Entity removedEntity = le.remove(entityID);
        if (removedEntity != null) { // The entity was found
            return;
        }
    }
}

public Collection<Entity> getEntities() {
    List<Entity> allEntities = new ArrayList<>();
    for (HashMap<String, Entity> le : entityMap.values()) {
        allEntities.addAll(le.values());
    }
    return allEntities;
}

public <E extends Entity> List<Entity> getEntities(Class<E>... entityTypes) {
    List<Entity> entitesFound = new ArrayList<>();
    for (Class<E> entityType : entityTypes) {
        HashMap<String, Entity> e = entityMap.get(entityType.toString());
        if (e != null) {
            entitesFound.addAll(e.values());
        }
    }

    return entitesFound;
}

public Entity getEntity(String ID) {
    for (HashMap<String, Entity> le : entityMap.values()) {
        Entity foundEntity = le.get(ID);
        if (foundEntity != null) {
            return foundEntity;
        }
    }
    return null;
}
```

Figure 8n: *The new optimized code for entities. Which brings* world.getEntities *from linear time to constant time.*

As can be seen in figure **8n** all methods, besides *getEntities(Class<E>... entityTypes)*, complexity has increased. The runtime of *getEntities* has now gone from linear time $O(e)$ to $O(1)$ because entityTypes will almost always only have a length of 1. And the lookup time for a *HashMap* is $O(1)$ **[3]**. This improvement is a compromise on less important methods in order to increase the performance of frequently used methods. This is often the case. It is important to optimize heavy performance hitters before optimizing anything else.

This means for the Core game loop with EnemyProcessingService used as an example it now has a runtime of $O(i \cdot en)$ where i is the amount of IEntityProcessingServices and en is the amount of

enemies in the world. This however is still not optimal as the game starts to stutter when we add more than 20 entities.

8.7 OSGi implementation details

In BugTD, all of the components are loaded using declarative services with OSGi. The way you declare services with this method is through an *osgi.bnd* file at the root of each component. The content of *osgi.bnd* from the Player component can be seen below on **figure 8o**. The last line points to two different META-INF files using their unique names inside the Player component.

```
1 Bundle-SymbolicName: Player
2 Bundle-ActivationPolicy: lazy
3 Service-Component: META-INF/entityprocessor.xml, META-INF/gameplugin.xml
```

Figure 8o: *osgi.bnd file for the Player module. It enables OSGi to load entityprocessor.xml and gameplugin.xml*

Each META-INF-file contains information needed to provide implementations for the interfaces the Player module realizes (in this case *IEntityProcessingService* and *IGamePluginService*). The content of the Player's *entityprocessor.xml META-INF* file can be seen below on **figure 8p**. On line 5, it states what interface it provides an implementation for (IEntityProcessingService) by using its unique name in the *Common* component. Line 3 states the unique name of the class that serves as the implementation of the interface (*PlayerControlSystem*). When the application starts, the Core module is able to load the services with its *addEntityProcessingService* and *addGamePluginService* methods. The *META-INF* file "*core.xml*" in the *Core* component binds the interfaces to each of these methods making it handle the injection automatically.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="dk.sdu.mmmi.playerControl">
3     <implementation class="dk.sdu.mmmi.player.PlayerControlSystem"/>
4     <service>
5         <provide interface="dk.sdu.mmmi.cbse.common.services.IEntityProcessingService"/>
6     </service>
7     <reference bind="setMapSPI" cardinality="0..1"
8                interface="dk.sdu.mmmi.commonmap.MapSPI"
9                name="MapSPI" policy="dynamic" unbind="removeMapSPI"/>
10 </scr:component>
```

Figure 8p: *Inside a META-INF file in the Player module. States the implementation for the interface IEntityProcessingService.*

With Apache Felix, you are able to dynamically unload and load individual components through the command line interface - even at runtime. When unloading a component, its stop-method inside its *IGamePluginService* implementation is called. This method is responsible for tearing the component down by for example removing any of its entities from the game (like the Player for instance). When it is loaded again, the start-method will be called which is in charge of reintroducing the component to the game, for example by adding the Player entity back in. The stop and start methods are called from inside Core's methods *removeGamePluginService* and *addGamePluginService* respectively.

Together this forms a completely dynamic game in which most components can be removed and added at will.

8.8 Known bugs

AI pathfinding to blocking tower

In certain situations when the path to the queen is blocked and enemies navigate to the nearest tower to clear the path, the game shuts down. This happens due to the way enemies decide which of the four corners of the tower they go to. They always navigate to the corner closest to them, but sometimes the closest corner is not accessible. This error is easily replicated in the third level by placing a tower with its upper left corner right in front of the small path's exit. Enemies that spawn will try to navigate to the upper right corner of the tower, but fail.

9 Testing

Tests have been written at the end of the development process, and have not played an important role in the project. Should the project be further developed, testing would have more importance. When a project grows, testing can help to ensure that no existing code breaks when adding new functionality to the codebase.

9.1 Unit Testing (Whitebox validation-testing)

The purpose of writing unit tests in the project is to validate that the individual functions or blocks or code perform as expected. It is therefore very important to be able to control the environment that the tests operate in. Some code might depend on other code, which behaviour we cannot guarantee, as it might change or have bugs. To solve this problem, a mocking framework (*Mockito*) is used, which allows you to define the behaviour of a given object, so that it becomes fully predictable.

In **figure 9a**, we want to test one of the functions on the *MapPlugin* class. The problem is, that it depends on external classes (*Entity*, *PositionPart*, *SpritePart*) with hidden implementations. Therefore, *Mockito* is used to script the behaviour of these classes on the method calls that are needed to run the test.

The test itself tries to make it clear how to calculate the number of tiles that a certain entity should collide with. It happens especially in the code section under the last comment "Manually calculate number of tiles mockEntity fully covers" Afterwards it calls the method it is testing, and checks if it gets the same result.

```java
// Unit testing of MapSPI method: getTilesEntityIsOn
@Test
public void testGetTilesEntityIsOn() {
    Entity mockedEntity = mock(Entity.class);

    // Setup mocked entity dimensions
    float entityHeight = 50;
    float entityWidth = 50;

    SpritePart mockedSpritePart = mock(SpritePart.class);
    when(mockedEntity.getPart(SpritePart.class)).thenReturn(mockedSpritePart);
    when(mockedSpritePart.getWidth()).thenReturn(entityWidth);
    when(mockedSpritePart.getHeight()).thenReturn(entityHeight);


    // Setup mocked entity position
    float entityX = 0;
    float entityY = 0;

    PositionPart mockedPositionpart = mock(PositionPart.class);
    when(mockedEntity.getPart(PositionPart.class)).thenReturn(mockedPositionpart);
    when(mockedPositionpart.getX()).thenReturn(entityX);
    when(mockedPositionpart.getX()).thenReturn(entityY);


    // Manually calculate number of tiles mockedEntity fully covers
    int tileSize = mapPlugin.getMapData().getTileSize();
    int collidingTilesX = Math.round(entityWidth / tileSize);
    int collidingTilesY = Math.round(entityHeight / tileSize);
    int collidingTiles = collidingTilesX * collidingTilesY;

    assert(mapPlugin.getTilesEntityIsOn(mockedEntity).size() == collidingTiles);
}
```

Figure 9a: *Unit test for getTilesEntityIsOn() method*

9.2 Integration Testing (Black box verification-testing)

It is especially important to test if components work together when developing a component-oriented architecture. Since components must be able to load and unload at runtime, we verify that they:

- Load even when a component it is dependent on is not yet loaded
- Access instances of other components through the OSGi framework or dependency injection
- Make sure to clean up spawned instances when unloading

In **figure 9b** is a test of the *Map* component which is responsible for reading a datafile containing game level data, and instantiating multiple components (here among the Queen of the map) which it depends on.
The test starts after the *Map* component is already instantiated in a setup method. The test checks for above rules, and makes sure the component acts as expected when interacting with other components.

```
@Test
public void mapAndQueenIntegration() {
    Queen queen = mapPlugin.getMapData().getQueen();

    // Map can create a queen from the loaded text file
    assert(queen != null);

    // Map adds the created queen to the world's list of entities
    assert(world.getEntity(queen.getID()) != null);

    mapPlugin.stop(mockedGameData, world);

    // Map can remove queen from the world.
    assert(world.getEntity(queen.getID()) == null);
}
```

Figure 9b: *Integration test for loading and unloading Map module*

9.3 Testing the OSGi runtime (Integration testing)

The previous test uses objects from an external component, hence we call it integration testing. However, these objects are instantiated manually in the test file. It would be a good idea to write the same test, but depending on the OSGi framework to load components, and then verify that they have been loaded correctly. That kind of test has not been written due to difficulties in the test in the current development environment and low prioritization.

10 Discussion

In this section each essential project requirement from the project description will be examined and discussed by first giving a short resume of how we solved the requirement, then the pros & cons of our solution will be discussed and evaluated.

10.1 High level requirement discussion

The game has to include Player, Enemy, Weapon, GameEngine and Map components

The game has the following components: Player, Enemy, Weapon, LibGDX, Map, Tower, Queen, AI, EnemySpawner, Core, CommonEnemy, CommonAI, CommonPlayer, CommonMap & Common. This requirement has been reached.

The Player, Enemy and Weapon components have to implement provided interfaces that allow the components to be updated and removed dynamically at run-time

This has been one of the primary purposes of the project. We have ensured that every component besides common components can be removed. If the Tower component is removed, every tower in the world is removed. If Enemy is removed, every enemy is removed and the AI will not calculate routes. If the AI is removed, enemies will not receive new instructions so enemies will walk over newly placed towers or stand still waiting for instructions. If the Queen is removed, the AI will not move enemies towards it. if Weapon is removed no entity is able to attack other entities. In order to ensure this unloading no component can have dependency on other components as that connection potentially can be broken if the dependent component is removed during runtime. Therefore no component has dependencies on other components.

A component framework has to be applied. You can use Netbeans Module System or OSGi
OSGi has been used. Specifically we have used Apache Felix OSGi implementation with the use of Dependency Injection through the usage of .xml and osgi.bnd files.

At least one component should implement an artificial intelligence technique
Based on the course "Artificial Intelligence", we concluded that the best AI algorithm to use is A-star (A*). The AI component implements this algorithm. It uses A* to control enemies by calculating the optimal route for enemies and sending instructions to the Enemy component which uses these instructions to move enemies around properly. It uses a weighted A-star with a startTile and a goalTile which often is the Queen in order to increase performance while still having a pretty optimal route.

Data-structures and algorithms have to be applied and documented
Data structures and algorithms have been helpful to reach a better run time. The most commonly used method had a linear run time, and was called every frame, this inspired us to make a new implementation which made the method to run in constant time. Besides this optimization. A lot of refactoring and rework has been done especially in regards to the AI implementation which all have improved the run time of the AI.

10.2 Low level requirements discussion

In this section our projects own goals and requirements will be discussed. All requirements with a priority of Won't have, will not be discussed.

F1.1 - The Player can buy and place, different towers with different attributes, on the Map's path
When the player hovers over tiles with the mouse, a preview is shown which indicates if the player is able to place a tower or not. There only exists one type of tower, it doesn't cost anything. Therefore this requirement is only solved partially.
Being able to buy towers requires economy. The plan was to create a shop component which handled everything related to economy, such as giving the player money when they kill an enemy and to subtract money when they buy towers with a price tag. The plan was also to create a shop overlay where the user would be able to point and buy towers with their mouse. Due to time constraints these objectives were not met, but the essential part of the requirement was met satisfyingly as the player is able to create mazes with towers in which enemies have to navigate through.

F1.2 - The Player can sell/remove Towers from the Map
Once a tower has been placed on the map, the player is not able to remove it from the Map. Only enemies can remove towers if they block the path. This requirement has not been met. In order to meet the requirement we could have enabled right mouse button clicks to remove towers. It only had a priority of Could.

F1.3 - The Player can pause/resume the game at any time
The player is able to choose from three different difficulties which will load three different maps. If the player hits 'ESC' it will exit the current loaded map. This requirement has not been met. The plan

was when the player hits 'ESC' a menu should be opened and the game paused. It only had a priority of Could.

F1.6 - The Player can earn gold in multiple ways

The player is not able to earn gold in any way as the economy is not implemented. This was considered an important requirement (Should priority). But due to time constraints it was neglected. Without economy the game does not really feel like a game. It should have been a higher priority in our development.

F2.3 - The Game is lost when the Queen dies & F4.4 - The Player can complete a map by surviving all waves of Enemies

When the Queen's health reaches 0, the player gets thrown back into the main menu where a label "Game Over" is inserted. When the player completes all waves of enemies, they are also thrown back into the main menu where a label "Game won" is inserted.

F3.1 - Enemies can inflict damage to Tower and Queen & F4.1 - Tower can inflict damage to enemies & F5.1 - Queen can damage Enemies

Enemies and towers have a WeaponPart and every entity with a WeaponPart is able to inflict damage to other entities. The enemy is controlled by instructions sent by AI, where if the AI sends an instruction with an Attack command, the enemy will walk to the target, which can either be the queen or a tower and attack it. The tower and queen works by constantly searching for enemies within a certain range. If the tower finds an enemy, it will start attacking that enemy until the enemy is dead or moves outside its range. This implementation has proven to work well.

F3.2 - Enemies can take damage and be destroyed & F5.3 - Queen can take damage and be destroyed

Enemies have a LifePart which reduces the health of the enemy when a tower attacks it. It works well.

F3.3 - Enemies can walk on the path and move ideally through the maze

The map contains two different types of tiles. One which is grass (path) and the other dirt. The enemy only follows the path given to them by the AI. The AI is only able to give valid instructions where the enemy only walks on grass which is not occupied by a tower. This requirement has especially taken a lot of development time, and has been one of the primary focuses of this project. The final version works very well, and the Enemy never oversteps their boundaries, and always follows a valid path. It will not move ideally, due to a weighted A* algorithm which is used, but it moves close to ideal.

F3.4 - If there is no path through the maze, enemies will destroy towers

If the AI cannot find a valid path for enemies it instead orders enemies to attack the closest tower around them. In our game without economy this however doesn't really have any consequences for the player as they can just continue to place towers without a cost. So this requirement is technically met, but it is not at the state we wanted it to be.

F4.2 - Towers are upgradable & F5.3 - The Queen is upgradable
This requirement is not met. If the economy was implemented these requirements would have been the next requirements to implement.

F6.1 - The game can load a Map & F6.2 - The game has waves of enemies & F6.3 - Enemies spawn increasingly difficult in waves
The game is able to read data from a file and load maps. Each map can have 1 to many waves of enemies. A wave is a number of enemies spawning, where after each wave a short delay occurs so the player has time to optimize their maze.
The game has three different difficulty levels, which is three different files which are loaded. These files include everything such as tiles placement, queen's health, enemy waves, enemy health etc. This is also how we control the increasingly difficult waves. This implementation has proven to work well.

10.3 Evaluation overall
The primary problem of this project was to ensure dynamic loading and unloading components and to create a 2D tower defense game where the player can make with towers which protects their queen. This has been done to an extent. The game doesn't really feel like a game, as economy is not implemented, if there had been time to implement economy the game would have felt a lot more like a tower defense game. As right now it does not.

11 Conclusion
The main issue
The main issue of the project is to build a component base tower defense game. It is required that the application can handle dynamic loading of those components at runtime, so it needs a framework to handle that.

The dynamic loading problem has been solved by using the OSGi design specification. It allows the application to load new dependencies at runtime. It also allows further extension of the application without needing to change the core logic.

The project had some given requirements for the content of the game (Player, Enemy, Weapon, GameEngine and Map), which were all implemented successfully. Additional requirements were defined in the requirements section of the report, but not all were implemented.

Algorithms and Data Structures
Throughout the development process several performance issues had surfaced due to complex loops running excessively. This caused the game to feel slow at times, which started to become a big problem. By troubleshooting these issues and analyzing the runtime of certain loops, it was possible to reduce the processing power required to run the game, and provide a smoother experience for the user. However it must be mentioned that stuttering can still happen at certain times in the game, if there are too many entities in the world.

Artificial Intelligence
In order for the game to work, the computer must be able to control the enemy units, and decide how to act based on the environment. To solve this problem we were able to find a suitable algorithm (A*) by using our knowledge of different problem-solving algorithms and their respective pros and cons.

11.1 Future work

The next step in this project would have been to implement an economy module in the game since this would have a huge impact on the players experience when playing the game. Being able to buy, sell, and upgrade towers would add a whole new level of playing the game, since the game doesn't feel like a game due to the ability to place as many towers as the players wish to.

If this project would have been created from the beginning, the consideration of implementing an Observer Pattern from the beginning would have been ideal. The Event pattern this project started out with was heavy for the system to run, where the Observer pattern is easier, since it only runs when it's needed to, which increases performance of the application. Another example of improving performance was when the world.getEntities(Class<E>) runtime was reduced from O(n) to O(1). Since the way this project was created from the beginning had a heavy workload, general considerations about performance and implementing the module in an effective way would have been prefered from the beginning.

12 References

[1] **Kozhuharov, D. (26. 05 2020).** ***Analysis of Algorithms | Set 1 (Asymptotic Analysis)*****. From GeeksForGeeks: https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/?ref=rp**

[2] **Jai, O. (26. 05 2020).** ***Analysis of Algorithms | Set 3 (Asymptotic Notations)*****. From GeeksForGeeks: https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/**

[3] **Big-O Cheat Sheet. (26. 05 2020).** ***Know Thy Complexities!*** **From Big-O Cheat Sheet: https://www.bigocheatsheet.com/**

[4] **Wong, S. (05. 26 2020).** ***Component-Framework Systems*****. From RICE University: https://www.clear.rice.edu/comp310/JavaResources/frameworks/**

[5] **Elye. (26. 05 2020).** ***Dependency Injection or Service Locator*****. From Medium: https://medium.com/analytics-vidhya/dependency-injection-and-service-locator-4dbe4559a3ba**

[6] **Lord, R. (26. 05 2020).** ***What is an Entity Component System architecture for game development?*** **From Richard Lord's blog: https://www.richardlord.net/blog/ecs/what-is-an-entity-framework**

[7] **Stuart, R., & Norvig, P. (2010). Artificial Intelligence A Modern Approach 3rd Edition. New Jersey: Pearson Education.**