University of Southern Denmark, Odense

The Faculty of Engineering / MMMI

BSc in Software Engineering

SB4-CBS Component-based Software Engineering

Component-based 2D Game
SDU Royale
Semester project F19

Project group 11

Name	Email
Benjamin Kjølby Parbst	bepar17@student.sdu.dk
Jonas Støve Rasmussen	jonar17@student.sdu.dk
Josef Ngoc Can Pham	jopha15@student.sdu.dk
Peter Stærdahl Andersen	pande15@student.sdu.dk
Sigurd Eugen Espersen	siesp17@student.sdu.dk
Thomas Finch Rasmussen	trasm17@student.sdu.dk

Supervisor:
Jonas Steffen Boserup <jobos16@student.sdu.dk>

Project period: 04/02/2019 - 27/05/2019
Number of pages: 35

Abstract

This paper examines the analysis, design, implementation and overall thoughts behind the project, SDU Royale. SDU Royale is intended to be a 2D component-based shooting game that can load/unload its components during runtime and utilize both artificial intelligence and algorithms to make a game with a dynamic gameflow.

The main purpose of this paper is to document the workflows and development process from the project start till final hand-in of a functional and running implementation of the desired game. The reader will get a better insight into the thoughts and effort the group has put into the project with the main theme of component-based software engineering.

Preface

This project on the 4th semester is about developing a 2D component-based game. The main requirements are to develop multiple components which should be implemented and maintained separately and also being able to be added or removed during runtime.

In addition, semester courses are also part of the project and contributes with varying techniques obtained from these courses such as Software Components (SB4-KOM), Algorithms and Data Structures (DM507) and Artificial Intelligence (SB4-KI).

The project started on February 4th and ended on May 27th 2019 and has been conducted at the University of Southern Denmark in Odense. Throughout the project, the group has been under guidance from supervisor Jonas Steffen Boserup.

This paper is written by all 6 members of group 11, all of whom study Software Engineering on their 4th semester at the Faculty of Engineering under the Maersk Mc-Kinney Møller Institute. All members acknowledge their active participation in the project mutually through their signatures below.

[illegible]	Jonas Støve Rasmussen	[illegible]
Benjamin Kjølby Parbst	Jonas Støve Rasmussen	Josef Pham
[illegible]	[illegible]	[illegible]
Peter Andersen	Sigurd Espersen	Thomas Finch Rasmussen

Table of Contents

Table of Figures and Tables

I Reading overview

This paper is written in a way that allows it to be read in a chronological order for the reader to get the best understanding of the overall project and system, SDU Royale. Subchapters are not independent and should not be read out of context from the full chapter as previous subchapters may contain information about the content of the following section. If read as a PDF, the links in both Table of Content/Figures/Tables and the various referencing throughout the paper, will work as references to specific pages and will direct the reader to the location of that reference.

It is recommended to have a copy of the Java code open when reading the implementation section since references to the code will be used. Classes and methods will be marked using cursive font. in addition methods will be tagged with the suffix ().

References throughout the paper will be listed using IEEE Referencing, which includes in-text citations, numbered in square brackets, which refer to the full citation listed in the reference list at the end of the paper. The reference list is organized numerically, not alphabetically.

II Glossary

This section will present and explain various terms, as this paper contains different abbreviations and acronyms.

Term	**Description**
Maven	Maven is a software project management and comprehension tool.
nbm	A file type like .jar .txt etc. nbm stands for NetBeans Module. Maven can for instance manage a project's build
jar	*.jar* (Java ARchive) is a package file used to aggregate Java class files, meta-data and ressources. Components become deployable units in form of jar files.
xml	*.xml* (eXtensible Markup Language) is a markup language that can define a set of rules for encoding documents in a format that is both human-readable and machine-readable.
NBMS	Abbreviation for NetBeans Module System. NBMS is the foundation of the NetBeans Platform.
OSGi	Acronym for The Open Service Gateway Initiative - and is likewise the NetBeans Module System also a Java-based component system.
Runtime container	A set of provided modules used to handle the system's life cycle.
Update Center	Folder consists of a set of the system's components in the form of NetBean Module files (*.nbm*). Also an autoupdate catalogue file in *.xml* format is found in the folder..
FBC	Fragile Base Class - occurs when having a deep inheritance hierarchy. Whenever the base class is changed, it causes ripple effect of changes. (which is a problem)
SPI	Abbreviation from Service Provided Interface.
Interface	An interface is a reference type in Java similar to class. It is a collection of abstract methods. Classes, which implement the interface, will inherit these methods.
AI	Abbreviation from Artificial Intelligence.
Algorithm	A process or a set of rules to be followed e.g. in calculations or problem-solving operations.
MoSCoW	Acronym for the prioritization technique: Must have, Should have, Could have, Won't have

III System Guide

Following this section, is an explanation and through guide for starting and running the final system. It is preferable that the reader has knowledge to the NetBeans IDE to some extent.

How to run the game without dynamic load/unload

- Clean and build “SDU_Royale-parent”.
- Run the “SDU_Royale-app (app)” under “SDU_Royale-parent”, by right-clicking and pressing run (App has dependencies to all components).

How to run the game with dynamic load/unload

- Make sure that app has dependencies to all components.
- Set the apps configuration to deployment.
- Clean and build. (update center is generated under *\/rootFolder/application/target/netbeans_site/).*
- Remove all dependencies except for SilentUpdate from app.
- **If a local update center is desired:**
    - Locate “netbeans_site” folder, where all snapshots are generated, and copy these to a local “Update Center” folder and specify that specific path in SilentUpdate (Bundle.properties). (org_netbeans_modules_autoupdate_silentupdate_update_center=PATH)
- **Else if the auto generated update center (netbeans_site) is desired:**
    - Continue!
- Run the game as before, however make sure the name of the project folder is exactly “Sem4_CBSE”.
- The application is now running, but to enable load/unload the application must be restarted, as the path to update.xml is being generated during runtime.
- At the second run, locate the “updates.xml” in the “netbeans_site” folder, and edit this file.
- Outcomment each component that you wish to unload.
- Add the uncommented code again to load the component once again.

1 Introduction

In this semester project the task is to develop a component-based 2D game. The project will include techniques from courses such as Software Components, Artificial Intelligence and Algorithms and Data Structures. The main objective is to develop a game, create and form various components which can be added or removed during runtime. Also the components should be implemented and maintained separately. The idea for the project is to develop a 2D top-down shooting game inspired by the game “The Binding of Isaac” - which is a roguelike video game. The game is called SDU Royale with the initial idea of a battle royale game between the Faculties of SDU. The idea later proceeded into only having two faculties: the Faculty of Engineering against the Faculty of Humanities.

1.1 Purpose and goals

The overall purpose and goal is to maximize the learning outcome throughout the semester for the project team. The ideal outcome would be to improve the knowledge and understanding gained through the various courses. The main learning objectives are the following:

- Using component frameworks for development.
- Using tools to develop component-based software.
- Applying algorithms and data-structures on concrete problems.
- Choosing suitable artificial intelligence techniques to solve problems.

These learning objectives should also help improve the group members’ software engineering skills by enhancing the problem solving capabilities through a more component-based approach in addition to optimizing code with the competences obtained from the algorithms and data-structures course. All of this would in return benefit the future lives as professional software engineers.

2 Requirements

Requirements are an essential part in software development. It is the first step out of five core workflows in the development process: Requirements, Analysis, Design, Implementation and Testing. Requirements are used to specify all functionality and any constraints of the desired software. In other words, requirements describe what the system should do and how it should do so. This is important as it helps stakeholders or any non-developers who might have an interest in the software to understand the product better. The requirements can be split into two categories: Functional requirements (functionality) and Non-functional requirements (constraints).

2.1 Functional requirements

Through brainstorming and discussion, the project team tries to come up with various functionalities of which an end user of the system should be able to make use of in-game, but is also based on the project description in appendix A2. The end user of the game will be referred to as the player. All functionalities will be written as user stories, which is a short and simple description of the feature told from the perspective of the player.

Name: Start game	**Name:** End game
ID: 1	**ID:** 2
Description: As a player, I am able to start a game	**Description:** As a player, I am able to end a game
Precondition: The game is not running	**Precondition:** The game is running
Postcondition: The game starts	**Postcondition:** The game stops

Name: Move player
ID: 3
Description: As a player, I am able to control and move the in-game player
Precondition: User input received
Postcondition: The in-game player does an action based on the user inputs

Name: Pick-up (objects)
ID: 4
Description: As a player, I am able to pick up objects such as items or weapons
Precondition: The in-game player is near or on top of the specific object
Postcondition: The in-game player possesses the object in the inventory (if weapons then equipped)

Name: Shoot weapon
ID: 5
Description: As a player, if I possess a weapon, I should be able to fire the weapon against enemies
Precondition: Possession of a weapon
Postcondition: The in-game player fires the weapon and spawn projectiles or bullets

Name: Hit enemies
ID: 6
Description: As a player, if I possess a weapon, I can fire the weapon and hit enemies to kill them
Precondition: Possession of a weapon
Postcondition: Enemy lost health

Name: Obtain points
ID: 7
Description: As a player i am able to obtain points
Precondition: Enemy has been hit with a weapon
Postcondition: Obtained points

Name: Hit by enemy (death)
ID: 8
Description: As a player, if I collide with an enemy or get hit by an enemy, I will lose HP and die (0 HP)
Precondition: Lost health below 0
Postcondition: Game over

Name: Obtain highscore
ID: 9
Description: As a player, I can watch my highscore after the game has ended
Precondition: The in-game player has died and game over
Postcondition: Highscore is visible

Name: Collision (borders, projectiles, enemies)
ID: 10
Description: As a player, I can collide with different objects such as borders, projectiles and enemies
Precondition: The in-game player moved into something
Postcondition: Collision returns true

Name: Win game	**Name:** Lose game
ID: 11	**ID:** 12
Description: As a player, I can win the game by fulfilling the win conditions	**Description:** As a player, I can lose the game by dying
Precondition: Killed x amount of enemies	**Precondition:** Collision with projectiles or enemies
Postcondition: Game ended, highscore is visible	**Postcondition:** Game over, highscore is visible

Name: Pause game	**Name:** Resume game
ID: 13	**ID:** 14
Description: As a player, I can pause the game	**Description:** As a player, I can resume game play
Precondition: Game is running	**Precondition:** Game state is paused
Postcondition: Game state is now paused	**Postcondition:** Game is now running

Table 2.1: Collection of requirements displayed as user stories.

2.2 Non-functional requirements

Non-functional requirements are specific criteria and constraints, which are put on the project either by the developers or stakeholders. These requirements are normally within categories such as functionality, usability, reliability, performance, supportability, security etc. These are also called quality attributes of the system. However, in this project the focus will mainly be on maintainability, functionality and a bit of performance for the component-based game. Some of the design and implementation constraints for the project are following:

- The game should run smoothly, thus ensuring minimal disturbance to the gameplay.
- The game has to include Player, Enemy, Weapon, GameEngine and Map components.
- The Player, Enemy and Weapon have to implement provided interfaces that allow the components to be updated and removed dynamically at run-time.
- Components must be handled and maintained separately.
- A component framework that supports multiple classloaders per component and component versioning has to be applied (NetBeans Module System or OSGi).
- At least one component should implement an artificial intelligence technique.
- Data-structures and algorithms have to be applied and documented.

2.3 Prioritization

The specified functional requirements are sorted by using MoSCoW. The MoSCoW method is a prioritization technique to sort and prioritize important core functionality apart from nice-to-have features. The term MoSCoW is an acronym derived from the first letter of each of four prioritization categories: Must have, Should have, Could have and Won’t have.

Must have	**Should have**	**Could have**	**Won’t have**
ID 1: Start game	ID 6: Hit enemies	ID 7: Obtain points	ID 4: Pick-up
ID 2: End game	ID 8: Hit by enemy	ID 9: Obtain highscore	ID 11: Win game
ID 3: Move player	ID 10: Collision		ID 12: Lose game
ID 5: Shoot weapon			ID 13: Pause game
			ID 14: Resume game

Table 2.2: Presentation of requirements listed using MoSCoW prioritization.

3 Analysis

This section contains an analysis of the functional and nonfunctional requirements listed in previous section. The purpose of this is to establish an understanding of what the system should do in order to fulfil the requirements. The analysis includes, among others, diagrams created to illustrate game functionalities and how they are prioritized. Furthermore, it includes a description of what the component framework provides for this system and what roles and responsibilities the different components should have.

3.1 Use Case Diagram

Shown below in Figure 3.1 is the use case diagram conducted by the project team. The model visualizes the functional requirements listed in the previous chapter together with MoSCow prioritization.

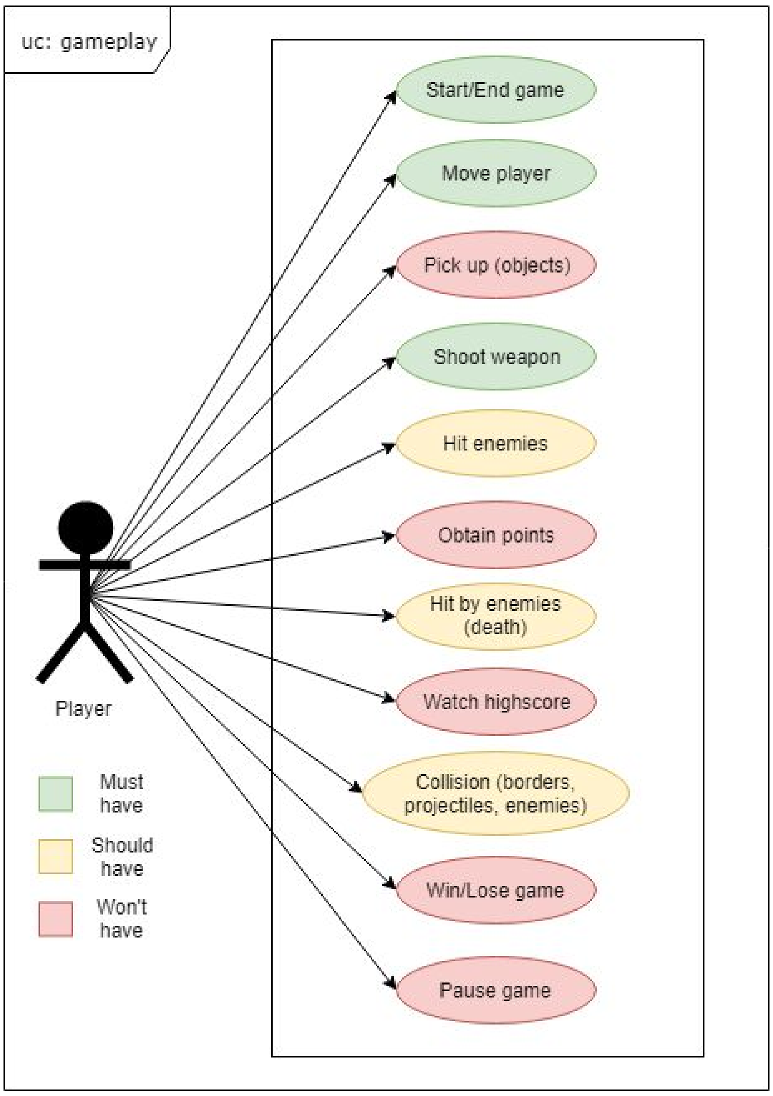


Figure 3.1: Illustration of the use case diagram included MoSCoW.

3.2 Component Framework

The main objective of the project is to develop a component-based game. Therefore the first choices were to go from a higher hierarchy of inheritance based entities and instead separate game logics into components.*'Fragile base class problem'* (FBC) should be avoided because of how systems normally are built with high hierarchy and inheritance. This causes problems with a number of dependencies through various subclasses that all depends on a superclass. The FBC problem states that, if a situation occurs where you need to change the base class, all the subclasses would also need to be changed. The problem is intended to be solved by using component-based development and more specifically by using a module system such as NetBeans Module System or OSGi Module System.

The component-based software development solves the FBC by handling your data in different components. A component library is often used to contain Service Provided Interfaces (SPI, which favors composition over inheritance. Based on this, the system should be flexible and extensible. (Illustrated by Figure 3.2)[1]

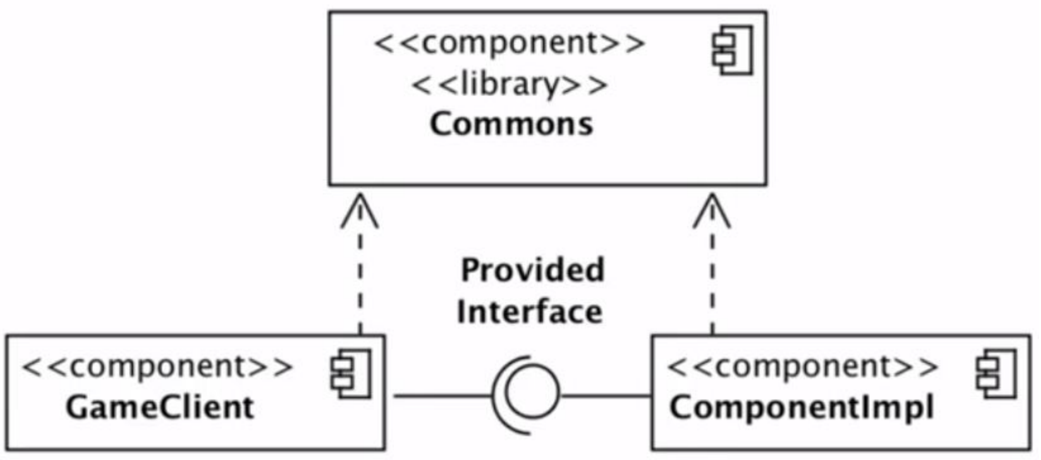


Figure 3.2: Illustration of component-based systems.

It is common to utilize a framework with component-based software support. One of these is the NetBeans Platform and its NetBeans Module System which will be used in this project.

The NetBeans Module System resides in the NetBeans Platform and has the ability to dynamically discover components and resolve dependencies between modules. The NetBeans Module System makes it possible to download new modules as well as deploy/redeploy modules at runtime. In this context, a module is a deployable JAR file with manifest entries that allows the NetBeans Module System to recognize it.
The NetBeans Module System is a runtime container designed for applications to discover their components at runtime which handles inter-library dependencies explicitly which results in applications being unable to start in a state where its dependencies are not satisfied.
In addition to allowing components to be deployed/removed at runtime, the NetBeans Module System enforces low coupling throughout the project by following API design.

A module cannot reference classes in different modules without declaring a specific dependency to that module. The other module also has to agree to be referenced which is done by exporting their package. This design choice ensures that different classes will not be referenced unnecessarily and thereby enforcing low coupling.[2]

3.3 Components

Before designing the program, it is important to know which components to include and what roles and responsibilities they each have. By analyzing both the functional and non-functional requirements it is possible to derive these components through relevant nouns mentioned. This analysis led to components such as; Player, Enemy, AI, Map, Weapon, Collision, AI, Highscore, Items, Point, Core (GameEngine).

However, this way to find all components is not complete. As a component framework is to be used for building this system it is required to also consider what sorts of components it typically includes. One goal when having this type of system is to avoid build time dependencies. Instead of having the application to depend on all service providers and their implementations, it is required to depend on interfaces instead. These shared interfaces should be located in a separate common library component which the service providers and the application depends upon. The service providers are then able to implement these interfaces from the new Common component. Another purpose of the Common component is to hold not only common interfaces but also common data so only one component is responsible of exposing a common library with other components instead of making all data visible to everyone.

Another important component to fulfil the requirement of having a game where it is possible to load/unload during runtime is the Auto Update Service with NetBeans Module System. The goal is to be independent off the Visual UI manager in NetBeans but still be able to load/unload modules during runtime. The SilentUpdate component is a good solution for this purpose. It will point to the update center which is generated through Maven and manage the lifecycle of other components. All other dependencies from the applications have to be removed except for SilentUpdate to make it responsible for this update handling. Through this analysis all the needed components for the project were found which include: Player, Enemy, AI, Map, Weapon, Collision, AI, Highscore, Items, Point, Core (GameEngine), Common and SilentUpdate. However, based on prioritization of the requirements through MoSCoW, the focus has been set on the following components:

Core

This module will contain the game engine and be responsible for maintaining the game loop. It will depend on the Common module which contains the applications service provided interfaces.

Common

All the service provided interfaces in the application will be stored here along with common state and behavior that is required by the Core module. The service providers will reside in independent modules that implements the service provided interfaces in this module.

Player

This module will be responsible for everything that is concerned with the player and how it behaves in the game. The service providers of this module will reside in the Common module and this will merely be a service provider of the player service in the game.

Enemy

Everything concerned with the enemies of the game will be in this module. The behaviour of the enemies will not be held here but in a separate AI component. This will hold the service provider of the enemy service as the service provided interface lies in the Common module.

Artificial Intelligence

The AI component will be responsible and hold everything concerned with programming behavior for components of the game, at first only enemy will use this component. This will only hold the service provider of the AI service because the service provider is held in the Common module.

Weapon

The weapon system of the game will be held in this component along with the functionality of the weapons. This functionality may vary with different weapons. The service provided interfaces resides in the Common module which is implemented by the service providers of this module.

Map

This component will hold the map or the level of the game on which the game is played. This will include state and behavior which may vary between different maps as something might be possible in one map but not in another. This will be the service provider of the map service and implement the service provided interface which resides in the Common module.

Collision

A constant collision check is needed in order to make sure the player and enemies do not wander beyond the scope of the map or into each other. This functionality will be held in this component which will check coordinates and make sure that nothing collides in the game.

The service provided interface lies in the Common module and is implemented by the collision service providers in this module.

3.4 Artificial Intelligence

Based on the description of the game idea and the requirements (section 2), an analysis about possible usage of artificial intelligence (AI) techniques is needed in order to fulfil these requirements. With reference to the requirement section a few hints are given to how certain entities should behave in the game including the enemy.

Requirements and AI

The purpose of the enemy is to make the gameplay harder and more troubled for the player. The use case ‘Hit by enemies’ directly implies that an enemy’s movement should affect the player’s health. This description of the enemy’s movement suggests a movement pattern that allows the enemy to follow the player through the map. Given from this, it should be the enemy component who should implement an AI technique as one of the official requirements demand. This component should be load- and unloadable during runtime in order to fulfil the requirement of having a complete component-based system. This implies that all use of this implementation of AI should be accessed through a SPI and not directly from the service provider.

AI Movement Analysis

The AI technique used for the enemy should be more advanced than using simple trigonometry. During the beginning of the analysis, it was initially considered to use a movement solution with trigonometry making the enemy move towards the player’s position in the shortest way possible. However, to adequately satisfy the AI requirement the game should make use of an AI technique that includes starting goals, end goals and a heuristic function that can calculate the costs of each move towards the end goal.

The movement of the enemy should depend on the enemy’s role in the game which as mentioned is to follow and damage the player. Combining this role of the enemy to what is known about AI, the enemy becomes a goal-based agent [3]. Followed by the role of the enemy, the environment for this agent should be considered. The map of 1200 x 700 pixels containing possible obstacles on the way and with each pixel being a location or coordinate for the enemy to enter makes the path to the goal rather difficult. Furthermore, the environment is dynamic as it is a game where the goal of the agent is constantly moving and also observable as the enemy is free to get information of the state of all coordinates/positions. Knowing the environment is critical for an AI technique to bring the agent to its goal. When the environment is modeled, the next step is to determine how to get from A to B. This is done by searching.

Searching is about looking for a sequence of actions that results in the best possible solution which in this case is the path with the lowest cost possible [4]. Here the lowest cost would be the least number of steps from the enemy's current position (initial state) to the goal which is the players position (goal state). A state is a snapshot of the world with all important information the enemy needs to know given the task at hand. In this game, the state should have information of the world's positions or coordinates. Given the dynamic gameplay, the state is constantly changing. Choosing the right type of search is important as it depends on what problem you want to solve. As mentioned, the cost variable should be used in this game and therefore it is not desirable to use an Uninformed Search Algorithm to generate a path for this problem-solving agent being the enemy. This type of search including Breadth-First Search and Depth-First Search offers a structured discovery of the world but it is not intelligent as it only offers a prioritized walk-through. This will result in problems when it comes to computation time and an unnecessary use of memory. Metrics can be useful as it allows to compare a state x with state y based on what is best. The best option here is entering the next state with the lowest cost towards the goal.

A Greedy Best-First Algorithm could improve this as it will make the agenda of the visited coordinates/states a priority queue that orders the next states to be visited based on metric of cost with the states that are least expensive explored first. This improves the intelligence as the algorithm pay attention to the cost of executing when changing between states and not visit the same states more than once. However, the problem with this type of algorithm is that it is short-sighted and only checks the distance or cost to the next circle of states ahead. By doing so it might think that the next state is the optimal but if it planned further it may realize that it would not be the case. Therefore, it should continually estimate the cost to the goal in order to make it optimal.

To control this goal, estimating an extra part is added to the process to understand how close the enemy is to its goal state [4].
This part is called a heuristic and is the estimated cost of the least expensive path from the current state to a goal state. By using this technique, the enemy would travel along the path of states where it always will choose the next state depending on its heuristic value. However, this technique alone will not result in the optimal path. Combining these two approaches would seem rather smart and actually give the targeted result which is desired in this game. A few search algorithms make use of this combination as they consider the value of the next state x to be the cost to reach it plus the estimated distance from this state to the goal making them both complete and optimal. It is important to notice that the algorithm will only be optimal if an admissible heuristic is used and if it is consistent.

Algorithm Performance

To find the right match for this game, another criteria should be considered before selecting an algorithm; *Performance*. Therefore, the purpose of this section is to get an idea about a few known pathfinding algorithms' performances. There will be no detailed explanations of the algorithm as the focus and goal is to find the algorithm which will perform the best in this game environment. Both algorithms mentioned below fulfils the requirement of being both complete and optimal.

The A* algorithm is an informed search algorithm used in e.g. computer games for pathfinding. It builds its calculations of finding the optimal path from initial state to goal state by combining the cost-focused approach with a heuristic function. D* Lite which is also an informed search algorithm has similar functionality to D* but is simpler and uses an incremental approach. This makes it faster than D* and therefore more interesting for this game. As mentioned, completeness and optimality are important but time complexity regarding the dynamic gameplay is just as important. Research shows that D* Lite is faster than A* when it comes to larger scaled maps with a huge amount of positions or nodes. One of the reasons for this is that D* Lite saves information about notes surrounding the current node being the current position of the player in the game. This makes the D* Lite able to do a fast re-planning. A* does not save this information and must rescan the whole map each time the goal state moves or if sudden blocks on the current calculates path appears. Due to this important difference in performance between the two, D* Lite should be implemented in this game [5]-[8].

4 Design

The section presents a detailed description of the design choices that have been made based on the analysis (previous section). One of these choices is about the structure and architecture of the system. Another choice revolves on how the system is build and is illustrated in the design component class diagram in addition to descriptions of the component contracts. Lastly, some of the key components will be described in details.

4.1 Software architecture

In the previous chapter, it was decided to use NetBeans Module System(NBMS) which is built on top of the NetBeans Rich Client Platform. As the platform forms the entire foundation, it is interesting to view the entire architecture and specify the core features. The NetBeans Platform architecture is quite extensive (illustrated in Figure 4.1) but most important of all is that it is modular. Thus making it easy to create robust, flexible and extensible application as many tools are already provided and found in the architecture.

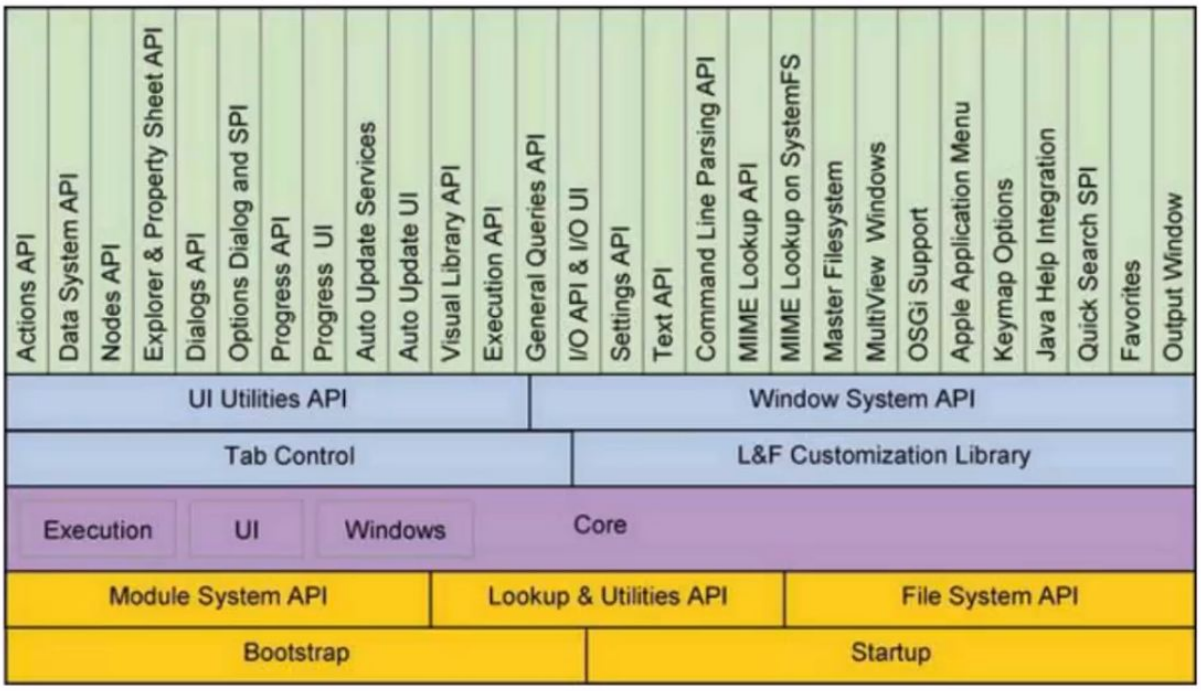


Figure 4.1: Presentation of the architecture of NetBeans Module System.[9]

At the bottom of the architecture you will see five yellow blocks. These modules constitutes the *runtime container* which can be seen as the very core of our component-based system. You will find two noteworthy additions such as Auto Update Services and OSGi Support among the green blocks at the top of the architecture figure. The Auto Update Services is another core feature needed in the project as it is used for dynamically loading/unloading modules. The OSGi Support can be viewed as a bonus feature for long-term development as it provides the opportunity to add and implement OSGi bundles or having both in future development.

The whiteboard component model is used since the project is using NetBeans Module System and it is desired to restrict it to pure *.nbm*. It utilizes the *Lookup* and Auto Update Service. The NetBeans Module System avoids a flat class-path by having an individual class loader in every module. The purpose of the project is not to design a NetBeans Desktop Application using the Visual UI manager so it will only depend on the core-startup (org-netbeans-core-startup) rather than NetBeans-Platform (org.netbeans.cluster). This will influence the way modules are loaded and unloaded but will be discussed later. *Lookup* will be used in conjunction with the service providers. The service providers will be used in the form of *annotation (@)* so that a service can register itself to the register (META-INF/services). The *Lookup* will look through the global register and use the key-value pair (key: interface, value: implementation) to find the implementation of a given service.

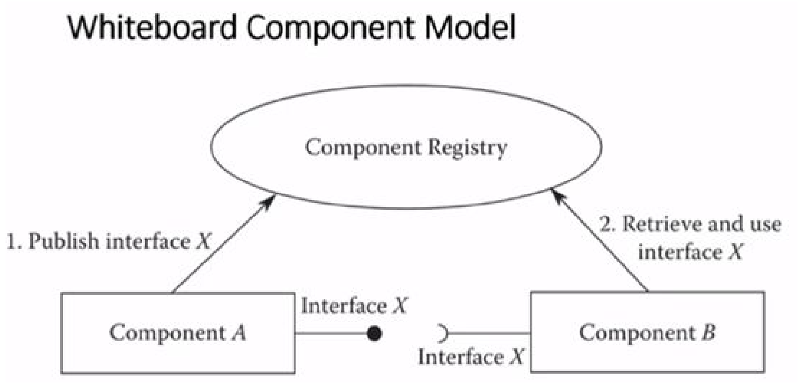


Figure 4.2: Presentation of the Whiteboard Component Model.[10]

By deploying the application, an update-center can be generated. This update-center contains the information about the different components together with their respective .jar file and META-data. Because the project does not depend on the NetBeans GUI, it is important to use another way to load and unload component. Since the NetBeans Platform supports multiple update-centres, it is important to specify which update-center you want to use to load in your components. In this project, a *Auto Update Service* will be used to look through the different modules and their current installation/uninstallation status.[9]

4.2 Components

The components of the application will be described here in the form of an overview, in the form of component contracts and how they interact with each other. The overview can be seen in the diagram below which describes how the *Core* module should depend on game logic from LibGDX library/API through maven and on *Common* which holds all the service provided interfaces. These should be implemented by all components which holds service providers of the application.

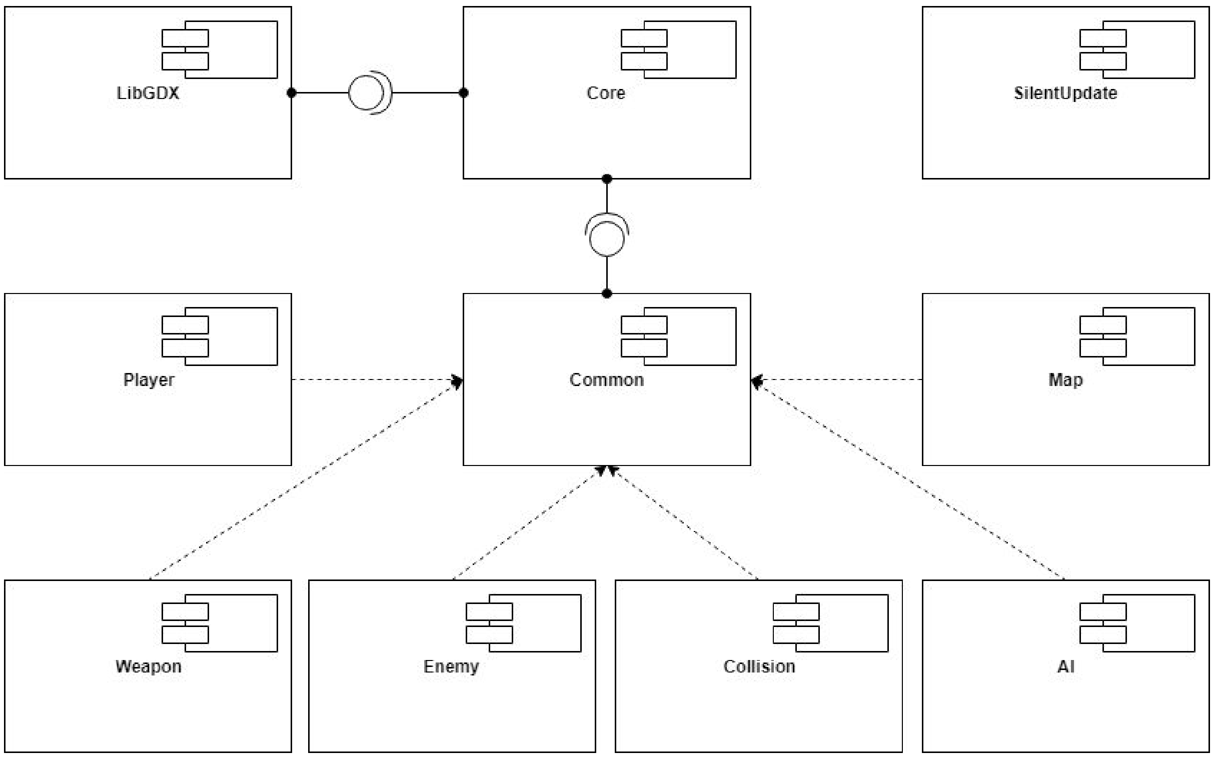


Figure 4.3: Illustration of the system's component diagram.

By having the SPIs centralized in the *Common* component it is much easier to load and unload during runtime with *SilentUpdate* because the SPIs of a given service is isolated from its service provider. This enables the components to avoid dependencies on each other and instead share a dependency to Common as shown in Figure 4.3.

Component Contracts

In this section, the service provided interfaces (SPI) that the game uses are further described. These SPI's are what differentiate the different classes in the components from each other.

SPI	IPluginService

Operation:	start(gameData:GameData, world:World)
	stop(gameData:GameData, world:World)

Description:	Handles the start and stop of components. Called by a component initializer, which listens for changes in the plugin services. If a new plugin (component) is added, that plugin is started and likewise stopped if removed from the list of plugins.
Parameters:	GameData - contains data important for the components cohesion between itself and the game. World - handles logic to add/remove from world.
Preconditions:	Lookup locates the newly added plugin that is not yet started, and calls the start method. Vice versa with stop.
Postconditions:	The plugin is either started or stopped depending on which method is called.

SPI	IControlService
Operation:	execute(gameData:GameData, world:World)
Description:	Called as the first thing by the game loop after it has rendered. The IControlService is responsible for handling inputs and positions for different movable entities. In some occasions, it is also used to create entities, made by different entities.
Parameters:	GameData - to access data important for the component's controls World - to get access to the list of entities
Preconditions:	The components have been loaded in by Lookup and their information added to the World
Postconditions:	The positions of all entities implementing IControlService have their respective action performed.

SPI	IPostProcessor
Operation:	execute(gameData:GameData, world:World)
Description:	Called at the end of game loop in every iteration, and is found by the component initializer. Handles logic for entities after they have been added to world.
Parameters:	gameData - handles game data such as time, user inputs, screen width and height World - handles logic to add/remove from world
Preconditions:	Entities have to be loaded into game by Lookup Update from IControlService has occurred beforehand
Postconditions:	Action between entities have occured

Table 4.1: Presentation of SPIs and conducted component contracts.

There are three different roles for the components of this application: *Core* which maintains the runtime and game loop, *Common* which holds all common data and SPIs and the service providers which implement the SPIs. *Core*, *Common* and *Player* represents these roles respectively.

Core

The game loop will run in LibGDX but should rely on implementation from the *Core* module. It is required that a class implements ApplicationListener from LibGDX and overrides methods that are called in the game loop. The most essential ones are create and render:

- Create: Instantiates and sets state before the game loop is started.
- Render: Calculates the positions for and draws all entities in the next frame

The module will depend on service provided interfaces from the *Common* module to process all the services of the application. The lookup will be used to find the service providers of these services and serves as our component registry in the whiteboard component model.

Common

All the service provided interfaces of the application will be held in the common module. This includes; one for processing entities of the game in the game loop, one for starting and stopping services of the game during runtime and one for processing services of the game in the game loop:

- IControlService:	An executable service for processing entities in the game.
- IPluginService:	An start- and stopable service for entities/services in the game.
- IPostProcessor:	An executable service for services in the game.

The module should also contain common data that is shared with all components. This should only include data that is not subject to change in order to avoid the fragile base class problem. This is includes key binding codes, display height and width of the window and coordinates of renderable entities:

- Entity:	The abstraction of most objects in the application. It contains the most common and crucial game logic to run the game.
- GameData:	Contains common data about the game in general such as display height and width.
- KeyBindings:	All key binding codes and control of the interaction with keys is contained here.
- State:	Represents a state for the artificial intelligence. It contains behavior to calculate a moving route for the enemy.
- Pair:	Contains information about the neighbour State coordinates.
- World:	Contains the state of the world in the game such as collections of all entities and behavior to access this state.

The section below is describes how SPIs are used by service providers. However *Player* will be used as an example.

Player

The player in the game along with its controls and graphics will be contained in this module. This includes methods to update the coordinates of the player based on input, as well as movement speed and how the player is controlled by the one playing the game. The graphics consists of textures to draw the player in the game. The *Player* component should include service providers of both *IPluginService* and *IControlService*. That is because the *Player* should be renderable but also stoppable as a service since it should be possible to unload the player component during runtime.

The interaction between these three types of components need to occur in a particular way in order to maintain the component-based principle in the application and avoid build-time dependencies. Here player represents the service provider components. The core which is the central point of the application module searches for service providers of the SPIs of the application. On these a method is called on the SPI which is then directed to the particular service provider and its implementation of the method. It will then return the desired output based on the SPI contract. This sequence will run in a loop and happen in each update of the game loop and is depicted in the sequential diagram (Figure 4.4):

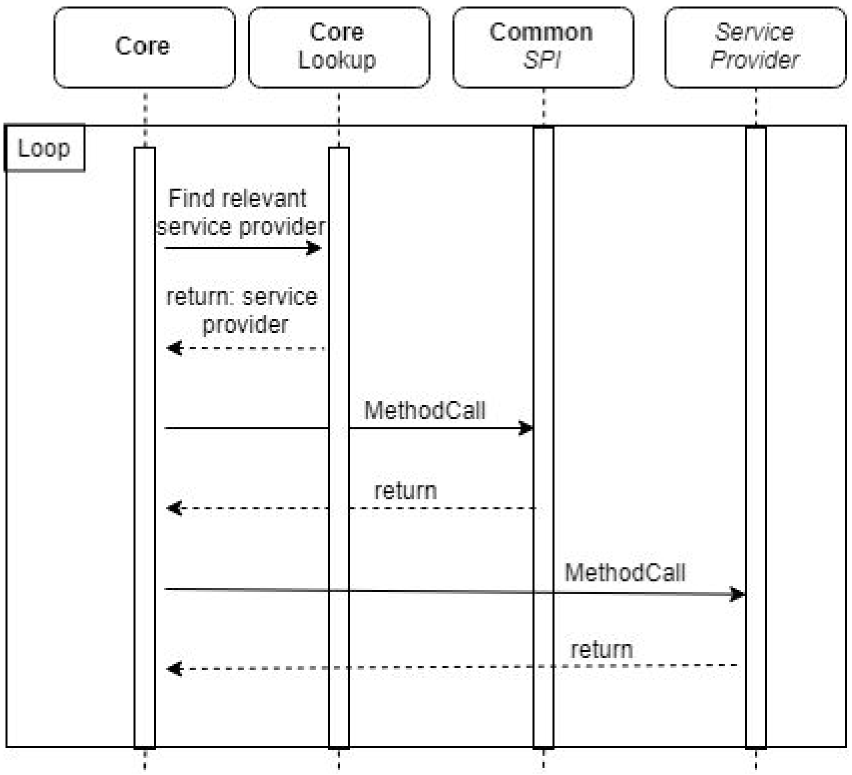


Figure 4.4: Sequence diagram of the communication between components.

4.3 GameEngine

The objective of the project was to obtain a better understanding for how component-based software engineering worked. To do so, the techniques required to create a computer game should be based upon the component strategies that help achieve that objective. Focus should therefore not be on game logic but instead developing component features. This required a game engine to save much of the work on game logic.

Many choices are possible when selecting a game engine but LibGDX was the one that ended up being used. Mostly because LibGDX is a library well known from the lectures but also because of its simple structure that is very intuitive. LibGDX is a library that uses a simple game loop as seen in Figure 4.5.

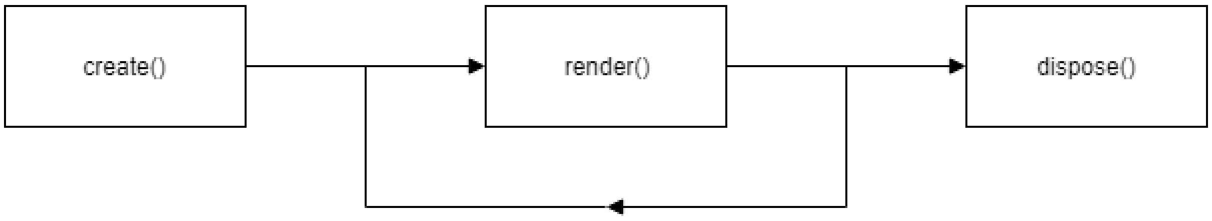


Figure 4.5: Illustration of the system's game loop.[11]

First it creates the resources needed, then renders/updates the game as long as the game is running and finally disposes of everything when the game is paused or shut down. Furthermore LibGDX also makes it possible to port the game from platform to platform and comes with predefined methods to create most of the game logic needed.

4.4 Artificial Intelligence

In the design of the Artificial Intelligence, it was decided that the *Enemy* is controlled by the AI. The *Enemy* component rotates towards the X and Y coordinates generated by the *AI* component. The *AI* component generates a new path every time the player's X and Y coordinates changes and the *Enemy* component is rotating towards those coordinates while moving in that direction. The *AI* component is designed to generate a grid based on the starting coordinate and the goal coordinate. By designing the components with this relationship, it is easily possible to unload/load the *AI* component and thereby enable/disable the AI functionality on the *Enemy* component.

Making the *AI* component responsible for the enemy's movements as emphasized in the analysis implies that that they are two separate components. However, the *Enemy* component must be able to make use of the *AI* component which serves as a service provider. It will do so via the common library located in the *Common* component. *Common* holds both the *AI* interface and the information about all common data needed in the game. By using the common library, the *Enemy* component will be able to get the mentioned game data and make use of needed functionalities provided by the *AI* service provider. To make use of mentioned AI functionalities the enemy will implement the service provided interface *IControlService* just like the *Player* component. This interface is implemented by all moving entities in the game. In its implementation of this, the enemy will state how it will make use of the AI technique.

5 Implementation

This section presents the implementation choices made based on the previous design section. Firstly key components in the game will be presented. Secondly the implementation of the AI technique and finally some important game-mechanics components and their implementation choices will be described.

5.1 Core

The *Core* component is responsible for defining user inputs from peripheral devices, through the *GameInputProcessor.class* in addition to defining the constraints for the game window in the *Installer.class*.

Most important is the *Game.class*. This class ensures the initialisation of some key services such as the Lookup api and the game window in the *create()* method. Furthermore the Game*.class* renders all entities in-game with sprites and updates them accordingly via *render()*-, *update()*- and *draw()* methods. Finally the *Game.class* also makes sure all plugins are correctly loaded or unloaded via the *LookupListener.class*.

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

Game Loop

The game loop in the *Game.class* overrides the *LibGDX ApplicationListener.class* methods. The *create()* method is called when a new game is run. This method makes sure to initialise everything from the camera and sprites to the plugins already created at the start. The *render()* method is called each time a new frame should be rendered. This method is called each game loop iteration and handles all updates of the game and draws the next frame thus making the game feel fluent. For instance the *render()* method handles the computation of the placement and rotation of entities and sprites in addition to calling the *update()* and *draw()* methods. The *update()* method ensures the call to both the *IControlService* and the

IPostProcessor SPI's *execute()* methods which will then make sure each component updates itself.

```
private void update() {
   // Update
   for (IControlService entityProcessorService : getEntityProcessingServices()) {
        entityProcessorService.execute(gameData, world);
   }
   // Post Update
   for (IPostProcessor postEntityProcessorService : getPostEntityProcessingServices()) {
            postEntityProcessorService.execute(gameData, world);
   }
}
```

5.2 Common

The *Common* component holds all the SPIs together with regular interfaces as well as the common data structures that other classes can use and inherit from. This is used to achieve loose coupling between the different components and remove dependencies otherwise needed. The *Common* component is one of the keys to letting components being loaded and unloaded during run-time since no objects are held as a direct reference but instead dependent on their contracts residing in *Common*.

Services

The three SPIs in Common are *IPluginService*, *IControlService* and *IPostProcessor*. The *IPluginService* is used by all components that should be able to be loaded and unloaded. It provides a start and a stop method for Lookup to call so that objects can be instantiated and added during start and removed during stop. In addition, the *IControlService* only provides an execute method, to be called when movement of the implemented object is intended. Since the execute method in our case is called in the game loop, it is called constantly and therefore used to update positions, check keystrokes as well as creating new objects in some instances. The *IPostProcessor* is used to process after *IControlService* has been executed. This is preferred to do in cases like collision where you want the newest positions to be checked for collision rather than the old positions.

Data

Data common for the different components is being kept in the package called *Data* in the *Common* component. This includes e.g. the *World*, *GameData* and *Entity*.
The World holds references to all the different implementations of the different interfaces currently loaded into the game. Whenever a plugin is started it adds the newly created object to the *World* together with its respective interface. This allows the *World* to be the gateway for the different components connection to each other using only the interfaces provided for the different components.

Every component with a sprite and a position is inheriting from the superclass called *Entity*. *Entity* provides the most basic information regarding positioning and sprites file-location.

Having all moveable components inherit from *Entity* allows all components to loop through all the entities contained in *World* regardless of dependencies and without violations.

Interfaces

Three interfaces are provided in the *Common* component. *ICombatEntity*, *IMap* and *IAI*. *ICombatEntity* has the information needed regarding entities that can engage combat. This includes health points, dying and shooting. It has an execute method as well to perform various types of actions depending on its implementation. *IMap* works as the blueprint of a spawning system. It has a list of different spawn locations for different entities and a check if it is time to spawn. The last interface provided in *Common* is the *IAI* which has the callable methods of the *AI* component. This allows a different component to utilize the *AI* without knowing the direct implementation.

5.3 SilentUpdate

The *SilentUpdate* component plays an important role in the system as it handles dynamic loading and unloading. This also means that the component is responsible of fulfilling the main objective of the project. The component contains two classes: *UpdateActivator.class* and *UpdateHandler.class*. First and foremost the *UpdateActivator* extends ModuleInstall which is a NetBeans class from the Module System API. The Module System API (org-openide-modules) is one out of six other modules that makes a runtime container (seen in section 4.1) - and it is this exact module that gives access to the lifecycle of all modules in the application.

The *UpdateActivator.class* has a *ScheduleExecutorService* that will get instantiated when the module is loaded. This service will then execute a task for every 5 seconds (5000 ms). The task is a *Runnable* and checks whether it is time to handle an update by a method *timeToCheck()* which returns a boolean. If the method returns true and it is time to check for an update it then calls for a static method inside the *UpdateHandler.class* called *checkAndHandleUpdates()*.

```
@Override
    public void restored() {
        exector.scheduleAtFixedRate(doCheck, 5000, 5000, TimeUnit.MILLISECONDS);
    }

    private static final Runnable doCheck = new Runnable() {
        @Override
        public void run() {
            if (UpdateHandler.timeToCheck()) {
                UpdateHandler.checkAndHandleUpdates();
            }
        }
    };
```

The very first step in *checkAndHandleUpdates()* checks if any modules have been installed locally by declaring a variable *locallyInstalled* equals the *findLocalInstalled()* method.

In the *findLocalInstalled()* a *Collection* of *UpdateElement* is instantiated and also a list of *UpdateUnits* - where the UpdateUnit is found by the method *getSilentUpdateProvider()*. Then to fetch all *UpdateProviders*, the *getSilentUpdateProvider()* method has a *UpdateUnitProviderFactory* which is a singleton that can be called to get a list of all *UpdateUnitProviders*. Then it searches for the Update Center by iterating through *UpdateUnitProvider* and checks whether the providers name equals the name of the specified Update Center which are specified as a final string at the very top in the *updateHandler.class*. Now that the Update Center is specified, the call returns to *findLocalInstalled()* and finds *UpdateUnits* with *getSilentUpdateProvider().getUpdateUnits().* Next step is to iterate through for each *UpdateUnit* and check whether that *UpdateUnit* has already been installed or else add it to the *locallyInstalled* variable. To ensure that the system is up to date at this point, a method is being called: *refreshSilentUpdateProvider()*. This method first checks if there is an *silentUpdateProvider* and then it calls *silentUpdateProvider.refresh()*.

```
public static void checkAndHandleUpdates() {

        locallyInstalled = findLocalInstalled();

        // refresh silent update center first
        refreshSilentUpdateProvider();

        Collection<UpdateElement> updates = findUpdates();
        Collection<UpdateElement> available = findNewModules();
        Collection<UpdateElement> uninstalls = findUnstalls();

        if (updates.isEmpty() && available.isEmpty() && uninstalls.isEmpty()) {
            // none for install
            LOGGER.info("None for update/install/uninstall");
            return;
        }
```

At this point there are 3 scenarios and 3 methods. If there are modules that needs to be updated *findUpdates()* is called. If there are available modules to be installed *findModules()* is called. And lastly if there are any modules to be uninstalled *findUnstalls()* is called. *findUpdates()* and *findModules()* has a similar approach. They both fetches *UpdateUnits*, iterates through them and checks if there are any updates or available modules and add them to a collection of *UpdateElement*.

- *findUpdates()*: fetches *UpdateUnits* with *getSilentUpdateProvider().getUpdateUnits()*, checks for updates if *getAvailableUpdates()* is not empty, then adds *getAvailableUpdates()*.
- *findModules()*: fetches *UpdateUnits* with *UpdateManger.getUpdateUnits()*, checks for updates if *getAvailableUpdates()* is not empty, then adds *getAvailableUpdates()*.

A continuous check is then made by checking whether there is any modules to update, modules to install or modules to uninstall. Else the case will be 3 scenarios again where

OperationContainer are created for install, update and uninstalls - and calls *handleInstall()* for install and updates, and *handleUninstall()* for uninstall. Both install and update uses the same method for *feeding* the container, *feedContainer()*. In this method, it checks whether it is for updates, then create container for updates by calling *OperationContainer.createForUpdate()*, and if not for updates, then it is for installs, *OperationContainer.createForInstalls()*. It then iterates through *UpdateElement* and adds the necessary ones to the container.

```
        // create a container for install
        OperationContainer<InstallSupport> containerForInstall = feedContainer(available, 
false);
        if (containerForInstall != null) {
            try {
                handleInstall(containerForInstall);
                LOGGER.info("Install new modules done.");
            } catch (UpdateHandlerException ex) {
                LOGGER.log(Level.INFO, ex.getLocalizedMessage(), ex.getCause());
                return;
            }
        }

        // create a container for update
        OperationContainer<InstallSupport> containerForUpdate = feedContainer(updates, 
true);
        if (containerForUpdate != null) {
            try {
                handleInstall(containerForUpdate);
                LOGGER.info("Update done.");
            } catch (UpdateHandlerException ex) {
                LOGGER.log(Level.INFO, ex.getLocalizedMessage(), ex.getCause());
                return;
            }
        }

        // create a container for uninstall
        OperationContainer<OperationSupport> containerForUninstall = 
feedUninstallContainer(uninstalls);
        if (containerForUninstall != null) {
            try {
                handleUninstall(containerForUninstall);
                LOGGER.info("Unstall modules done.");
            } catch (UpdateHandlerException ex) {
                LOGGER.log(Level.INFO, ex.getLocalizedMessage(), ex.getCause());
                return;
            }
        }
    }
```

(See figure 2 in Appendix A3).

5.4 Collision

The collision component is responsible for handling collision between all entities in the world. The component has a single class (*Collision.class*) that implements the SPI *IPostProcessor* - and have two methods:

- *execute()* method that is being called in every game iteration in the game loop.
- *detectCollision()* method with two entities as parameter and returns a boolean.

The collision detection is a simple circle collision method which basically checks if the distance between two center points of our entities are less than the sum of their radius. By having two center points from two entities: $E_1 = (x_1, y_1)$ *and* r_1, $E_2 = (x_2, y_2)$ *and* r_2. The distance of the two entities points and radius is then calculated with: $d_{entity} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ and $d_{radius} = \sqrt{(r_1 - r_2)^2}$.

The collision will then be when the distance between two entities are less than the distance between the two radius. $collision = d_{entity} < d_{radius}$. The implementation can be seen in the code snippet below. Instead of using Math.sqrt(square root) and exponentiation, the function Math.abs is used to do the same resulting in getting the absolute value.

```
    public boolean detectCollsion(Entity e1, Entity e2) {

        if ((Math.abs(e1.getPositionX() - e2.getPositionX()) +
             Math.abs(e1.getPositionY() - e2.getPositionY())) <
             Math.abs(e1.getRadius() + e2.getRadius()))
        {
            return true;
        }

        return false;
    }
```

In the *execute()* method the first step is to iterate through two loops to get both entities in the world with the function call *world.getEntities()*. Hereafter a check is made to see whether it is the same entity or else a collision will be detected between the two entities. Whenever there is a collision between two entities their life should be set using *setLife(entity.getLife() - 1)* and there should also be a check if the entity is dead using *entity.isDead()*. The entity is then removed from the world using *world.removeEntity(entity)*.

5.5 Artificial Intelligence

The implementation of AI is done through a library with the classes *DStarLite*, *State*, *Pair* and *Node*. To use the library in a component-oriented way the group created a component called *AI* making it possible to load/unload the AI functionality. Inside the component the group kept the *DStarLite.class* and created a new class called *DStarLitePlugin.class* which contains the methods for adding and removing the component from the world. The remaining classes were added to the Common folder. After adding the classes to the corresponding folders the group added the functionality to the Enemy component in the *EnemyControlSystem.class*.

The AI component is retrieved from the World and the method *updateStart()* is applied. The method *updateStart()* is a method from the AI library and it is used to tell the D* algorithm where to start generating a path from X and Y coordinates. The method takes two integers as parameters in this case being the Enemy's X and Y coordinates as the D* should start at the Enemy's current position and move towards the Player. Furthermore the method *updateGoal()* is called and it works in the same way as *updateStart()* with the only difference being that it tells the D* where the goal X and Y coordinate is. In our case it is the Player's X and Y position. Code examples of the use can be seen below.

```
ai.updateStart((int) enemy.getPositionX(), (int) enemy.getPositionY());
ai.updateGoal((int) enemy.getPlayerX(), (int) enemy.getPlayerY());
```

After making sure the algorithm is updated with the latest starting position and goal, the method *replan()* is called. This method recalculates the path from the new X, Y coordinates given from the two methods mentioned before. It is important to do so otherwise the components using the AI would walk in a wrong path each time their position got updated as well as when the player moved. Finally the last method *getPath()* is called and it returns a list containing all the X and Y coordinates for the generated path between the components and the Player. The X and Y coordinates in the list are updated constantly and the components therefore always move towards the Player in a path determined by the algorithm. It is also possible to add obstacles that the algorithm will avoid during the planning of the path. The method *updateCell()* makes it possible to type in X, Y coordinates that should be avoided and the components would then move around those if applied.

5.6 Player

The *Player.class* is an *Entity.class* which can be controlled by the user of the game representing a list of commands reflected by the use-case diagram (section 3.1).
The *Player.class* contains the information regarding those actions as well as implementing the *ICombatEntity* interface (section 5.2). In addition, the *Player* component need to consume/implement the *IControlService* in order to allow movement and the *IPluginService* in order for the component to be loaded/unloaded during runtime.

The *PlayerPlugin* consumes *IPluginService* and overrides the *start()* and *stop()* methods.
The *start()* method instantiates the *Player.class* through a method called *createPlayer()* which is in charge of setting the starting position of the player, the speed and rotation etc.
When the *createPlayer()* has been executed, the *start()* method adds the *Player* object to the *World*.
The *stop()* method is called when the module is unloaded and removes the *Player* object from the *World.*

The *PlayerControlSystem* consumes *IControlService* and is in charge of constantly updating what is being pressed by the user. The *IControlService* has an *execute()* method that is called by every game iteration and the *PlayerControlSystem* overrides it so it can fetch the currently pressed keys from *GameData* and thereby update it at the *Player.class*. At the end of the *execute()* method, it will call the execute*()* method of the *ICombatEntity* that

Player.class is which allows the player to update its positions through the *Entity.class* coordinates. Since the information is now available through the *Entity.class*, a dependency will not need to be exposed from the player since the game loop can draw based on the *Entity.class* positions. (See figure 1 in Appendix A3)

5.7 Enemy and Map

Enemies are the main obstacles for the player in the game. The enemy implementation shares many similarities to the player implementation as enemy is also an Entity and makes use of the SPIs *IControlService*, *IPluginService* and *ICombatEntity*. However, it differs as the *Enemy* should move and act on its own and gets the logic from *AI* and *Map* unlike the *Player* which are controlled by an end user.

Similarly to the player, the *Enemy.class* also extends the *Entity.class* and implements the interface *ICombatEntity*. The *Enemy* component also needs to consume the two SPIs *IControlService* and *IPluginService* as previously explained in *Player*.

It is the *EnemyPlugin.class* that implements the *IPluginService* and this class overrides the *start()* and *stop()* methods. Firstly, the *start()* method is empty as the *Enemy* should not be loaded into the world at the same time as the *Player*. *Enemy* implements a different spawning system. Secondly, the *stop() method* ensures that the *Enemy* gets removed when the module gets unloaded. This is done by calling *removeEntity(enemy)* from *World*.

The *EnemyControlSystem.class* implements the *IControlService* and include a bit more implmenentation than *EnemyPlugin.* This class also has a bit more content as it implements some of *Map* and *AI* components' methods. It has two interesting methods: Firstly the *execute()* is being called in every iteration in the game loop and holds implementation about *Map* and *AI.* Secondly there is the *createEnemy()* with the parameters *Game, World, floats x* and *y* and returns an *Enemy*. *createEnemy()* sets fixed details about the enemy such as the *speed*, *rotation* and the *x/y* values for position.

The *execute()* method has two parts - first part handles the spawning of enemies into the world but based on coordinates from the *Map* component. The second part adds movement logics to the *Enemy* through methods from the *AI* component. First part looks whether if there is a *Map* loaded into the world, then references and cast that map as an *IMap* object and gets the reference with the method *getMapArray()* from World. Then a boolean is checked in the method *isSpawning* and create the enemies with *createEnemy()*. The x and y parameters is being retrieved from *Map* with *getEnemyCoordinatesX()* and *getEnemyCoordinatesY()* with a random set on the x coordinates using *Random.nextInt(3)*. The enemy's y coordinates are fixed to being spawned at the top of the window application. The x coordinates on the other hand is set to being 4 different values - meaning that the enemy can spawn from 4 different location at the top of the windows application. Finally after the creation of enemy it can be added to the world through the method *addEntity()* from World. Second part of the *execute()* is described in details in section 5.5.

5.8 Weapon

SDU Royale contains one weapon: a projectile which is an entity that also consumes the *ICombatEntity* interface. Furthermore, it extends the functionality of a weapon with checks of being hit or killed and most importantly whether or not the weapon is shooting. The reason for weapon being an entity is so that it can be rendered through the game loop with sprites and coordinates and move in-game when fired.

The *Weapon* consists of the following three classes: The *Weapon* itself, *WeaponControlSystem* and the *WeaponPlugin*. The *WeaponControlSystem* handles the creation of new weapons whenever the player shoots. This is done through the *ICombatEntity* interface that the player and weapon share. When the player presses space an boolean is set to true from which the *WeaponControlSystem* creates a new weapon sets the boolean to false once again and lastly adds the weapon to the game.

The *WeaponPlugin* handles the start and stop of each weapon via the *IPluginService*. Finally the *Weapon.class* itself starts a timer when a new weapon is created and two seconds after creation this weapon is deleted thus making sure that the game is not flooded with too many weapons. In addition, the class also defines the movement logic for each weapon so that it moves directly forward upon creation.
Both the *WeaponControlSystem* and the *WeaponPlugin* uses the *ServiceProviders* notation to make the component visible to the game.

6 Testing

In this section the overall test process of the game is described. The testing will commence when a component is completed and is realized through unit tests and performance tests of the AI. Furthermore the load and unload requirements are explained in details in addition to the possible bugs found in the system.

6.1 Bugs

Currently a bug regarding the establishment of the project is present. When cloning the source code, the projects root folder must be named exactly “Sem4_CBSE”. The reason for this, is that all paths that the system uses, is build with a String of the absolute path containing “Sem4_CBSE”. The absolute path will then be splitted at “Sem4_CBSE”, solving the problem that the first part of the path is different depending on the end user. However this solution requires the root folders name to be exactly “Sem4_CBSE” as any other name will not be tolerated. This solution is not optimal, as the generated path is hardcoded, since it is specified at the split name - but it was a quick fix to an auto generated path that is compatible for both Windows and Mac users. The more correct solution would be a generated path that is not restricted to a certain name.

Another bug is that the application have to be run at least one time before the auto generated path is updated and used. At the first run, the application will use the last modified path, meaning that your specific path to the updates.xml will first be applied at the second run. Therefore, to dynamically load/unload, the application have to be run twice.

6.2 Load/Unload components

In order to test the functionality of loading and unloading the modules separately, the first step is to create the complete update center, which includes data in form of *.jar* files of every single module in the system. The following steps are already explained in the System Guide. This used approach is applied in testing of the individual modules, and requires no maintenance and provides the results required. The tests results in all components being able to load and unload during runtime. To see an example of this, see figure 1 in Appendix A4. However the testing environment could be expanded and optimized at a later point with integration tests. The benefits of adding integration testing are that it can automate and ease the process of testing the individual modules and their ability to be loaded and unloaded during run-time.

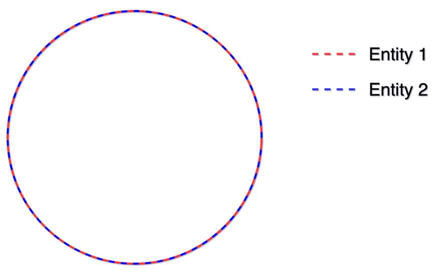
6.3 AI Performance Testing

Performance testing was applied to both A*[12] and D* Lite[13] and the testing was done in order to see if the expected results were on par with the theory gathered from the analysis. The tests were conducted in a console environment outside the game. Both of the algorithms A* and D* Lite were tested on a 1200 x 700 grid with the starting coordinate being (0,0) and the goal being (1200, 700). The runtime gave us an idea of the efficiency of each algorithm and the test results showed that A* had a runtime of 4.909 seconds and D* Lite had a runtime of 16 milliseconds in the same environment. Both of the algorithms were also tested in the game environment and during the test of A* with a 1200 x 700 grid a *OutOfMemoryException* was caught. D* Lite on the other hand performed well as expected. The performance test results matched with our findings during our analysis of the AI algorithms.

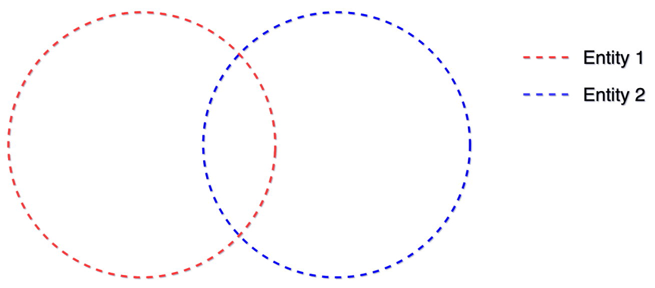
6.4 Unit Testing

Unit testing is used in the *Collision* module to check if the collision works in three different cases. The reason collision is tested with three different fixed test cases is that collision is not subject to change. However it depends on information from entity so therefore an automated test is preferable. That way the developer can rely on collision to work until told otherwise. The collision method that is used is the circle collision method and it is applied in the three following cases (following images are self produced):

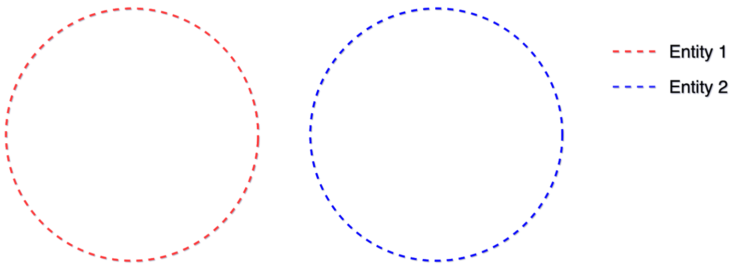
1. Two entities overlap by exact same position and size: Same x, same y, same radius, same rotation. Expected result is true.



2. Two entities overlap but is not located in the same position however the circle colliders of the two entities overlaps. Expected result is true.



3. Two entities do not overlap and is located far from each other with no circle collider overlap. Expected result is false.



The test is implemented using the jUnit framework. The method that is tested is called *detectCollision()* and can be seen below:

```
public class Collision {
...
public boolean detectCollsion(Entity e1, Entity e2) {
        if ((Math.abs(e1.getPositionX() - e2.getPositionX()) *
             Math.abs((e1.getPositionX() - e2.getPositionX())) +
             Math.abs(e1.getPositionY() - e2.getPositionY())) *
             Math.abs(e1.getPositionY() - e2.getPositionY()) <
             (e1.getRadius() + e2.getRadius()) *
             (e1.getRadius() + e2.getRadius()))
        {
            return true;
        }

        return false;

    }
...
}
```

The three test cases are implemented as individual tests on the *Collider.class*. In each test method an instance of *Collision* is created with new entities for each test case that fits the state of the individual test cases. All the tests are concerned with the *detectCollision()* method:

```
public class CollisionTest {
...
@Test
public void testDetectCollisionCase1() {
    Collision instance = new Collision();

    // entity 1 test case state
    float x1 = 400;
    float y1 = 300;
    float radians1 = 0;
    float radius1 = 10;

    Entity entity1 = new Entity();

    entity1.setRadius(radius1);
    entity1.setRadians(radians1);
    entity1.setPositionX(x1);
    entity1.setPositionY(y1);

    assertTrue(instance.detectCollsion(entity1, entity1));
}
```

```java
@Test
public void testDetectCollisionCase2() {
    Collision instance = new Collision();

    // entity 1 test case state
    float x1 = 400;
    float y1 = 400;
    float radians1 = 0;
    float radius1 = 10;

    // entity 2 test case state
    float x2 = 405;
    float y2 = 390;
    float radians2 = 0;
    float radius2 = 12;

    Entity entity1 = new Entity();
    Entity entity2 = new Entity();

    entity1.setRadius(radius1);
    entity1.setRadians(radians1);
    entity1.setPositionX(x1);
    entity1.setPositionY(y1);

    entity2.setRadius(radius2);
    entity2.setRadians(radians2);
    entity2.setPositionX(x2);
    entity2.setPositionY(y2);

    assertTrue(instance.detectCollsion(entity1, entity2));
}

@Test
public void testDetectCollisionCase3() {
    Collision instance = new Collision();

    // entity 1 test case state
    float x1 = 400;
    float y1 = 300;
    float radians1 = 0;
    float radius1 = 10;

    // entity 2 test case state
    float x2 = 6000;
    float y2 = 500;
    float radians2 = 0;
    float radius2 = 12;

    Entity entity1 = new Entity();
    Entity entity2 = new Entity();

    entity1.setRadius(radius1);
    entity1.setRadians(radians1);
    entity1.setPositionX(x1);
    entity1.setPositionY(y1);

    entity2.setRadius(radius2);
    entity2.setRadians(radians2);
    entity2.setPositionX(x2);
    entity2.setPositionY(y2);

    assertFalse(instance.detectCollsion(entity1, entity2));
}
...
}
```

7 Discussion and evaluation

This section contains a discussion about the design choices of the application along with the alternative solutions that have been discussed throughout the development. This will concern the choice of module system and design choices in regards to minimize the level of abstraction in the development versus minimizing the fragile base class problem. Lastly it will concern the design choices of the AI and which algorithms that should be used.

7.1 NetBeans Modules vs OSGi

The benefits of using the NetBeans Module System is that you have a framework that supports the Whiteboard Model directly. This is achieved with “JDK-ServiceLocator” or its successor “LookUp”. The major difference between the two is that LookUp creates the meta-information where the information has to be updated and handled manually with the ServiceLocator.

OSGi supports both *Dependency Injection (DI)* with *Inversion of Control (IoC)* and the *Whiteboard Model*. The DI is prefered if you don’t want your code to depend on a specific framework however no such constraint was put on this project.
NetBeans Module System supports OSGi components but a component that translates the OSGi bundle has to be used. The current solution would support OSGi components if such a module was created. However, it would mostly serve as a demonstration since not much would be gained by using OSGi bundles. It would allow components to use DI within the translation module instead of pure Whiteboard. But in the end, the project would still use the Whiteboard Model. The mix between the two module systems would solve no given problem within the project.
Therefore, the optimal solution for the project is the NetBeans Module System using LookUp since it supports the project the most.

7.2 Fragile Base Class

One of the main advantages of using a component-based system is to avoid having a fragile base class. This was not completely achieved since all entities of the system inherits from the thick base class *Entity*. This base class holds functionality that will not change throughout development or the future since it only contains the most basic functionality common for all the entities. A way to avoid this is to have a completely empty *Entity.class* so that nothing can change and thereby avoid having a fragile base class. It would however result in increased complexity since other data-classes would be needed to hold the previous functionality stored in Entity. The reason why they would be necessary is because the components should not hold any references to each other so common data-classes would be the only way for them to communicate in a general way. Entity functions as the product of those data-classes and therefore decreases the overall complexity of the system. The result of this is that inheritance is used which may cause ripple effect of changes if the base class is modified. However, no deep inheritance hierarchies will be formed using this format.

7.3 AI

In the *AI* component a path is generated. An advantage of that is that it can be used to constantly generate the most optimized path from A to B. However it can be discussed whether or not the method is required to be called for each time the goal in the path changes. In our project the method that regenerates the path is called whenever needed without any condition checking if the re planning is necessary or not. For example if the player moves from the initial goal to another goal then the path needs to be updated but the X, Y from where the component using the AI is located to the next X, Y coordinate in the current path might not need to be altered. In this case the method call could be reduced to only be called when the component reached a point where a new coordinate was needed. It can also be argued that the resources that are spent generating a new path might not always be worth it if the optimization is minimal.

Another thing worth noticing is that the component using the AI moves towards the *Player* in the generated path but the component is only making use of one X, Y coordinate from the path. The reason for this is that the component rotates and moves towards a point and if it were to rotate towards all of the coordinates it would never make it to the *Player* because the list of coordinates are updated so often. By only choosing one coordinate from the list it ensures that the component is not hindered by the often path regeneration.

In the requirement and analysis (section 2 and 3) the expectations for an implemented AI technique were described. The technique needed to be complete and optimal. The D* Lite implementation used in the project delivers both. The *Enemy* component uses the provided functionality from the *AI* component which makes it possible to move the enemy towards the players position making it complete. It is optimal due to the cost-efficient path calculation which ensures the enemy in taking the shortest path possible.

The goal was also to have a well-performing AI technique with high speed and low computation costs. The performance test (section 6.2) confirmed the research mentioned in section 3.4 with D* Lite being quicker than A*. The results showed that D* Lite was significantly faster which is why it was the best AI technique to implement for this specific game.

8 Conclusion

In this semester project about “Component-based Software Engineering” we wanted to create a 2D component-based game in order to enhance our learning of the component-based software development. We achieved this by prioritizing the creation of components and fully utilize the component framework in addition to incorporating LibGDX API to minimize time spent on creating game-logic.

We achieved this by focussing on the NetBeans Module System that enables us to separate different kind of game-logic and functionality into their respective components and through the whiteboard model fully integrate the components into one system.

A feature that we wanted to implement was to utilize artificial intelligence techniques in the game. We achieved that by applying AI through pathfinding with an algorithm that makes the gameplay feel more fluent as components such as the enemy component should act on its own calculations to determine its actions in-game. The decision of choosing D* Lite helped performance as A* was too inefficient for our game.

Another core feature was to realize the possibility of loading/unloading components during runtime. This was achieved through the SilentUpdate component in conjunction with the LookUp API and the Auto Update Services.

Finally we can conclude that the project is satisfiedly completed as a functioning 2D component-based game (SDU Royale) that fulfills the requirements and our expectations.

9 References

[1] J. Corfixen(2017). *Component and DATA-Oriented Game Design*. (Feb. 9, 2017). Accessed: May 27, 2019. [Online Video] Available: https://drive.google.com/file/d/0B6Mo6Uok0on9VnV1LXI0RXZuM2M/edit.

[2] T. Boudreau, *Not Just an IDE: Working with the NetBeans Platform and the NetBeans Module System*, Netbeans.org, 2006. [Online]. Available: https://netbeans.org/download/magazine/02/nb02-part3-platform.pdf. [Accessed: 27 May 2019].

[3] S. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach*, 3rd ed. Pearson, 2010, Chapter 2

[4] S. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach*, 3rd ed. Pearson, 2010, Chapter 3.

[5] "D*", *En.wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/D*. [Accessed: 27 May 2019].

[6] Howie Choset, Carnegie Mellon University (2007). *Robotic Motion Planning: A* and D* Search.* [PowerPoint slides] Available: https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf. [Accessed: 27 May 2019]

[7] MIT OpenCourseWare, YouTube (2016). *Advanced 1. Incremental Path Planning*. [online] Available at: https://www.youtube.com/watch?v=_4u9W1xOuts. [Accessed: 27 May 2019].

[8] En.wikipedia.org. (2019). *A* search algorithm*. [online] Available at: https://en.wikipedia.org/wiki/A*_search_algorithm. [Accessed: 27 May 2019].

[9] J. Corfixen, *NetBeans*. (Mar. 3, 2017). Accessed: May 27, 2019. [Online Video] Available: https://drive.google.com/file/d/0B6Mo6Uok0on9RHhFbkFHVnRzOXc/view.
[10] J. Corfixen, Java and Components. (Feb. 20, 2017). Accessed: May 27, 2019. [Online Video] Available: https://drive.google.com/file/d/0B6Mo6Uok0on9UXRGbkd0c2Z1Sms/view.

[11] J. Corfixen, *CBSE 2D Games*. (Feb. 8, 2017). Accessed: May 27, 2019. [Online Video] Available: https://drive.google.com/file/d/0B6Mo6Uok0on9SFB1ZjJ2cWVkSkk/edit.

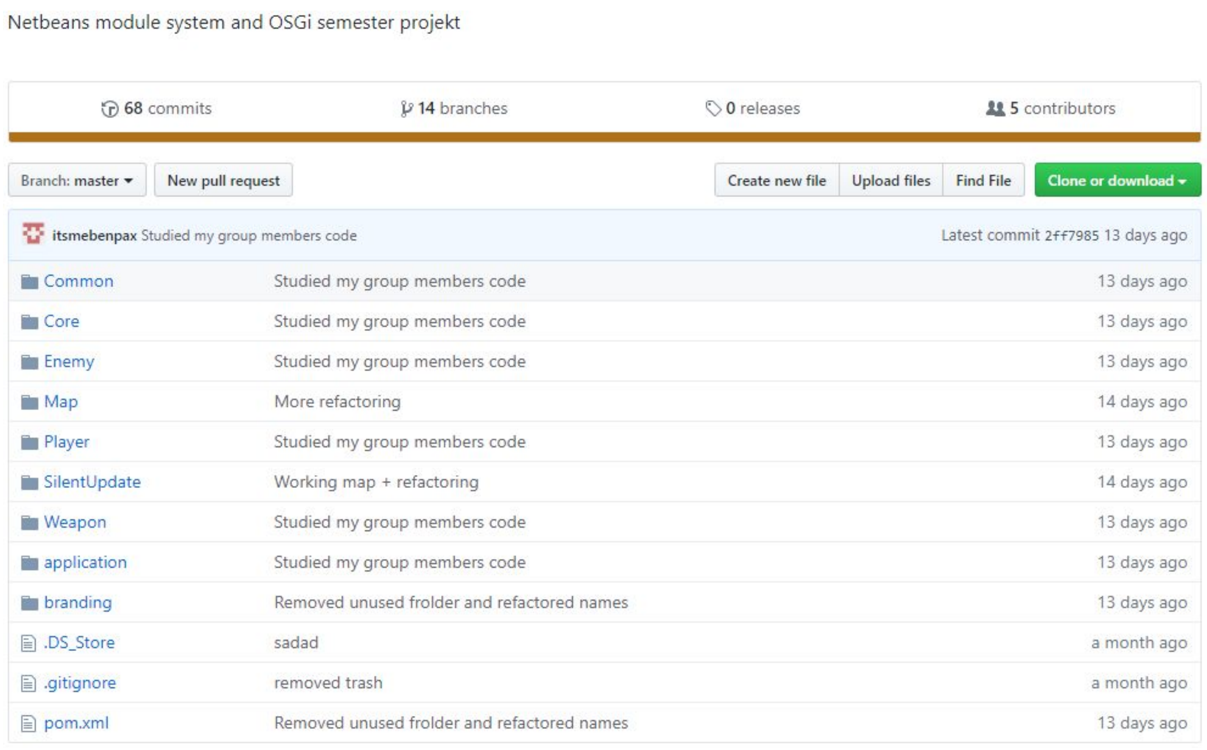
[12] GitHub. (2019). *marcelo-s/A-Star-Java-Implementation*. [online] Available at: https://github.com/marcelo-s/A-Star-Java-Implementation. [Accessed: 27 May 2019].

[13] GitHub. (2019). *daniel-beard/DStarLiteJava*. [online] Available at: https://github.com/daniel-beard/DStarLiteJava. [Accessed: 27 May 2019

Appendix

A1 Source code

Project source code can be found in the groups' GitHub.
https://github.com/SigurdEEspersen/Sem4_CBSE


Netbeans module system and OSGi semester projekt

68 commits | 14 branches | 0 releases | 5 contributors

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

itsmebenpax Studied my group members code — Latest commit 2ff7985 13 days ago

Common	Studied my group members code	13 days ago
Core	Studied my group members code	13 days ago
Enemy	Studied my group members code	13 days ago
Map	More refactoring	14 days ago
Player	Studied my group members code	13 days ago
SilentUpdate	Working map + refactoring	14 days ago
Weapon	Studied my group members code	13 days ago
application	Studied my group members code	13 days ago
branding	Removed unused frolder and refactored names	13 days ago
.DS_Store	sadad	a month ago
.gitignore	removed trash	a month ago
pom.xml	Removed unused frolder and refactored names	13 days ago

Figure A1.1: Screenshot from the project teams GitHub.

A2 Project Description

The project description is from the initial startup phase in this semester project. It was conducted through discussion and brainstorming, and forms the foundation of the project.

Project description

In this semester project, our idea for a 2D game is a top-down shooting game inspired by a game called “The Binding of Isaac”.[1]
The main theme of the game is a battle between Faculties in the University of Southern Denmark. The ideas of the game is written below, where the **sections** or text marked with **bold** could be possible components for later implementation.

Gameplay/Core

You are playing as an engineer-student (student from the Faculty of Engineering) on SDU, where the objective of the game is to ‘slay’ as many Humaniora-students (students from the Faculty of Humanities), obtain a higher score, until you reach the end boss (still not figured yet) and win the game. The enemies are the Humaniora-students’, as they try to dominate SDU! Your mission is to stop this madness!
The main player will have a finite amount of HP(Health points), where enemies will reduce your HP by attacking you, and the game will be lost when you reach 0 HP.
You obtain points by slaying enemies. Additional features could be missions or quests where the rewards could be either points or bonus items. An example could be saving hostages, fellow engineer-students, who are captured by Humaniora-students.

Enemies

The main enemies are the Humaniora-students. It is planned to incorporate Artificial Intelligence into this component. AI techniques or algorithms should determine the behavior of the enemy for example the way the enemies are chasing the main player etc.

WorldMap/Levels

The game world is based on chosen locations in SDU on Campus, for example the Faculty of Engineering-building, main entrance, main hallway etc.The starting point is the Faculty of Engineering-building and the end point is the Faculty of Humanities-building where the end boss is located. As you progress in the game, you will get to a higher level where the starting point can be seen as the first level and the end point as the last level. As you complete one level or one room, you have the opportunity of entering the next until you reach the last one.

[1] https://store.steampowered.com/app/113200/The_Binding_of_Isaac/

Weapons

We have a variety of ideas for weapons. It is also planned to incorporate Artificial Intelligence into this component. Standard weapons or upgrades are yet to be settled - but some of the discussed weapons are:

- As an engineer-student you can throw/shoot with computers against enemies
- Another option is using a LAN cable as a form for whip as weapon against enemies
- A bonus weapon could be some drone as a non-playing character, whose role is an ally who follows you and shoots enemies for a limited time
- As enemies or Humaniora-student, they will throw/shoot books against you, that will diminish your health points (HP).

Highscore/points

The highscore is point-based. You obtain points by slaying enemies, or doing missions/quests etc. You can choose to spend your points in the item-shop - meaning that you will get a lower highscore by purchasing items, thus increasing difficulty by having to choose between a high highscore or easy gameplay.

Items/Item-shop

The item-shop which is located as the cafeteria will give the player the opportunity of purchasing items that have different functions and different prices. An example of items could be CPUs, GPUs, RAMs, Keyboards, Monitors etc. which could function as some kind of power-up.
The player is able to purchase items with the points from his/hers highscore.
Additional items could appear randomly or in specific rooms (or maybe bought in the item-shop) and could function as power-ups. Ideas for such item could be: coffee/energy drink (movement speed, limited time), pizza (+max HP), LOGO-bolsjer (HP-regen, limited time), ØL (100% immortal, - 50% accuracy, limited time, permanent -max HP)

A3 Sequence Diagrams

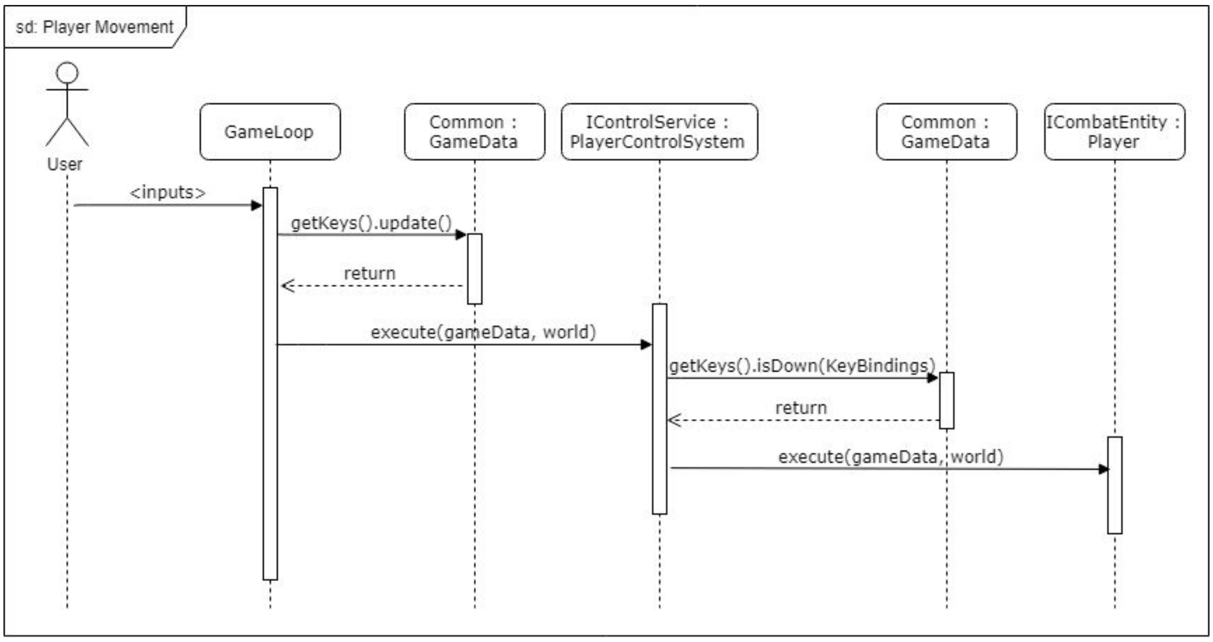


Figure A3.1: Sequence diagram that displays flow of player movement.

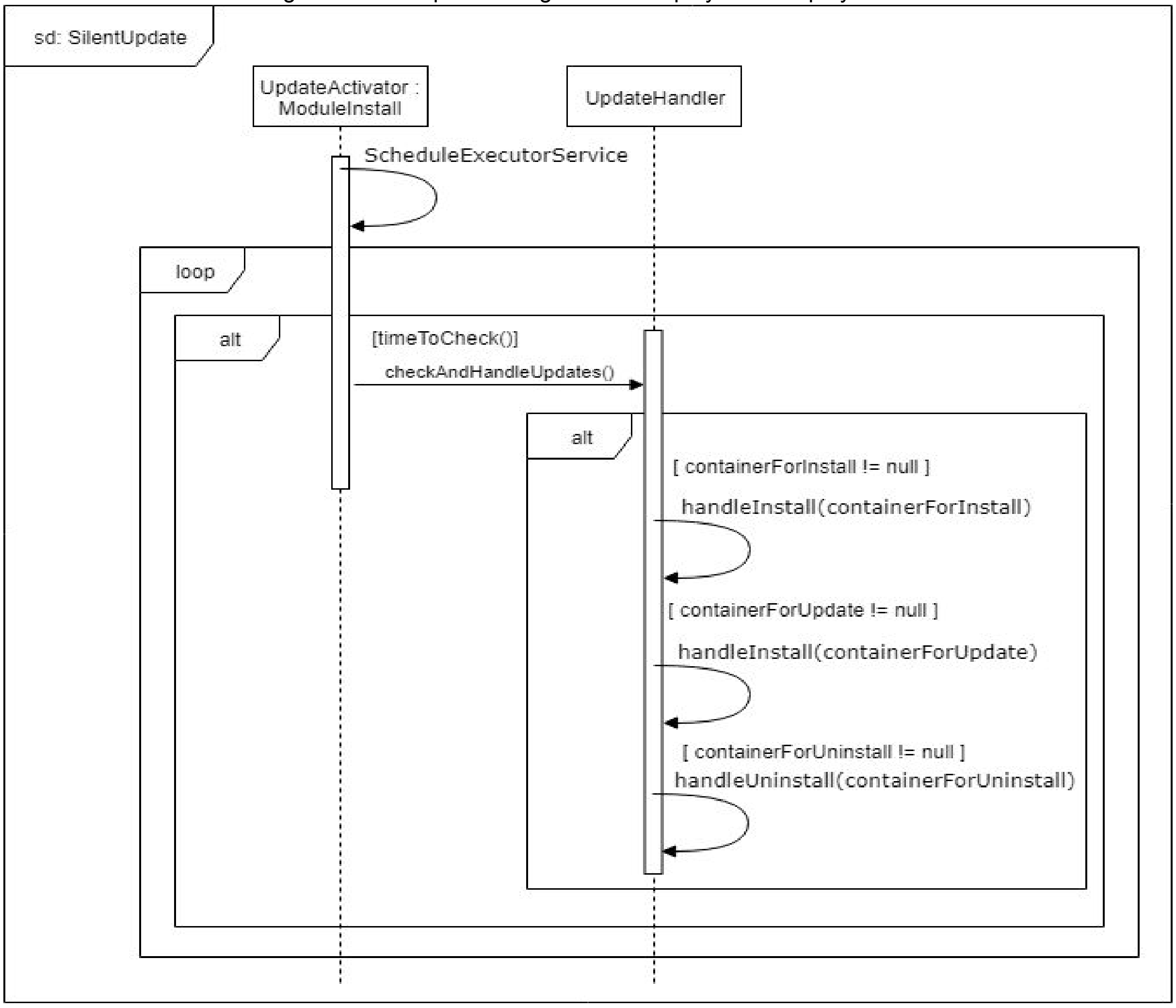
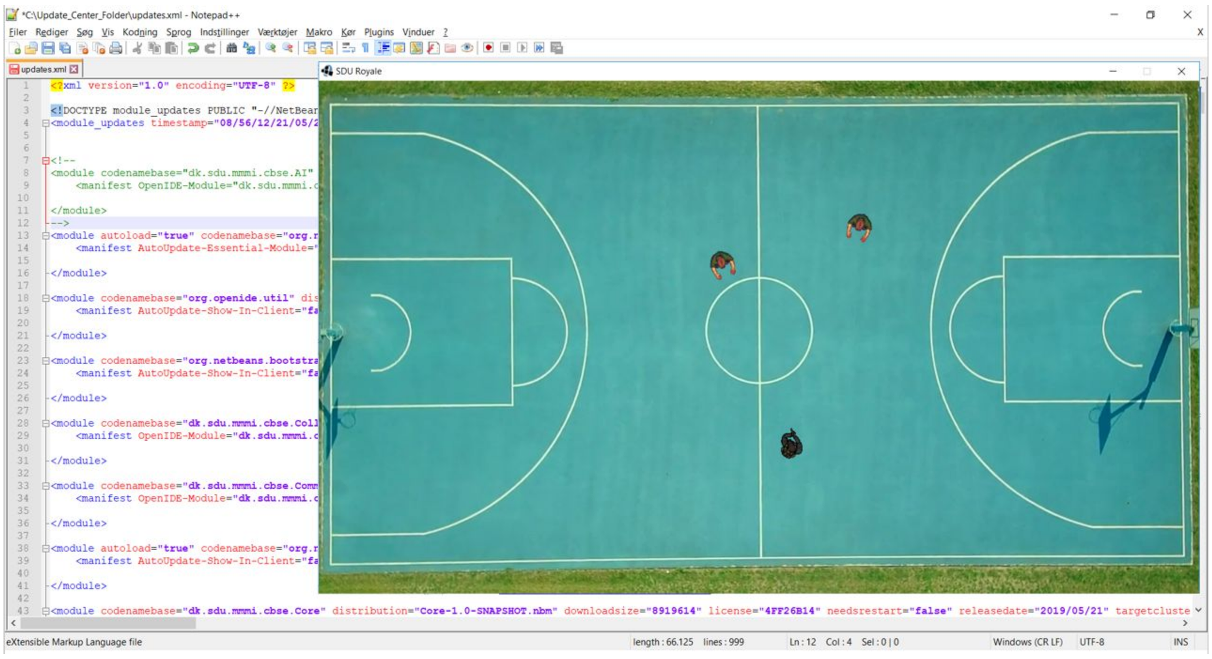

sd: SilentUpdate
UpdateActivator : ModuleInstall
UpdateHandler
ScheduleExecutorService
loop
alt
[timeToCheck()]
checkAndHandleUpdates()
alt
[containerForInstall != null]
handleInstall(containerForInstall)
[containerForUpdate != null]
handleInstall(containerForUpdate)
[containerForUninstall != null]
handleUninstall(containerForUninstall)

Figure A3.2: Sequence diagram that displays flow in SilentUpdate.

A4 SDU Royale


*C:\Update_Center_Folder\updates.xml - Notepad++
Filer Rediger Søg Vis Kodning Sprog Indstillinger Værktøjer Makro Kør Plugins Vinduer ?
updates.xml
SDU Royale
eXtensible Markup Language file
length : 66.125 lines : 999
Ln : 12 Col : 4 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE module_updates PUBLIC "-//NetBear
<module_updates timestamp="08/56/12/21/05/2


<!--
<module codenamebase="dk.sdu.mmmi.cbse.AI"
    <manifest OpenIDE-Module="dk.sdu.mmmi.c

</module>
-->
<module autoload="true" codenamebase="org.r
    <manifest AutoUpdate-Essential-Module="

</module>

<module codenamebase="org.openide.util" dis
    <manifest AutoUpdate-Show-In-Client="fa

</module>

<module codenamebase="org.netbeans.bootstra
    <manifest AutoUpdate-Show-In-Client="fa

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Coll
    <manifest OpenIDE-Module="dk.sdu.mmmi.c

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Comm
    <manifest OpenIDE-Module="dk.sdu.mmmi.c

</module>

<module autoload="true" codenamebase="org.r
    <manifest AutoUpdate-Show-In-Client="fa

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Core" distribution="Core-1.0-SNAPSHOT.nbm" downloadsize="8919614" license="4FF26B14" needsrestart="false" releasedate="2019/05/21" targetcluste
```

Figure A4.1: Screenshot of running application.

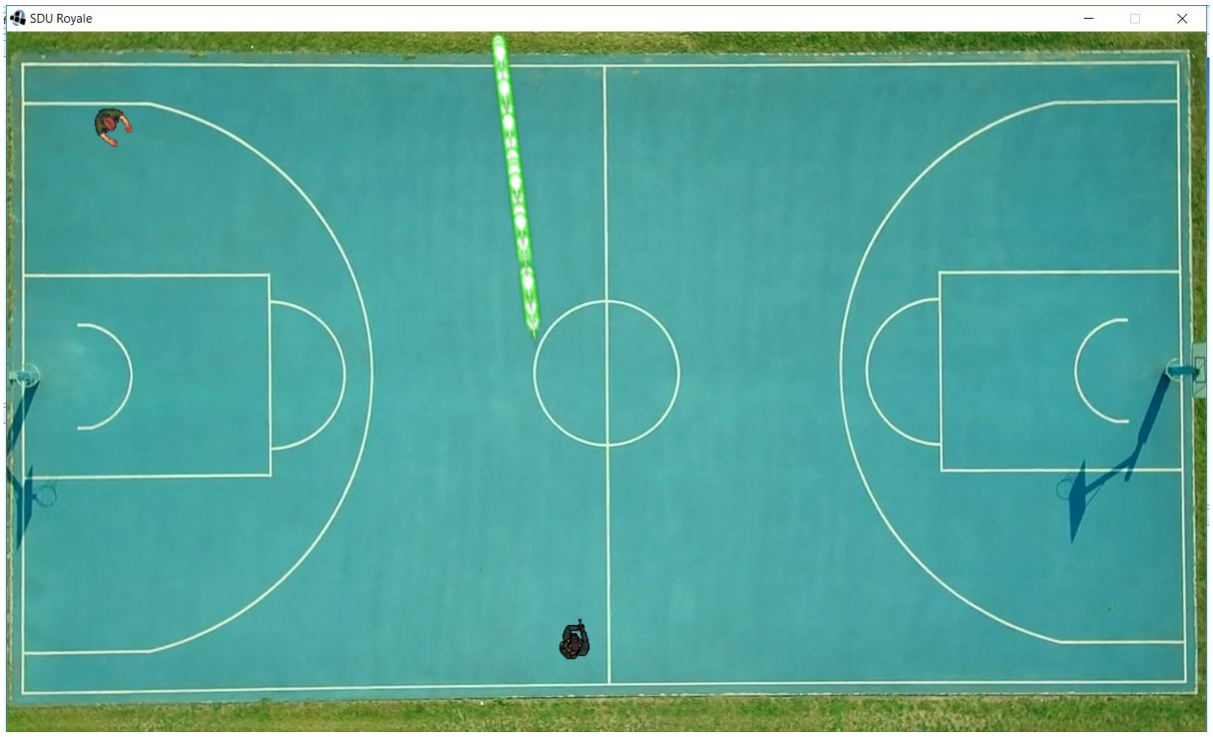


Figure A4.2: Screenshot with all components present.