



Defend your stuff

Component Based Software Engineering

Software Engineering 4. semester project

Semester project group 8

Alexander Nytofte Markussen

[alexm15@student.sdu.dk](mailto:alexm15@student.sdu.dk)

---

Joachim Hejlesen

[johej15@student.sdu.dk](mailto:johej15@student.sdu.dk)

---

Mads Berggreen

[madbe15@student.sdu.dk](mailto:madbe15@student.sdu.dk)

---

Martin Fabricius

[mfabr14@student.sdu.dk](mailto:mfabr14@student.sdu.dk)

---

Karim Møller

[karmo15@student.sdu.dk](mailto:karmo15@student.sdu.dk)

---

Sebastian Bjerre Jørgensen

[sejoe13@student.sdu.dk](mailto:sejoe13@student.sdu.dk)

---

Project Advisor

Daniel Bjerring Jørgensen

[dbj@mmpi.sdu.dk](mailto:dbj@mmpi.sdu.dk)

Project Period

07.02.2017 - 23.5.2017

# 1. Abstract

Currently one of the most common ways of game development is monolithic game development. This development style comes with a variety of issues, such as worse maintainability, code readability and a limit to the possibility of further development. As a solution to these issues, alternate development styles are explored, one of these are component based game development.

When developing games with a monolithic design, some of the classes and methods are often long unreadable walls of code. This makes the game nearly impossible to maintain later, this is especially the case if the team doing the maintenance is not the team who started developing the game. In many cases the team who did the development might even have difficulties understanding the code after a while. Therefore further development is difficult and time consuming, this makes for a weak future for the game. Especially in game development the possibility for further development and maintenance is crucial, because the requirements of the user changes rapidly. A game must be developed in a way that allows for easy changes and updates, monolithic developed games hardly falls in that category.

DefendYourStuff is developed using an Entity System Framework. This makes DefendYourStuff a game developed with a collection of components. These components only include the minimal amount of information, and the entities are placed in a common library and several sub-common libraries. These sub-common libraries are connected through inheritance, and this ensures that the entities contain only the necessary amounts of data. However because of the inheritance these common libraries each have dependency depth with the largest being a depth of five components.

The communication between the components is mainly done through SPIs, the four interfaces required by GameEngine are responsible for ensuring the core of the game operations. While the variety of component specific interfaces are responsible for the necessary communication between the components.

Not everything in the solution are optimal as there are other implementations that might solve the problem in a better way. For example a different and more advanced solution can be considered for the systems AI. Likewise there are multiple design problems such as fragile base class and leakage of implementation details. These are only part of the problematical subjects that have been discovered during development of the system, and not all have been discovered.

Despite the known issues with the implementation, the system that have been developed solves most of the problems, that the monolithic style of software development present. The system solved most of the issues by using components that depend only on service provider interfaces, with specified pre- and postconditions. This will then allow for a better maintainability as a whole. Each contract specified in the system also provides easier ways for replacing current components with alternative implementations, without redeploying the

complete system. Another quality the solution adds, is for updating parts of a system without restarting the system. This is possible as components can be loaded and unloaded individually at runtime.

# Table of Contents

<b>1. Abstract</b>	<b>2</b>
<b>Table of Contents</b>	<b>4</b>
<b>2. Introduction</b>	<b>7</b>
2.1 Game description	7
2.2 Git repository	7
2.2.1 Compiling with SilentUpdate	8
2.2.2 Running the game via the executable	9
2.3 Demo video of the game	9
<b>3. Requirements</b>	<b>10</b>
3.1 Functional requirements	10
3.2 Non-functional requirements	11
<b>4. Analysis</b>	<b>11</b>
4.1 Entity System Framework	11
4.1.1 Advantages:	12
4.1.2 Disadvantages:	12
4.2 The Common Library	12
4.3 Interfaces	14
4.3.1 Interface table	15
4.4 Component responsibility	16
<b>5. Design</b>	<b>17</b>
5.1 Sub libraries of the Common library	17
5.2 Components of the system	18
5.3 Action interfaces	18
5.4 Component-specific SPIs	19
5.5 Class Inheritance and dependency depth	20
5.5.1 Class dependency depth	21
<b>6. Implementation</b>	<b>22</b>
6.1 Map generation	22
6.1.1 The Chunk	24
6.2 LibGDX render functionality	25
6.3 Player implementation	25
6.3.1 Player movement	25
6.3.2 Player abilities	26
6.3.3 Player weapon	27
6.4 Enemy implementation	28

6.4.1 The spawning of enemies	28
6.4.2 The Behavior of Enemies	30
6.5 Collision detection	32
6.6 SilentUpdate	33
<b>7. Tests</b>	<b>34</b>
7.1 Exception Handling at creation	34
7.2 GUI Testing	34
7.3 Unit Testing	34
7.4 Interface testing	35
7.5 Integration Testing	35
<b>8. Discussion</b>	<b>36</b>
8.1 Current AI vs more advanced AI	36
8.2 Leakage of implementation details between components	37
8.3 Fragile base class	39
8.4 Ability Component	40
8.5 Pros vs. cons for different component models	41
8.5.1 Netbeans Module System	41
8.5.2 OSGi	41
<b>9. Conclusion</b>	<b>42</b>
9.1 Future work	43
<b>10. References</b>	<b>44</b>
<b>Appendix A: Dictionary</b>	<b>45</b>
Video Game terminology	45
System terminology	46
<b>Appendix B: Game manual</b>	<b>47</b>
Keyboard input:	47
Goals:	47
Instructions:	47
<b>Appendix C: Tables</b>	<b>49</b>
C.1 - All functional requirements	49
C.2 Component dependency depth	51
<b>Appendix D: Diagrams</b>	<b>54</b>
D.1 IGameProcess sequence diagram	54
D.2 IGamePlugin sequence diagram	54
D.3 IGamePostProcess sequence diagram	54
D.4 IGamePreStartPlugin sequence diagram	55
D.5 Entity inheritance diagram	56
D.6 useAbility sequence diagram	56

D.7 Ranged enemy behaviour tree	58
<b>Appendix E: Pre/Postconditions</b>	<b>59</b>

## 2. Introduction

This chapter describes the idea and high level features of the game. It also gives a guide on how to run the game either by building the project or running an executable file. A link to all demo videos have also been included.

### 2.1 Game description

“DefendYourStuff, is a sidescroller with RPG elements. The goal of the game is to protect your castle against the never ending horde of enemies. To accomplish this, build and strengthen your kingdom, enlist strong allies and equip epic weapons containing powerful abilities. Kill your enemies, earn gold, then spend this on amazing upgrades that make it even easier to defend your stuff”. This is an outline for how the game description would have been, if the game would be completed.

Right now, enemies spawn from both sides of the castle where the player can try to defend it for as long as possible. The player can change between two weapons; one containing a ranged fireball ability, and the other containing a melee slash ability. When enemies are slain, the player receives gold. The more gold the player earns before he dies or the castle is destroyed, the better the score. The game ends when either the castle or player dies.

When developing software systems, a big concern is reusability and maintenance. In order to address both problems, this game is designed using component-based software engineering practices. These components communicate exclusively through interfaces, and various shared components. These shared components uses the naming convention “Common”+”SomeName”. By using these components where the dependencies are low, it is easier to maintain components without having to worry about other components. These components can also easily be modified and used in another system. There are also these predefined requirements where some components should be able to be loaded and unloaded during runtime.

To create these different components, the Netbeans Module System (NBM) has been used together with Maven. Here Maven has created the Project Object Model (POM) file, that handles various issues about the code. These issues include the dependencies the different components have.

### 2.2 Git repository

The Github repository for the system implementation, is on the following link:  
<https://github.com/alexm15/DefendYourStuff>.

From the link it is possible to either download the complete project as a zip file, or clone the project to one's favorite IDE. It is recommended to use the Netbeans IDE to view the code, as this is the IDE used for the development of the project.

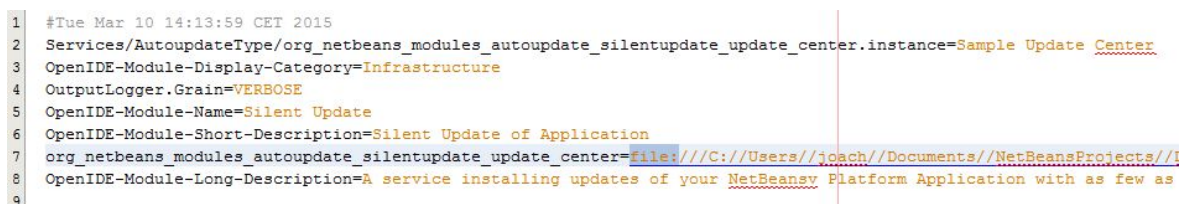
When opening the project, either via the downloaded zip or the cloned git project, the Maven parent project has each component of the project in the module folder. If you choose to run the game, there are two ways to do this; the first way is to build the update center, then compiling the code with the SilentUpdate module; the second way is to run the game via the included *DefendYourStuff-app-1.0.0-SNAPSHOT.zip* folder, which contains an executable file.

## 2.2.1 Compiling with SilentUpdate

Before you are able to clean and build the project, remember to skip unit tests in the Netbeans IDE. This is due to the ApplicationTest being set up with a specific file path. To skip unit tests when cleaning and building the project, go to Tools → Options → Java → Maven tab, and then check “Skip Test for any build executions not directly related to testing”. Now you should be able to clean and build the DefendYourStuff-parent project, to download all necessary dependencies.

The current project setup represent all code implemented with the SilentUpdate module, for dynamic loading and unloading of modules. In order to use this module the file path for the update center needs to be updated, as the file path is specific to your own system. It is in the *Bundle.properties* file of the SilentUpdate module, at line 7 (see figure 2.2.1), that the file path to the update center should be updated. It is located at the following location:

[drive chosen]:\[directory location chosen]  
\\DefendYourStuff\\DefendYourStuff\\SilentUpdate\\src\\main\\resources\\org\\netbeans\\modules\\autoupdate\\silentupdate\\resources



```
1 #Tue Mar 10 14:13:59 CET 2015
2 Services/AutoupdateType/org_netbeans_modules_autoupdate_silentupdate_update_center.instance=Sample Update Center
3 OpenIDE-Module-Display-Category=Infrastructure
4 OutputLogger.Grain=VERBOSE
5 OpenIDE-Module-Name=Silent Update
6 OpenIDE-Module-Short-Description=Silent Update of Application
7 org_netbeans_modules_autoupdate_silentupdate_update_center="file:///C:/Users/joach/Documents/NetBeansProjects/DefendYourStuff-App/src/main/resources/org/netbeans/modules/autoupdate/silentupdate/resources"
8 OpenIDE-Module-Long-Description=A service installing updates of your NetBeans Platform Application with as few as
9
```

Figure 2.2.1: Shows the content of the *Bundle.properties* file. From the highlighted text and the rest of the line is where the file path of the *update.xml* file is to be specified for the given computer the game is ran on.

The update center file path should be written as:

file:///drive chosen]/[directory location chosen]  
//DefendYourStuff//DefendYourStuff/GameEngine/src/main/resources/netbeans\_site

Remember the usage of forward slashes instead of backward-slashes, and use double slashes for ensuring a correct path is used by the java program as mention in Oracle [1].

To build the update center, open the DefendYourStuff-App module and right click on it, choose Set Configuration and set it to Deployment, then clean and build the DefendYourStuff-App. This should create the *netbeans\_site* which need to be moved into



the file path set in the Bundle.properties as done previously. The netbeans\_site can be found in the following file path:

..\DefendYourStuff\DefendYourStuff\application\target\

Now set the configuration to 'Run' instead of 'Deployment' and compile the project.  
Now the program can be executed from the app.

## 2.2.2 Running the game via the executable

To run the game without SilentUpdate, unpackage the *DefendYourStuff-app-1.0.0-SNAPSHOT.zip* folder, and go to defendyourstuff/bin and run either the program defendyourstuff64.exe if your system is 64 bit, or the defendyourstuff.exe if your system is 32 bit. Look at figure 2.2.2.

Navn	Ændringsdato	Type
defendyourstuff	22-05-2017 15:08	Fil
defendyourstuff	22-05-2017 15:08	Program
defendyourstuff64	22-05-2017 15:08	Program

Figure 2.2.2. Shows the location of the 32 and 64 bit executable of the game.

## 2.3 Demo video of the game

There are three videos, one showing the final game, one showing the application test, and the last showing loading/unloading of components with SilentUpdate. These can be found in the following Google Drive folder:

<https://drive.google.com/drive/folders/0B6Mo6Uok0on9U0hqLVJFZkxCSVk>

## 3. Requirements

This section documents both the functional requirements and the non-functional requirements specified for the developed game.

### 3.1 Functional requirements

The functional requirements included in this snippet ensures the basic gameplay, most of these requirements incorporates core functionality crucial to the gameplay. However, many of these requirements constructs the foundation for the implementation of many other requirements, this is the reason why they were prioritized with a must priority.

This snippet of the table of requirements show most of the must requirements realized by the user stories in the game. All requirements have an ID, a name and a description. The requirements were furthermore put through a MoSCoW analysis and after that given story points. The full table of requirements can be found in Appendix C.1: All functional requirements.

ID	Name	Description	MoSCoW	Story Points
03	Player: Move direction	Player can move left and right, or up when jumping.	Must	5
04	Player: Aim with mouse	Locate the mouse in the screen grid and use that location as the current aim point.	Must	5
05	Player: Attack	Adds an ability to the world.	Must	3
13	Enemy: AI behavior	Defines the specific AI behavior for different types of enemies.	Must	40
14	Enemy: Attack	When an enemy ability collides with a building or the player, some damage is done.	Must	5
27	Weapon: Generate weapon	Generate the weapons that can be bought or found.	Must	8
29	Weapon: Abilities	Abilities that the weapon can use, and how the weapon empowers the abilities in certain ways.	Must	13
33	Building: Health	Buildings have health that can be updated when damage is done to the building.	Must	3
34	Building: Castle	A main castle building is created for the game with rules for the main objectives of the game, like the game over condition.	Must	2
35	Building: Buildings can be destroyed	When buildings health reaches zero they get destroyed and turns into a rubble building.	Must	2
42	Map: Generate map	Generate the rest of the map with blocks such as rubble and earth outside of the predefined chunks.	Must	21
43	Map: Predefined	A collection of blocks defined for a specific part of the map, that will work as templates to be chosen	Must	13

	"Chunks"	for generation of the map.		
44	Collision: Collision detection	Detect when two objects collide.	Must	13

## 3.2 Non-functional requirements

This table defines all the non-functional requirements for the game. Each of these non-functional requirements have no MoSCoW priority or story points assigned, because they are all mandatory for the project setup- and assignment given.

The non-functional requirements specifies some constraints that the system should adhere to.

ID	Name	Description
NF01	Component framework	The system must use the Netbeans Module System or OSGi as the component framework.
NF02	Game framework	The system must use libGDX as the game framework.
NF03	Mandatory components	The game has to include the following components: Player, Enemy, Weapon, GameEngine and Map.
NF04	AI course	The game is required to implement some AI techniques.
NF05	Algorithms and data-structures course	The game is required to implement some techniques from the algorithms and data-structures course.
NF06	Exception handling	Exception handling must be implemented, where game crashes can occur.
NF07	Loadable and unloadable modules	The following components must be loadable and unloadable during runtime: Player, Enemy, Weapon, Ability, Building, AI, Collision and DayNight.
NF08	Required input	The game must be playable with mouse and keyboard.
NF09	Coding language	The game has to be coded in Java.

In the requirement NF01 it is stated that the game must be developed in Netbeans Module System, this requirement was added due to the semester's course planning.

Lastly in the unload/load requirement NF07, that is satisfied, when all entities and functions are either removed when unloaded or reinstated when loaded.

## 4. Analysis

In this chapter, contracts for various components and interfaces is described, as well describing some diagrams about the overall architecture of the program.

### 4.1 Entity System Framework

DefendYourStuff is built on an Entity System Framework. This minimizes the amount of shared data present in the system. These entities are defined in their own components, and when an instance of the entity should be used, it is added to a world instance located in the

Common library. This enables the opportunity to search through the world for all entities. The entities data can be manipulated and used in a variety of components, with a variety of different methods [2].

Some of the advantages and disadvantages about the Entity System Framework are listed here.

#### 4.1.1 Advantages:

- Because there is no need to define all entity relationships at start of project, the system can easily be expanded according to needs.
- Easy to understand and contact the location of the systems data.
- Very agile way to develop code
- Possibilities for applying multithreading to program.

#### 4.1.2 Disadvantages:

- The list of entities can get very big, when the system then iterates through the list. If the number of entities is large, the total iteration cost can get very high [3], [4].

### 4.2 The Common Library

In the early stages the project was developed by using Enums, and these were used to share the necessary information between the components. However in the middle stages of the project it was decided to redo the analysis, in conclusion the use of Enums were discarded.

Instead the use of an extended common library were implemented. With this implementation the necessary information could be searched for using the extended versions of the entity class themselves. This way the components no longer leaked unnecessary information.

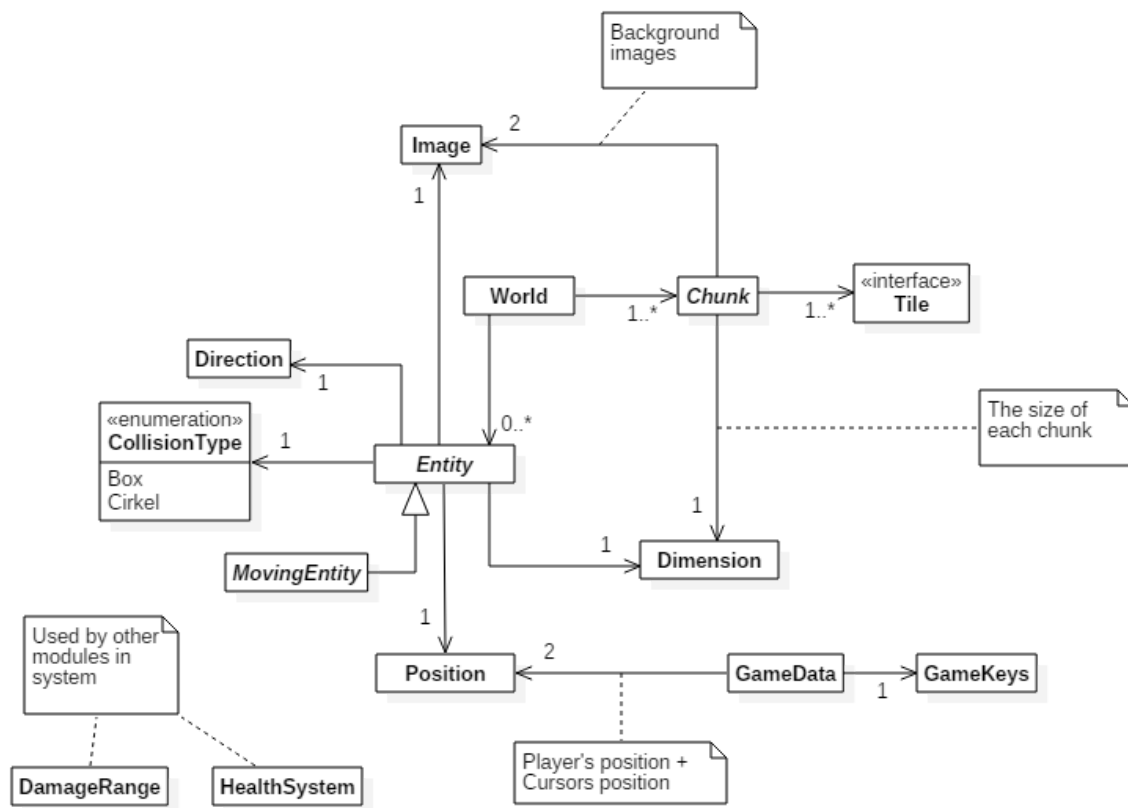


Figure 4.2.1: Common library without the sub-common libraries besides MovingEntity.

Within all the common libraries, most of the classes extends from a superclass (figure 4.2.1). For example Enemy extends from Character which extends from MovingEntity which then again extends from the Entity. Likewise Player also extends Character. This way entities does not contain unnecessary information, and provides the possibility for a specific search of entities in world. Each of the sub-common components extends a general common component. The communication between the components is done with a service provider interface, placed in the respective common component. This creates clean communication and makes way for easy extensions to the system; in addition it also greatly lowers the chances of illegal communication between components.

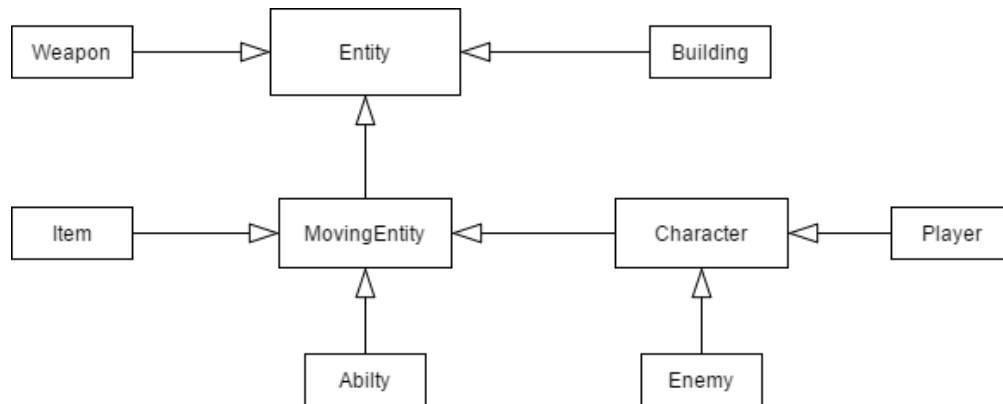


Figure 4.2.2: Simplified common and sub-common library containing only class types.

## 4.3 Interfaces

The components of DefendYourStuff are connected by a variety of interfaces, four of these interfaces are exclusively required by GameEngine. These four interfaces: IGamePlugin, IGameProcess, IPostProcess and IPreStartPlugin are provided by the other components in the system. In addition to these four interfaces the system contains a lot of component specific interfaces.

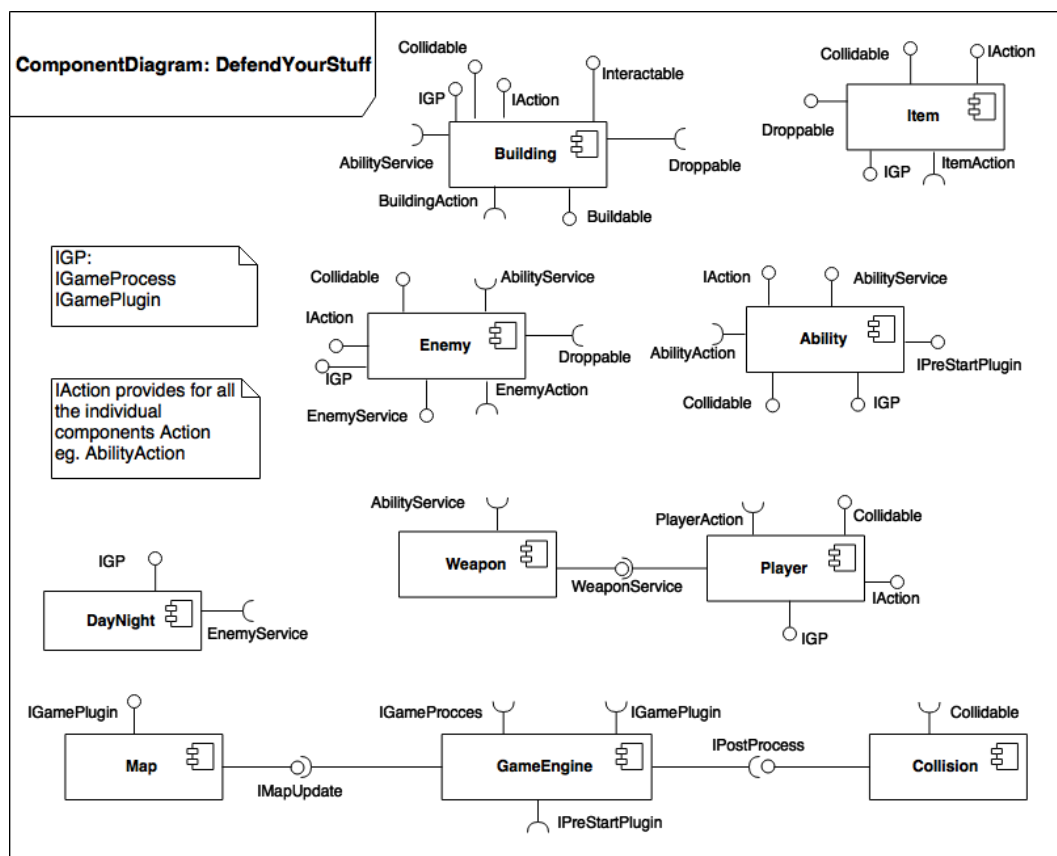


Figure 4.3.1: Component diagram showing which components realises which interfaces.

All the components shown in figure 4.3.1 containing entity data, that needs to be updated every game loop, implements the IGameProcessingService. These components include Player, Building, DayNight, Ability and the Enemy component. IGameProcessingService is however not only related to movement of entities in the game (as seen in Player, Ability and Enemy), but also static entities and data like Building and DayNight. Through these and the previous movable entities other data is updated, like the entities HealthSystem (see figure 4.2.1).

Besides the processing of data, most of these components also implement the IGamePlugin service used for dynamic loading and unloading of these components by the SilentUpdater component. These interfaces, plus the IMapUpdate interface are used by the GameEngine component for communicating with other components, making it able to process, start and stop each of those components.

### 4.3.1 Interface table

The following table contains the pre and post-conditions for the interfaces IGamePlugin, IGameProcess, IGamePostProcess and IPreStartPlugin, along with their methods. Appendixes has also been added with sequence diagrams for usage of the interfaces.

Method name:	Precondition:	Postcondition:	In interface:
start(gameData, world)	The component is currently not present in the system	The component is running in the system and data realized in the start method has been deployed to the GameData, and World class	IGamePlugin [see Appendix D.2]
stop(gameData, world)	A component implementing this interface is currently running in the system	The component is removed from the running system. Through the stop method all the associated data with the component is terminated or halted in some way.	IGamePlugin
start(gameData)	The component currently not present in the system	The component is running in the system and data realized in the start method has been deployed to the GameData class	IPreStartPlugin [see Appendix D.4]

process(gameData, world)	Entities and other data from the component is available for modifications through the World and GameData class	The components data and entity data is updated, the data for the given component in World and GameData is then replaced with the modified data.	IGameProcess [see Appendix D.1]
process(gameData, world)	The initial game cycle of IGameProcess is completed	The components data and entity data is updated, the data for the given component in World and GameData is then replaced with the modified data.	IGamePostProcess [see Appendix D.3]

## 4.4 Component responsibility

Each component in the game, have their own purpose and responsibilities. Here the scope of each component along with what service they provide. Below is a table with each component listed.

Component Name:	Responsibility:	Provides:
GameEngine	Render textures, create instance of world, create instance of GameData, move camera and move background	Drawing texture on screen
Player	Player movement and behavior	A user controlled player
Collision	Detects a collision between two entities & notifying the two entities	Collision Detection
Map	Creating chunks in world	Algorithm for creating chunks for map
Ability	Creating abilities, destroying them, their removal from the world and movement of abilities	Creation and processing of abilities
Building	Creating buildings, destroying them and handling what happens when they are destroyed	Creating and removal of buildings



Enemy	Sets enemies on ground level, creating and destroying Enemy entities and handling what happens when they are destroyed	Creating and removal of enemies
NPC	Creating and destroying NPC's and handling of what happens when they are destroyed	Creating and removal of NPCs
Item	Creating items, updating their position in the game when they are dropped by enemies and performing their abilities.	Creating, updating, and performing abilities of items
AI	The behavior of enemies, creating/casting their ability and controlling its cooldown & behavior for NPC	Behavior for NPCs and enemies
Weapon	To generate Weapons for the player to use	Weapons for player
DayNight	To spawn entities at the right time and place & remove enemies and other entities at day	-

## 5. Design

In this chapter, a more detailed structure of the system is described, including the relationship between common and the sub-common libraries, the component structure and how each component communicate with each other.

### 5.1 Sub libraries of the Common library

In section 4.2 of this rapport it was explained, how the common library component is the foundation for most of the classes to be used between other components. In order to separate responsibility and development of these components even further, the Common library has been separated into several Sub-common Libraries. They are then added as dependencies to the components that needs to share necessary data between each other. This is done in order to minimize the leakage of implementation details, so that for instance, the Map component (see figure 4.3.1) doesn't have to know details about classes or interfaces unrelated to those used by the GameEngine and Building component. If later development would then reveal that the Map component needs to communicate with other components, it will be easy to add the implementation of the appropriate sub-common library.

The following diagram shows the classes and interfaces of the sub-common library, and how some of these classes inherit from the classes of the common library. Details about this will be explained in section 5.5.

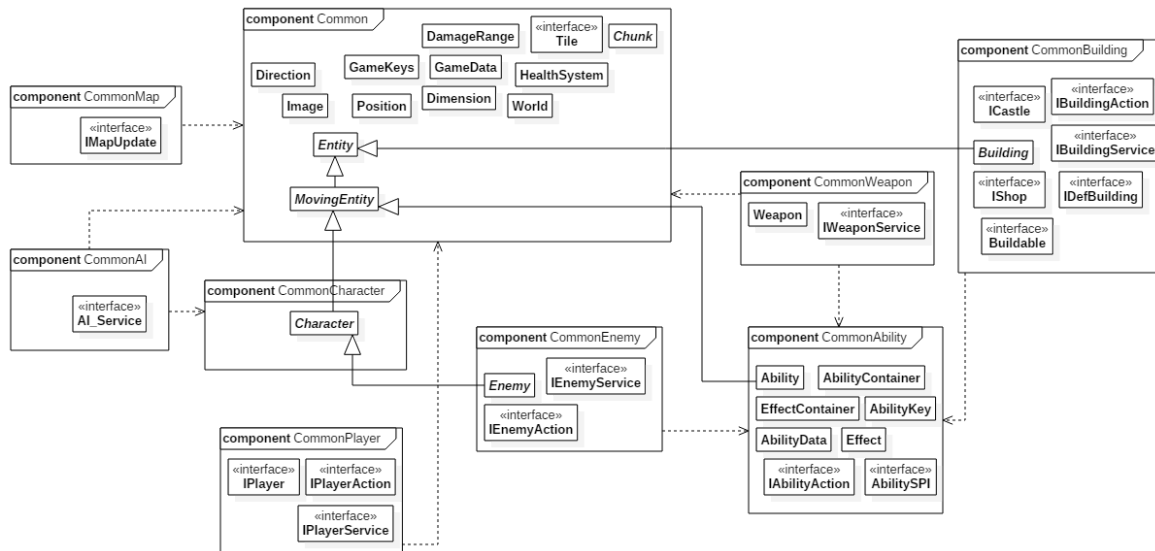


Figure 5.1.1: Diagram showing the relationship between the Common component, and the different sub-common components.

## 5.2 Components of the system

The interaction between components are done by component specific SPIs shown in figure 4.3.1. These are shown as required and provided interfaces, an example of a SPI is the IEnemyService SPI, that the DayNight component uses to spawn enemies. For different components to communicate with each other, they implement an action interface. The SPIs and action interfaces make it possible to avoid tight coupling between components, because the components depend upon the interfaces and the entity system framework .

## 5.3 Action interfaces

Each action interface provides a way for an entity located in a component to identify a specific entity located in another component, and make it perform an action related to the first entity. This is done by using the Collision component as an intermediate for communications. Below is a sequence diagram showing an example of how this interaction is performed by the system.

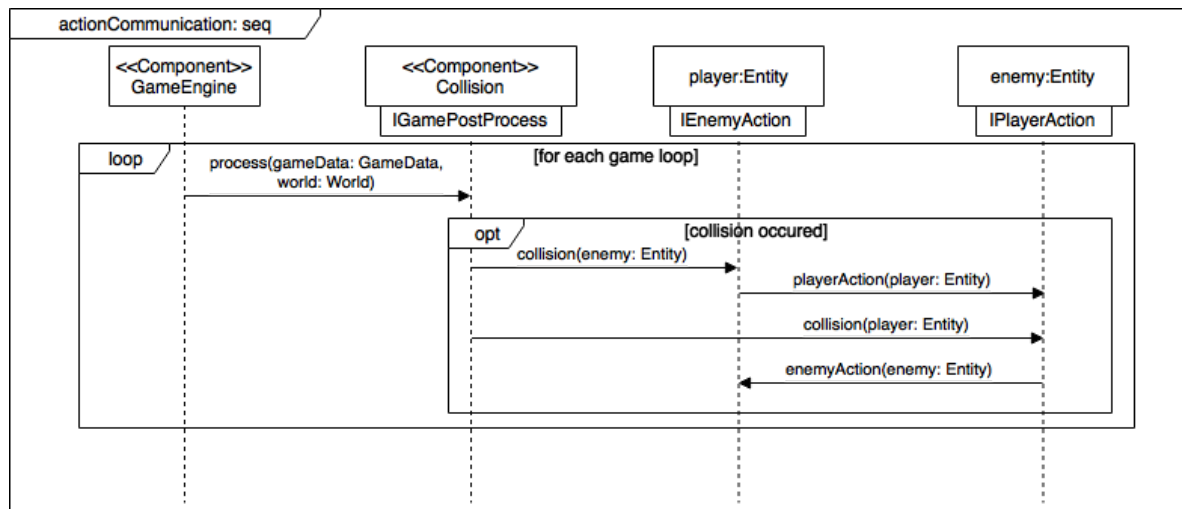


Figure 5.3.1: This sequence diagram shows method call for a collision between a player and an enemy.

Through the collision method the entities are able to call the other entities action interface and make it perform a specific action. For example if a player has collided with an enemy, then the Collision component will call the player's collision method with the enemy entity as parameter, and the enemy's collision method with the player entity as parameter. The player will then call playerAction on the enemy entity, and the same sequence of actions is performed for the enemy entity. Therefore player provides the IEnemyAction interface containing the *enemyAction(enemy Entity)* that makes the player able to perform actions related to a collision with an enemy, and vice versa. This is done by all subclasses of Entity existing in the different components of the game. Through this way of identifying entities from each other, it is also possible to identify a specific entity between multiple instances of the entity subclass (e.g. a specific building amongst all buildings currently present in the game, that the player has collided with).

## 5.4 Component-specific SPIs

As described in section 5.1 the component uses more specific interfaces from the sub-common libraries (figure 5.1.1) for performing specific tasks than necessarily updating data of entities. The table below gives a brief description of these interfaces with their primary purpose.

Interface name	Main purpose
IMapUpdate	Is used for updating the current game world with new chunks when the player reaches the end of the current chunk array (see section 6.1).
IPlayerService	Is used for communication with the player entity, which is needed for instance by the GameEngine component to determine the movement speed of the game-camera in relation the player's movement speed.

IPlayer	Is used by other entities and components for identifying the player of the game.
IBuildingService	Is used for communication with all building entities of the game, making other entities able to upgrade the specific building.
IDefBuilding	Is a sub-interface of IBuildingService related to only the defensive buildings of the game. This interface includes an operation for repairing damaged buildings.
IShop	Is a sub-interface of IBuildingService related to the shop-buildings of the game. This interface is used for buying different items in the game, and unlocking new items, based on some specific cost.
Buildable	This interface is used in the Map component for choosing which building to create at which location in the game.
ICastle	Is used for retrieving the health system of the castle-building. This is for the GameEngine to monitor the health of the castle-building and perform specific actions such as the Game Over state, and displaying the Castle's health on the GUI.
AbilitySPI	Is used for all communication related to using Abilities in the game. This interface includes operations used for retrieving a categorized list of available abilities in the game, and also for performing these abilities.
AI_Service	Is used for assigning specific AI behaviour for entities of the game, which is to be used for NPCs and enemies.
IEnemyService	Is used for creating specific types of enemies in the game, plus removal of these.
IWeaponService	Is used for creating specific types of weapons in the game, which will be used by the shops when the player tries to buy a weapon.

Most of these interfaces includes a list of methods, with specific pre- and postconditions for their usage. These are included in Appendix E.

## 5.5 Class Inheritance and dependency depth

The class inheritance in this game can be seen in figure 5.1.1. The inheritance is used to avoid a single entity class that contains all fields that only some specific kind of entity needs. Thus resulting in some of these fields will be empty in another implementation of entity, making it impossible to know which field is available and how to manipulate it. One way to solve this is by using inheritance; when doing so, one must be careful not to use this to avoid code reuse but use it for type reuse. The diagram is a snippet of the full diagram which can be seen in Appendix D.5

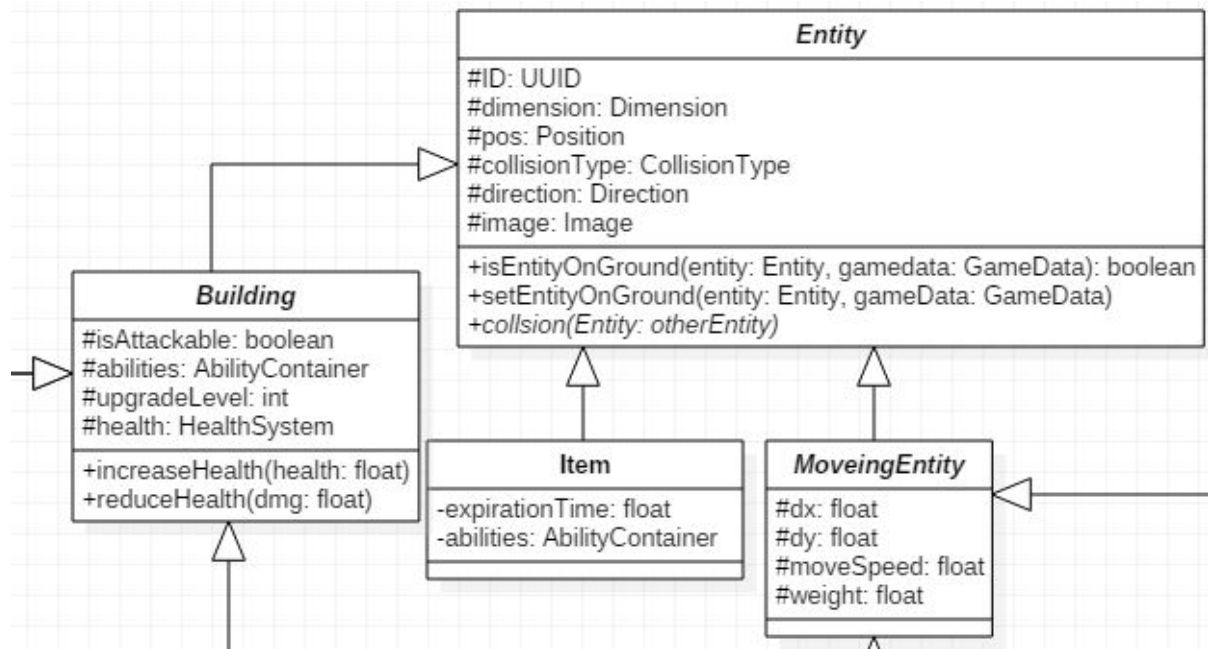


Figure 5.5.1: Snippet of entity inheritance diagram

The snippet shows the differences in the first sub classes. The way the entity framework is implemented makes it easy to implement a new type of enemy, since this new enemy will extend **Character**; the only change will be inside the enemy component. To other modules this enemy is just another type of entity.

One of the drawbacks by using class inheritance is that the higher classes in this hierarchy must be stable and contain only the essential fields and methods to avoid changes. Even a small change in one of these classes could result in a multitude of changes that affect multiple classes [5].

### 5.5.1 Class dependency depth

It is desirable to keep the dependency depth as low as possible [6, Ch. 4, pp. 70-72] for the reasons mentioned in the previous section. The dependency depth in this game is:

Number	Class	Depends on	Dependency depth
1	Entity	Dimension, Position, Image, Direction, CollisionType	1
2	Building	Entity, AbilityContainer, Healthsystem	2
3	Item	Entity, AbilityContainer	2
4	MoveingEntity	Entity	2

5	Character	MoveingEntity, AbilityContainer, Healthsystem	3
6	Ability	MovingEntity, EffectContainer, DamageRange, Entity	3
7	RangedAbility	Ability	4
8	MeleeAbility	Ability	4
9	PositioningAbility	Ability	4
10	SummoningAbility	Ability	4
11	Arrow	RangedAbility	5
12	Fireball	RangedAbility	5
13	Slash	MeleeAbility	5
14	DefensiveBuilding	Building	3
15	Castle	DefensiveBuilding	4
16	Wall	DefensiveBuilding	4
17	Tower	DefensiveBuilding	4
18	DestroyedCastle	DefensiveBuilding	4
19	Rubble	Building	3
20	Portal	Building	3
21	Enemy	Character	4
22	BigMeleeEnemy	Enemy	5
23	MediumEnemy	Enemy	5

The highest dependency depth in the game is 5. This might make the system harder to maintain because of the ripple effect previously explained [5].

## 6. Implementation

In this chapter, the implementation of the different components mentioned in the design chapter, will be explained by the implementation of relevant scenarios.

The chosen scenarios are the implementation of the Map generation, the game mechanic for Player and Enemy, the LibGDX render mechanic, the collision detection system, and the functionality of SilentUpdate.

## 6.1 Map generation

The Map component implements two Service Provider Interfaces; IMapUpdate and IGamePluginService.

With IGamePluginService, the Map will be able to instantiate chunks which is part of the map, when the application starts, and remove all chunks when the Map component is unloaded. In the start method, the map generates multiple Chunks on both the left and the right side of the world, until a portal is to be placed.

The IMapUpdate however, is only called when the need for a new chunk in either the left or the right side of the map, should be created. This is done in the GameEngine component, that checks if the player entity is entering the last chunk on either side. If it is, then GameEngine calls Lookup for IMapUpdate's update method, this then generates chunk and places it in the correct ArrayList that contains either all left or right Chunks, based on the position (0,0) in the game world.

```
// Update Map if camera is near the end at the left side
if (camPositionX < secondLastOffsetX) {
    for (IMapUpdate service : getIMapUpdate()) {
        service.update(world, true);
    }
}
```

Figure 6.1.1: Snippet of the check for the left side of the game world. "secondLastOffsetX" is the position offset of the second last chunk.

The way to generate chunks for the map, was inspired by Darius Kazemi's map generation for his game "Spelunky". His map generation [7] would appear to randomly generate each Chunk, however it only randomly chooses a predefined chunk, then it randomly chooses a template scheme for its tilegrid, and at last it randomly chooses the appearance of each tiles in the chunk.

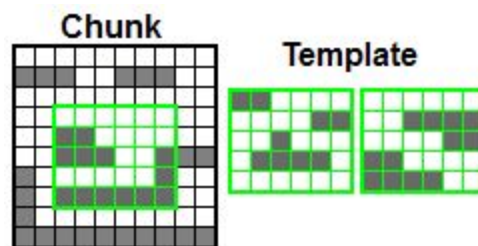


Figure 6.1.2: The Chunk is predefined with the placement of a template and some tiles. The template is also predefined with the placement of tiles.

As his example, all Chunks in this project are predefined with a tilegrid. However the similarities end there as the template functionality was never implemented, because it was not a must requirement. The Chunk instead communicates with the Buildable interface to instantiate building entities in the game.

The Map component's controller class, generate random Chunks with different switch cases. First a general switch is called, that contains calls to other more specific switches, which then generate a specific chunk based on an integer chosen randomly in an interval.

The general switch contains all biome switches. In this case only the forest and grassland has been implemented. The biome switch then chooses a random case again, based on the same principle as before. This time it creates a new specific Chunk and returns it all the way up the call-chain, and then it is added to the ArrayList in World.

Because the Map only deals with two biomes that only contains two different chunks each, the switch cases have not been moved into its own singleton factory classes.

### 6.1.1 The Chunk

The Chunk class that is used by the map, contains a tile grid with images and a method to instantiate necessary Building entities for that chunk, such as the Castle for the base chunk. The tile map is a two-dimensional array of Tile instances, which will specify the width and height of the Chunk. The placement of the tiles in the tilegrid, is rotated 90 degrees compared to the game world. (see figure 6.1.3)

```
private Tile air = new Tile_Air();
private Tile d01 = new Tile_Dirt();
private Tile w01 = new Tile_BrickWall();

public final Tile[][] BG_BASE = new Tile[][]{
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air},
    {d01, w01, air, air, air, air}
};
```

*Figure 6.1.3: Shown is the 2D tile grid for the base chunk.*

This is because of the way a for loop iterates, so the earth has to be placed vertically at the left, instead of horizontally on the bottom, in the double array.

A Tile only contains a string URL path to an image, and not a Building entity; those are created in the createEntities method of the Chunk. This method uses Lookup with the Buildable interface, and creates the necessary entities with the specified position. (See figure 6.1.4)



```
@Override
public void createEntities(World world) {

    Position wallLeft = new Position(this.getPositionOffset(), TILE_SIZE);
    Position wallRight = new Position((getDimension().getWidth() + this.getPositionOffset()), TILE_SIZE);
    Position CastleDoor = new Position((getDimension().getWidth() / 2) + this.getPositionOffset(), TILE_SIZE);

    for (Buildable buildable : lookup.lookupAll(Buildable.class)) {
        buildable.createWallBuilding(world, wallLeft);
        buildable.createCastleBuilding(world, CastleDoor);
        buildable.createWallBuilding(world, wallRight);
    }
}
```

Figure 6.1.4: Shows how entities is created using Lookup in the createEntities method from a chunk.

The controller class for the Building component, is then responsible for creating these entities and add them to the world hashmap.

## 6.2 LibGDX render functionality

The core implementation for the game to work, is the GameEngine. This is the only component to implement libgdx which is used for drawing the screen. Before libGDX can render anything, a Lightweight Java game library (lwjgl) application must be set up and configured, since it uses an ApplicationListener, that contains the functionality to render the game. The LwjglApplication is created at startup when GameEngine is loaded.

When the LwjglApplication is created, resumed, rendering, paused or destroyed; the ApplicationListener is called using methods with said names. The most important ones are the create() and render() methods. The create methods ensures all the important game logic is set up, such as the InputProcessor, AssetManager and the different renderers such as ShapeRenderer and the SpriteBatch. It is also here Lookup is used for calling all preStart() and start() implementations using the service provider interfaces.

The render() method is called every frame the application is running, this is why the first thing to do, is to clear the screen using the OpenGL buffer, so a new frame is ready to be rendered. Before a frame is to be drawn on screen, all entities must be updated using Lookup on all implementations of the IGameProcessingService and afterwards the IGamePostProcessingService. When all entities are updated, a draw method can be called, that loops through all entities and draws their image using the AssetManager and SpriteBatch.

## 6.3 Player implementation

This subsection explains the implementation of the different functionality of the player, such as movement, abilities and weapons.

### 6.3.1 Player movement

The player is implemented in the Player component, which uses the IGamePluginService to create an instance of a Player entity; while the IGameProcessingService SPI updates said instance each frame.

The player controller checks each frame for different keyboard and mouse input, so that the player can move and shoot accordingly. This is done by using the GameKeys instantiated in GameData, which the GameEngine updates when libGDX checks for keyboard and mouse input using the InputProcessor.

The player should move in either left or right direction, and upwards when jumping. Depending on what direction the player entity is going, a horizontal and vertical velocity is set, which is used to calculate the player position.

```
handleKeyboardInput(gameData, world, player);  
Position position = new Position(player.getX() + horizontalVelocity,  
                                player.getY() + verticalVelocity);  
player.setPosition(position);
```

*Figure 6.3.1: This snippet shows how the player's position is set with horizontal and vertical velocity*

To set the velocity, the player controller checks the keyboard input for: W when jumping, that sets the vertical velocity based on a vertical force and the player's weight; A and D for left and right direction, that sets the horizontal speed based on the move speed specified of the player class.

### 6.3.2 Player abilities

Most entities have an AbilityContainer, which hold specific AbilityData instances. If the player should have the ability to shoot Fireballs, the player's AbilityContainer would then have a FireBallData instance. This instance contains a display name, cooldown information and an AbilityKey used when creating an ability entity.

```
public void useAbility(Entity owner, int abilityIndex, World world) {  
    abilites.get(abilityIndex).useAbility(owner, world);  
}
```

*Figure 6.3.2: This snippet shows how the player's position is set with horizontal and vertical velocity*

To use an ability, the player uses his AbilityContainer with the index being the ability he wants to create; this keeps the implementation dynamic, as the player does not need to know what is stored in the AbilityContainer, only that when he presses a key to use an ability, that ability is created. An example can be seen in figure 6.3.3.

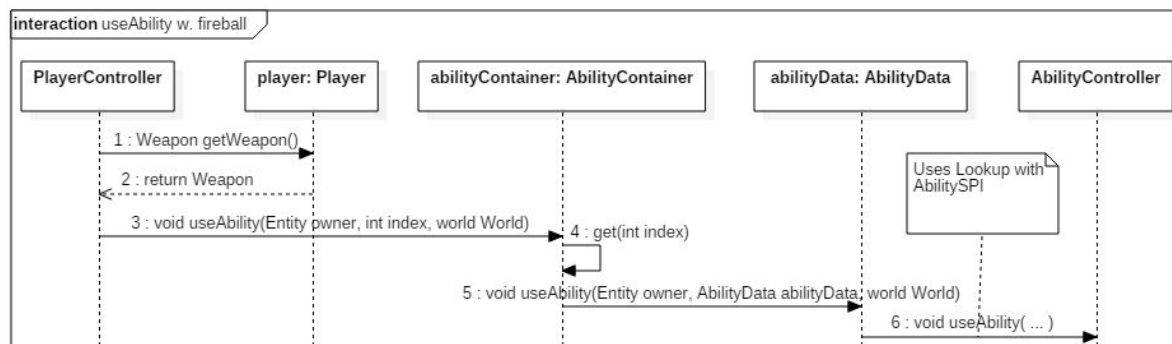


Figure 6.3.3: This snippet shows how the player shoots an ability. The full sequence diagram can be seen in Appendix D.6.

Looking at the diagram, it can be seen that when the player calls useAbility, the method uses Lookup on the AbilitySPI, which the Ability component implements. From here a new Ability entity is created, by getting a reference to the correct Ability class using the AbilityCatalogue ConcurrentHashMap, then calling createNewInstance on that reference. (Shown in figure 6.3.4)

```

Ability ab = abilityCatalog.getAbilityForType(abilityData).
    getNewInstance(owner, x, y, directionLeft);
    
```

Figure 6.3.4: This line of code show how the createAbility method in the AbilityController, creates an ability using the AbilityCatalogue and getNewInstance.

The AbilityCatalogue uses two ConcurrentHashMap, which will return either a reference to a specific Ability entity or an AbilityData. To get this value in the ConcurrentHashMap, an AbilityKey reference must be used as a key. It is this way a correct Ability entity can be created using the useAbility, since the AbilityKey reference is stored in the AbilityData sent with the method.

```

@Override
public void preStart(GameData gameData) {
    abilityDataCatalog.addAbility(new FireballData(new AbilityKey(), new Fireball());
    abilityDataCatalog.addAbility(new SlashData(new AbilityKey(), new Slash());
}
    
```

Figure 6.3.5: This shows how the AbilityCatalogue is instantiated with abilities and AbilityData, that both uses its own object on AbilityKey, when preStart is called.

The Ability plugin uses IPreStartPluginService to insert all abilities and ability data into the AbilityCatalogue. As all entities that uses an AbilityContainer heavily depend on this AbilityCatalogue containing data, it needed to be instantiated before the other components, hence the usage of IPreStartPluginService, instead of IGamePluginService.

### 6.3.3 Player weapon

The player gets his AbilityContainer from a weapon instance, which also hold information on damage multipliers. The abilityData that is stored into the AbilityContainer of the weapon, is based on the weapon type, which can be either ranged or melee. The weapon does not

store a weapon type, instead the weapon is created with different methods using Lookup on the IWeaponService SPI, either with the createRanged or createMelee method.

```
@Override
public Weapon createRanged() {
    AbilityData[] abilityList = new AbilityData[4];
    AbilitySPI abilityProvider = lookup.lookup(AbilitySPI.class);
    AbilityData ab = abilityProvider.getRangedAbilities().get(0);
    abilityList[0] = ab;
    Weapon rangedWep = new Weapon(randomMultiplier(80, 120),
                                   (randomMultiplier(80, 120)) * percentage, abilityList);

    return rangedWep;
}
```

Figure 6.3.6: This shows that Lookup is used to get ranged AbilityData from the Ability component.

It is the Weapon component that implements the IWeaponService SPI, and when a create method is called, it creates a new AbilityContainer and calls Lookup on the AbilitySPI which returns an abilityData list based on the method called. The method then creates the weapon with a damage multiplier and the AbilityContainer, and then returns it to the player. This can be seen in figure 6.3.7.

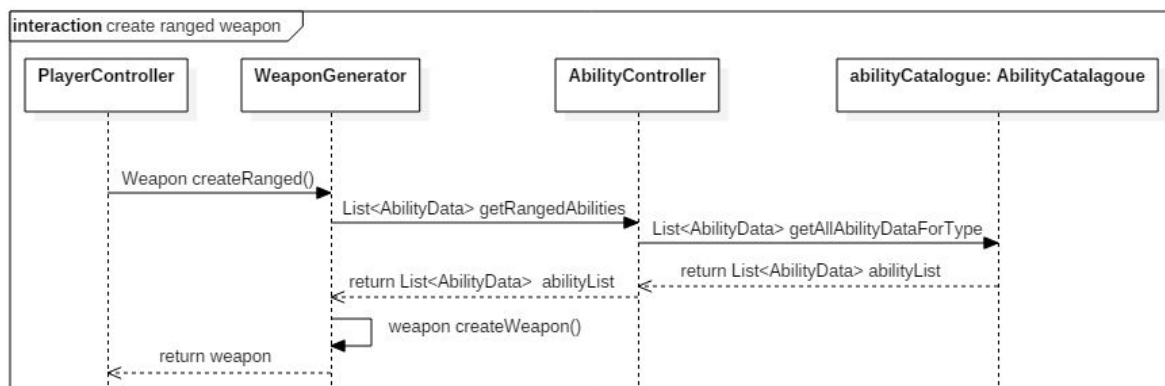


Figure 6.3.7: This shows the calls between components when creating a weapon.

## 6.4 Enemy implementation

This subsection will explain how the enemies are created, the behavior of them and how enemies are removed.

### 6.4.1 The spawning of enemies

The DayNight component has the responsibility to spawn enemies at a specific time and on a given location. The component achieve this by counting down roughly 10 seconds, then calculating what enemies to spawn, the amount of enemies to spawn and finally setting a random location for each enemy to spawn on.

The component calculates this using a greedy algorithm, this algorithm is designed to output the strongest combination of enemies in the game. Below, will the proof and the implementation of the algorithm be presented.

The following table presents the different enemy's weight and usefulness value. This is used to develop the algorithm. The weight is used to calculate how many enemies that will be spawned and the usefulness value is used to calculate which enemy to spawn. A higher usefulness value is always preferred.

Weight to value table:

Type	Big + Medium Enemy (B+M)	Big Enemy (B)	Medium Enemy (M)
Weight	15	10	5
Usefulness value	7	3	1

Figure 6.4.1: Weight to value table for enemy spawner

The reasoning behind the "Usefulness value" is to prioritize what combination of enemies or which enemy that will be spawned. The best combination in this game is pairs of big- and medium enemies. As the big enemy will try to get close to the player and absorb the player's damage, while the medium enemy will stop at a distance to shoot the at player. Therefore, this combination has the highest usefulness value. This algorithm should strive upon always spawning a pair of Big and Medium enemies before any other combination or lone enemy. The implementation of this algorithm is shown below:

```
83 private void greedyEnemySpawner(float enemyCap, IEnemyService spawner,
84     World world, GameData gameData) {
85     while (enemyCap >= MEDIUM_ENEMY_COST) {
86         if (enemyCap >= MEDIUM_ENEMY_COST + BIG_ENEMY_COST) {
87             spawnBigAndMediumEnemy(spawner, world, gameData);
88             enemyCap -= MEDIUM_ENEMY_COST + BIG_ENEMY_COST;
89         } else if (enemyCap >= BIG_ENEMY_COST) {
90             spawnBigEnemy(spawner, world, gameData);
91             enemyCap -= BIG_ENEMY_COST;
92         } else {
93             spawnMediumEnemy(spawner, world, gameData);
94             enemyCap -= MEDIUM_ENEMY_COST;
95         }
96     }
97 }
98
99 }
```

Figure 6.4.2: The greedy algorithm used for spawning enemies is illustrated in this code snippet..

The algorithm is given an input *enemyCap* that determines the current limit for how many enemies can be spawned in the current iteration of the greedy algorithm. As seen in figure 6.4.2, line 84, the algorithm wants to minimize the value of *enemyCap* to a value below 5, that is smaller than any choice the algorithm can use. In figure 6.4.2, line 86, 90 and 94, three cases are considered by the algorithm:

- Case 1:  $\text{enemyCap} \geq \text{weight value of Big Enemy} + \text{Medium Enemy}$  (15)
- Case 2:  $\text{enemyCap} < 15$  and  $\text{enemyCap} > \text{weight value of Big Enemy}$  (10)
- Case 3:  $\text{enemyCap} < 10$

The following will show how the algorithm always chooses the optimal solution, through a proof by contradiction.

#### Case 1:

$$\text{enemyCap} = 20$$

$$\text{enemyCap}' = 20 \Rightarrow (B + M) + (M) \Rightarrow (15 + 5) - \text{enemyCap} = 0$$

$$\text{enemyCap}'(\text{usefulness value}) = 7 + 1 = 8$$

Now to proof that there isn't a more optimal solution, this can be done with a simple substitution. If assuming that a more optimal choice exist, then by substituting the combination (B+M) with 4\*(M) it is possible to calculate this combinations usefulness value:

$$\text{enemyCap} = 20$$

$$\text{enemyCap}' = 20 \Rightarrow 4 * (M) \Rightarrow (4 * 5) - \text{enemyCap} = 0$$

$$\text{enemyCap}'(\text{usefulness value}) = 4 * 1 = 4 \Rightarrow \text{poorer usefulness value}$$

The only other choice is two Big enemies this gives the usefulness value of 6. Since the value of 6 is still a poorer choice than the original choice made by the algorithm, and no other choices exists for this *enemyCap*, the algorithm has chosen the optimal solution. Case 1 will only change if the *enemyCap* is increased, in this case the best choice will be multiple (B+M) choices over the other possible choices. This means that case one is proven.

#### Case 2:

$$\text{enemyCap} = 14$$

$$\text{enemyCap}' = 14 \Rightarrow (B) \Rightarrow 10 - \text{enemyCap} = 4$$

$$\text{enemyCap}'(\text{usefulness value}) = 3$$

The only other choice is two medium enemies which give a usefulness value of 2. Since no other choice is possible when  $15 > \text{enemyCap} > 10$ , one big enemy is preferred over two medium enemies. This means that case two is proven

#### Case 3:

In case 3 where  $10 > \text{enemyCap} > 5$ , the algorithm will only have the choice of selecting a Medium enemy. This means that case three is proven.



This means that for any given enemyCap this algorithm will always spawn as many (B+M) as possible and first when this is no longer possible the algorithm will try to spawn a big enemy, and only if this is impossible it will spawn a medium enemy. Therefore the algorithm must give the optimal solution for any given enemyCap, given the restrictions of figure 6.4.1. This means that the greedy algorithms is proven.

To spawn and remove enemies, the DayNight component uses the interface IEnemyService. This interface is being realized by the class "EnemyController" in the enemy component.

The enemy component realize the IEnemyService and thus have responsibility to instantiate and destroy enemies. The component does this by implementing subtypes of the class Enemy and putting instance of these in the game world.

## 6.4.2 The Behavior of Enemies

The primary responsibility of the enemy component is handling the lifecycle of every enemy. The behavior of each enemy is then outsourced to another component, which is the AI component that can receive any Character class, and apply AI behaviour to that object.

```
public interface AI_Service {  
    /**  
     *  
     * Is used to give an enemy AI behavior.  
     *  
     * @param enemy the enemy.  
     * @param world the world that the enemy exist in.  
     * @param gameData gamedata is used for delta time.  
     */  
    void assignAttackAndDodgeEnemyAI(Character enemy, World world,  
        GameData gameData);  
    /**  
     *  
     * @param enemy the enemy.  
     * @param world the world that the enemy exist in.  
     * @param gameData gamedata is used for delta time.  
     * @param minShootDistance the minmum shooting distance for the enemy to  
     * stand still and shoot.  
     * @param maxShootDistance the maximun shooting distance for the enemy to  
     * stand still and shoot.  
     */  
    void rangedAI(Character enemy, World world, GameData gameData,  
        int minShootDistance, int maxShootDistance);  
}
```

Figure 6.4.3: Show the two interface methods current present in the AI\_Service SPI.

In this case, the "AI\_ControlSystem" is the component realizing two different AI behaviour as stated in the interface above. One for the melee enemies, and one for the ranged enemies. The behavior for these two types is quite different, but the implementation technique for both

is achieved through a behaviour tree. The tree for Big melee enemy (see figure 6.4.4) the behaviour tree for ranged enemies can be found in appendix D.7:

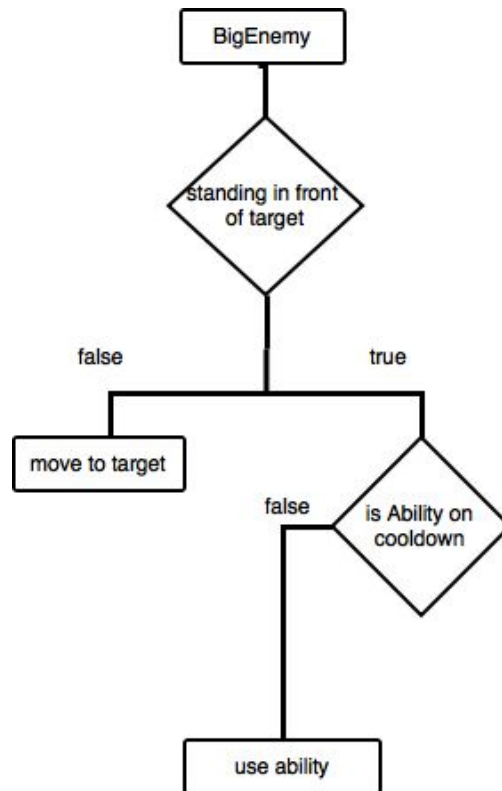


Figure 6.4.4: Behaviour tree seen for the AI of Big Enemy

The AI behaviour tree technique produces a simple AI that takes decisions on a True/False basis without any consideration for the cost of these choices. This AI fits the current enemies of game well since every enemy's action is simply "go left", "go right" and "attack". This behavior is seen in the behaviour tree where the melee AI is a simple move to and attack. The ranged AI is a bit more complicated since it moves to a set distance from the target and from there attacks its target. The implemented AI operates on a fully observable environment, this has the advantage that it can always find a target in world if there is one.

## 6.5 Collision detection

The Collision component handles collision checks for all entities in `world.getEntities()`. This component only implements `IGamePostProcessingService`, since it only needs to be run after every entity has been processed.

The collision detection runs an  $O(n^2)$  loop once, but uses multiple if-statements to check what `CollisionType` both entities have, so that a correct collision check can be made. If both entities use box-collision or circle-collision, only a collision check with box or circle collision is made. However if the entities does not have the same `CollisionType`, a `boxCircle` collision check is made, that correctly checks the corner of the square for collision.



## 6.6 SilentUpdate

The SilentUpdate component, is a component that was given by the lecturer. The SilentUpdate works together with an Update Center, that contains Netbeans Module System files (.nbm). These files contain information about a component and the components jar files. The Update Center also contains the update.xml file that specifies what components exists in the Update Center, and some information about these components.

The SilentUpdate component then listens to changes in this xml-file, and when one occur, it tries to install the component. To install the component it must have a class that extends the ModuleInstall class. When the component is installed, the rest of the program can use its functionalities.

```
public Collection<? extends IGameProcessingService> getGameProcesses() {  
    return lookup.lookupAll(IGameProcessingService.class);  
}
```

Figure 6.6.1: Snippet from gameEngine the gets GameProcessers

When a component is installed, lookup-methods of the Netbeans Module System's ServiceLocator class can find the newly installed implementations, and then run their methods. However, the start method that only runs when a component is installed, it is implemented differently. Here a listener is used, that registers when a new implementation of IGamePlugin is added to a list, and then calls the start method, for updating the list of running components in the system.

```
private final LookupListener lookupListener = new LookupListener() {  
    @Override  
    public void resultChanged(LookupEvent le) {  
  
        Collection<? extends IGamePluginService> updated = result.allInstances();  
  
        for (IGamePluginService us : updated) {  
            // Newly installed modules  
            if (!gamePlugins.contains(us)) {  
                us.start(gameData, world);  
                gamePlugins.add(us);  
            }  
        }  
    }  
}
```

Figure 6.6.2: Codesnippet from the lookuplistener in gameEngine

The same happens when it registers, that a component is uninstalled, the listener then makes sure that the stop method is called on the component to be removed from the list.

## 7. Tests

In this chapter the tests done for the system are documented and described. The tests include GUI Testing, Unit Testing, Interface Testing and Integration Testing.

### 7.1 Exception Handling at creation

When some new objects are created, they contain a check for illegal arguments. These exceptions makes it easier to spot the root of possible errors in the system.

```
if (expiration < 0) {  
    throw new IllegalArgumentException("Expiration time cannot be negative");  
}
```

*Figure 7.1.1 Shows a snippet of a check from the Ability class.*

### 7.2 GUI Testing

Some exceptions however cannot be avoided. There are two methods of handling those exceptions. When lookup is used, a check is made to see if the returned provider is null. This can be used, since lookup will automatically return null, if no implementing class can be found. This will for example happen, if the implementation of a service providing component is unloaded. Exception handling is also used when getting entries from an arraylist. Also in this case, an error can occur if the components are not loaded correctly. When that happens, the program will crash with an `IndexOutOfBoundsException` exception. Therefore handling of such errors are necessary, though only if the cause of the error is known.

In order to minimize the amounts of errors several methods of testing are used. The most prevalent method, but also the least reliable is GUI testing. This testing method works by implementing some code, and then observing the game to see if it runs as expected. Even though this method is not the best to test, if the program works, it is necessary when methods have to “feel” right like jump distance and movement speed.

### 7.3 Unit Testing

Another method is Unit testing. The unit test reside in each of the components and function as a whitebox test on the individual component and classes of that component. These unit tests can test public methods available to other components, as well as methods only available inside its own component. These tests ensure that the methods of the classes in the components still works after any modifications.

```
/**
 * Test Dimension of chunks
 */
@Test
public void testChunkDimension() {
    float expectedW = 12.0f;
    float expectedH = 5.0f;
    float resultW = chunkBase.getDimension().getWidth();
    float resultH = chunkBase.getDimension().getHeight();
    assertTrue(expectedH == resultH);
    assertTrue(expectedW == resultW);
}
```

Figure 7.3.1: Here is a test in *BuildingControllerIT*, that checks if the dimension for the base chunk is as expected.

## 7.4 Interface testing

Some interface tests have also been created. These interface tests are located outside of the component providing the implementation of the interface and therefore only verifies the interface methods. These tests work as a black box test, and tests if the components interfaces behave as expected. If these tests pass, then the chances that the other components can use the interfaces without error is better.

```
public static Test suite() {
    return NbModuleSuite.createConfiguration(SilentUpdateTest.class)
        .gui(false)
        .failOnMessage(Level.WARNING)
        //failOnException(Level.INFO)
        .enableClasspathModules(false)
        .clusters(".+").
        suite();
}
```

Figure 7.4.1: Code Snippet of the *Integrationtest*.

## 7.5 Integration Testing

The last method used for testing of the software is integration testing. This tests if the components can be loaded and unloaded with the *SilentUpdate* component through an automated test. This is done by simulating changes in the *update.xml* file that the *SilentUpdate* component uses to install and uninstall components in runtime.

A problem with this test is that it will always fail whenever an exception is caught due to the method added from the *NBModuleSuite* configuration, seen in the code snippet above. In order to circumvent this, the test has been set not to fail when these errors happen, but this decreases the overall usefulness of the test. However the test is still important, because it still tests if loading and unloading of components work.

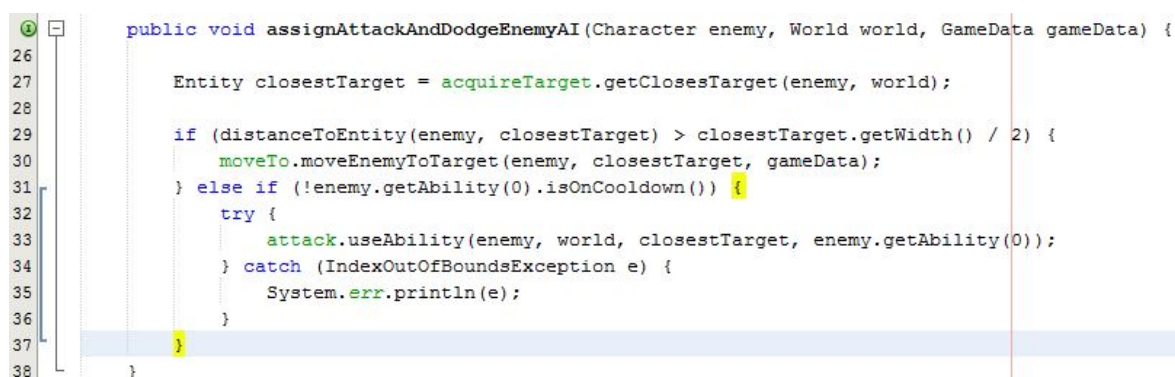
## 8. Discussion

In this chapter, some of the implementation details are compared to alternatives. The quality of the code, and choices are also discussed here along with pros and cons.

### 8.1 Current AI vs more advanced AI

As previously described the current AI implemented in this game functions through a behaviour-tree [8] making the AI able to determine the appropriate action, based on a series of decision. Behaviour-trees are typically considered a good tool for implementing a simple AI in the game, and one could argue that behaviour-trees are also a viable solution for fairly complex AI. But as seen in the previous diagram (see figure 6.4.4) the BigEnemy AI containing only two decisions, already has a total of four nodes for the complete behaviour tree without considering additional alternatives that might be needed. One could also argue that the actions themselves after being performed, might lead to further decisions to be made which the BigEnemy might become restricted by different constraints put on it. The amount of nodes present in the decision tree will quickly contain intertwined connections to each other, which will give the AI little control over what decisions might be considered the best. The behaviour-tree might also become more difficult to understand, when more nodes are added, thus making it more vulnerable to potential bugs.

To avoid this, measures have been taken to make the architecture use more easily implemented frameworks for developing the AI of the game. If for instance one aspect of the behaviour-tree is analyzed like the code of the `assignAttackAndDodgeEnemyAI()` method seen below, it is seen how each appropriate action of a given decision is performed by another class. The classes in figure 8.1.1 are `AcquireTarget`, `MoveTo` and the `Attack` class. These have been build with relations to the State Design pattern [9]<sup>1</sup>, which for AI purposes can evolve into a Finite State Machine (FSM) pattern.



```
public void assignAttackAndDodgeEnemyAI(Character enemy, World world, GameData gameData) {
    Entity closestTarget = acquireTarget.getClosesTarget(enemy, world);

    if (distanceToEntity(enemy, closestTarget) > closestTarget.getWidth() / 2) {
        moveTo.moveEnemyToTarget(enemy, closestTarget, gameData);
    } else if (!enemy.getAbility(0).isOnCooldown()) {
        try {
            attack.useAbility(enemy, world, closestTarget, enemy.getAbility(0));
        } catch (IndexOutOfBoundsException e) {
            System.err.println(e);
        }
    }
}
```

Figure 8.1.1 The code snippet shows the AI of the big enemy

In many ways this FSM pattern is already present in the code with these state classes. Typically the entity operated by these states are supposed to have a general reference to an

<sup>1</sup> [https://sourcemaking.com/design\\_patterns/state](https://sourcemaking.com/design_patterns/state)

abstract state in some way, which is then swapped out with another state class when an action of one state makes that state transition into another state. But as explained in section 6.4.2., the more AI behaviour implemented in the game, the more complex the behaviour trees will become. This also holds true for the usage of FSM, especially if a series of actions are required before a specific action can occur [10]. This is where the approach of Goal-based action planning (GOAP) AI can be considered.

When the code reaches a certain level of complexity through evolving the state transition even further for each action and assigning a cost to the action performed. GOAP can then use a planner class for specifying a goal for the AI to accomplish and try making the AI reach this goal through the cheapest cost possible [11]. GOAP-based AI however would in the case of the current game, be considered needless complexity, because most AI behaviour present in the game can be fitted into small behaviour-trees as previously described in section 6.4.2.

## 8.2 Leakage of implementation details between components

Through the development of the game a few different ways have been used as identification of entities between components. Because good component design implicates a minimum leakage of implementation details, the preferred way of communicating is through interfaces. As seen in the project where the components primarily communicates through the interfaces `IGameProcessingService`, the component specific services (e.g. `IWeaponService`), and the Action interfaces (see section 5.3).

At first the project used enums between components for communication and identification. This made all communication between components run through `GameData` and `World`. The identification would then run inside each implementation of the process method in each of the components. If new classes were to be added to the game, then the list of enums would be updated. The problem with this however is that enum has no restrictions on modification, and therefore important enums of the game had a high risk of being accidentally deleted or misused. This could quickly become a big source of errors in the system.

Instead the current system uses a combination of both classes and interfaces placed in respective sub-common libraries. Even if enums were kept in the system, this sub-common separation would serve as a restriction on the leakage of implementation details, because components only depends on the libraries that it needs to have knowledge about. This serves as a better solution compared to having all components know about all shared data used throughout the system, because the components only need to know about a fraction of the shared data. Below is a table that shows the dependency depth at component level using multiple common libraries (the full table can be found in appendix C.2).

ID	Component	Depends on	Dependency Depth
1.	GameEngine	Common, CommonMap, CommonPlayer	5
2.	Player	Common, CommonAbility, CommonBuilding, CommonCharacter, CommonEnemy, CommonItem, CommonPlayer, CommonWeapon	4
3.	Collision	Common, CommonCharacter	3
4.	Map	Common, CommonBuilding, CommonMap	5
5.	Ability	Common, CommonAbility	2
6.	Building	Common, CommonAbility, CommonBuilding, CommonEnemy, CommonPlayer	4
7.	Enemy	Common, CommonAI, CommonBuilding, CommonCharacter, CommonEnemy, CommonPlayer, CommonAbility	5
8.	AI	Common, CommonAI, CommonAbility, CommonCharacter, CommonEnemy	5
9.	Weapon	Common, CommonAbility, CommonWeapon	3
10.	DayNight	Common, CommonCharacter, CommonEnemy	3
11.	Common	-	0

Through certain sub-common libraries, an abstract or concrete class is exposed to other components. For instance `CommonWeapon` exposes the concrete class `Weapon` to other components depending on this library compared to `CommonEnemy` that exposes an abstract enemy class. The risk of the concrete `Weapon` class is that components like the `Player` component has no restriction for creating `Weapon` instances manually in the component itself, even though correct implementation and usage of weapons in the game requires the specific component to use the method's `createMelee()`, or `createRanged()` from the interface `IWeaponService`. This usage of the `Weapon` class can potentially cause maintenance issues, and additional bugs at both the current development state, and the future development of the game. However if the exposed class to the component is abstract it restricts the development of the component from creating instances of the class, and might implicate a more indirect usage (for instance as parameters to methods). A problem with abstract classes are that they might represent some kind of implementation compared to interfaces, which only define a contract for usage. If for instance a change is made to methods with implementation details in the abstract class then this change will affect all subclasses created (see section 8.3).

Another alternative to classes is the use of interfaces for identification. In the game both the action interfaces are used for this purpose, but also a separate interface have been used in some cases. In the `CommonPlayer` component the interface `IPlayer` is exposed. This interface is used only as an identification interface, which the `Player`-entity class of the `Player` component implements as a kind of wrapper interface. The benefit of this solution is the complete restriction this puts on other components needing to communicate with the player component. Because the interface is only used as identification of the player entity in other components without exposing the possibilities for manipulating with the players data. One could argue about the resemblance this approach has to the usage of enums. The restrictions provided by interfaces however make them a more preferred choice over enums. An overpopulation of these wrapper interfaces could however quickly occur if every entity, at some point in the game development needs this specific identification mechanism. The wrapper interfaces could instead also be used as more general descriptions about functionality used by for instance all enemies that other components need to know details about. This would remove the problem with implementation details residing in an abstract superclass. And make additions of new functionality require additions of methods to be implemented from the interface or make additional interfaces required to be implemented by classes. Due to the late discovery of the advantages given from sole usage of interfaces, this approach have only been partially implemented in the project.

## 8.3 Fragile base class

One of the weak points in the game is its class inheritance tree. The inheritance in this game is used as type reuse and not code reuse. This complies with the Liskov substitution principle in the OO basic principles S.O.L.I.D. which means that every subtype of `Entity` should be replaceable with entity without altering the program [12, Ch. 10]. This makes it easy for other components to handle the different sup implementation of entity since other



components often only need to know the more general implementation of entity or an interface.

The problem appears in the maintainability of the system since one change in the Entity class(base class) requires change in every class that depend on entity or one of its subclass. This problem will only grow bigger because every class is now a place for refactoring and with every added class the overview becomes harder to maintain. This will result in a program that is very hard to maintain. A possible solution to this is to implement interfaces instead of an abstract class where this is possible.

## 8.4 Ability Component

Due to the complexity of the ability component, it is easy to find places of improvement. The first thing to notice is that it is the only component that implements the `IPreStartPluginService`. This SPI is needed to ensure that abilities are always loaded before anything else. This is because many other objects contain instances of abilities, and these objects would not work if Ability was loaded after them.

However another solution could be to accept that, objects can be created without abilities or let the Ability component put abilities, in predefined slots. While that solution might work, it is considered to be more work compared to the current solution in place.

```
@Override
public Ability getNewInstance(Entity owner, float x, float y, boolean directionLeft) {
    //TODO: implement this method
    throw new UnsupportedOperationException("Not supported yet.");
}
```

*Figure 8.4.1 Code snippet from RangedAbility.*

Another interesting thing is the way new abilities are created. The abilities are created by copying an ability, and then adding the copied version to the map in World. The new abilities are created using the `getNewInstance` method. This method does have some code that is not very optimal though. For starters the methods in the superclasses `RangedAbility`, `MeleeAbility`, `SummoningAbility` and `PositioningAbility` all throw an `UnsupportedOperationException`. This exception is misleading, as the methods are supported, but just in a subclass. A better exception to throw would be the `InstantiationException`, when the game tries to instantiate the wrong object.

```
@Override
public Ability getNewInstance(Entity owner, float x, float y,
    boolean directionLeft) throws InstantiationException {
    throw new InstantiationException("Not supported yet.");
}
```

*Figure 8.4.2 Code snippet that shows an improvement of getNewInstance method.*

The `InstantiationException` is primarily used when there is an attempt at instantiating an abstract class or interface. This hints at another solution, where instead of having a



getNewInstance method in the superclass, the method is only in the subclass. To do this, the superclass and method needs to be abstract, for instance making the RangedAbility class and its getNewInstance method abstract.

These solutions seems to be a better solution than the way it is currently done, and some versions of them would most likely be implemented in a later version.

## 8.5 Pros vs. cons for different component models

One of the formal requirements of this project is to use a component framework which had to be either the Netbeans Module System or OSGi. The choice for this project fell upon Netbeans, because it was the component framework, that was first introduced in the component course. In this subsection a pros vs cons list will be made for both the Netbeans and the OSGi component frameworks.

### 8.5.1 Netbeans Module System

#### **Advantages:**

- Netbeans allows for insertion and/or removal of component at runtime, either with the build in installer and uninstaller or by a SilentUpdate component.
- It also manages the dependencies automatically and makes sure to avoid circular dependencies. A circular dependency is when two or more modules either directly or indirectly depend on each other which can cause some problems.
- Netbeans allows metadata for components like version numbers, descriptions and so on.
- A component can be close to anything, and can even include resources like images to use as sprites for a game.
- Netbeans allows for distinguishing between service APIs and service implementations that can be in different components.
- Netbeans has the possibility of integrating with OSGi, through an extender component added to the system..

#### **Disadvantages:**

- The Netbeans Modules System is only optimized for use in the Netbeans IDE, and is significantly harder to use in other IDEs.
- Netbeans Module system has the possibility of using whiteboard registry for registering implementations of interfaces, which forces the usage of the components to include API code specific to the system.

### 8.5.2 OSGi

#### **Advantages:**

- OSGi allows bundles to be installed, uninstalled, stopped, started or replaced at runtime.

- It automatically handles dependencies at the package level. A bundle can declare that it exports one or more packages, and another bundle can declare that it imports those packages.
- OSGi also allows metadata for its bundles like version numbers, descriptions, packages, imported/exported and so on.
- OSGi is supported by Eclipse, IntelliJ and Netbeans, making it a less restricted choice for general development of java components, without being specific to one IDE.
- OSGi supports both the usage of whiteboard registry and dependency injection for wiring interfaces to implementation.
- Through usage of dependency injection in declarative services, API code is hidden from developed code and resides only in the xml files.

**Disadvantages:**

- OSGi relies heavily on the META-INF/MANIFEST:MF files found in every bundle making the developers use a lot of time working with these files and not the actual application. This makes the user responsible for a bit of manual wiring of the components.
- When writing the xml files for declarative services there is an additional risk of making type mistakes which can make the debugging of error's long and tedious.

By comparing these pros and cons it can be concluded that while Netbeans Modules and OSGi are very similar in terms of capabilities they are different in implementation.

## 9. Conclusion

In this chapter, the conclusion is written. The planned functionality that has not been implemented, will be described in future work.

Throughout this report and project, component-based software development has been applied together with game development. This has been explored in the report through the typical areas of software development including requirements, analysis, design, implementation and testing. These development steps has provided insight in advantages and disadvantages of various implementation methods. One of these disadvantages is the unnatural constraints of creating a game where all data is not available to other components, which was especially noticeable when trying to construct the interaction between two different entities. For instance if a building component needs to know what kind of ability it is handling, it has to import identifying parts of the given object. An easy solution would be to use enums or having a direct dependency on other types of entities. Although they seem simple from a object oriented perspective, they violate principles in component oriented software development. In order to circumvent many of the problems encountered, interfaces and inheritance were used. The usage of inheritance was problematic due to the fragile base class problem, where the maintenance gets harder despite working on a component-based software. Another problem have been to keep a good component oriented design while adhering to the principle of not leaking implementation details. These problems were a challenge and demanded constant learning for better ways to improve the abstractions

present in the system for maximizing the component aspect to its full potential through easy maintainability, low coupling and high reusability.

## 9.1 Future work

DefendYourStuff is still missing important functionality, one of the more crucial points is the interaction between player and the world. The original plan was that the player could interact with buildings and NPCs through the GUI, the plan was then that the buildings could be upgraded, repaired or used as shops. Initially the plan was also, that the player could hire NPCs in these buildings, and then order the NPCs through interactions with them. The types of NPCs should depend on the types of buildings the player owned, for example if the player owned a church and a barracks paladins should be available to hire.

A later idea for the NPCs behavior would be the ability to perform complex behaviour such as running errands for the player, i.e. selling supplies, gathering supplies, buying or crafting items and performing other complex goals. While doing this the NPCs should be able to react to sudden changes occurring in the world around them. When these changes constrain them from reaching their goal, they should then choose appropriate alternatives with the most effective cost. This is where the previously described GOAP-based AI (see section 8.1) would be used to give a more smart AI that these NPCs would use [11]. Future development of enemies including bosses of the game could also benefit from the usage of GOAP-based AI to make the game more difficult, and avoiding stupid AIs, that might ruin the game experience.

Other development plans include the development of items, these items could be dropped by enemies and includes coins, chests, potions or perhaps recipes for new weapons. On top of these items, the development of a backpack to contain the picked up items were planned.

The plan for the ability system was to have the opportunity to change the abilities that the weapons can have upon purchase, so the same weapon types can have different abilities. However for this to be realised interaction with shops, more abilities need to be added and a more complex weapon creation system is required. Even though this is part of the first ideas about the game it is still very far from being complete.

The GameEngine has an enumeration GameState, which is used set the game to either a play or a "game over" state, so the appropriate game mechanic can run depending on the game state. To change the game state, the GameEngine checks all entities for the instance of ICastle and IPlayer to get the castle building's and the player's health. A better implementation might be for the player and castle themselves to call GameEngine via a service provided interface; this would effectively eliminate a loop each frame. The downside would be that player and castle would have to know that when they are dead or destroyed, they cause the game to be over.

To give the game more visual flavor the plan was to further develop the day/night cycle, so that enemies spawn only during night. Besides controlling spawning of enemies the intention

was that the game visually had a sun and moon that moved across the sky depending on the current ingame time.

## 10. References

- [1] Oracle. (2015) Characters [Online]  
Available: <https://docs.oracle.com/javase/tutorial/java/data/characters.html>
- [2] L. Richard. (2012, January, 19) What is an entity system framework for game development?  
[Online] Available: <http://www.richardlord.net/blog/ecs/what-is-an-entity-framework.html>
- [3] L. Richard. (2012, February, 16) Why use an entity system framework for game development?  
[Online] Available: <http://www.richardlord.net/blog/ecs/why-use-an-entity-framework.html>
- [4] B. Scott. (2002) A Data-Driven Game Object System [Online] Available:  
[http://scottbilas.com/files/2002/gdc\\_san\\_jose/game\\_objects\\_slides.pdf](http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf)
- [5] H. Allen. (2003, August, 1) Why extends is evil [Online] Available:  
<http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html>
- [6] P. Manickam, S. Sangeetha, and S.V. Subrahmanya. Component-Oriented Development and Assembly: Paradigm, Principles, and Practice using Java. Infosys Press. Taylor & Francis, 2013
- [7] K. Darius. (2017) Spelunky Generator Lessons. Part 2: Generating the Rooms [Online] Available:  
<http://tinysubversions.com/spelunkyGen2/>
- [8] S. Chris. (2014, July, 17) Behavior trees for AI: How they work [Online] Available:  
[http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior\\_trees\\_for\\_AI\\_How\\_they\\_work.php](http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php)
- [9] SourceMaking (2017) State Design Pattern [Online] Available:  
[https://sourcemaking.com/design\\_patterns/state](https://sourcemaking.com/design_patterns/state)
- [10] B. Fernando (2013, October, 24) Finite-State Machines: Theory and Implementation [Online]  
Available:  
<https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--game-dev-11867>
- [11] O. Brent (2014, April, 23) Goal Oriented Action Planning for a Smarter AI [Online] Available:  
<https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--game-dev-11867>
- [12] M. Micah, M.C. Robert. Agile Principles, Patterns, and Practices in C#. Prentice Hall, 2006, July, 20

## Appendix A: Dictionary

This appendix describes the special words used throughout the report and system, words that are not expected to be known to the reader.

### Video Game terminology

Word	Description
Sidescroller	A video game genre, for games that only moves in few directions in 2D space, typically only from the left to right direction.
AoE	Area of effect. An effect that covers a large area, such as the effect of a grenade, whereas a bullet has no area of effect.
DMG	Damage.
Chunk	A matrix of tiles (besides the middle chunk all chunks are 800x600 pixels)
Tile	The parts used to fill the chunk (each tile is 100x100 pixels)
NPC	Non-playable character. Only refers to characters who are neutral or allied to the player.
Player	The playable entity that the user controls.
Enemy	A non-playable entity that is an enemy to the player and allies.
RPG	Role Playing Video Game. Typically includes RPG elements such as items, abilities and quests.
Item	An entity the user/player can pick up.
Quest	A task the user/player need to complete, so that he can receive a reward.
Ability	Either an entity or an action a character, building or item, can perform to manipulate themselves, the game world or other entities.
Melee	Close ranged attack object, such as a sword or knife, or an object that move only within the arm's length of a character.
Ranged	Long ranged attack object, such as a bow, or an object that moves over great lengths.
Spawning	The action of creating an entity at a position.
Health	The health value of a character or building.

Healthbar	The GUI element of some health, showing both the max health and the current health.
Paladins	A holy soldier; often described as a melee soldier who wield healing abilities
Barracks	A place where the player can recruit soldiers
Chests	An object that contains treasures(e.g. new weapons)
Biome	A biome is an ecological community(e.g. forest, grassland, swamp or desert)
World	Represents the game world containing all entities present in the game.

## System terminology

Word	Description
SilentUpdate	The component used for unloading and loading components dynamically.
Common Packages	A component library for shared data to be used between other components.
SPI	Service Provider Interface
API	Application Programming Interface

## Appendix B: Game manual

Here the keyboard input and the instructions of the game will be described, to know how the game is played. A picture describing the different game features will also be included.

### Keyboard input:



### Goals:

You need to protect your castle gate from being destroyed by the enemies. They will appear from both the left and right side of the castle gate. The game ends when either you, the player, is dead or the castle gates are destroyed; when this happens, the game is over and your gold is your score.

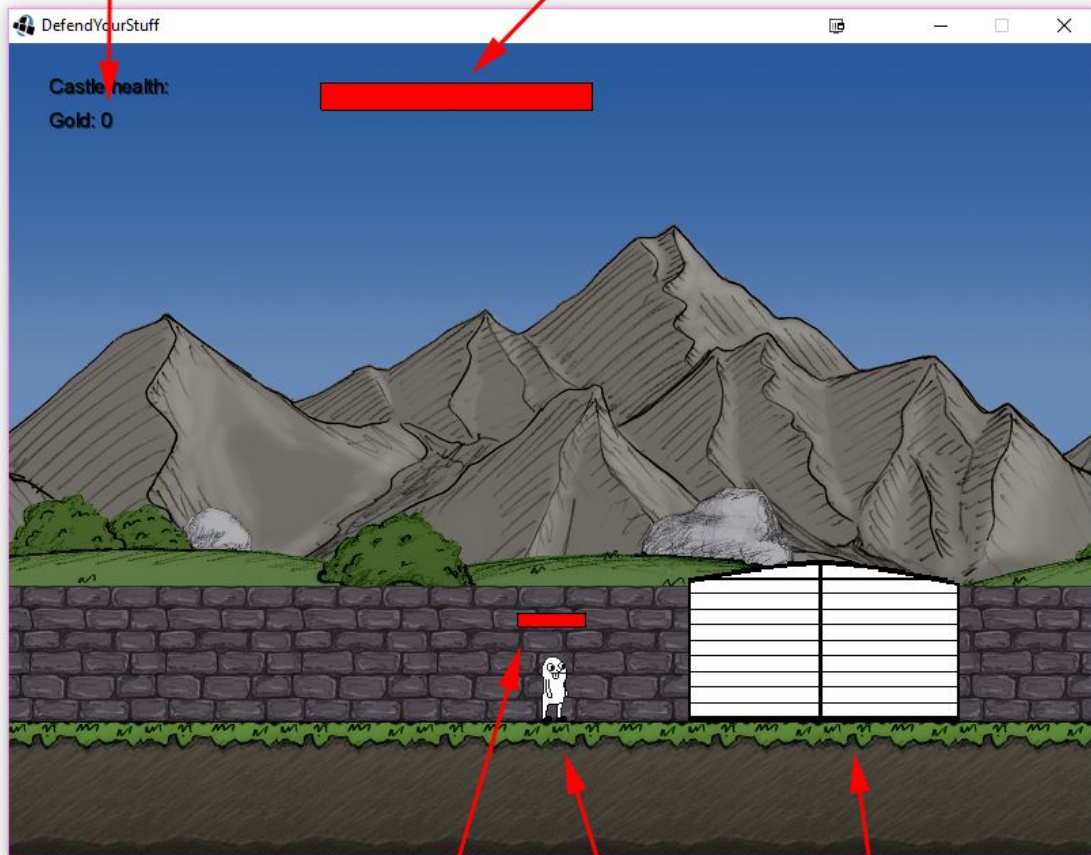
### Instructions:

Move with the keyboard and shoot at the oncoming enemies to keep them from destroying the castle doors. Each enemy kill, will increase the player's gold.



Gold amount (score)

Castle door health



Player healthbar

Player

Castle doors

## Appendix C: Tables

### C.1 - All functional requirements

ID	Name	Description	MoSCoW	Story Points
01	Player: Health	Player have health that can be updated.	Must	3
02	Player: Mana	Player have mana that can be updated.	Should	3
03	Player: Move direction	Player can move left and right or up when jumping.	Must	5
04	Player: Aim with mouse	Locate the mouse in the screen grid and use that location as the current aim point.	Must	5
05	Player: Attack	Adds an ability to the world	Must	3
06	Player: Current weapon	Player has a reference to the weapon object it is using.	Must	5
07	Player: Gold	The player has gold that it can use to buy weapons and NPCs or upgrade buildings.	Must	3
08	Player: Interact with world	When the player hitbox collides with a building the player is given the option to interact with the building. Depending on the building different options are given.	Should	21
09	Player: Experience	The player gets experience when killing an enemy.	Should	5
10	Player: Level up	The player levels up when enough experience is gained.	Should	3
11	Player: Backpack	The player has an inventory with items and weapons gathered.	Could	8
12	Player: Use item	The player can use items in their inventory.	Could	3
13	Enemy: AI behavior	Defines the specific AI behavior for different types of enemies.	Must	40
14	Enemy: Attack	When an enemy ability collides with a building or the player, some damage is done.	Must	5
15	Enemy: Give gold	When an enemy is killed, the player receives a set amount of gold.	Must	2
16	Enemy: Give experience	When an enemy die, the player receives a set amount of experience points.	Should	2
17	Enemy: Drop item	Enemies can drop items upon death.	Would	2
18	NPC: Move direction	The NPC can move left or right.	Could	5
19	NPC: Health	The NPC has health that can be updated.	Should	3

20	NPC: AI behavior	Defines the specific AI behavior for different types of NPCs.	Should	40
21	NPC: Attack	The NPC can deal damage to enemies.	Should	5
22	NPC: Interaction with player	When the player collides with the NPC the player is given the option to interact with the NPC. Depending on the NPC different options is given.	Could	8
23	Item: Coins	The enemies can drop coins which gives gold if the player collides with them.	Could	3
24	Item: Spawn treasures	Treasures can be randomly spawned in the game.	Could	3
25	Item: Tooltip	Items can have tooltips which explains what the item does when used.	Could	5
26	Item: Generate item	Generate the items that can be dropped or found.	Would	8
27	Weapon: Generate weapon	Generate the weapons that can be bought or found.	Must	8
28	Weapon: Base stats	Variables that define the range and damage of the weapon.	Must	3
29	Weapon: Abilities	Abilities that the weapon can use, and how the weapon empowers the abilities in certain ways.	Must	13
30	Weapon: Kill counter	A weapon will count how many enemies that have been killed.	Would	2
31	Weapon: Level up	A weapon can level up after a set amount of enemies have been killed.	Would	2
32	Weapon: Passive abilities	Passive abilities that the weapon can have.	Could	5
33	Building: Health	Buildings have health that can be updated when damage is done to the building.	Must	3
34	Building: Castle	A main castle building is created for the game with rules for the main objectives of the game, like the game over condition.	Must	2
35	Building: Buildings can be destroyed	When buildings health reaches zero they get destroyed and turns into a rubble building.	Must	2
36	Building: Upgrade buildings	Upgrade the building interacted with.	Should	8
37	Building: Upgrade hierachy	A hierarchy over available or already done upgrades.	Could	8
38	Building: Buy weapon.	The player can buy a weapon by interacting with the blacksmith building.	Should	3
39	Building: Sell weapon	The player can sell a weapon by interacting with the blacksmith building.	Could	3
40	Building: Hire NPC	The player can hire a NPC by interacting with the mercenaries building.	Could	3

4 1	Building: Repair	The buildings can be repaired after taking damage.	Could	3
4 2	Map: Generate map	Generate the rest of the map with blocks such as rubble and earth outside of the predefined chunks.	Must	21
4 3	Map: Predefined "Chunks"	A collection of blocks defined for a specific part of the map, that will work as templates to be chosen for generation of the map.	Must	13
4 4	Map: Generate dungeon	Generates a dungeon that contains enemies.	Could	21
4 5	Collision: Collision detection	Detect when two objects collide	Must	13
4 6	Timer: Time system	A time system that defines a day and night cycle in the game.	Should	5
4 7	Game engine: Main menu	The game has a main menu.	Would	2
4 8	Game engine: Start game	The game can be started from the main menu.	Would	2
4 9	Game engine: Save game	The game can be saved.	Would	2
5 0	Game engine: Load game	The game can be started from a save file.	Would	2
5 1	Game engine: Pause game	The game can be paused.	Would	2
5 2	Game engine: Difficulty	The difficulty can be changed in the main menu.	Would	2

## C.2 Component dependency depth

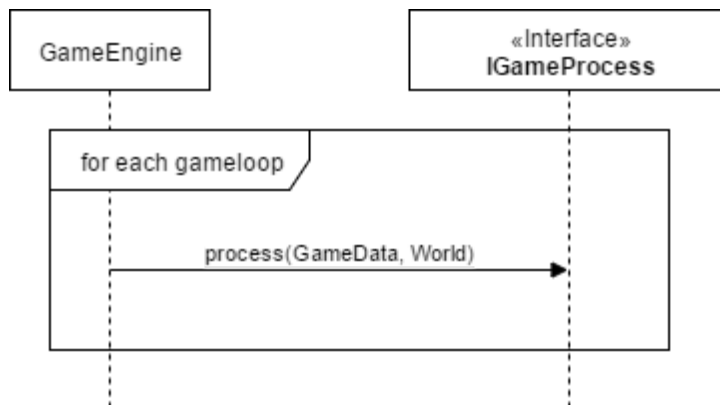
Number:	Componenten Name:	Depends on:	Dependency depth:
1.	GameEngine	Common, CommonMap, CommonPlayer	5
2.	Player	Common, CommonAbility, CommonBuilding, CommonCharacter, CommonEnemy, CommonItem, CommonPlayer, CommonWeapon	4

3.	Collision	Common, CommonCharacter	3
4.	Map	Common, CommonBuilding, CommonMap	5
5.	Ability	Common, CommonAbility	2
6.	Building	Common, CommonAbility, CommonBuilding, CommonEnemy, CommonPlayer	4
7.	Enemy	Common, CommonAI, CommonBuilding, CommonCharacter, CommonEnemy, CommonPlayer, CommonAbility	5
8.	AI	Common, CommonAI, CommonAbility, CommonCharacter, CommonEnemy	5
9.	Weapon	Common, CommonAbility, CommonWeapon	3
10.	DayNight	Common, CommonCharacter, CommonEnemy	3
11.	Common	-	0
12.	CommonWeapon	Common, CommonAbility	2
13.	CommonBuilding	Common, CommonAbility, CommonWeapon	3
14.	CommonPlayer	Common, CommonCharacter, CommonWeapon	3
15.	CommonEnemy	Common, CommonCharacter, CommonAbility	3
16.	CommonAlly	Common, CommonCharacter	3

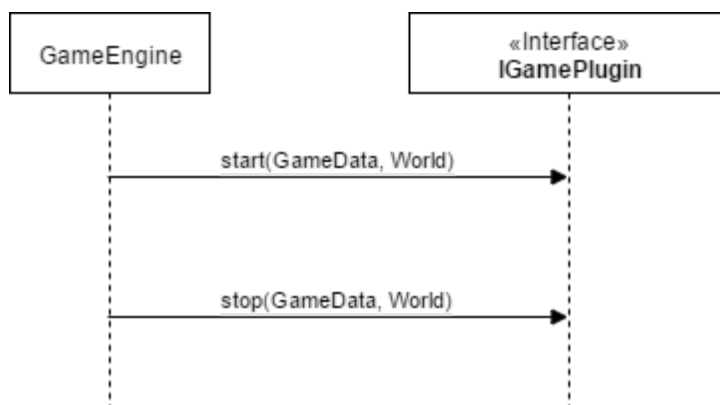
17.	CommonItem	Common, CommonAbility	2
18.	CommonCharacter	Common, CommonAbility	2
19.	CommonMap	Common, CommonBuilding	4
20.	CommonAI	Common, CommonCharacter, CommonEnemy	4
22.	CommonAbility	Common	1

## Appendix D: Diagrams

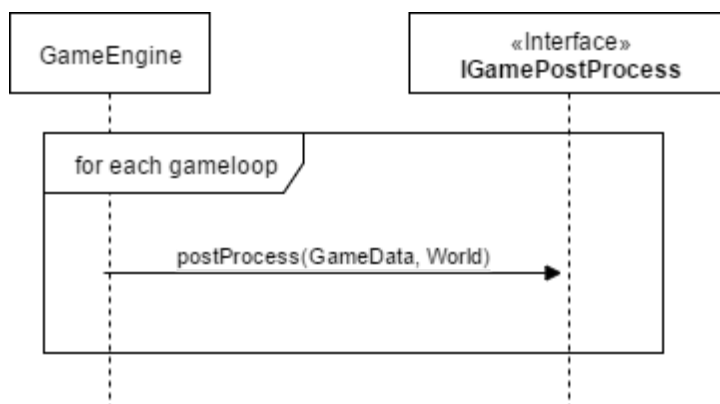
### D.1 IGameProcess sequence diagram



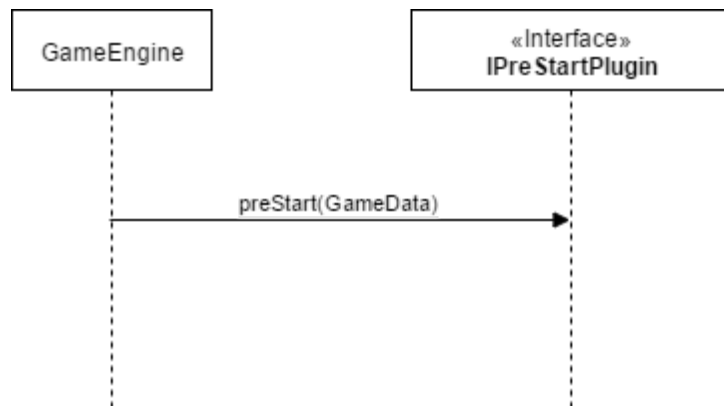
### D.2 IGamePlugin sequence diagram



### D.3 IGamePostProcess sequence diagram

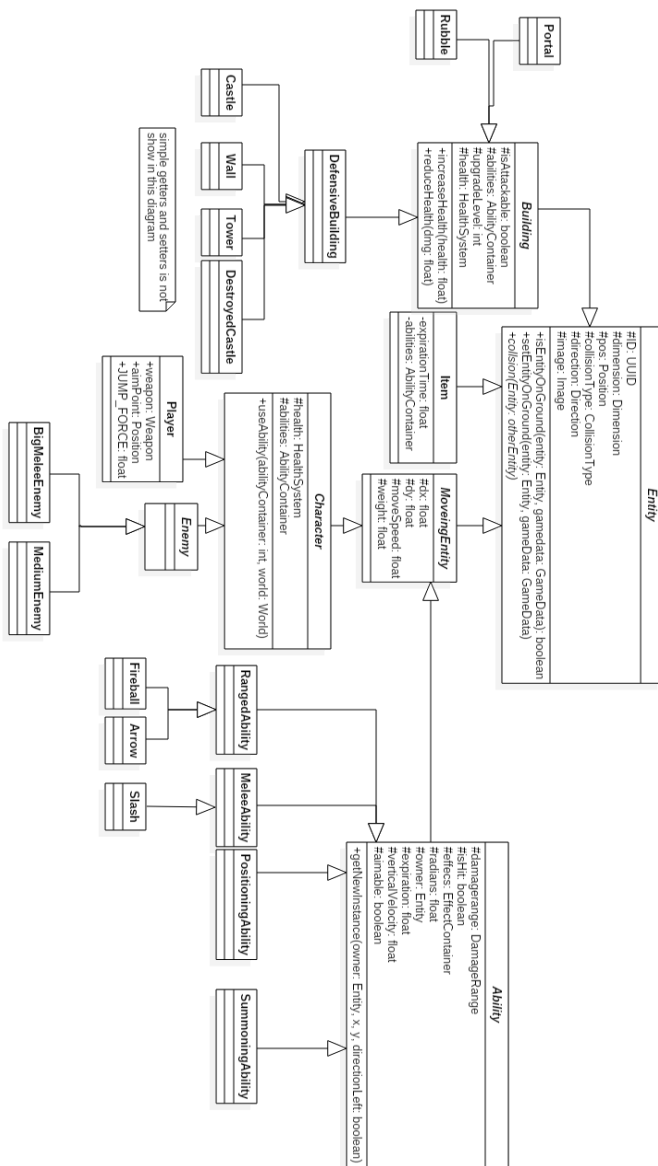


## D.4 IGamePreStartPlugin sequence diagram

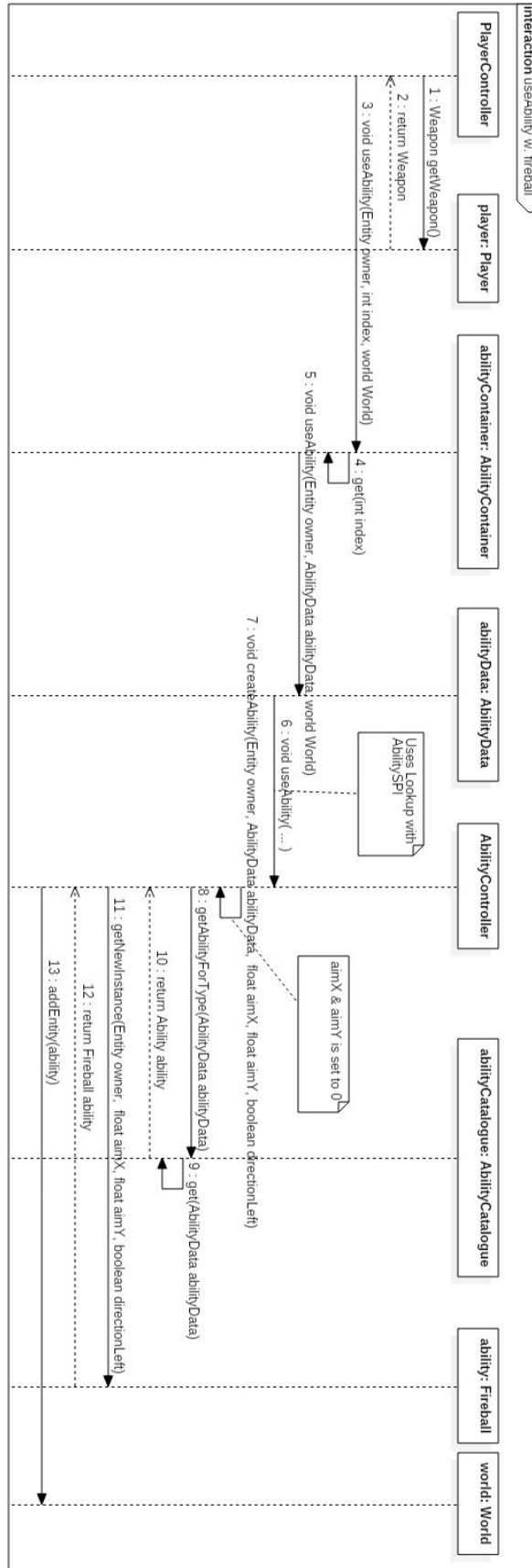




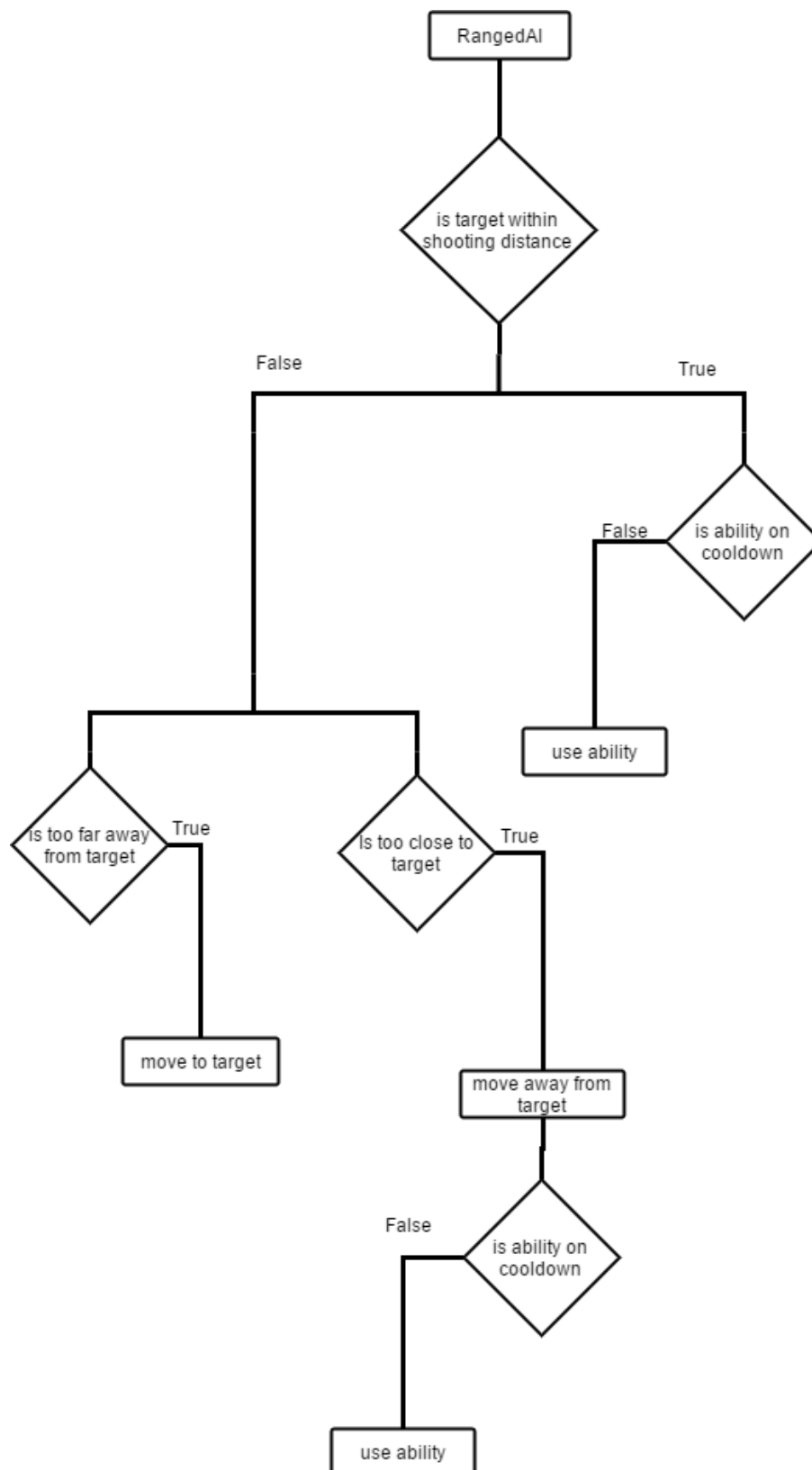
## D.5 Entity inheritance diagram



## D.6 useAbility sequence diagram



## D.7 Ranged enemy behaviour tree



## Appendix E: Pre/Postconditions

Interface/ Method	Precondition	PostCondition
<b>IMapUpdate</b>		
update()	The game needs another chunk	Another chunk have been added to the game
<b>IPlayerService</b>		
getPlayerMoveSpeed()	A player exists in the game	The player's moveSpeed has been returned to the caller
<b>IPlayer</b>		
<b>IBuildingService</b>		
upgradeBuilding()	There is a building in world, that can be upgraded	The building is upgraded.
<b>IDefBuilding</b>		
repair()	An entity is interacting with a building implementing this interface, and the building has reduced health.	The building's health is increased up to a certain maximum defined by the building.
<b>IShop</b>		
buyWeapon()	The player is interacting with a building implementing this interface.	The players weapon object is replaced with the bought weapon.
unlockWeapon()	Player has collided with an item and the item has called item action on the building containing items to be unlocked.	The next locked item in the shop is unlocked.

<b>Buildable</b>		
createXBuilding() x= some building	There is a position in the game for the building	A building now exist at the desired position.
<b>ICastle</b>		
getHealthSystem()	The castle should have health	A healthSystem is returned.
<b>AbilitySPI</b>		
useAbility()	An entity have an abilityData matching an ability	An ability have been added to the world
getAbilities()		A collection of all abilities are returned.
getXAbilities() X = subtype of ability		A collection of a subtype of abilities are returned.
<b>AI_Service</b>		
assignAttackAndDodgeEnemyAI()	A NPC uses this AI, is being processed	The NPC have acted according to the AI
rangedAI()	A NPC uses this AI, is being processed	The NPC have acted according to the AI
<b>IEnemyService</b>		
createMediumEnemy()		A MediumEnemy has been added to the world
createBigEnemy()		A BigEnemy has been added to the world
removeAllEnemies()		There are no enemies in world
<b>IWeaponService</b>		
createRanged()		A ranged weapon is returned to the caller
createMelee()		A melee weapon is returned to the caller