# Semester Project: Intelligent Software Systems

Software Engineering SDU

4th Semester 2022


**Written by**

Cecilie Fredsgaard - Cefre17

Troels Kaldau – Trkal20

Nicolaj Aalykke Hansen - Nicol20

Sandra Malling-Larsen – Small11

Theis Juul Langlands - Thlan20

# 1. Abstract

This report covers the work of developing a 2D Tower Defence game, where a player places towers to shoot enemies attempting to walk from one end of a map to another. The player loses when enough enemies have reached the end of the map.

The main purpose is to use the learning outcome from the courses 'Algorithms and Data structures', 'Artificial Intelligence' and 'Component Based Software Engineering'.

The game is engineered using interface-oriented development and the OSGi framework. It contains the components Map, Core, Player, Enemy, Tower, and Projectile, whereof Enemy, Tower, Projectile, and Player can be loaded and unloaded during runtime without restarting the application.

The report contains descriptions of the implementation of the A* algorithm used for path finding on the map. Specific data structures are considered for efficiently sorting and storing the enemies within a tower's shooting range.

The requirements with the highest priority, containing the basic functionality of a tower defence game, has been tested for their functionality with expected values.

The system design is flexible and maintainable, so further functionality can easily be added in the future to improve the game experience.

# Contents

# 2. Project Description

The aim of this project is to create a classic Tower Defence 2D game described in section 2.1 with the constraints given by the description in section 2.2, and shaped by the team's focus for the project as described in section 2.3.

As this project contains both a player entity moving around the map, as well as a user, who plays the game by key inputs, it has been decided to use the word "player" for the entity in the game, and the word "user" for the person playing the game, to avoid confusion.

## 2.1. Gameplay

The type of game chosen for this project is a Tower Defence game. In a Tower Defence game, a map is defined, usually with a single path running from one edge to another. On this path, enemies are spawned on one end, and walk to the other. The user of the game places towers along the sides of the path, which shoots the enemies when they are within range. The objective is to kill the enemies before they reach the end of the path. As they reach the end, the user loses life points. When all life points are lost, the game is over. The enemies come in waves of increasing numbers, making the game gradually harder, which inevitably leads to the user losing the game.

In this version, the user must move the player entity to a location on the map where they wish to build a tower, instead of placing towers freely, which is more common for the genre.

The player entity dies on contact with the enemies, which makes the user unable to create new towers. Once the player entity is dead, the game runs until the towers already placed cannot hold back the waves of enemies, and life points reach zero. A screenshot from a simple tower defence game used for inspiration can be seen in Figure 1

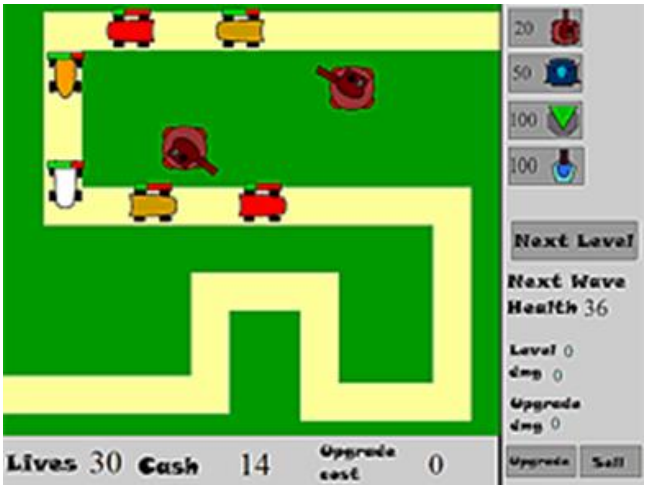


*Figure 1 – mocked example of a Tower Defence game*
*source: https://www.y8.com/games/simple_tower_defense*

## 2.2. Constraints

The following constraints for this project are given by the project description and consist of the following:

- This project must be built using a component-oriented structure, using a Component Framework, with support for multiple ClassLoaders and component versioning.
- It must contain the components Player, Enemy, Weapon, GameEngine, and Map.
- It must be possible to add and remove the Player, Enemy and Weapon components during runtime.
- The Project must also contain an aspect of AI and a description of algorithms and data-structures used in the project.

## 2.3. Project Selection

The purpose of this project is to implement the knowledge and skills obtained through the courses of the 4th semester of Software Engineering at SDU. These are 'Component Base, software Engineering', 'Artificial Intelligence', and 'Algorithms and Data structures.

In selecting the project, the focus is to keep the gameplay itself simple, utilizing the aspects described in the project limitations in section 2.2

The priority is to work with the functionality of components, rather than advanced graphics, sound, and gameplay. It should be possible to work in an agile way, implementing functionality gradually, with focus on ease of extension and meeting the requirements.

# 3. Requirements

The requirements are derived from the project description in section 2, covering the gameplay and project constraints. They are categorized into non-functional and functional requirements.

## 3.1. Non-functional requirements

The non-functional requirements contain the design constraints described in section 2.2, and can be found in Table 1 below.

*Table 1 – non-functional requirements*

| ID | Category | Non-functional requirement |
|---|---|---|
| NF1 | Environment | Must use component framework that can load and unload components during runtime, OSGi or NetBeans |
| NF2 | Design | Must contain well defined and defended data structures |
| NF3 | Design | Must contain algorithms |
| NF4 | Design | Must use algorithms for Artificial intelligence |

## 3.2. Functional requirements

The functional requirements cover the basic gameplay as described in section 2.1. They are prioritized using the MoSCoW model, with the must-haves representing the minimum viable product. All must-have requirements are tested and verified, see section 7.

*Table 2 - functional requirements*

| ID | Functional requirements | MoSCoW | Test# |
|---|---|---|---|
| F1 | Must be possible to dynamically load/unload Player, Enemy, Weapon | M | 7.1 |
| F2 | Must contain components Player, Weapon, Enemy, Map, Core | M | - |
| F3 | Player | | |
| F3.1 | A player must be able to move all over the map | M | 7.3 |
| F3.2 | A player must be able to build towers | M | 7.4 |
| F3.3 | A player could be able to repair a tower | C | - |
| F3.4 | A player could be able shoot at enemies | C | - |
| F3.5 | A player must have a health system | M | 7.6 |

| | | | |
|---|---|---|---|
| F4 | Enemy | | |
| F4.1 | An Enemy must be able to move through the path on the map using AI | M | 7.5 |
| F4.2 | An Enemy must have a health system | M | 7.6 |
| F4.3 | An Enemy could be able to shoot at the player | C | - |
| F4.4 | An Enemy might be able to shoot at a tower | W | - |
| F4.5 | An Enemy should have a walking speed | S | - |
| F4.6 | An EnemySystem could decide which enemies to add to the map | C | - |
| F5 | Tower | | |
| F5.1 | A Tower must be able to shoot at enemies | M | 7.7 |
| F5.2 | A Tower might be able to use different kinds of weapons | W | - |
| F5.3 | A Tower might be able to have a shooting speed | W | - |
| F5.4 | A Tower could have a health system | C | - |
| F5.5 | A Tower should be able to choose the correct enemy as a target | S | - |
| F6 | Map | | |
| F6.1 | A Map must have a fixed path through the board | M | 7.8 |
| F6.2 | A Map could be generated dynamically, so that it is different for every game | C | - |
| F7 | Collision | | |
| F7.1 | Collisions between player, enemies and projectiles must be detected and handled | M | 7.9 |
| F8 | Projectile | | |
| F8.1 | A Projectile must be able to be generated | M | 7.10 |
| F8.2 | A Projectile must be able to move in a straight line with a defined speed | M | 7.11 |
| F8.3 | A Projectile must die when it reaches the end of the map | M | 7.12 |
| F9 | Core (GameEngine) | | |
| F9.1 | A core must initialize and run the game | M | 7.13 |
| F9.2 | There could be a menu at start-up | C | - |
| F9.3 | It might be possible to change game settings in the menu | W | - |

# 4. Analysis

In this section, the functionality of the individual components composing the game will be analysed. The required interfaces for controlling and communication between components are specified, and the implementation of artificial intelligence is analysed.

## 4.1. Component analysis

The game will consist of the following components derived from the requirements: Player, Enemy, Tower, Map, Collision, Projectile, Core, and a Common component for shared common global attributes and interfaces. They will be described in the following sections.

### 4.1.1. Core

The Core component is the GameEngine responsible for creating the game, controlling the game flow, and ending the game.

Initially a map is loaded, relevant entities are added to the game, and the game data is set, e.g., life, score, and money. During the gameplay, the Core will handle user input, generate attacks, update entities, and the screen will continuously render with updated entity positions. After each update the Core will check the consequences, e.g., collision between entities, update of scores, and check if the game has ended. When the game ends, the Core will stop any active processes and prompt the user to start a new game. The flow can be seen in the diagram in Figure 2.

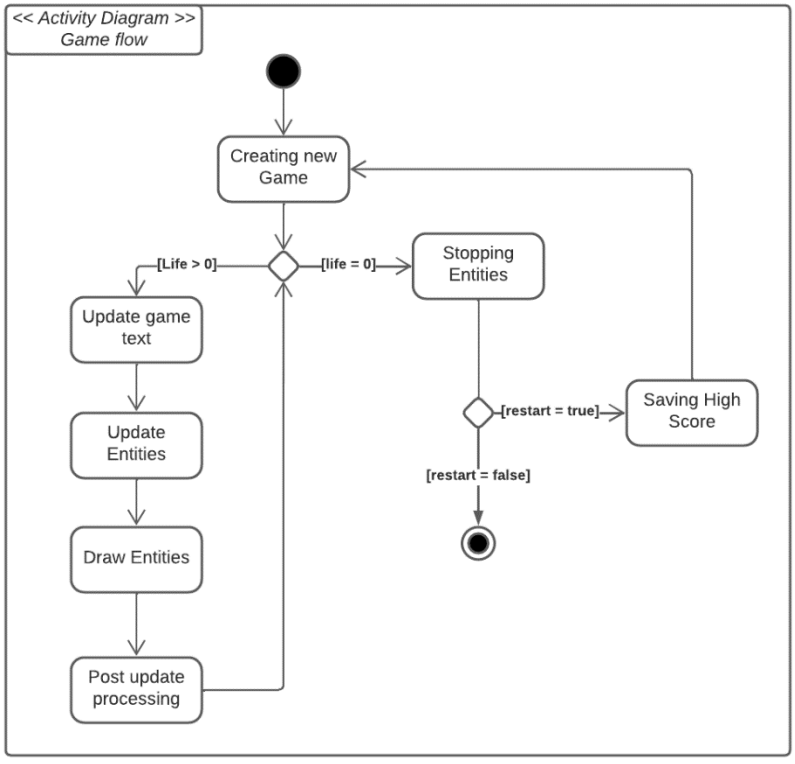


*Figure 2 - Game flow activity diagram*

### 4.1.2. Map

The map is built from square tiles of a fixed size, organized into a coordinate system with standard x- and y-axes. Each tile will have a property used to divide them into types: start, end, path, grass, or tower. Each tile property will have its own image, to allow a user to visually differentiate between them.

### 4.1.3. Tower

The Tower is a non-movable entity which can be placed by a player on grass tiles on the map. It can shoot projectiles at enemies within its range.

The tower will select the ideal enemy to target based on four criteria; the enemy's straight-line distance to the end- and start tiles, the health of the enemy and the distance from the enemy to the tower. An evaluation function calculates a value based on these criteria, with each criterion being multiplied by a weight to balance their importance. The distance to the end tile and the enemy’s remaining health is deemed the most important, so they will have the highest contribution to the decision. The tower then selects an enemy by comparing the values of the enemies within its attack range and targets the enemy with the lowest value.

### 4.1.4. Projectile

The projectile will be a moving entity created by towers when they wish to shoot at enemies. It will move in a set linear direction with a specified speed and range.

The projectile will travel to the edge of the map where it terminates, unless it has already been destroyed by hitting an enemy or reaching the end of the projectile's range.

### 4.1.5. Enemy

The enemies will be movable entities, which move along the path tiles on the map. The path is detected based on the layout of the map. There will be a health counter, which decreases when hit by projectiles. The enemy will search for an end tile, and once it is reached, the enemy will be removed from the world, and do damage to the user by subtracting life points.

### 4.1.6. Player

The player will be a move- and controllable entity, which will move according to the input given by the user. The player will be able to place towers on tiles and can move across the entire map. Furthermore, it will take damage if intercepted by enemies.

### 4.1.7. Common

The common module will contain functionality shared between all the components and information about the game, including commonly used entities and interfaces present in the game, as well as information about the game session, such as the life and score of the user.

## 4.2. Interface analysis

For the components to have independent implementation, communication between them is done through interfaces. This makes it possible to replace one component with another if the requirements of the interface contracts are met. This makes the system flexible and easy to extend or modify, if e.g., another type of enemy or tower is wanted. The interfaces are derived from the component analysis and the requirements.

An interface will need to be implemented to make it possible to start and stop components, as derived from requirement F1.

Interfaces for updating entities and processing what happens after an update during gameplay is needed, e.g., for moving the entities and handling collisions between them.

A player needs to be able to build towers, and a tower needs to create and shoot projectiles, so interfaces for creating towers and projectiles are needed.

Most components will need to access the map and its methods, which an interface is needed for. A table of the required interfaces is seen in Table 3 with name, description and which components provides the service.

*Table 3 - List of necessary interfaces*

| ID | Name | Description | Component providers |
|---|---|---|---|
| IF1 | IGamePluginService | Start and stop components | Tower, Projectile, Enemy, Player, Map |
| IF2 | IEntityProcessingService | Update entities during gameplay | Tower, Projectile, Enemy, Player |
| IF3 | IPostEntityProcessingService | Handle consequences of an update | Collision |
| IF4 | TowerSPI | Create towers | Tower |
| IF5 | ProjectileSPI | Create Projectile | Projectile |
| IF6 | IMap | Expose map functionality | Map |

An illustration of the interfaces required and provided by the components in the system can be seen in Figure 3.

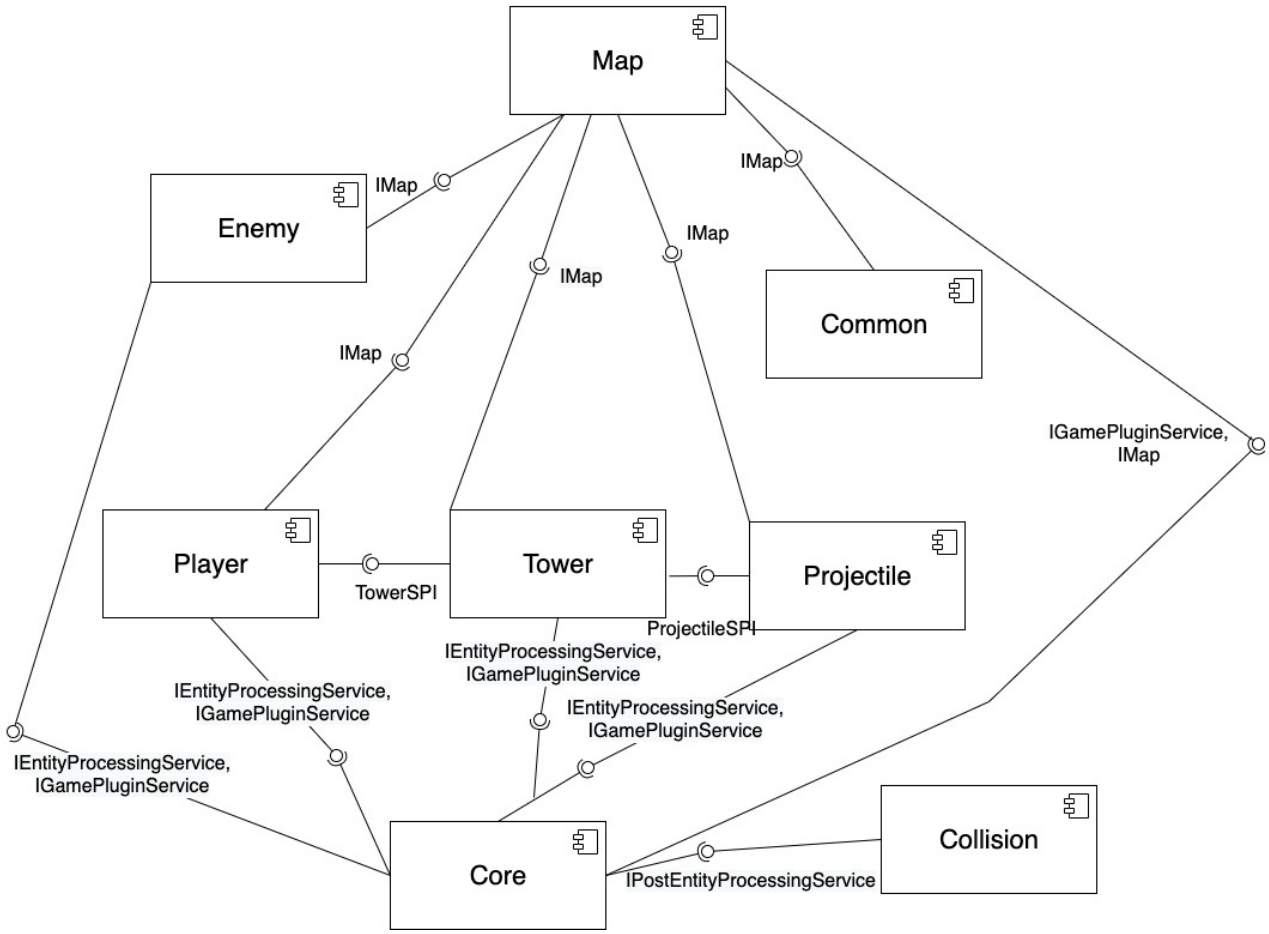


*Figure 3 - Diagram showing the required and proved interfaces for the different components*

Lollipop notation is used to show which interfaces are required and provided by the components, e.g., it is visible that the map component is a service provider of the IMap interface, and it is used by all other components except Collision. Likewise, the TowerSPI interface is required by the Player and is provided by the Tower. Where two interfaces are specified, two interfaces are provided and required. E.g., the Player component provides two interfaces: IEntityProcessingService and IGamePluginService, and the Core component requires both from the Player.

## 4.3. Artificial Intelligence

To uphold requirement F4.1, there must be AI in the path finding of enemies. This means that the algorithm should calculate the path that allows the enemies to walk from the start tile to the end tile of the map. To assess which algorithm to use, performance and completeness is investigated. The path is finite, which means there is an end and a start.

### 4.3.1. Performance

A* is most commonly the first algorithm considered when looking at performance. This is because A* is considerably faster than, for instance, Depth-first Search and Breadth-first Search. There are only very few cases in which they will outperform A*. This occurs in very few cases, for example, when the first node in each level of the search tree is always part of the path to the goal; then depth is the best.

The easiest way to compare the algorithms is to look at their Big-O-Notation.

*Table 4 - Comparison between different search methods [1]*

| Algorithm | Big-O | Completeness |
|---|---|---|
| Depth-first-search | $O(b^m)$ | Yes, given the path is finite |
| Breadth-first-search | $O(b^d)$ | Yes |
| A*-search | Number of nodes for which $f(n) \leq C^*(exponential)$ | Yes |
| b: maximum branching factor of the search tree<br>d: depth of the optimal solution<br>m: maximum length of any path in the state space<br>C*: cost of optimal solution | | |

All the algorithms under consideration are complete. Depth-first and breadth-first can grow very fast, but so can A*. In most cases, A* will be much faster than the other two, but it very much depends on its heuristic functions. A* can go as low as $\Theta(1)$ in very rare cases. Since the number of heuristic functions and nodes are very limited, A* would be the optimal choice.

# 5. Design

This section will cover design considerations when building the system including:

- Selection of a component framework
- Interface and component design
- Interaction between components
- Design of artificial intelligence.

Since the project is composed of components, a component framework needs to be selected that covers the non-functional requirement NF1, see section 3.1, of being able to load and unload components at runtime, which is covered in section 5.1

The system is build using interface-oriented design, with the components interacting through interfaces as mentioned in section 4.2. Therefore, it is important to have strong and well-defined interface contracts, see section 5.2. The concrete implementation of a component can be changed as long as it fulfills the terms of the interface contract. Therefore, the focus of design will be on those contracts.

The design and interaction between the components found in section 4.1 is described in section 5.3, 5.4 and 5.6, and the design of the AI path-finding algorithm can be seen in section 5.5

## 5.1. Selection of Component Framework

To use the component-based architecture, a component-framework must be selected. The decision lies between the NetBeans module system and OSGi module system. In this project the OSGi module system will be used, as it is the de facto industry standard for dynamic loading and unloading [2], and therefore more relevant from a learning perspective. Component lookup and dynamic loading and unloading of modules is more intuitive in OSGi because of dependency injection, which is seen as the best choice since implementation is simpler. Dependency injection is highly supported in OSGi, which is another reason why OSGi is the chosen framework for this project.

## 5.2. Interface Contracts

Having strong and well-defined contracts before implementation is important since it prevents the need for changing the interface during development. Changes could potentially ruin a system if it depends on specific functionality defined in interfaces. Having a stable and well-defined contract prevents this, since every component complies with the interface. In general, it is allowed to add more functionality to an interface, but deleting functionality should be avoided.

The interfaces described in section 4.2 are specified with methods and system operation contracts for each method it contains. The following chapter details the interfaces used in this project.

### 5.2.1. IGamePluginService

The IGamePluginService is used for adding and removing components to and from the game using the methods *start* and *stop*. See description in Table 5.

*Table 5 - Interface contract for IGamePluginService*

| IF1: IGamePluginService | |
|---|---|
| Operation | *start*(GameData gameData, World world) |
| Description | Adds elements to the game. |
| Parameters | gameData: global information about the game.<br>world: list of all entities in the game |
| Preconditions | A game has been created with gameData and world |
| Postconditions | An entity has been added to the world |
| Operation | *stop*(GameData gameData, World world) |
| Description | Removing elements from the game. |
| Parameters | gameData: global information about the game.<br>world: list of all entities in the game |
| Preconditions | A game has been created with gameData and world |
| Postconditions | An entity has been removed from the world |

### 5.2.2. IEntityProcessingService

Offers functionality to update entities during gameplay and consists of a *process* operation, which updates fields in entities, and a draw method, which draws the entity on the screen. See full description in Table 6.

*Table 6 - Interface contract for IEntityProcessingService*

| IF2: IEntityProcessingService | |
|---|---|
| Operation | *process*(GameData gameData, World world) |
| Description | controlling elements added to the game, e.g., movement of player and enemy. |
| Parameters | gameData: global information about the game.<br>world: list of all entities in the game |
| Preconditions | - |
| Postconditions | Entities has been processed and updated |
| Operation | *draw*(Image image, World world) |
| Description | updating the entity image placement on the map |
| Parameters | image - image representing the entity.<br>world - list of all entities in the game |
| Preconditions | - |
| Postconditions | Image has drawn on screen with correct position |

### 5.2.3. IPostProcessingService

Handles consequences of an update and should be called after updating entities using the IEntityProcessingService. E.g., used for handling collision between entities. The details can be found in Table 7 below.

*Table 7 - Interface contract for IPostEntityProcessingService*

| IF3: IPostEntityProcessingService | |
|---|---|
| Operation | *process*(GameData gameData, World world) |
| Description | Handle consequences of an update |
| Parameters | gameData: global information about the game.<br>world: list of all entities in the game |
| Preconditions | A game has been created with gameData and world<br>The IEntityProcessingService has been processed |
| Postconditions | The entities have been further processed in the world |

### 5.2.4. TowerSPI

The TowerSPI is used for creating a tower at a specific tile if the position adheres to the specified requirements. Full detail can be found in Table 8 below.

*Table 8 - Interface contract for TowerSPI*

| IF4: TowerSPI | |
|---|---|
| Operation | *createTower*(GameData gameData, World world, int x, int y) |
| Description | verify that a tower can be built at the given location and create a tower. |
| Parameters | gameData: global information about the game.<br>world: list of all entities in the game<br>x,y: tile coordinates where tower is placed |
| Preconditions | The gamedata and world can be accessed. |
| Postconditions | A tower has been created from the location and returned. |

### 5.2.5. ProjectileSPI

The ProjectileSPI is used for creating a projectile with the position and direction of the shooter, and the speed defined by the weapon of the shooter. See Table 9 for further details.

*Table 9 - Interface contract for ProjectileSPI*

| IF5: ProjectileSPI | |
|---|---|
| Operation | *createProjectile*(Entity shooter, GameData gameData, World world) |
| Description | Responsible for creating bullets when fired by an entity. |
| Parameters | shooter: entity who creates the projectile<br>gameData: global information about the game.<br>world: list of all entities in the game |
| Preconditions | A game must have been initialized.<br>The shooter entity must contain position coordinates, a direction of movement and a weapon description defining the speed of the created projectiles. |
| Postconditions | A projectile has been created |

### 5.2.6. IMap

Provides a map and methods for accessing its properties as listed in Table 10. A full specification can be found in the interface contact in Appendix A .

*Table 10 - Resumé of the IMap interface contract*

| IMap methods - resumé |
| --- |
| get basic map properties, e.g height, width, tilesize |
| verify if a point is inside the map |
| get a list of the coordinates containing the path |
| change between tile coordinates and map coordinates |
| change the tile type |
| get coordinates of start and end tile |
| get the properties for a specific tile, e.g., grass, tower. |

## 5.3. Component Design

In this section the design of all components will be described including the concrete design of how methods described in the interface contracts above will function in the individual components

### 5.3.1. Core

The Core components main responsibility is controlling the game flow, see Figure 2 in section 4.1.1.

First a game needs to be initialized, setting global variables in the GameData class, e.g., life, money, and score, as well as loading a map, loading fonts, sprites and creating entities.

The game loop starts; updating game text, updating the entities by calling the *process* method on all classes implementing IEntityProcessingService, followed by the *draw* method to update the screen. During the game the score will be saved as a new high score, provided it has surpassed the previous high score. Finally, the *process* method on classes implementing IPostEntityService is called. This loop continues while the user still has life left, and once the user runs out of life, it is possible to restart the game.

The Core component consists of the classes Game, which is responsible for initializing and updating the Game, and a helper class GameInputProcessor, which registers input from the keyboard. The class specifications for Game and GameInputProcessor are found in Table 11 and Table 12 respectively.

*Table 11 - Class specification for Game class*

| <<class>><br>Game | |
|---|---|
| Purpose | Responsible for initializing the game and updating the game logic. |
| Functionalities | Initializing the game<br>Configuring the graphics and fonts<br>Setting the initial GameData, e.g., life, money etc.<br>Configuring attack rate<br>Calls *start* method on IGamePluginService<br>Update during gameplay<br>Drawing text<br>Update and draw entities<br>Re-rendering the screen |
| SPI's | - |

*Table 12 - Class specification for GameInputProcessor class*

| <<class>><br>GameInputProcessor | |
|---|---|
| Purpose | The input class is responsible for registering user inputs from the keyboard. |
| Functionalities | The class provides functions for detecting whether specified keys are down or up. |
| SPI's | - |

### 5.3.2. Tower

The Tower component is responsible for creating and removing tower entities in the game, as well as detecting enemies within its range, and prioritizing which enemies to shoot at. It consists of two classes, TowerControlSystem and TowerPlugin.

The TowerControlSystem class is responsible for targeting enemies and creating towers, see class specification in Table 13.

*Table 13 - Class specification for TowerControlSystem*

| << class >> TowerControlSystem | |
|---|---|
| Purpose | Create tower and shoot projectiles at enemies |
| Functionalities | Shoot enemies<br>Detecting and selecting enemy to target<br>Create projectile through ProjectileSPI.<br>Creating tower entities<br>verify if placed on valid location<br>Create new towers |
| SPI's | IEntityProcessingService, TowerSPI |

The *createTower* method is implementing the method from the TowerSPI interface. It verifies that the tower can be built on the specified location and creates a new tower entity.

The *process* method of IEntityProcessingService controls the behavior of a tower. It checks whether there are enemies within its range. The reachable enemies are stored using a min-heap; the reachable enemy entities are sorted after the value of their heuristics as described in section 4.1.3.
This will keep the enemy with the lowest value as root of the search tree, so that it is possible to retrieve the enemy from the heap in $O(1)$ running time. Adding an enemy to the min-heap can be done in $O(log\ n)$ time [3]. Alternatively, this could be implemented with a for loop iterating over the list of enemies, keeping track of the enemy with the lowest heuristic value, but this solution would have a running time of $O(n)$. Seeing the min-heap allows for faster search for best enemy as well as faster addition of enemies, this approach was chosen instead of the for loop.
If an enemy is selected, the angle between the tower and enemy is calculated, the tower is rotated to point at the enemy, and a projectile is created using the ProjectileSPI.
A sequence diagram depicting the *process* method can be found below in Figure 4.

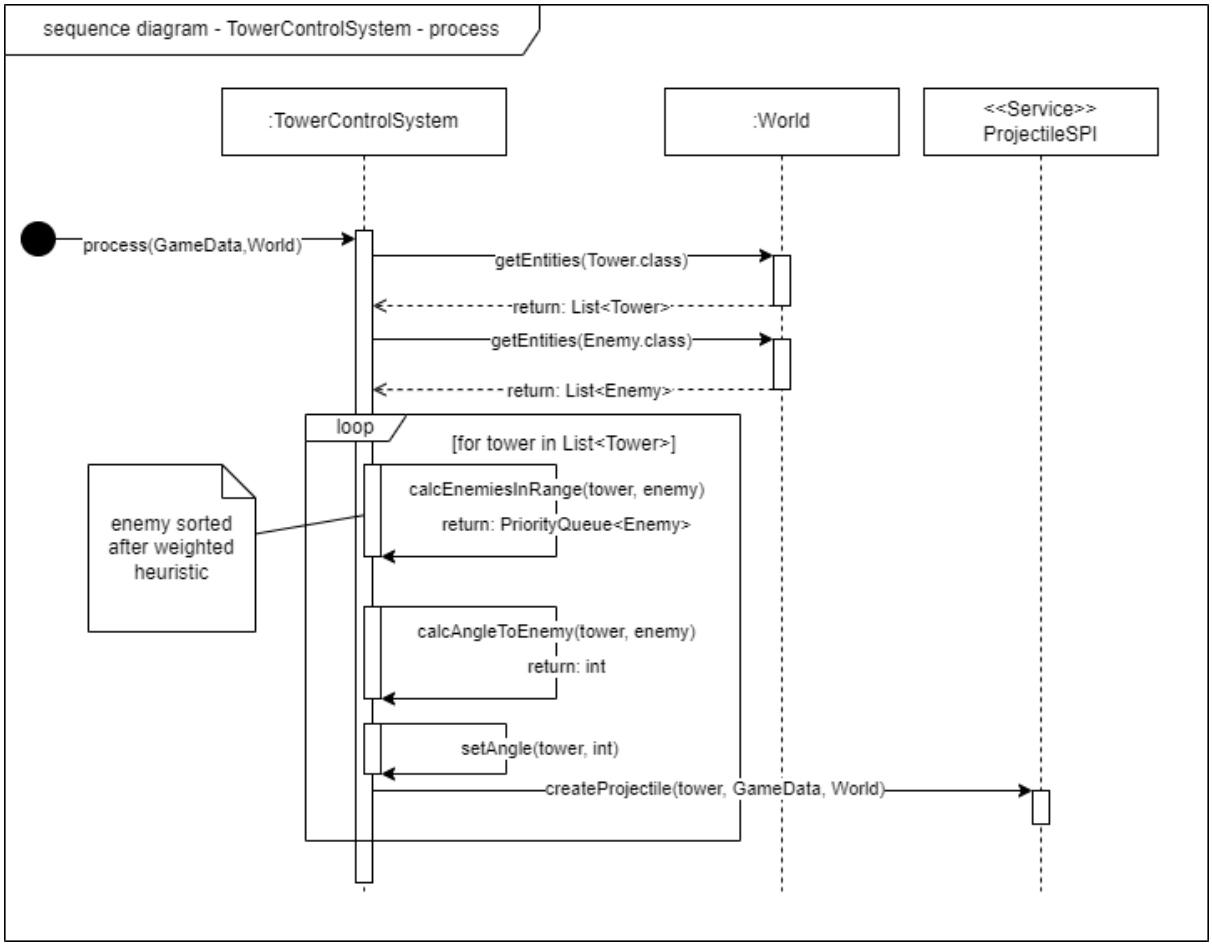


*Figure 4 - Sequence diagram for process method in TowerControlSystem*

To remove tower components from the game, the TowerPlugin class is used. See class specification in Table 14.

*Table 14 - Class specification for TowerPlugin*

| << class >><br>TowerPlugin | |
|---|---|
| Purpose | Removing all towers when components stopped. |
| Functionalities | Provides methods for stopping the component. |
| SPI's | IGamePluginService. |

### 5.3.3. Player

The Player component is responsible for moving the Player entity on the map, based on the user's input, and requesting tower builds. The Player component contains two classes; PlayerControlSystem, found in Table 15, and PlayerPlugin, found in Table 16.

*Table 15 - Class specification for PlayerControlSystem*

| <<class>><br>PlayerControlSystem | |
|---|---|
| Purpose | Responsible for updating the player during gameplay. |
| Functionalities | Provides a method for updating the Player component. |
| SPI's | IEntityProcessingService |

*Table 16 - Class specification for PlayerPlugin*

| <<class>><br>PlayerPlugin | |
|---|---|
| Purpose | Responsible for creating the player on start-up and removing the player on shutdown. |
| Functionalities | Provides methods for starting and stopping the Player component |
| SPI's | IGamePluginService |

The PlayerControlSystem implements one method: *process*. This method is used to execute all the responsibilities of the Player component. This means updating the Player values according to the peripheral input, as well as building towers and updating the general game data accordingly. A step-by-step overview of the *process* method can be seen in the sequence diagram in Figure 5.

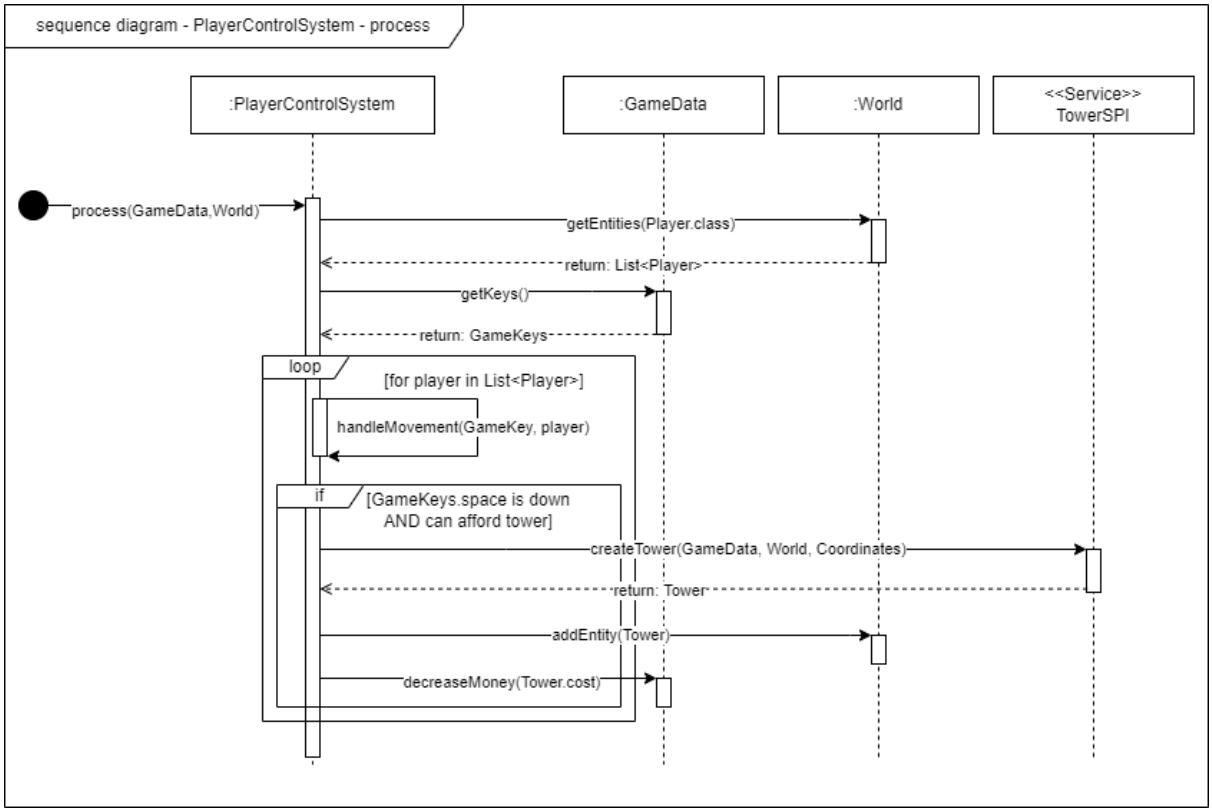


*Figure 5 - Sequence diagram for the process method in PlayerControlSystem*

### 5.3.4. Enemy

The Enemy component is responsible for instantiating enemies, guiding them through the map, and destroying them. It is also responsible for updating the common GameData as a result of the enemy's creation, movement, and destruction. The class specifications for EnemyPlugin and EnemyControlSystem can be found in Table 17 and Table 18 respectively.

*Table 17 - Class specification for EnemyPlugin*

| <<class>><br>EnemyPlugin | |
|---|---|
| Purpose | Responsible for handling the start and stop of the Enemy component, cleaning up any resources used, and updating any game data relevant to the stopping of the module. |
| Functionalities | Provides methods for starting and stopping the Enemy component |
| SPI's | IGamePluginService |

*Table 18 - Class specification EnemyControlSystem*

| <<class>><br>EnemyControlSystem | |
|---|---|
| Purpose | Responsible for processing any changes to the enemies, updating them accordingly, and finding the way through the map. |
| Functionalities | Provides a method for updating the Enemy component. |
| SPI's | IEntityProcessingService |

The EnemyControlSystem implements only one method: *process*. This method is used to execute all the responsibilities of the Enemy module. This includes instantiating enemies, updating their properties, and removing them if certain conditions are met. The sequence diagram for the *process* method can be seen on Figure 6.

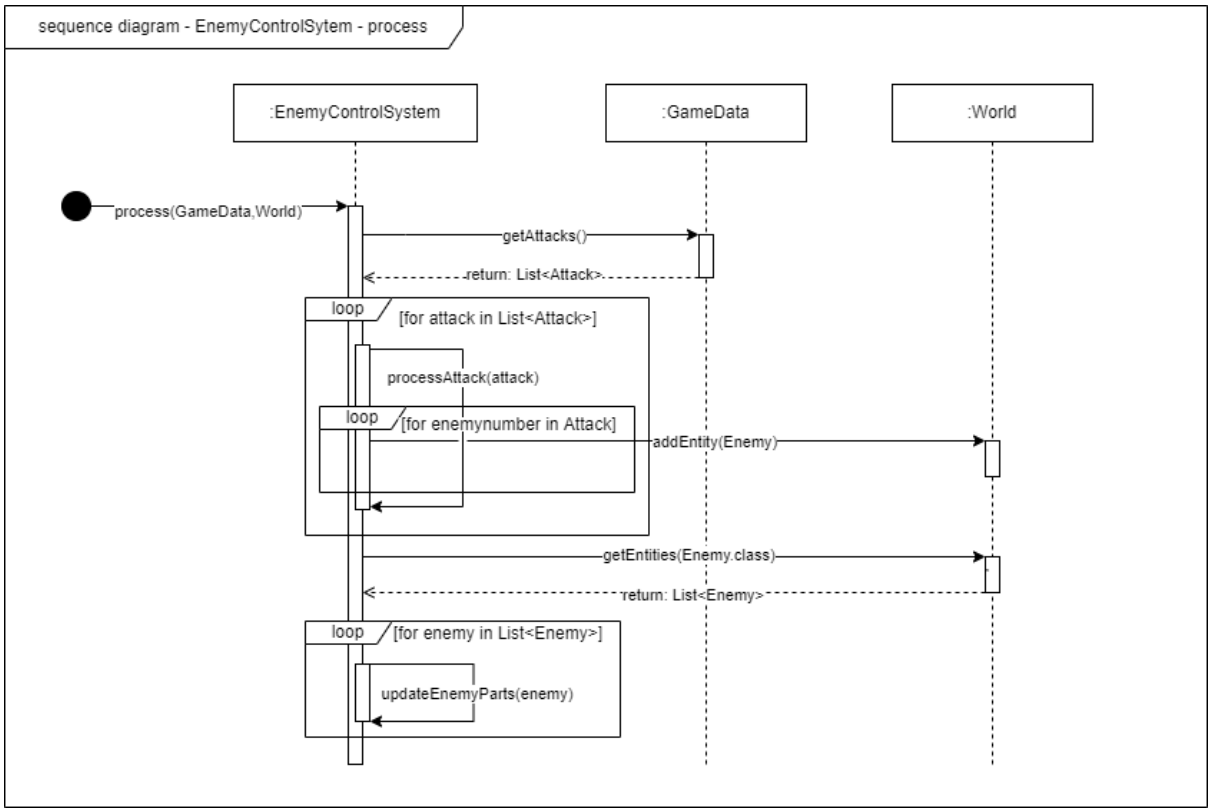


*Figure 6 - Sequence diagram for process enemies*

### 5.3.5. Collision

The Collision component is responsible for detecting collisions between different entities, subtracting lives from the appropriate entities, and removing dead entities from the world, see class specification in Table 19.

*Table 19 - Class specification for CollisionManager*

| <<class>><br>CollisionManager | |
|---|---|
| Purpose | Responsible for detecting all collisions and damaging the lifepart of the affected entities. |
| Functionalities | Provides a method for detecting collisions, handle consequences, and removing dead entities from the world. |
| SPI's | IPostEntityProcessingService |

To determine whether entities collide, circle collision is used [4]. The distance between enemies is calculated using Pythagoras and compared to the sum of both entities' radii. If the distance is less than this value, the entities are colliding as seen in Figure 7 - detecting collision using circle collision.

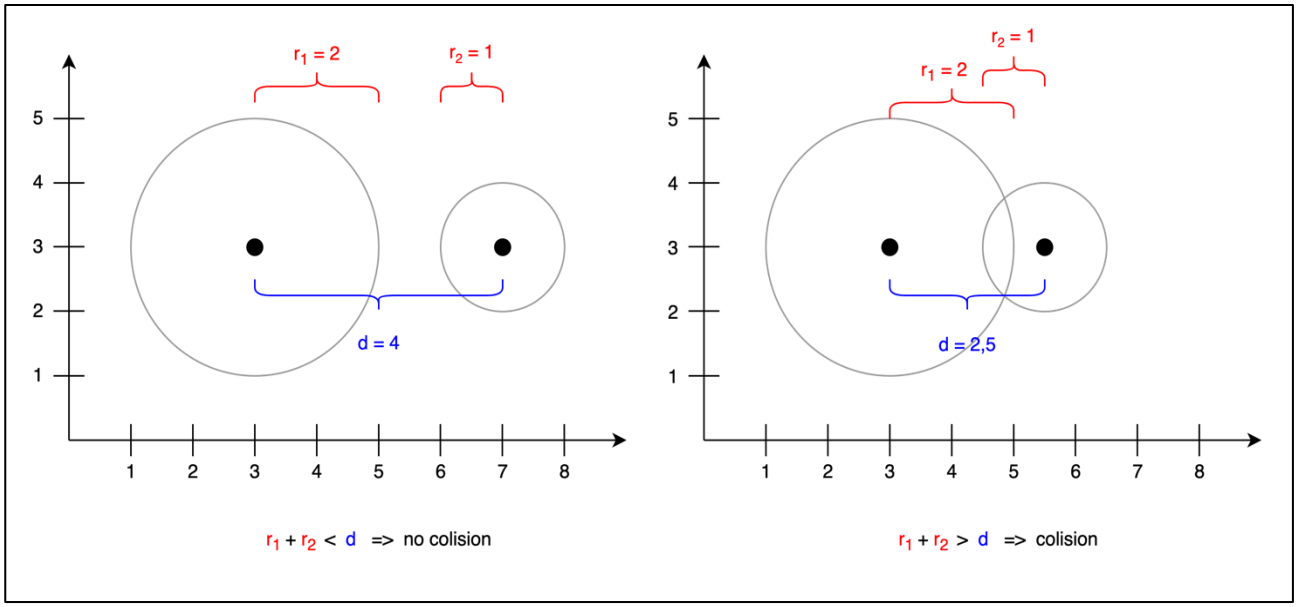


*Figure 7 - detecting collision using circle collision*

If a collision is detected, entities are damaged and life should be subtracted from the entities, depending on different conditions. E.g., the player should not be damaged by a tower, enemies do not affect each other when colliding, and nothing should collide with towers.

If an entity dies after having a life point deducted, different actions happen depending on which entity it is. If a projectile collides with anything or an enemy is dead, it is removed from the world and the score updated accordingly. If the player dies, a marker should be set, which is to be handled in the game loop.

### 5.3.6. Projectile

The Projectile component is responsible for instantiating projectiles an controlling their movement during gameplay. It contains two classes ProjectilePlugin and ProjectileControlSystem.

The ProjectilePlugin class is responsible for removing projectiles from the world, see specification in Table 20.

*Table 20 - Class specification for ProjectilePlugin*

| <<class>><br>ProjectilePlugin | |
|---|---|
| Purpose | Responsible for removing projectiles when stopped. |
| Functionalities | Provides methods for removing the Projectile component |
| SPI's | IGamePluginService |

The ProjectileControlSystem handles updates of projectiles during gameplay, removes projectiles if the exceed their range or flying outside the map and is responsible for creating a new projectile, with properties defined by the shooters weapon.

*Table 21 - Class specification for ProjectileControlSystem*

| <<class>><br>ProjectileControlSystem | |
|---|---|
| Purpose | Handles update of projectiles during gameplay and creates new projectiles |
| Functionalities | Provides a method for updating and drawing the projectiles, and a method to create projectiles when requested |
| SPI's | IEntityProcessingService, ProjectileSPI |

### 5.3.7. Common

The Common component contains common interfaces, and common POJO's, which most other game components depend on.

The common interfaces include IEntityProcessingService, IPostEntityProcessingService, and IGamePluginService described in section 4.2 Interface analysis.

The POJO classes represent custom data types; the most important being the World and GameData classes. The World class contains a map of entities added to the game, and a map of textures representing entities in the game, see Table 22.

*Table 22 - Class specification for World class*

| <<class>><br>World | |
|---|---|
| Purpose | Contains a list of active entities added to the game |
| Functionalities | Getters and setters |

The GameData class holds global game variables, e.g., score, health, screen messages, gamekeys, attacks etc., see Table 23.

*Table 23 - Class specification for GameData class*

| << class >><br>GameData | |
|---|---|
| Purpose | Contains information of the active game |
| Functionalities | Getters and setters |

The Attack class holds information about attacks, when to launch the attack compared to the game's start time and how many enemies should be deployed in an attack, see Table 24

*Table 24 - Class Specification for Attack class*

| << class >><br>Attack | |
|---|---|
| Purpose | Contains information about an attack |
| Functionalities | Getters and setters |

The PathDirection class contains information relating to the way in which the EnemyControlSystem moves on the map. It contains a direction to move and a goal point where the direction should change., see Table 25.

*Table 25 - Class specification for PathDirection class*

| << class >><br>PathDirection | |
|---|---|
| Purpose | Used by the entities to find direction |
| Functionalities | Getters and setters |

The entities are constructed using object composition with the Entity class as blueprint for all entities in the game see Table 26.

*Table 26 - Class specification for Entity class*

| << class >><br>Entity | |
|---|---|
| Purpose | Contains common information of the entity, and a list of entity parts containing relevant attributes |
| Functionalities | Getters and setters |

An EntityPart contains attributes and a *process* method if relevant. These are added to an Entity according to its functionality. Each entity part implements the EntityPart interface.
Table 27 contains a list of classes implementing the EntityPart interface, and a description of the content. E.g., the LifePart contains the life attribute of an entity, and a MovingPart handles the movement of entities.

*Table 27 - List of EntityPart classes*

| << class >><br>EntityPart | Purpose |
|---|---|
| LifePart | Contains life of an entity |
| MovingPart | Entity movement and speed<br>Process method updates PositionPart |
| PathPart | Information about the path the entity is to follow |
| PositionPart | Contains information of the entities position and orientation |
| WeaponPart | Contains information of the weapons range and speed of projectiles |

### 5.3.8. LibGDX

A component must be created that contains the LibGDX-packages, which is the graphical-user-interface used in the project. LibGDX is intuitive and well documented and used in this semester. To represent the entities etc. as images, sprite**,** and textures are used.

The texture decodes an image file and thereafter it is loaded into the GPU memory. I.e., it takes an image and transforms it to be drawable.

The sprite wraps around a texture, and defines where it is going to be drawn, its position, etc.

To draw a sprite the batch must be used. All drawing in LibGDX must occur between a *start* and *end* call on the batch object. [5]

The LibGDX TiledMap will also be used; it is composed of layers which have tiles.

### 5.3.9. Map

Each tile on the map will have a code representing the content of the field. Each field can be either grass, path, tower, start tile or end tile, and will affect the properties of the game, see description in Table 28.

*Table 28 - Description of tile value codes*

| Code | Represents | Description |
|---|---|---|
| 2 | End Tile | Tile where enemies leave the map; health decreases |
| 3 | Grass | Basic field on the map, that the player can move on |
| 4 | Path | The path on the map, where the enemies and player can move |
| 5 | Start Tile | Tile where enemies enter the map |
| 6 | Tower | A field containing a tower |

The map will be represented in a two-dimensional array, with each cell containing the codes of each tile. When drawing the map, it will be built by individual images according to the cell.

This ensures flexibility so it's possible to give the map a new visual identity by changing the images, without altering the code. Furthermore, the size of the map can be adjusted by changing the dimension of the array. An example of a 7x7 map consisting of tiles can be seen in Figure 8.

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 3 | 3 | 3 | 3 | 6 | 3 |
| 3 | 4 | 4 | 4 | 4 | 4 | 5 |
| 3 | 4 | 6 | 3 | 3 | 3 | 3 |
| 3 | 4 | 4 | 4 | 3 | 3 | 3 |
| 3 | 3 | 6 | 4 | 3 | 3 | 3 |
| 2 | 4 | 4 | 4 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 |

*Figure 8 - 7x7 map with field, path, tower, start and end tiles*

An example of a 12 x 12 map containing field, path, end and start tiles, and towers generated with tiles can be seen in Figure 9. To the right of the map is dedicated space for showing score, health, and money.

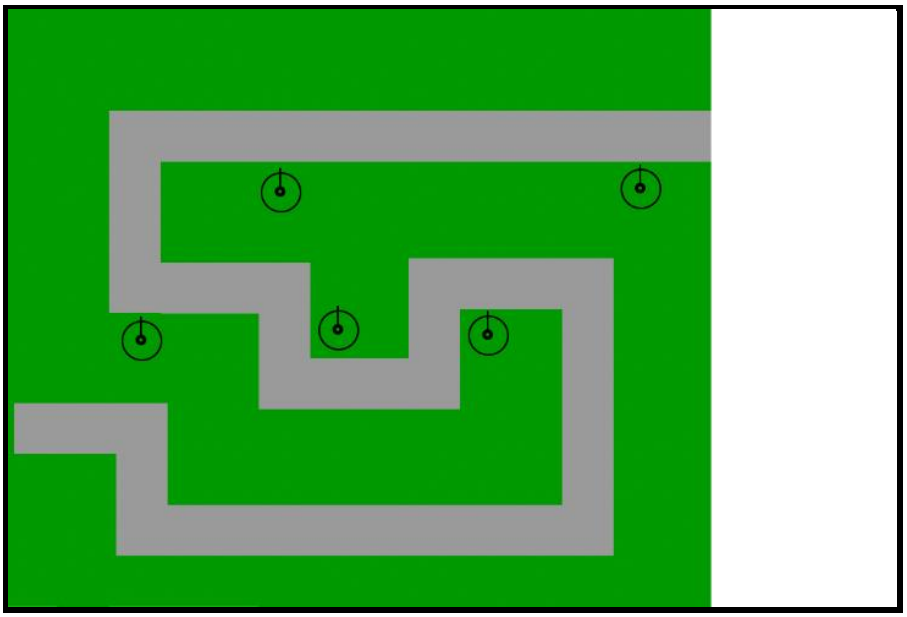

*Figure 9 - 12x12 map*

## 5.4. Interaction between Components

A component diagram visualizing the dependencies between the components can be seen in Figure 10. The arrows indicates a dependency on the other component. For example, Common Map depends on LibGDX.

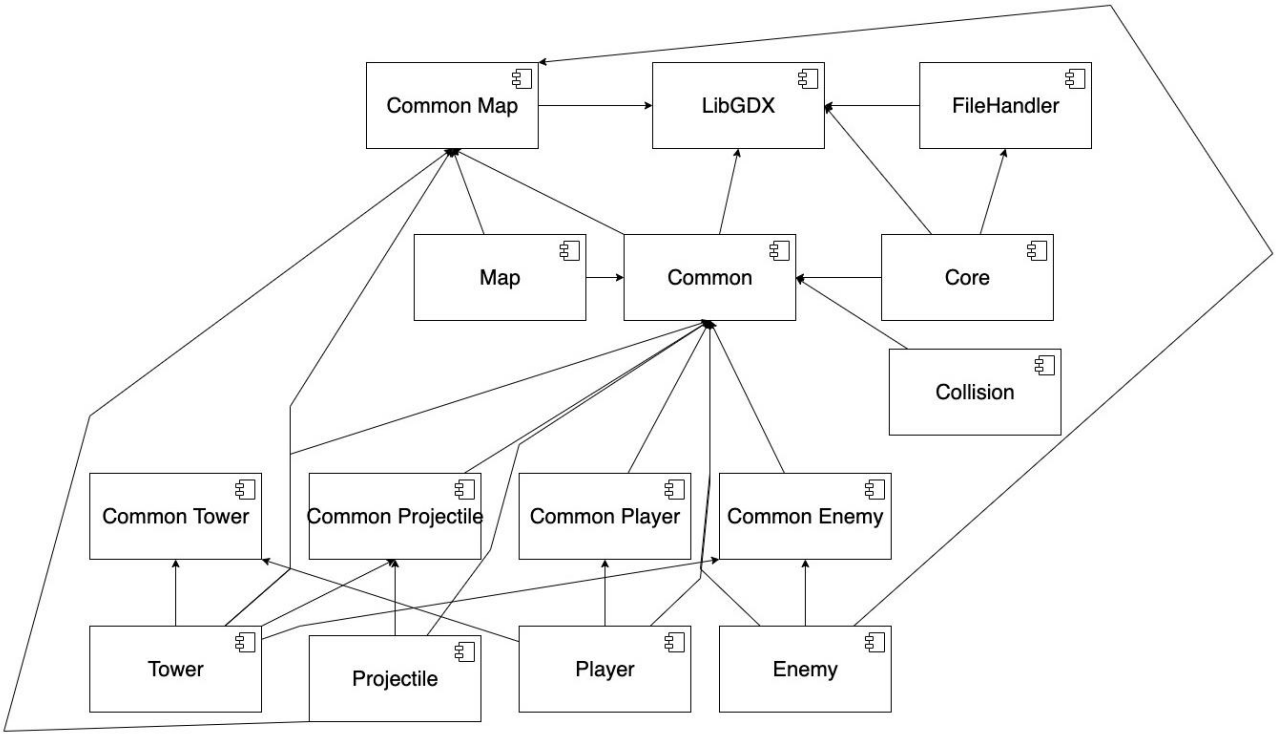


*Figure 10 - Component dependencies*

Common is the basis of most the components. It has the classes that most components need access to. I.e., most components should have a dependency on this component.

## 5.5. Artificial Intelligence

Based on the analysis of different algorithms, A* is seen as the most optimal, and thereby it is used in the project. A* is a variation of the Dijkstra algorithm with a heuristic function. The simplest heuristic, and used in this project, is the straight-line distance heuristic, which calculates the length of a straight line from current state to goal.

The goal is the tile defined in the map with the type of End. First step is therefore to get the end tile. Next, the start of the search must be found, which is defined in the map as a tile with the type Start. The Start tile is added to a list containing all unsearched tiles, now called fringe. While the fringe is not empty, the tile with the lowest summed value, of the distance to the goal and the path distance, is selected and searched.

The first step in the search is checking the type of the tile. If the type is other than Path, End or Start, the tile is moved from the fringe to a list of already searched tiles, now called explored. If the type is End, the path has been found and the search stops. If the type is path, the surrounding 4 tiles are examined.

If any of them share coordinates with a tile in the fringe or explored and has a longer path distance, they are skipped. If any share coordinates with a tile in the fringe, and has a lower path distance, it replaces the tile in the fringe. If they are not explored and not in the fringe, they are added to the fringe.

The tile must contain four pieces of information:

- tile coordinates
- tile type
- parent
- path distance

The parent is the tile searched before the specific tile. This is used to trace back the path. The path distance is the number of tiles on the path to the specific tile. This is used to find the shortest distance.

If the fringe is emptied without finding the End tile, null is returned. If the End is found, the path is tracked back from the parent pointers, and a list of tile coordinates is returned.

The overall overview of finding the path can be seen on Figure 11, which shows the sequence diagram for calculating the path.

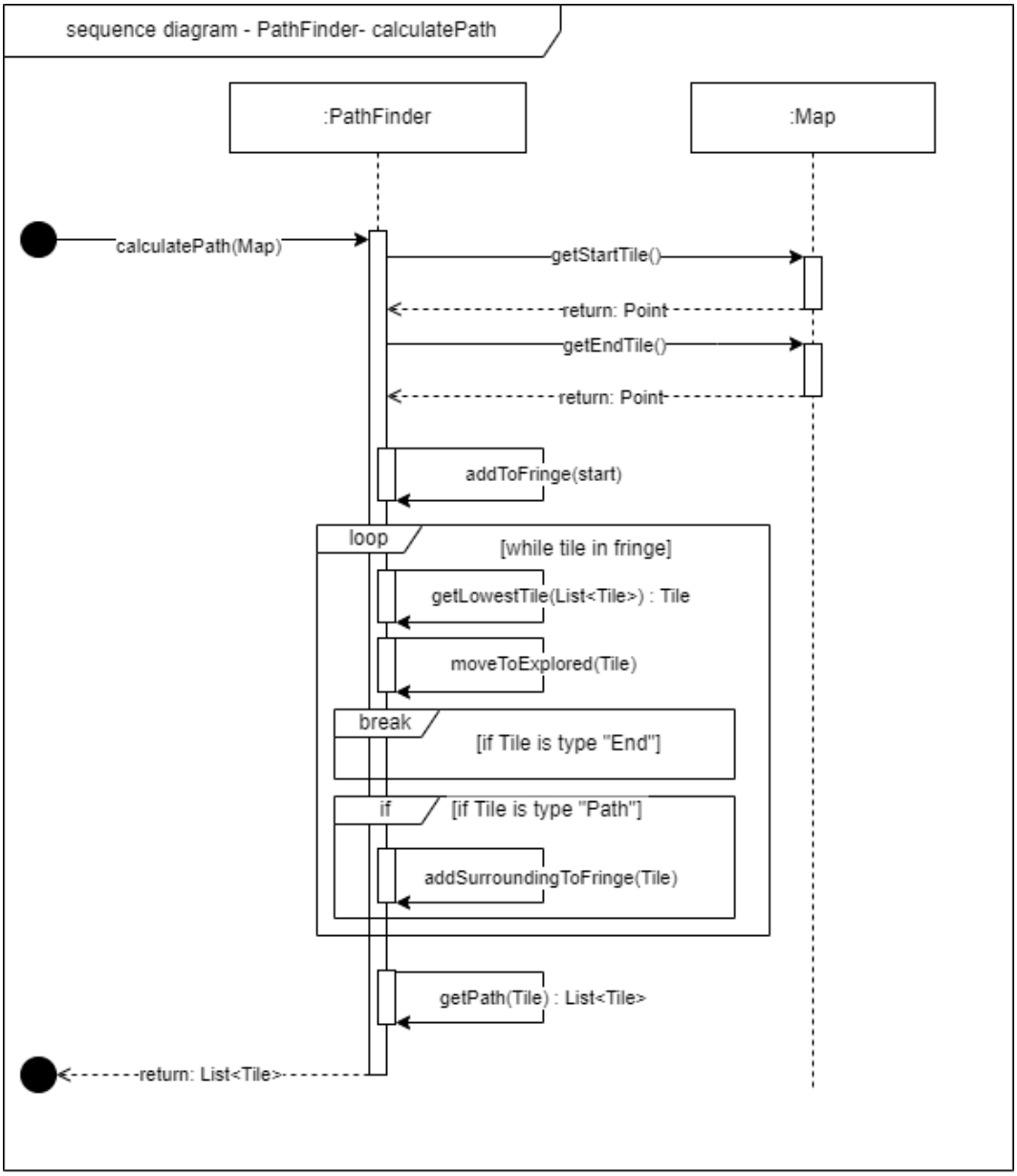


*Figure 11 - Sequence diagram for calculating the path*

## 5.6. Entity Overview

A class diagram showing the inheritance between the entities in the system, and the provided methods can be seen in Figure 12.

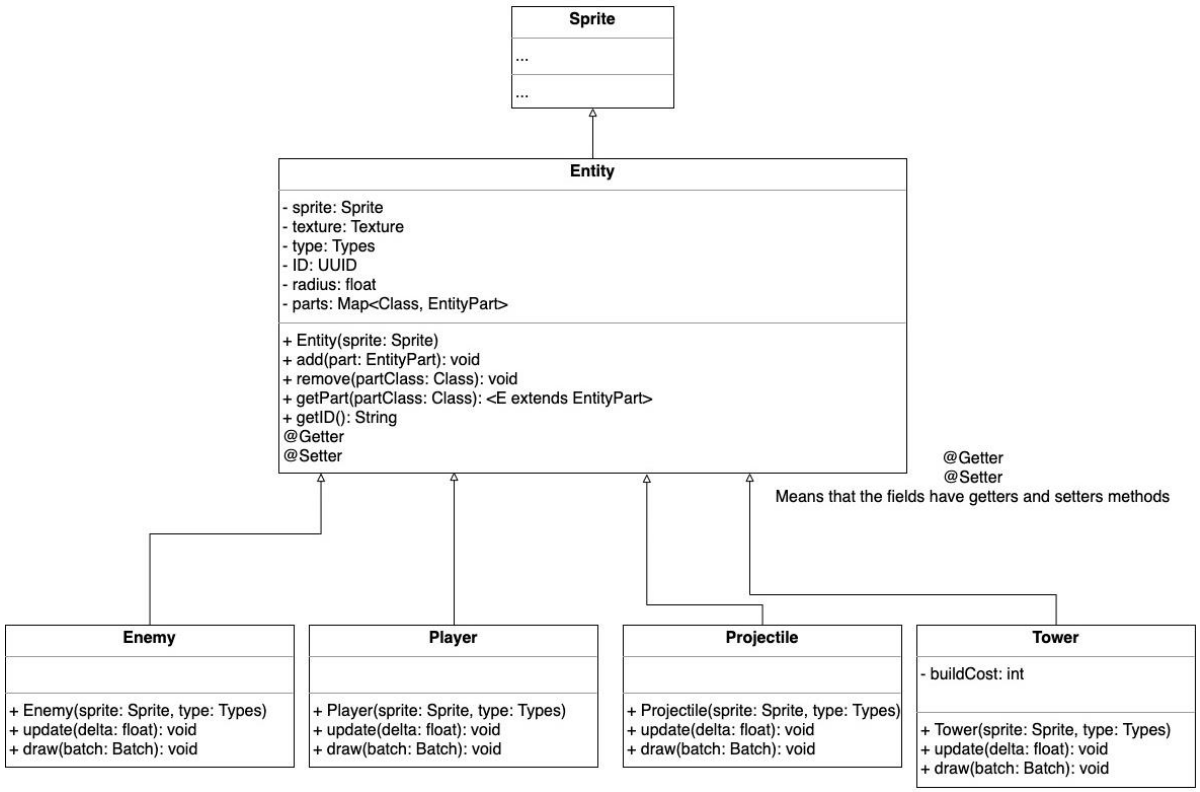


*Figure 12 - Class diagram over entities*

All the entities should be able to be drawn, and therefore, should all extend the Sprite class. This enables them to be drawn, since it will give them the necessary attributes. More values are added in the Entity class, such as a map of parts, and some attributes to define different things; for example, the sprite and texture is used to draw it.

All the entities in the game then extend the entity class, and they have some additional methods, which are overridden. For example, both *draw* and *update* are overridden. Some of them have more individual attributes giving them the necessary functionality needed. For example, the tower has a buildCost**.**

Furthermore, there will be getters and setters which is implemented by the lombok library, which makes code much cleaner and easier to read. There are just annotations over the class to replace all the getters and setters. I.e., they are there, just not written explicitly in the class. The following annotations is used:

- @Getter
- @Setter

# 6. Implementation

The following section is separated into the components that the system consists of. The implementation of each component is described to the extent where it is relevant and not repeated.

## 6.1. Core

The Core module contains two classes GameInputProcessor and Game; these are responsible for processing user input and running the game respectively.

### 6.1.1. GameInputProcessor

The game needs to be able to track user input through the keyboard. This has been done by creating the class GameInputProcessor which extends InputAdapter, see Code snippet 1, which in turn implements InputProcessor; both are part of LibGDX and uses an observer pattern for event handling.

The LibGDX interface InputProcessor is an interface containing all the methods need to handle the different types of user input, keyboard, mouse etc. The class InputAdapter implements InputProcessor and makes all the methods return false, this makes it possible to extend the InputAdaptor class and only override the methods relevant for the game. Because this Tower Defence game only uses the keyboard as input, only two methods are needed and therefor the approach of using InputAdapter instead of InputProcessor is chosen. [6]

```
public class GameInputProcessor extends InputAdapter {
    private final GameData gameData;

    public GameInputProcessor(GameData gameData) {
        this.gameData = gameData;
    }
        .
        .
        .
}
```

*Code snippet 1 – The class GameInputProcessor and its constructor.*
*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

The GameInputProcessor has an instance of GameData, which has an instance of GameKeys. It is these GameKeys in GameData that are changed by GameInputProcessor when a key is pressed or lifted. This makes it possible for all components with a dependency on the Common component to access what GameKey is pressed.

The value in GameKeys is updated through the two methods *keyDown* and *keyUp* in GameInputProcessor.

```
public boolean keyDown(int k) {
    if (k == Keys.UP) {
        gameData.getKeys().setKey(GameKeys.UP, true);
    }
    if (k == Keys.LEFT) {
        gameData.getKeys().setKey(GameKeys.LEFT, true);
    }
        .
        .
        .
    return true;
}
```

*Code snippet 2 – keyDown method used to set values of keys when a key on the keyboard is pressed. The dots represent code that have been left out.*
*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

The *keyDown* method detects when a key is being pressed by setting the GameKeys variable value to true, see Code snippet 2; the dots represent code that have been left out. The *keyUp* method is almost identical to *keyDown*; the only difference is that is it sets the value to false. Without the *keyUp* method the game keeps reading a key as pressed when it has been pressed once.

### 6.1.2. Game

The Game class is responsible for running the game. This is done by implementing the ApplicationListener interface from LibGDX which hooks into the lifecycle of the game. The instance of ApplicationListener is then passed to the LibGDX's back-end Application, here LwjglApplication is used. It is done by using the keyword *this* to refer to the current class. [7, 8]

```
public void init() {
    LwjglApplicationConfiguration cfg = new LwjglApplicationConfiguration();
    cfg.title = "TowerDefense";
    cfg.width = SCREEN_WIDTH;
    cfg.height = MAP_HEIGHT;
    cfg.useGL30 = false;
    cfg.resizable = false;

    new LwjglApplication(this, cfg);
}
```

*Code snippet 3 - The init method creates an instance of LwjglApplication and passes it the current class as the implementation of ApplicationListener*

*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

ApplicationListener contains the methods *create*, *render*, *dispose*, *resize*, *pause,* and *resume*. All of these are overridden in Game, but only create and render are implemented, the rest have been left empty.

### Create method

The *create* method is called when the application is created, it is responsible for:

- Setting the initial values for GameData
- Loading the tiled map
- Setting the start time
- Creating the enemy attacks
- Creating the batch to hold the sprites
- Creating the BitmapFonts for text on the screen
- Creating the OrthgraphicCamera and placing it correctly on the screen
- Setting the InputProcessor to GameInputProcessor
- Adding sprites to textures

### Render method

LibGDX does not have an explicit game loop, because it is event-driven by nature. The render method works like the game loop, since it is called by Application every time rendering should be performed. All game logic updates are done by this method. [7]

```
@Override
public void render() {
    Gdx.gl.glClearColor(0, 0, 0, 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    gameData.setDelta(Gdx.graphics.getDeltaTime());
    gameData.getKeys().update();

    mapRenderer.setView(camera);
    mapRenderer.render();

    batch.begin();
    drawFonts();
    batch.end();

    update();
}
```

*Code snippet 4 - The render method.*

*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

The *render* method sets some necessary LibGDX values, draws text to the screen, and updates the game logic by calling the *update* method.

### Update method

The *update* method is a private method in the Game class, created to separate out the logic needed to be updated every time the *render* method is called.

```
private void update() {
    boolean restart = false;

    if (gameData.getLife() <= 0) {
        // Stop all entities
        for (IGamePluginService iGamePluginService : gamePluginList) {
            iGamePluginService.stop(gameData, world);
        }
        // Set restart if enter is pressed
        if(gameData.getKeys().isDown(GameKeys.ENTER)){
            restart = true;
        }
        mapRenderer.render();
    }

    if (restart) {
        gameData.setPlayerDead(false);
        gameData.setScreenMessage("");
        create();
        for (IGamePluginService iGamePluginService : gamePluginList) {
            iGamePluginService.start(gameData, world, textures);
        }
        return;
    }

    // Update
    for (IEntityProcessingService entityProcessorService :
entityProcessorList) {
        entityProcessorService.process(gameData, world);
        entityProcessorService.draw(batch, world);
    }

    // Post Update
    for (IPostEntityProcessingService postEntityProcessorService :
postEntityProcessorList) {
        postEntityProcessorService.process(gameData, world);
    }
}
```

*Code snippet 5 - The update method is called by the render method and is responsible for updating the game logic.*
*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

The *Update* method tracks the need for a restart of the game. If the gameData variable life becomes zero or below, the *stop* method is called for all IGamePluginServices, leading to all entities implementing this interface being stopped and removed from the game. The user can then press Enter leading to restart being set to true.

When restart is true, the variable isPlayerDead in GameData is set to false, and the screen message is removed by setting it to an empty string. The *create* method, which was called when the application started, is called again, and all values are set back to their original start values, excluding highest score.

After the evaluation of restart, a loop iterates through all entities that implements IEntityProcessingService and calls the *process* and *draw* methods. This is followed by another loop which iterates through all entities that implements IPostEntityProcessingService and call the *process* method. This order makes it possible to first change the states of entities and then immediately after process the consequences of those changes, instead of waiting for the next call to *render*.

An example of this being done in the code is movement processed by IEntityProcessingService, and collision being processed by IPostEntityProcessingService; this leads to a smooth gameplay with less visible overlap between entities before collision is detected.

## 6.2. Common

The common module primarily contains POJO's, and interfaces used by all the other components, see section 5.3.7, and is therefore not necessary to describe in detail.

## 6.3. Common components

The common components are the following:

- CommonEnemy
- CommonPlayer
- CommonMap
- CommonProjectile
- CommonTower

All the above components are almost identical except CommonTower, CommonMap, and CommonProjectile, which have a service-provider interface that follows the exact contract defined in design. I.e., it is just an interface class with the methods described earlier. For example, the following code-snippet shows the ProjectileSPI:

```
public interface ProjectileSPI {
    void createProjectile(Entity e, GameData gameData, World world);
}
```

*Code snippet 6 - ProjectileSPI interface*
*File: OSGiCommonProjectile/src/main/java/dk/sdu/mmmi/cbse/commonprojectile/ProjectileSPI.java*

Each interface has the methods described earlier, which is why only one is shown here.

The entity classes, here Player, Enemy, Projectile, and Tower, extend the Entity class, which contains some methods that are identical. For example, Code snippet 7 shows the Player entity.

```
public class Player extends Entity {
    public Player(Sprite sprite, Types type) {
        super(sprite, type);
    }

    @Override
    public void draw(Batch batch) {
        batch.begin();
        update(Gdx.graphics.getDeltaTime());
        super.draw(batch);
        batch.end();
    }

    public void update(float delta) {
        PositionPart positionPart = this.getPart(PositionPart.class);
        this.setPosition(positionPart.getX(), positionPart.getY());
        this.setRotation((positionPart.getAngle() + 270) % 360);
    }
}
```

*Code snippet 7 - The Player class*
*File: OSGiCommonPlayer/src/main/java/dk/sdu/mmmi/cbse/commonplayer/Player.java*

As can be seen, the Player class contains the methods from the Entity class, and this is the same method for Enemy, Tower, and Projectile.

## 6.4. Collision

The CollisionManager class implements the IPostEntityProcessingService, since this is run once all the entities have been updated, to ensure correct collision happens. It starts by getting all the entities from the world, and thereafter iterating over all entities in a nested for loop. Thereby, all entities are checked against each other. This can be seen on Code snippet 8. There are some scenarios in which no changes should be made; these are:

- If one of the entities is a tower, since towers should not be affected by collisions
- If the entities are of the same type, e.g. two enemies should not damage each other
- If it is a player and projectile; projectiles should not damage the player

These conditions are checked first, whereafter it gets the position of the entities. If the distance between the two entities is less than their radius added together, then they are colliding, as mentioned in section 5.3.5.

The LifePart of the first entity is then received, whereafter different things can happen depending on the entity type:

- If it is a projectile, it is removed; a projectile should only hit once
- Life is decreased by one for the rest of the cases, and if its life is zero or less, the entity dies and is removed. Further processing is done based on some cases:
    - If it is the player, the player is set as dead in the GameData
    - If it is an enemy, the score goes up since an enemy was killed. Thereby, the game data score is increased.

```
@Override
public void process(GameData gameData, World world) {
    ArrayList<Entity> entities = new ArrayList<>(world.getEntities());
    for (Entity iEntity : entities) {
        for (Entity oEntity : entities) {
            if (iEntity.getType() == Types.TOWER || oEntity.getType() ==
Types.TOWER)
                continue;
            if (iEntity.getType() == oEntity.getType()) //Should not collide
when equal type
                continue;
            if ((iEntity.getType() == Types.PLAYER || oEntity.getType() ==
Types.PLAYER) && (iEntity.getType() == Types.PROJECTILE || oEntity.getType()
== Types.PROJECTILE))
                continue; //Player and projectile should not collide

            // Get the position part for the entities
            PositionPart iPosition = iEntity.getPart(PositionPart.class);
            PositionPart oPosition = oEntity.getPart(PositionPart.class);

            // Calculate distance between two entities
            double distance = Math.sqrt(Math.pow((iPosition.getX() -
oPosition.getX()), 2) + Math.pow((iPosition.getY() - oPosition.getY()), 2));

            if (distance < (iEntity.getRadius() + oEntity.getRadius())) {
//Collides
                LifePart iLifePart = iEntity.getPart(LifePart.class);
                if (iEntity.getType() == Types.PROJECTILE)
                    world.removeEntity(iEntity);
                iLifePart.setLife(iLifePart.getLife() - 1);
                if (iLifePart.getLife() <= 0) {
                    if (iEntity.getType() == Types.PLAYER)
                        gameData.setPlayerDead(true);
                    // checking if enemy is dead & updates score
                    if (iEntity.getType() == Types.ENEMY)
                        gameData.setScore(gameData.getScore() + 1);
                    world.removeEntity(iEntity);
                }
            }
        }
    }
}
```

*Code snippet 8 - Process method from collision*
*File: OSGiCollision/src/main/java/dk/sdu/mmmi/cbse/collision/CollisionManager.java*

## 6.5. Map

The map component is based on the LibGDX TiledMap class. A Tileset containing the individual tiles described in section 5.3.9 is saved as Tiles.tsx, containing the tile id and property description of each tile.

The two-dimensional map array is defined in Map.tmx with cells encoded as csv and metadata about the map. E.g., dimensions and tilesize.

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.8" tiledversion="1.8.4" orientation="orthogonal"
renderorder="right-down" width="12" height="12" tilewidth="58" tileheight="58"
infinite="0" nextlayerid="2" nextobjectid="1">
<tileset firstgid="1" source="Tiles.tsx"/>
<layer id="1" name="Tile Layer 1" width="12" height="12">
 <data encoding="csv">
3,3,3,3,3,3,3,3,3,3,3,3,
3,3,3,3,3,3,3,3,3,3,3,3,
3,3,4,4,4,4,4,4,4,4,4,5,
3,3,4,3,3,3,3,3,3,3,3,3,
3,3,4,3,3,3,3,3,3,3,3,3,
3,3,4,4,4,4,3,3,4,4,4,3,
3,3,3,3,3,4,3,3,4,3,4,3,
2,4,4,3,3,4,4,4,4,3,4,3,
3,3,4,3,3,3,3,3,3,3,4,3,
3,3,4,3,3,3,3,3,3,3,4,3,
3,3,4,4,4,4,4,4,4,4,4,3,
3,3,3,3,3,3,3,3,3,3,3,3
</data>
</layer>
</map>
```

*Code snippet 9 - Map.tmx, containing metadata of map*
*File: runner/Map.tmx*

The map is loaded when the game is created in the *create* method in Core component's Game class and added to the map component from Core.

```
public void create() {
.
.
   map.setTiledMap(new TmxMapLoader().load("Map.tmx"));
   mapRenderer = new OrthogonalTiledMapRenderer(map.getTiledMap());
.
```

*Code snippet 10 - loading of map in create*
*File: OSGiCore/src/main/java/dk/sdu/mmmi/cbse/Game.java*

The map contains two sets of coordinate systems; tile-coordinates, which refers to the tiles of the map, in this case 12 x 12, and map-coordinates used for positioning of entities, that refers to the actual pixels on the screen in this case 696 x 696px,

The mehods in the map interface is implemented using the main properties from the map, which can be achieved by methods supplied by LibGDX. An example is the *getTileTypeByCoor* function, which returns the tile property of a tile at a map coordinate (x, y).

```
public String getTileTypeByCoor(int x, int y) {
    //get first layer of map
    TiledMapTileLayer layer = (TiledMapTileLayer) tiledMap.getLayers().get(0);

    // Get tile at position (x,y)
    Point point = getTileCoordinates(x, y);
    TiledMapTile tile = layer.getCell(point.x, point.y).getTile();

    // getting properties of tile
    return tile.getProperties().get("Tag", String.class);
}
```

*Code snippet 11 - method to get tile type by coordinate*
*File: OSGiMap/src/main/java/dk/sdu/mmmi/cbse/map/MapType.java*

The first layer of the map is accessed, as the game only contains one layer. The map coordinates are translated into tile coordinates, and the specific tile is accessed. Finally, the property for the tile is returned.

## 6.6. Tower

A tower's primary purpose is shooting at enemies. To select the correct enemy to target, the tower uses an evaluation, *getHeuristic.* which calculates a value for an enemy by the parameters described in the analysis, see section 5.3.2. The distance is calculated using various helper methods not described here e.g., *getDistanceToEnd*. Each value is weighted with a weight to balance the result, so the preferred enemy is selected.

```
private float getHeuristic(Enemy enemy, Tower tower){
    // calculate heuristics of an enemy
    LifePart enemyLifePart = enemy.getPart(LifePart.class);
    float enemyHeuristic = 0;
    enemyHeuristic += weightDistanceToEnd * getDistanceToEnd(enemy);
    enemyHeuristic += weightDistanceToStart * getDistanceToStart(enemy);
    enemyHeuristic += weightDistanceToTower * getDistanceBetweenEntities(enemy,
tower);
    enemyHeuristic += weightLife * enemyLifePart.getLife();
    return enemyHeuristic;
}
```

*Code snippet 12 - get heuristic from enemy and tower*
*File: OSGiTower/src/main/java/dk/sdu/mmmi/cbse/tower/TowerControlSyste.java*

To be able to compare the heuristics of two enemies an enemy comparator is used; returning a negative integer if enemy1 has the lowest heuristic and positive if enemy2.

```
class enemyComparator implements Comparator<Enemy> {

   @Override
   public int compare(Enemy enemy1, Enemy enemy2) {
       float heuristicEnemy1 = getHeuristic(enemy1, (Tower) selectedTower);
       float heuristicEnemy2 = getHeuristic(enemy2, (Tower) selectedTower);
       return (int) (heuristicEnemy1 - heuristicEnemy2);
   }
}
```

*Code snippet 13 - comparator for enemy*
*File: OSGiTower/src/main/java/dk/sdu/mmmi/cbse/tower/TowerControlSystem.java*

A PriorityQueue<E> of reachable enemies is used, as it implements the desired min-heap described in section 5.3.2. It is initialized with the initial capacity of 10 and uses the enemyComparator for the ordering.

The list of entities belonging to the Enemy class is iterated, and each enemy within the towers range is added to the PriorityQueue of reachable enemies, using the add method.

```
List<Entity> enemies = world.getEntities(Enemy.class);
PriorityQueue<Enemy> reachableEnemies = new PriorityQueue<>(10, new
enemyComparator());

if (enemies != null) {
   for (Entity enemy : enemies) {
      int distance = getDistanceBetweenEntities(enemy, tower);
      if (distance < weaponPart.getRange()){
         reachableEnemies.add((Enemy) enemy);
     }
   }
}
```

*Code snippet 14 - Adding to priority queue*
*File: OSGiTower/src/main/java/dk/sdu/mmmi/cbse/tower/TowerControlSystem.java*

To select the enemy to target, the peek method is used to get the first element in the queue, which will have the lowest heuristic value due to the min heap property; first element in a min-heap is the smallest.

```
Enemy selectedEnemy = reachableEnemies.peek();
```

*Code snippet 15 - Selected enemy*
*File: OSGiTower/src/main/java/dk/sdu/mmmi/cbse/tower/TowerControlSystem.java*

## 6.7. Projectile

The projectile component has the class ProjectileControlSystem, which implements the IEntityProcessing and ProjectileSPI interfaces. The IEntityProcessing interface enables it to get processed by the core. The process method iterates over all the projectile entities and processes the

MovingPart of the projectile. Furthermore, it checks whether the projectile has reached the end of its range and checks if it is outside the map. In either case it is removed from the world, as can be seen in Code snippet 16.

```
@Override
public void process(GameData gameData, World world) {
    for (Entity projectile : world.getEntities(Projectile.class)) {

        PositionPart positionPart = projectile.getPart(PositionPart.class);
        MovingPart movingPart = projectile.getPart(MovingPart.class);
        WeaponPart weaponPart = projectile.getPart(WeaponPart.class);

        movingPart.process(gameData, projectile);

        // checks if it has reached it's range
        if (positionPart.getDistanceFromOrigin() > weaponPart.getRange()) {
            world.removeEntity(projectile);
        }

        // check if outside map
        float radius = projectile.getRadius();
        float centerX = positionPart.getX() + map.getTileSize()/2f;
        float centerY = positionPart.getY() + map.getTileSize()/2f;

        if (!(map.isInsideMap(centerX+2+radius, centerY+2+radius))) {
            world.removeEntity(projectile);
        }

        if (!(map.isInsideMap(centerX+2-radius, centerY+2-radius))) {
            world.removeEntity(projectile);
        }
    }
}
```

*Code snippet 16 - Process in projectile*
*File: OSGiProjectile/src/main/java/dk/sdu/mmmi/cbse/projectile/ProjectileControlSystem.java*

It has a *draw* method, which all entities must have to be drawn; the method simply iterates over all the projectiles, and calls their *draw* method, which is then handled in their common component. It also has a *createProjectile* method which creates the projectile. The start coordinates for the projectile and which direction it should move is derived from the entity creating the projectile. Thereafter, it gets the texture for the projectile, creates the sprite, adds the parts to the projectile, and adds it to the world, as can be seen in Code snippet 17.

```
// getting sprite for projectile
Texture texture = world.getTextureHashMap().get(Types.PROJECTILE);
Sprite sprite = new Sprite(texture);
sprite.setCenter(sprite.getHeight() / 2, sprite.getWidth() / 2);

// creating new projectile with entity parts and add to world
Entity projectile = new Projectile(sprite, Types.PROJECTILE);
projectile.setRadius(4);
projectile.add(new MovingPart(projectileSpeed, true));
projectile.add(new PositionPart(projX, projY, radians));
projectile.add(new LifePart(1));
projectile.add(weaponPart);

world.addEntity(projectile);
```

*Code snippet 17 - Creating projectile*
*File: OSGiProjectile/src/main/java/dk/sdu/mmmi/cbse/projectile/ProjectileControlSystem.java*

# 6.8. Enemy

The enemy component is responsible for creating the enemies on the map, sending enemies through the map, and navigating them.

## 6.8.1. Path Navigation

The enemy module is given a path of tile coordinates. This must be converted into enemy navigation. For this, the tiles are processed two by two to find the correct direction to walk, to get from the first tile to the next as seen in Code snippet 18. This direction is saved in a PathDirection POJO along with the coordinates of when the direction should change again. When the entity reaches this goal, it will take directions from the next PathDirection object in the stack.

```
protected Stack<PathDirection> getPathDirectionStack(ArrayList<Point> path) {
    Stack<PathDirection> pathDirections = new Stack<>();
    for (int x = 1; x < path.size(); x++) {
        Point currentTile = path.get(x);
        Point newTile = path.get(x - 1);
        PathDirection direction = new PathDirection(getDirection(currentTile,
newTile), map.getTileCenter(newTile));

        pathDirections.add(direction);
    }

    return pathDirections;
}
```

*Code snippet 18 - getting the path direction*
*File: OSGiEnemy/src/main/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystem.java*

To check when the goal is reached, the method *CheckEnemyOutOfBounds* is used, as seen in Code snippet 19. The method takes 2 parameters; where the entity is, the PositionPart, and where the goal is, the PathPart. There need to be different checks for different directions. If the entity is moving left, the

method checks if it has reached the right side of the goal. Opposite, if it is moving right, the method checks if it has reached the left side of the goal, and so forth.

```
private boolean checkEnemyOutOfBounds(PositionPart positionPart, PathPart
pathPart){
    if (positionPart.getAngle() == PositionPart.left && positionPart.getX() >
pathPart.getGoal().x)
        return false;
    if (positionPart.getAngle() == PositionPart.right && positionPart.getX()
< pathPart.getGoal().x)
        return false;
    if (positionPart.getAngle() == PositionPart.down && positionPart.getY() >
pathPart.getGoal().y)
        return false;
    if (positionPart.getAngle() == PositionPart.up && positionPart.getY() <
pathPart.getGoal().y)
        return false;

    return  true;
```

*Code snippet 19 - checking if enemies is out of bounds*
*File: OSGiEnemy/src/main/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystem.java*

## 6.8.2. Asynchronous navigation

The path navigation described in section 6.8.1 requires a certain amount of computation. The game is unwilling to draw sufficient resources from the system running it and so the path computation causes lag. This is less relevant for computations happening before the introduction of a new entity than for computations recurring through the lifecycle of an entity. Nevertheless, it is an issue and should be addressed sooner rather than later. The solution used is a ThreadPool seen in Code snippet 20 where the creation of enemies is added as tasks. This ensures a smooth-running game, as the processing can continue while the heavier computation finishes.

```
for (Entity enemy : enemies) {
    executor.submit(() -> {
        PathPart pathPart = enemy.getPart(PathPart.class);
        PositionPart positionPart = enemy.getPart(PositionPart.class);

        if(checkEnemyOutOfBounds(positionPart, pathPart)) {
            setNewEnemyPath(enemy, () -> goalReached(enemy, gameData,
world));
        }
    });
```

*Code snippet 20 - Committing to the threadpool*
*File: OSGiEnemy/src/main/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystem.java*

## 6.9. Player

The Player component contains two classes: PlayerControlSystem and PlayerPlugin. It is based on the Player defined in the CommonPlayer component, extending the Entity class from the Common component.

The PlayerControlSystem implements the IEntityProcessingService interface and is responsible for handling and processing any user inputs relating to controlling the Player in the game.

The draw method from the IEntityProcessingService interface is overwritten to loop through all entities present in the world, find those that are of the Player class, and use the draw method defined in CommonPlayer to render the necessary sprites.

The method *handleInput,* checks for user input that would affect the Player's movements, here the arrow keys, and adjusts the Player accordingly. If any of the arrow keys are pressed, the angle of the Player's PositionPart is updated to point in the indicated direction. The Player's MovingPart is set to moving as well, making it possible for the Player to move around the map. If any other keys are pressed, this method will not affect the Player or its parts.

The PlayerControlSystem overwrites the *process* method from the IEntityProcessingService interface. This method handles any key input given by the user, using the predefined key mapping and the *handleInput* method to determine the correct response.

This includes moving the Player around the map, by updating the Player's MovingPart and PositionPart, but also the placement of towers on the map. To do this, the PlayerControlSystem is injected with an instance of IMap and a TowerSPI, from the Map component and Tower component, respectively.

This allows the PlayerControlSystem to place a new tower on the map, by calling the TowerSPI, and utilizing its *createTower* method. Likewise, it is possible to use the *changeTileType* method from the IMap interface to change the tile the Player is positioned on from a Grass tile to a Tower tile. Part of the code for this action can be found in Code snippet 21.

```
…
if (gameData.getKeys().isDown(SPACE) && towerSPI != null) {
    Tower tower;

    Point coordinates = map.mapCoorToTileCoor(positionPart.getX(),
positionPart.getY());
    tower = (Tower) towerSPI.createTower(gameData, world, coordinates.x,
coordinates.y);

    if (tower != null && tower.getBuildCost() > gameData.getMoney()) {
        int buildCost = tower.getBuildCost();
        tower = null;
        map.changeTileType(coordinates.x, coordinates.y, "Grass");
        gameData.setScreenMessage("You don't have enough \nmoney to buy a
Tower \n\nTower cost: " + buildCost);
    }
…
```

*Code snippet 21 - Creating the tower from player*
*File: OSGiPlayer/src/main/java/dk/sdu/mmmi/cbse/player/PlayerControlSystem.java*

The *process* method in the PlayerControlSystem then adds the newly created tower to the world, and handles updating the money attribute found in GameData, to reflect the cost of building a tower.

The second class in the Player component is the PlayerPlugin. This class implements the IGamePluginService interface and is responsible for starting and stopping the component. The *start* method handles the creation of a Player object based on predefined values, as well as adding the Player object to the world upon creation of the game.

Likewise, the *stop* method removes the Player from the world, and is used if the component is deactivated during runtime and once the game ends.

## 6.10. Test

The test component is created to make it possible to run integration tests. Integration tests cannot be placed inside any other module since it must test the system as a whole. Therefore, it is placed in its own component, so it can easily be changed, and is not dependent on other implementations. The component only contains a test folder with the necessary dependencies to run those tests.

## 6.11. FileHandler

The FileHandler is created to help with accessing files from the project using LibGDX. LibGDX has an internal filesystem, which can cause problems finding the files in the resources folder, because LibGDX looks elsewhere in an internal assets folder. To solve this problem, a component was created, that helped locating the files in the correct path. It contains all the relevant resources, and a way to access

them by the file name. This FileHandler extends FileHandle from the LibGDX library and overrides the *readBytes* and *read* methods.

## 6.12. LibGDX

The LibGDX component contains all the dependencies necessary from LibGDX framework. These are packed into a module, since it is easier and better to have a dependency on another module, than on all the LibGDX packages, as it would have to be specified in POM files of each module.

## 6.13. The final game

The final game ended up looking like Figure 13.


*Figure 13 - The final game*

The design led to the implementation of the game, which ended up looking like the above. It resembled much the game described in the analysis and design, which was expected and desired.

# 7. Test

The following section explains how the Must-Have requirement have been tested. The framework used for testing is Mockito together with JUnit, which provides powerful mocking and hooking. Once all tests pass, it is verified that the code executes as expected, and the requirements are fulfilled.

**The non-functional requirements are fulfilled:**

- NF1: The project uses OSGi as the component framework, which can unload and load components during runtime. Furthermore, the OSGi framework has a ClassLoader in each component, and supports versioning
- NF2: Contains data structures that are well defended and documented, in this project min-heap is used
- NF3: It contains algorithms for calculating which enemy to select and the min-heap methods are used
- NF4: The AI uses an A* algorithm

Thereby, it can be concluded that the non-functional requirements are fulfilled.

The following sections tests the different functional requirements.

All the tests are run when the game is started, to ensure that the game functions as expected.

## 7.1. Dynamic loading and unloading

**Requirement:** F1 - Must be possible to dynamically load/unload Player, Enemy, Weapon.

**File:** OSGiTest/src/test/java/dk/sdu/mmmi/cbse/ApplicationITest.java

**Type:** Integration test

**Description:**

To fulfill the requirement it must be possible to load and unload the Player, Enemy and Weapon , here called Projectile, components during runtime. This can be tested by hooking into the BundleContext API, which is injected by Pax Exam; Pax Exam is a testing framework for OSGi bundles, which allows to access the bundles and load and unload them without running the whole application.

**Preconditions:**

The modules are compiled

**Test steps:**

1. At start, the bundles are loaded into Pax Exam container, as seen in Code snippet 22.

```
//importing all the bundles
@Configuration
public static Option[] configuration() {
    return options(
            provision(
                    bundle(toFileURI("bundles/OSGiLibGDX.jar")),
                    bundle(toFileURI("bundles/OSGiPlayer.jar")),
                    bundle(toFileURI("bundles/OSGiEnemy.jar")),
                    bundle(toFileURI("bundles/OSGiProjectile.jar")),
                    bundle(toFileURI("bundles/OSGiCore.jar")),
                    bundle(toFileURI("bundles/OSGiMap.jar")),
                    mavenBundle("org.osgi", "org.osgi.compendium", "4.2.0"),
                    mavenBundle("org.ops4j.pax.swissbox", "pax-swissbox-
tinybundles", "1.0.0")
            )
    );
}
```

*Code snippet 22 - Loading the bundles*
*File: OSGiTest/src/test/java/dk/sdu/mmmi/cbse/ApplicationITest.java*

2. Thereafter, the bundles are loaded in, and the test method can execute. For example, it tests that the bundle is active, and when it is stopped, it is tested that the bundle has stopped.

```
@Test
public void Test_Loading_Unloading_Contains_Enemy() throws Exception {
    for (Bundle bundle_started : bundleContext.getBundles()) {
        if (bundle_started.getHeaders().get("Bundle-Name") != null &&
bundle_started.getHeaders().get("Bundle-Name").equals("OSGiEnemy")) {
            log.info("Bundle found: OSGiEnemy");
            if (bundle_started.getState() != Bundle.ACTIVE) {
                log.error("BUNDLE NOT ACTIVE");
                throw new Exception("NOT ACTIVE");
            }
            bundle_started.stop();
            log.info("Bundle stopped");
            if (bundle_started.getState() != Bundle.RESOLVED) {
                log.error("BUNDLE NOT RESOLVED");
                throw new Exception("NOT RESOLVED");
            }
```

*Code snippet 23 - Test for unloading and loading enemy*
*File: OSGiTest/src/test/java/dk/sdu/mmmi/cbse/ApplicationITest.java*

3. Tests for uninstalling; uninstalling the component, and thereafter verifying it is uninstalled
4. Tests for installing; reinstalling the component, and thereafter verifying it is installed again

**Expected result:**

The test can be concluded to run as expected, since no exceptions are thrown, i.e., the test does not fail.

The same test is done for Enemy, Player, and Projectile.

## 7.2. Components

**Requirement:** F2 Must contain components Player, Weapon, Enemy, Map, Core

**File:** OSGiTest/src/test/java/dk/sdu/mmmi/cbse/ApplicationITest.java

**Type:** Integration test

**Description:**

The game must include the Player, Weapon, here Projectile, Enemy, Map and Core components, which should be tested to verify that they are loaded in. This is done using Pax Exam, as described in 7.1. Code snippet 22 shows how the modules are loaded in. Thereafter, the same tests are used for the Player, Enemy and Projectile to verify whether they are loaded in; otherwise it would not be possible to deactivate them. Code snippet 24 shows the test to confirm the game contains the map component.

```
/**
 * Checks that the map-bundle exists
 * @throws Exception thrown if it fails
 */
@Test
public void containsMap() throws Exception {
    for (Bundle bundle : bundleContext.getBundles()){
        if(bundle.getHeaders().get("Bundle-Name") != null &&
bundle.getHeaders().get("Bundle-Name").equals("OSGiMap")){
            log.info("CONTAINS OSGiMAP");
            return;
        }
    }
    log.error("DOES NOT CONTAIN OSGiMAP");
    throw new Exception("DOES NOT CONTAIN MAP");
}
```

*Code snippet 24 - Test that it contains the map component*
*File: OSGiTest/src/test/java/dk/sdu/mmmi/cbse/ApplicationITest.java*

**Preconditions:**

The modules are compiled

**Test steps:**

1. The bundles from the bundlecontext are iterated over
2. It finds the bundle OSGiMap, and if it cannot find it, an exception will be thrown, i.e., the test fails. If it does find it, the test terminates as successful

The same test is done for the Core-module, and thereby they are all tested.

**Expected result:**

The components are found, and no exceptions are thrown.

## 7.3. Player movement

**Requirement:** F3.1 A Player must be able to move around the map

**File:** OSGiPlayer/src/test/dk/sdu/mmmi/cbse/player/PlayerControlSystemTest.java

**Type:** Unit test

**Description:**

The PlayerControlSystem must be able to receive key input from the user playing the game and process those, so that the MovingPart can update PositionPart with the Player's new coordinates on the map.

**Preconditions:**

A Player is situated on a Map, within the Map's boundaries.

**Test steps:**

1. Prepare mock objects for the test, including GameData, World, Entity, and Keys. An instantiated PlayerControlSystem is also needed.
2. Ensure the mocked MovingPart- and PostitionPart objects are returned when the method *getPart* is called on the mocked Entity object.
3. Ensure the mocked Entity object is returned when the method *getEntities* is called on the mocked World object.
4. Ensure the mocked Keys object is returned when the method *getKeys* is called on the mocked GameData object.

5. Process the mocked GamaData and World objects. The process method for PlayerControlSystem will then call the process methods for MovingPart and PositionPart, both of which are tested separately in their own module.
    a. MovingPartTest
        i. Prepare mock objects for the test, including GameData, Entity, and IMap, as well as instantiated instances of MovingPart and PositionPart.
        ii. Assign the mocked IMap to the MovingPart and ensure true is returned when *isMoving* is called on the MovingPart mock
        iii. Ensure the instantiated PositionPart is returned when *getPart* is called on the mocked Entity.
        iv. Ensure the size of a maptile, here 58 px, is returned, when *getTileSize* is called on the mocked IMap.
        v. Ensure that *isInsideMap* returns true, when called on the mocked IMap.
        vi. Call *process* on MovingPart, using the mocked GameData and Entity as parameters.
        vii. Compare the initial PositionPart with the Entity's updated PositionPart to ascertain whether MovingPart.*process* updated the coordinates for PositionPart based on the Entity's speed and direction.

    b. PositionPartTest
        i. Prepare for the test by instantiating a PositionPart, with known x and y coordinates.
        ii. Call *setPosition* on the instantiated PositionPart, using different x and y values.
        iii. Extract the new x and y coordinates, ascertain whether they match the new x and y values given in the previous call of *setPosition*.
6. Verify that the mocked MovingPart- and PositionParts each calls their *process* method once and only once

```
@Test
public void testPlayerMovement(){
    PlayerControlSystem playerControlSystem = new PlayerControlSystem();

    when(entity1.getPart(MovingPart.class)).thenReturn(movingPart);
    when(entity1.getPart(PositionPart.class)).thenReturn(positionPart);
    world.addEntity(entity1);

    when(world.getEntities(any())).thenReturn(new ArrayList<Entity>(){{
        add(entity1);
}});
    when(gameData.getKeys()).thenReturn(keys);

    playerControlSystem.process(gameData, world);

    verify(movingPart,times(1)).process(any(), any());
    verify(positionPart, times(1)).process(any(), any());
}
```

*Code snippet 25 - Test player movement*
*File: OSGiPlayer/src/test/java/dk/sdu/mmmi/cbse/player/PlayerControlSystemTest.java*

**Expected result:**

If the two verifications in step 6 both return true, the test will pass, meaning that the process methods for MovingPart and PositionPart are each called once and only once. Should either or both process methods not be called, or called multiple times, the test will fail.

## 7.4. Building towers

**Requirement:** F3.2 A player must be able to build towers

**File:** OSGiPlayer/src/test/java/dk/sdu/mmmi/cbse/player/PlayerControlSystem.java

**Type:** Unit test

**Description:**

It must be possible for the player to build towers. This can be done by verifying that the player calls the TowerSPI to create a tower once the space bar is pressed.

**Preconditions:**

The TowerSPI is injected, and the player exists in the world

```
@Test
public void placeTowerTest(){
    when(entity1.getPart(MovingPart.class)).thenReturn(movingPart);
    when(entity1.getPart(PositionPart.class)).thenReturn(positionPart);
    world.addEntity(entity1);

    when(world.getEntities(any())).thenReturn(new ArrayList<Entity>(){{
        add(entity1);
    }});
    PlayerControlSystem playerControlSystem = new PlayerControlSystem();
    playerControlSystem.setTowerSPI(towerSPI);
    playerControlSystem.setIMap(map);
    when(keys.isDown(anyInt())).thenReturn(true);
    when(gameData.getKeys()).thenReturn(keys);
    when(map.mapCoorToTileCoor(anyFloat(), anyFloat())).thenReturn(point);
    playerControlSystem.process(gameData, world);
    verify(towerSPI, times(1)).createTower(any(), any(), anyInt(), anyInt());
}
```

*Code snippet 26 - Test of place tower*
*File: OSGiPlayer/src/test/java/dk/sdu/mmmi/cbse/player/PlayerControlSystemTest.java*

**Test steps:**

1. Mocking of objects are created, and thereafter, the mocked objects can be hooked into.
2. The class under test, PlayerControlSystem, is instantiated
3. The mocked TowerSPI and IMap are injected into the ControlSystem
4. The method *process* is called to start the method
5. It is verified that the TowerSPI has been called 1 time with the method *createTower* with any arguments
6. To verify that the tower is placed correctly, it can be checked in the tests for tower whether the tile changed to Tower. This can be seen on Code snippet 27. It verifies that the tile type is changed to Tower.

```
@Test
public void placeTowerTest(){
    TowerControlSystem towerControlSystem = new TowerControlSystem();
    towerControlSystem.setIMap(mockMap);
    when(mockMap.getTileType(anyInt(), anyInt())).thenReturn("Grass");
    HashMap<Types, Texture> hashMap = new HashMap<Types, Texture>(){{
        put(Types.TOWER, texture);
    }};
    when(mockMap.tileCoorToMapCoor(anyFloat(), anyFloat())).thenReturn(new
Point(1,1));
    when(world.getTextureHashMap()).thenReturn(hashMap);
    when(mockMap.getTileSize()).thenReturn(58);
    // injecting mock dependencies
    towerControlSystem.setIMap(mockMap);
    towerControlSystem.createTower(gameData, world, 1, 1);
    verify(mockMap, times(1)).changeTileType(1,1, "Tower");
}
```

*Code snippet 27 - Test for placing tower*
*File: OSGiEnemy/src/test/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystemTest.java*

**Expected result:**

The TowerSPI *createTower* method is called with arguments, and that the tile type is changed to Tower, and thereby, a tower is created.

## 7.5. Enemy movement with AI

**Requirements:** F4.1 An Enemy must be able to move through the path on the map using an AI.

**Description:**

To fulfill this requirement, it is necessary that the EnemyControlSystem updates the MovingPart when the *process* method is called. The MovingPart is used to update the Enemies position on the map, which is updated using data calculated by an AI system.

### 7.5.1. EnemyControlSystem

**File:** OSGiEnemy/src/test/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystemTest.java

**Type:** Unit test

**Preconditions:** None

**Test steps:**

1. EnemyControlSystem is created
2. EnemyControlSystem instance is given a Mockito mock of IMap
3. On *getCurrentAttack* method called on GameData instance, return empty list.

4. On *getEntities(Enemy.class)* called on world, return list with Mockito instance of enemy
5. On *getPart(MovingPart.class)* called on enemy, return Mockito mock of MovingPart
6. Run *process* method on EnemyControlSystem instance.
7. Verify that the *process* method is called on MovingPart

```
@Test
public void testEnemyMovement(){
    EnemyControlSystem enemyControlSystem = new EnemyControlSystem();
    enemyControlSystem.setIMap(map);
    when(gameData.getCurrentAttacks()).thenReturn(new ArrayList<>());
    when(world.getEntities(Enemy.class)).thenReturn(new ArrayList<Entity>(){{
        add(enemy);
    }});
    when(enemy.getPart(MovingPart.class)).thenReturn(movingPart);
    enemyControlSystem.process(gameData, world);
    verify(movingPart, times(1)).process(any(), any());
}
```

*Code snippet 28 - Test for EnemyMovement*
*file: OSGiEnemy/src/test/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystemTest.java*

**Expected result:**

The *process* method on the MovingPart instance is expected to be called during the *process* method call on EnemyControlSystem.

### 7.5.2. MovingPart

**File:** OSGiCommon/src/test/java/dk/sdu/mmmi/cbse/common/MovingPartTest.java

**Type:** Unit test

**Preconditions:** None

**Test steps:**

1. Mockito mocks have been created
2. A MovingPart instance is created with a speed of 1
3. It is given a Mockito mock of map.
4. It is set to “moving”
5. A position part is created with a known position
6. A Mockito mock of an Entity is given the PositionPart
7. The Entity instance is set to return the PositionPart when queried.
8. Map is set to return a specific tilesize
9. Entity is set to return a specific radius

10. Map is set to always verify any position as inside the map's boundaries.
11. The *Process* method is called on MovingPart, with the Mockito Entity.
12. It is verified that the PositionPart is changed as a result

```
@Test
public void movingPartProcessTest(){
    MovingPart movingPart = new MovingPart(1);
    movingPart.setIMap(map);
    movingPart.setMoving(true);
    PositionPart positionPart = new PositionPart(12, 10, 90);
    entity.add(positionPart);

    when(entity.getPart(PositionPart.class)).thenReturn(positionPart);

    when(map.getTileSize()).thenReturn(58);

    when(entity.getRadius()).thenReturn(5f);

    when(map.isInsideMap(anyFloat(), anyFloat())).thenReturn(true);

    movingPart.process(gameData, entity);
    PositionPart positionPart2 = entity.getPart(PositionPart.class);

    assertTrue(positionPart2.getX()!=12 || positionPart2.getY()!=10);
}
```

*Code snippet 29 - Test for the process method in MovingPart*
*file: OSGiCommon/src/test/java/dk/sdu/mmmi/cbse/common/MovingPartTest.java*

**Expected result:**

When the MovingPart *process* method is called, it updates the PositionPart values contained in the given Entity.

### 7.5.3. Map

**File:** OSGiMap/src/test/java/dk/sdu/mmmi/cbse/map/MapTest.java

**Type:** Unit test

**Preconditions:** A map exists

**Test steps:**

1. Mockito mocks have been created
2. A list of expected path points is created
3. A Mockito mock of Map is changed to return values matching a drawn map
4. An instance of PathFinder is created
5. The *calculatePath* method is called on the PathFinder instance

6. It is checked that the path is not empty
7. It is checked that the returned path is identical to the expected path

```
@Test
    public void testAStar() {
        ArrayList<Point> expectedPath = new ArrayList<>();

        expectedPath.add(new Point(0, 2));
        expectedPath.add(new Point(1,2));
        expectedPath.add(new Point(2,2));
        expectedPath.add(new Point(3, 2));

        when(map.getStartTileCoor()).thenReturn(new Point(3,2));
        when(map.getEndTileCoor()).thenReturn(new Point(0,2));
        when(map.getTileType(0,2)).thenReturn("End");
        when(map.getTileType(1,1)).thenReturn("Path");
        when(map.getTileType(1,2)).thenReturn("Path");
        when(map.getTileType(1,3)).thenReturn("Grass");
        when(map.getTileType(2,1)).thenReturn("Path");
        when(map.getTileType(2,2)).thenReturn("Path");
        when(map.getTileType(2,3)).thenReturn("Grass");
        when(map.getTileType(3,1)).thenReturn("Grass");
        when(map.getTileType(3,2)).thenReturn("Start");
        when(map.getTileType(3,3)).thenReturn("Grass");
        when(map.getTileType(4,2)).thenReturn("Grass");

        PathFinder pathFinder = new PathFinder(map);
        ArrayList<Point> path = pathFinder.calculatePath();
        assertEquals(expectedPath.size(), path.size());

        for (int x = 0; x < path.size(); x++) {
            assertEquals(expectedPath.get(x), path.get(x));
        }
    }
```

*Code snippet 30 - Test for A* algorithm*
*File: OSGiMap/src/test/java/dk/sdu/mmmi/cbse/map/MapTest.java*

**Expected result:**

The *calculatePath* method returns, not just a usable path, but the shortest path.

## 7.5.4. Path to direction

**File:** OSGiEnemy/src/test/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystemTest.java

**Type:** Unit Test

**Preconditions:** None

**Test steps:**

1. Mockito mocks have been created
2. An EnemyControlSystem instance is created
3. It is given a Mockito mock of IMap
4. Map is set to return (0,0) when prompted for the center of any tile, as the return value is irrelevant to the test.
5. A list of Points is created.
6. The method *getPathDirectionStack* is called on the EnemyControlSystem instance with the list
7. It is verified that the returned stack of directions is the right length
8. It is verified that each direction is correct.

```
@Test
public void testPathToDirectionConversion(){
    EnemyControlSystem enemyControlSystem = new EnemyControlSystem();
    enemyControlSystem.setIMap(map);
    when(map.getTileCenter(any())).thenReturn(new Point(0,0));

    ArrayList<Point> input = new ArrayList<>();

    input.add(new Point(0,0));
    input.add(new Point(1,0));
    input.add(new Point(1,1));
    input.add(new Point(0,1));

    Stack<PathDirection> output = enemyControlSystem.getPathDirectionStack(input);

    assertEquals(3, output.size());
    assertEquals(output.pop().getDirection(), PositionPart.right);
    assertEquals(output.pop().getDirection(),PositionPart.down);
    assertEquals(output.pop().getDirection(),PositionPart.left);
}
```

*Code snippet 31 - Test of PathToDirectionConversion*
*file: OSGiEnemy/src/test/java/dk/sdu/mmmi/cbse/enemy/EnemyControlSystemTest.java*

**Expected result:**

The method *getPathDirectionStack* maps a list of points, corresponding to tiles on the map, into directions for the Enemy to follow

## 7.6. Health system

**Requirements:**

F3.5 A Player must have a health system

F4.2 An Enemy must have a health system

**File**: OSGiCommon/src/test/java/dk/sdu/mmmi/cbse/common/LifePartTest.java

**Type**: Unit test

**Description:**

Both Player and Enemy use LifePart as their health system; both are therefore tested through the LifePart test.

LifePart is altered by collision, see 7.9

**Preconditions:** None

**Test steps:**

1. Create an entity and a LifePart
2. Check that LifePart's *getter* method returns the expected value
3. Update the LifePart value using the *setter* method
4. Add LifePart to an entity
5. Get LifePart from the entity and check that the value is as expected
6. Update LifePart's value through the entity

```
@Test
public void lifePartTest(){
    Entity entity = new Entity(new Sprite(), Types.PLAYER);
    LifePart lifePart = new LifePart(0);

    assertEquals(0,lifePart.getLife());

    lifePart.setLife(1);

    assertEquals(1,lifePart.getLife());

    entity.add(lifePart);

    assertEquals(1, ((LifePart) entity.getPart(LifePart.class)).getLife());

    ((LifePart) entity.getPart(LifePart.class)).setLife(2);

    assertEquals(2, ((LifePart) entity.getPart(LifePart.class)).getLife());
}
```

*Code snippet 32 - Test of lifepart*
*File: OSGiCommon/src/test/java/dk/sdu/mmmi/cbse/common/LifePartTest.java*

**Expected result:**

The LifePart *getter* method should return the value set by the LifePart *setter* method.

## 7.7. Tower shooting

**Requirement:** F5.1**:** A Tower must be able to shoot at enemies

**File:** OSGiTower/src/test/java/TowerTest.java

**Type:** Unit test

**Description:**

The aim of the test is to verify if the *createProjectile* method is called on the projectile SPI. This is done when the *process* method is called in the TowerControlSystem, and an enemy is within the towers range.

**Preconditions:**

An instance of ProjectileSPI.

An instance of Tower and Enemy, with a PositionPart containing the point (100,100)

A WeaponPart with a range of 30 for tower.

**Test Steps:**

1. Prepare test data, with mocks of entities described in preconditions
2. Create an instance of TowerControlSystem and inject the instance of ProjectileSPI
3. The *process* method of TowerControlSystem is called with the test data.
4. Verify the *createProjectile* method is called

**Expected result:**

As the tower and enemy instance is using the same PositionPart, the enemy is within the range of the tower, so if the *createProjectile* is called, the test should pass.

## 7.8. Continues path in map

**Requirement:** F6.1 A Map must have a fixed path through the board

**File:** OSGiMap/src/test/java/dk/sdu/mmmi/cbse/map/PathTest.java

**Type:** Unit test

**Description:**

The map must have a continuous path from start to end for the enemies to move on.

This has been tested with a unit test

**Preconditions:**

The map is a tmx file.

The following ids must be used to denote tile type: ‘start = 5’, ‘path = 4’ and ‘end = 2’.

The map only has one start tile and one end tile.

The map must be a rectangle.

```
// While there is still unexplored path, look for adjacent paths
while (!unexploredPath.isEmpty()) {
    // Initialize current point to the first point in unexplored path
    final Point currentPoint = unexploredPath.get(0); // Breadth First
    final int x = currentPoint.x;
    final int y = currentPoint.y;

    // currentPoint is about to be explored, add it to explored path
    exploredPath.add(currentPoint);

    // Ternary statements to stay inside mapArray boundaries
    for (int dx = (x > 0) ? -1 : 0; dx <= ((x < mapArray.length - 1) ? 1 :
0); ++dx) {
        for (int dy = (y > 0) ? -1 : 0; dy <= ((y < mapArray[0].length - 1) ?
1 : 0); ++dy) {
            // Makes sure points diagonal to currentPoint is not explored
            if (dx == 0 ^ dy == 0) {
                final Point adjacentPoint = new Point(x + dx, y + dy);
                //Checks if adjacentPoint is a path and that it has not
already been explored
                if (mapArray[adjacentPoint.x][adjacentPoint.y].equals(path)
&& !exploredPath.contains(adjacentPoint)) {
                    unexploredPath.add(adjacentPoint);
                }
                // Checks if the adjacentPoint is the end point
                if (mapArray[adjacentPoint.x][adjacentPoint.y].equals(end)) {
                    // Adds it directly to exploredPath because there is no
reason to explore the end
                    exploredPath.add(adjacentPoint);
                }
            }
        }
    }
    // currentPoint has just been explored, remove it from unexplored path
    unexploredPath.remove(currentPoint);
}
```

*Code snippet 33 - the while loop where adjacent points are explored*
*File: OSGiMap/src/test/java/dk/sdu/mmmi/cbse/map/PathTest.java*

**Test steps:**

1. Read the map from the tmx file into a 2D array
2. Verify that it has a start and end position and create them as Points.
3. Assert that both points are not null.
4. Create lists containing Points to hold explored- and unexplored path
5. Add the start point to the unexplored list.
6. Loop through the list of unexplored points, set the current point to the first element in the list, and add it to the list of explored points.
7. Check if points adjacent to the current point are of type path, if they are, add them to unexplored points, unless they already are in the explored points.
8. Check if the adjacent point is the end point, if it is, add it to explored points.
9. Remove the current point from the unexplored list.
10. Check that the last element in explored points is equal to the endpoint.

**Expected result:**

The last point in the list of explored points is the end point.

## 7.9. Collision detection

**Requirement:** F7.1 A Collision system must be able to detect collisions between the player, enemies, and projectiles

**File:** OSGiCollision/src/test/dk/sdu/mmmi/cbse/collision/CollisionManagerTest.java

**Type:** Unit test

**Description:** There must be a collision system which can detect the collision between entities, and furthermore, it handles the subtracting of life from LifePart. There are created different cases where the entities are same object, same type etc., to ensure the system is running as expected.

**Preconditions:** There are two entities in the world in the same place, and i.e., colliding.

```
…
when(entity2.getRadius())
        .thenReturn(5f);
collisionManager.process(gameData, world);
LifePart lifePartEntity1 = entity1.getPart(LifePart.class);
LifePart lifePartEntity2 = entity2.getPart(LifePart.class);
assertEquals(4, lifePartEntity1.getLife());
assertEquals(1, lifePartEntity2.getLife());
verify(world, atLeast(1)).getEntities();
verify(entity1, atLeast(2)).getPart(any());
verify(entity2, atLeast(2)).getPart(any());
…
```

*Code snippet 34 - Subtracting life from entities that collides*
*File: OSGiCollision/src/test/java/dk/sdu/mmmi/cbse/collision/CollisionManagerTest.java*

**Test Steps:**

1. Relevant mocks are created, and hooking is done
2. The collisionManager is instantiated
3. The *process* method is called on the instantiated collisionManager with the mock gameData and mock world.
4. It should handle the collision
5. The lifeparts of the entities are checked to verify that they lost a life; they were 5 and 2 before.
6. It is checked that the expected methods are called

There are cases for normal collision, no collision, same type, collision with tower, collision between player and projectile, and when an entity dies.

**Expected result:** The entities life is decremented and thereby the processing of collision has been done.

## 7.10. Generating projectiles

**Requirement:** F8.1 A Projectile must be able to be generated

**File:** OSGiProjectile/src/test/java/ProjectileTest.java

**Type:** Unit test

**Description:** It must be possible for entities equipped with a WeaponPart to generate new projectiles.

**Preconditions:** An Entity must have a WeaponPart and a dependency on ProjectileSPI.

**Test steps:**

1. Prepare the necessary mocked objects, including Entity, PositionPart, WeaponPart, Texture, and World, as well as instantiate a ProjectileControlSystem object.
2. Ensure valid coordinates and angle are returned, when *getX*, *getY*, and *getAngle* is called on the mocked PositionPart.
3. Ensure the mocked PositionPart and WeaponPart are returned, when *getPart* is called on the mocked Entity
4. Create a HashMap containing the Projectile type and the mocked Texture, and ensure this HashMap is returned, when *getTextureHashMap* is called on the mocked World
5. Call the *createProjectile* method on the instantiated ProjectileConstrolSystem, using the mocked Entity, GameData, and World objects
6. Verify that a Projectile has been added to the mocked World.

**Expected result:** If the verification confirms that a single Projectile entity has been added to the World object, the test should pass. If, for some reason, the verification shows that zero or more than one Projectile is added to the World, the test will fail.

## 7.11. Projectile movement

**Requirement:** F8.2 A Projectile must be able to move in a straight line with a defined speed

**File:** OSGiProjectile/src/test/java/ProjectileTest.java

**Type:** Unit test

**Description:** The test will verify that a projectile moves in a straight line from a specific point at a defined speed. Both movements along the x and y axis will be asserted. The movement of a projectile is handled when the *process* method is called in the ProjectileControlSystem.

**Preconditions:** An instance of a Projectile, with a position at (10,10) and angle of movement at 0 with the speed of 6 is created, along with mocks of World, GameData, IMap.

**Test Steps:**

1. Prepare test data, with mocks of entities described in preconditions
2. Add the Projectile to the World
3. Create an instance of ProjectileControlSystem and inject the Map into it
4. Call the *process* method of ProjectileControlSystem
5. Assert that the sum of the original x-value and the speed matches the current x-value.

6. Change the direction of movement to 90.
7. Call the *process* method of ProjectileControlSystem
8. Assert that the sum of the original y-value and the speed matches the current y-value.

**Expected result:** As the angle of movement is 0, it is expected that the projectile moves along the x-axis, increasing the x-value with the given speed of 6, when the *process* method is called. When angle is changed to 90 at step 6, the projectile will follow the y-axis and should increase by 6.

## 7.12. Projectile out of bounds

**Requirement:** F8.3 A Projectile must die when it reaches the end of the map

**File:** OSGiProjectile/src/test/java/ProjectileTest.class

**Type:** Unit test

**Description:** The projectile should disappear once it reaches the boundaries of the map.

**Preconditions:** The projectile exists and is outside the map/at the boundaries.

```
@Test
public void TestProjectileOutsideMap(){
    ArrayList<Entity> projectiles = new ArrayList<Entity>(){{
        add(projectile);
    }};
    when(projectile.getPart(MovingPart.class)).thenReturn(movingPart);

when(projectile.getPart(PositionPart.class)).thenReturn(positionPartMock);
    when(projectile.getPart(WeaponPart.class)).thenReturn(weaponPartMock);
    when(worldMock.getEntities(Projectile.class)).thenReturn(projectiles);
    when(mapMock.isInsideMap(anyFloat(), anyFloat())).thenReturn(false);
    projectileControlSystem.process(gameDataMock, worldMock);
    verify(worldMock, times(2)).removeEntity(projectile);
}
```

*Code snippet 35 - Test when projectile is outside map*
*File: OSGiProjectile/src/test/java/ProjectileTest.java*

**Test steps:**

1. Mocks are created
2. Projectile are added to a list that is used in the mocking
3. Hooking into the execution of the method
4. Ensuring that the *isInsideMap* returns false
5. Call the *process* method on the ControlSystem that is instantiated
6. Verify that the world removes the entity

It is checked that the projectile is called to be removed two times, since in this case it reaches both the bottom and top of the map and should be removed in either case, as can be seen on Code snippet 35.

**Expected result:** The projectile is removed from the world.

## 7.13. Game initialization

**Requirement:** F.9.1 The core must initialize and run the game

**File:** OSGiCore/src/test/java/GameTest.java

**Type:** Unit test

**Description:** The aim of the test is to ensure that the game is initialized. Since this is done by the LibGDX framework, it is not possible to test it directly. LibGDX is a well-tested and well-documented framework, and therefore, there is no need to test it. The method simply tests that the game gets registered as the LibGDX ApplicationListener. Furthermore, it is to be tested that it updates the game flow, which is handled by LibGDX, and therefore not tested either.

**Preconditions:** The game can be instantiated

```
@Test
void gameInitializedUpdated(){
    Game game = new Game();
    assertEquals(game, Gdx.app.getApplicationListener());
}
```

*Code snippet 36 - Testing that the game instance is found by LibGDX*
*File: OSGiCore/src/test/java/GameTest.java*

**Test steps:**

1. The game is instantiated
2. It is asserted that the instantiated game is the one LibGDX uses, see Code snippet 36

**Expected result:** The LibGDX framework picks up the game since it implements ApplicationListener, and thereby, LibGDX should see this as the game. Thereby, the method is expected to return the game. I.e., the game is initialized and LibGDX also updates the game flow automatically.

## 7.14. Results

When the game is run, all the tests run successfully, since it would stop the execution otherwise. It can therefore be concluded that the game behaves as expected. If bugs are introduced, it would stop the execution of the game. Figure 14 shows that the tests are passed, since the build succeeded, and thereby each component is built correctly. I.e., all the tests have passed.

```
[INFO] OSGiLibGDX ..................................... SUCCESS [  1.675 s]
[INFO] OSGiCommonMap .................................. SUCCESS [  0.329 s]
[INFO] OSGiCommon ..................................... SUCCESS [  4.505 s]
[INFO] OSGiFileHandler ................................ SUCCESS [  0.217 s]
[INFO] OSGiCore ....................................... SUCCESS [  4.086 s]
[INFO] OSGiCommonPlayer ............................... SUCCESS [  0.205 s]
[INFO] OSGiCommonTower ................................ SUCCESS [  1.070 s]
[INFO] OSGiPlayer ..................................... SUCCESS [  3.447 s]
[INFO] OSGiCommonEnemy ................................ SUCCESS [  0.158 s]
[INFO] OSGiEnemy ...................................... SUCCESS [  2.642 s]
[INFO] OSGiCommonProjectile ........................... SUCCESS [  0.187 s]
[INFO] OSGiTower ...................................... SUCCESS [  2.863 s]
[INFO] OSGiProjectile ................................. SUCCESS [  2.909 s]
[INFO] OSGiCollision .................................. SUCCESS [  2.366 s]
[INFO] OSGiMap ........................................ SUCCESS [  1.932 s]
[INFO] OSGiTest ....................................... SUCCESS [  3.983 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  34.077 s
[INFO] Finished at: 2022-05-25T08:56:17+02:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

*Figure 14 - Test results*

# 8. Discussion

The following sections discuss different aspects that could be improved or done different, but also whether the requirements were fulfilled, and how further maintenance could be done.

## 8.1. Requirements

The project solved the problem of dynamic unloading and loading by using the Felix gogo shell. It is thereby possible to install new modules by moving them into the bundles folder, after which loading/unloading is possible. It could be possible by a graphical user interface, but this was not seen as something that would contribute to the final product, and not a requirement, and therefore, this has not been implemented. The requirement about unloading/loading is fulfilled.

The Must-have and Should-have requirements have been fulfilled according to the tests. There are some Could-have and Will-Not-Have requirements which can be fulfilled if future development of the game would happen. Components can also easily be changed by just providing the service-provider interface, which is the essence of component frameworks. There are a lot of opportunities to extend the game; for example, different types of enemies and towers could be added, more advanced levels etc.

The A* AI is implemented, and it is documented using algorithms. Furthermore, algorithms and data structures are used for the min-heap, and therefore, these requirements are also fulfilled.

## 8.2. Test

The tests should be extended to support more cases. Generally, it should follow boundary cases, and tests should be made for a lot more cases. This was not done in all cases because of the time constraints. It would give a higher code quality and ensure better verification. Furthermore, when changes are made, it would ensure that the system would not break down, and the bugs would be caught before the game was even started. Many of the cases are tested though, and a lot of bugs would be caught. It is mostly edge cases that could potentially break the game.

## 8.3. Component framework

It would be more relevant to use a component framework like Spring, which is much more used in the industry and much easier to write. Generally, the code is much cleaner. This was not possible since one of the requirements for the project was that it must be possible to dynamically unload and load components, which Spring does not allow. An argument could be made that this dynamic loading and unloading is not relevant to the performance and playability of the game itself, it is mainly a learning objective that the game must fulfill.

Another component framework that could be relevant is the java-modules framework that came with JDK 9. This was not an option to choose in the project, as it also does not support dynamic unloading and loading, but could be more relevant in terms of gaining experience with newer frameworks. Additionally, it would solve many of the problems that had to be overcome in the start of the process; for example, correct dependencies of OSGi were a major issue.

An old version of LibGDX with newer Java libraries caused problems, and the OSGi framework together with LibGDX bound it to be on JDK 1.8. I.e., it is very outdated, and should be updated for better maintenance and to protect against deprecation and old bugs that are fixed in newer versions.

Furthermore, an issue which has given a lot of problems is that LibGDX is not designed to be used for components. It is based on a monolithic game, which is why there were many issues with using LibGDX over many components. The time could therefore have been spent elsewhere to improve testing etc. Another game framework could have been used, but this was not done, since the only game framework introduced in the lectures has been LibGDX. To not spend too much time learning another game framework, which is not the goal of the semester project, LibGDX was used.

## 8.4. Artificial Intelligence

Arguments could be made that using artificial intelligence for this game is overkill, at least in the way it is utilized in this project. The pathfinding algorithm in the Enemy component could easily be replaced by a path attribute in the Map component and have the Enemy access this during runtime. This would increase the requirements for any future alternative Map components, but this path attribute would be a part of the map interface, which any new Map component would need to adhere to in any case.

By using a fixed path registered in the Map component, the game would need less resources in terms of processing power, as the number of calculations needed will drastically decrease.

Alternatively, artificial intelligence could be made more relevant by changing the structure of the Map component. If, instead of using a fixed map, the map was randomly generated each time the Map component is started, using a pathfinding algorithm would make more sense, as there would not be a predefined path to store in the Map component.

Another way to use artificial intelligence and maintain a fixed map, would be to change the layout of the map. If, instead of the enemy being restricted to only walk on path tiles, the enemy could walk on any tile, except barrier tiles, there would not be one single correct path for all enemies to take. Here different heuristics could be introduced, to influence a singular enemy's path, such as distance to end tile, distance to player, and distance to towers, making it possible for an enemy to find either the shortest or the least dangerous path.

There is a long-term opportunity to use artificial intelligence to optimize the heuristics for the towers' selection of enemies. After each game, it could check how many enemies were killed or came through the map, and use that data to adjust the weights of the different parameters, deciding which enemy to target. This could be taking the one with the most health, the enemy closest to the end etc. Thereby, the tower would get increasingly more intelligent for each game played.

## 8.5. Maintainability

Since OSGi is used, it is easy to maintain the system, if it would be running all the time. If a module is not working as expected, it can simply be unloaded, after which a new module could be installed and started. This can all happen while the application is running. Therefore, maintainability is easier, which is one of the primary reasons to use OSGi. Updates to components can be done in the same manner. Simply removing the old component and inserting the new component will be all it takes to update.

It is simple to develop new components to replace older ones, since they would just have to implement the service-provider interfaces already defined. Thereby, further development is made easy, and changing many of the components will not ruin the game.

The game can continue to operate, even when some components are unloaded. The functionality of an unloaded component is of course not present, but it is still possible for the game to continue operating without failures, as it is checked if service-provider interfaces are provided before each use.

## 8.6. Importance of predefined interface contracts

During development it was decided that the map had to be a separate component instead of included in the Common component, where the GameData class would contain the map. This was changed due to the requirements, specifically F2, and the possible flexibility of being able to change the map.

Since this was done in the middle of the development, it made the interface contract unstable. This made it contain too many methods and made it complex to refactor and develop, which shows how important it is to have good, predefined contracts. A lot of code was duplicated, which then had to be refactored. Had the interface contract been stable from the beginning, it would have made the development much easier, as with the other interface contracts.

# 9. Conclusion

A functional Tower Defense 2D game has been created containing all the required components; Map, Enemy, Player, Tower, Projectile, and Core as derived from the project description in section 2.

Artificial intelligence has been used for finding a path on the map using the A* algorithm as seen in section 4.3, and further AI could be used to optimize which enemy a tower is targeting, as described in section 8.4.

The use of a priority queue data structure implemented as a min-heap, used for towers selecting its target, has been described in section 5.3.2

Interface-oriented design methods has been used to make the system flexible and extendable, with contracts specified for all interfaces used in the system, as seen in section 5.2

The OSGi component framework has been chosen due to its dependency injection feature and ability to load and unload components during runtime, which is possible for the Player, Enemy and Projectile components, see section 5.1. This makes the system easy maintainable for further development.

All functional must-have requirements concerning the gameplay described in section 2.1, have been implemented and the game is working as intended.

Automatic tests have been created for all must-have requirements, which all passes, however most tests are testing “happy day” scenario and more tests including boundary cases should be added.

Future work on the game could be implementing the extra features mentioned in the Functional requirements section 3.2, which are prioritized as should- and could-haves, e.g., create maps dynamically or make enemies able to attack the player and the towers.

# 10. Bibliography


[1] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Pearson, 2010.

[2] J. C. Américo, W. Rudametkin and D. Donsez, "Managing the dynamism of the OGSi Service Platform in Real-Time Java Applications," *Proceedings of the ACM Symposium on Applied Computing,* March 2012.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third Edition, MIT Press, 2009.

[4] J. Johnson, "FlatRedBBall: Circle Collision," FlatRedBall, 2016. [Online]. Available: https://flatredball.com/documentation/tutorials/math/circle-collision/. [Accessed 18 March 2022].

[5] LibGDX, "Spritebatch, Textureregions, and Sprites," LibGDX, 9 March 2022. [Online]. Available: https://libgdx.com/wiki/graphics/2d/spritebatch-textureregions-and-sprites. [Accessed 14 March 2022].

[6] libGDX, "Event handling," libGDX, 22 May 2022. [Online]. Available: https://libgdx.com/wiki/input/event-handling. [Accessed 24 May 2022].

[7] libGDX, "The life cycle," libGDX, 22 May 2022. [Online]. Available: https://libgdx.com/wiki/app/the-life-cycle. [Accessed 24 May 2022].

[8] libGDX, "The application framework," libGDX, 5 May 2202. [Online]. Available: https://libgdx.com/wiki/app/the-application-framework. [Accessed 24 May 2022].

# Appendix A IMap interface specification

*Table 29: full interface contract for IMap*

| **IF6: IMap** | |
|---|---|
| **Operation** | ***getTiledMap()*** |
| Description | used to get the TiledMap instance |
| Parameters | - |
| Preconditions | - |
| Postconditions | returns a TiledMap or null |
| **Operation** | ***setTiledMap*(Tiledmap tiledMap)** |
| Description | sets the tiledmap, which the component class will use, and calculates the path from start to end tile |
| Parameters | tiledMap: a map object from the libgdx framework |
| Preconditions | - |
| Postconditions | the map used in the component is set |
| **Operation** | ***getTileType*(int x, int y)** |
| Description | returns tile property based on map coordinates |
| Parameters | x,y: map coordinates |
| Preconditions | a tiled map exists |
| Postconditions | the tile property has been returned |
| **Operation** | ***getTileTypeByCoor(int x, int y)*** |
| Description | returns tile property based on cell coordinates |
| Parameters | x,y: cell coordinates on map |
| Preconditions | a tiled map exists |
| Postconditions | the tile property has been returned |
| **Operation** | ***tileCoorToMapCoor*(float x, float y)** |
| Description | Converts tile coordinates to map coordinates |
| Parameters | x, y: tile coordinates |
| Preconditions | a tiled map exists |
| Postconditions | map coordinates have been returned |
| **Operation** | ***getStartTileCoor()*** |

| | |
|---|---|
| Description | Finds coordinates of the start tile of path |
| Parameters | - |
| Preconditions | the map must contain a path |
| Postconditions | the cell coordinates of the start tile have been returned |
| **Operation** | ***getEndTileCoor()*** |
| Description | Finds coordinates of the end tile of path |
| Parameters | |
| Preconditions | the map must contain a path |
| Postconditions | the cell coordinates of the end tile has been returned |
| **Operation** | ***getTilesOfType*(String property)** |
| Description | Finds tiles with a given property |
| Parameters | property: a string containing the wanted property |
| Preconditions | A map must exist |
| Postconditions | returns a list of tiles with the given property |
| **Operation** | ***getLayers()*** |
| Description | Used to get the layers of the TiledMap |
| Parameters | - |
| Preconditions | a Tiled ap must have been set |
| Postconditions | the layers of the TiledMap have been returned |
| **Operation** | ***getPath()*** |
| Description | Returns the path through the map from start tile to end tile |
| Parameters | - |
| Preconditions | a Tiledmap including a path must have been created |
| Postconditions | Returns an array containing the coordinates the path |
| **Operation** | ***changeTileType*(int x, int y, String tileType)** |
| Description | Used to change the tileType at a specific cell coordinate |

| | |
|---|---|
| Parameters | x,y: cell coordinates<br>tileType: the tile property |
| Preconditions | a TileMap must exist |
| Postconditions | A tile has changed it's tileType |
| **Operation** | ***getTileSize()*** |
| Description | Used to get the pixelsize of individual tiles |
| Parameters | - |
| Preconditions | a TileMap must exist |
| Postconditions | Returns the size of tiles in the map |
| **Operation** | ***mapCoorToTileCoor*(float x, float y)** |
| Description | Converts map coordinates to tile coordinates |
| Parameters | x, y: map coordinates |
| Preconditions | a tiled map exists |
| Postconditions | tile coordinates have been returned |
| **Operation** | ***getMapHeight()*** |
| Description | returns the height of the map in pixels |
| Parameters | - |
| Preconditions | a tiled map exists |
| Postconditions | the height has been returned |
| **Operation** | ***getMapWidth()*** |
| Description | returns the width of the map in pixels |
| Parameters | - |
| Preconditions | a tiled map exists |
| Postconditions | the width has been returned |
| **Operation** | ***getTileCenter*(Point point)** |
| Description | calculates the centre of a tile in map coordinates |
| Parameters | point: contains cell coordinates of a Tile |
| Preconditions | a TiledMap must exist |
| Postconditions | returns map coordinates for the centre of the tile |
| **Operation** | ***isInsideMap*(float x, float y)** |

| | |
|---|---|
| Description | checks if a given point is inside or outside the map |
| Parameters | x,y: coordinates of point to verify |
| Preconditions | a TiledMap must exist |
| Postconditions | returns “true” if point is inside map otherwise “false” |