Component Based Software Engineering

**AsteroidsFX**

**Course:**

- T510035101

**Examiner:**

- Jan Corfixen Sørensen

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Student:**

- Marcus Brink Thomsen

**Exam Nr:**

- 4118643

**Github username:**

- VinciDa33

**Project repository:**

- <https://github.com/VinciDa33/AsteroidsFX_MBT>



# Abstract

Describe the problem that the report addresses in context of the game domain.

Outline how the developed game addresses the requirement – its key characteristics and fundamental principles (establishing a solution).

This report describes the creation of a project which aims to comply with the open-closed principle known from SOLID. The goal of the project by doing so, is to avoid the often undesirable traits of developing a monolithic system.

Monolithic systems are often hard to change and maintain, due to how closely connected all components of the codebase are. Furthermore, expansion of a monolithic system might become bloated, with no way for users to deliberately select specific features that they need. It becomes an all or nothing package.

By using the Java Platforms Module System this project will have its codebase separated into modules with each their own responsibility. This approach makes the project easier to maintain and change as each module can be managed independently from each other. Extension of the system will become possible even without recompilation, thus it will also successfully follow the open-closed principle.

Contents

[Abstract 1](#_Toc165890100)

[1 Introduction 3](#_Toc165890101)

[2 Requirements 4](#_Toc165890102)

[Functional Requirements 4](#_Toc165890103)

[Non-functional Requirements 5](#_Toc165890104)

[3 Analysis 6](#_Toc165890105)

[Technologies 6](#_Toc165890106)

[JavaFX 6](#_Toc165890107)

[JPMS 6](#_Toc165890108)

[Apache Maven 6](#_Toc165890109)

[System Use Cases 7](#_Toc165890110)

[Module Communication 9](#_Toc165890111)

[Object Model 10](#_Toc165890112)

[4 Design 11](#_Toc165890113)

[5 Implementation 12](#_Toc165890114)

[6 Test 13](#_Toc165890115)

[Unit Testing 13](#_Toc165890116)

[Integration Testing 13](#_Toc165890117)

[Portability Testing 14](#_Toc165890118)

[7 Discussion 15](#_Toc165890119)

[8 Conclusion 16](#_Toc165890120)

[References 17](#_Toc165890121)

# 1 Introduction

The introduction must describe the game.

This project intends to mimic the classic asteroid game by Atari from 1979, using the Java language. The game consists of a simple playing field with a player character, enemy spaceships and asteroids. Both the player and enemies have the ability to shoot, splitting asteroids in two or destroying each other.

The game loop is potentially infinite, with the goal of getting the highest score possible. The difficulty of the game goes up over time, with more enemies and asteroids spawning. Each time an asteroid is split or destroyed, as well as when an enemy perishes, points are awarded to the player’s overall score.

The project uses JavaFX as its graphics framework, and game elements are rendered using simple polygon shapes with flat colours.

The game is created using Java with Apache Maven as build system, with a focus on the open-closed principle. Using the Java Platform Module System, game elements are split out into separate modules. This modular approach allows for extension modules to be integrated at a later point, without the need for recompilation of the core system.

# 2 Requirements

Describe the component-based game in terms **of interface contracts**, functional and non-functional requirements.   
The game must include **Player**, **Enemy**, Asteroids, **Weapon** and Map components.

The Player, Enemy and Weapon components must implement service provided interfaces that allow the components to be updated and removed without recompilation.

## Functional Requirements

The functional requirements describe the overarching product features of the project. These requirements must be implemented for the product to be completed. Most required modules have sub-requirements, detailing what features should be present within that module.

The table below was created based on the requirements for this course, as well as the assignments in the lab exercises.

|  |  |  |
| --- | --- | --- |
| **Functional requirements** | | |
| **ID** | **Title** | **Description** |
| F1 | Service interfaces | The project must contain service interfaces for module interactions |
| F2 | JPMS | The project must use the Java Platform Module System |
| F3 | JavaFX application | The project must use JavaFX for graphics and user interfacing |
| F4 | Player module | The project must contain a player module |
| F4-a | - Player movement | The player shall be able to move around the screen |
| F4-b | - Player shooting | The player shall be able to shoot bullets |
| F5 | Enemy module | The project must contain an enemy module |
| F5-a | - Enemy movement | The enemy/enemies shall be autonomously moving about the screen |
| F5-b | - Enemy shooting | The enemy/enemies shall be able to shoot bullets |
| F6 | Asteroid module | The project must contain an asteroid module |
| F6-a | - Asteroid movement | Asteroids shall move about the screen in random directions |
| F6-b | - Asteroid splitting | Asteroids shall split into smaller asteroids when hit by bullets |
| F7 | Bullet system | The project must contain a system for spawning and handling bullets |
| F8 | Pythagorean collision detection | The project must have a collision system using the Pythagorean theorem to check for circular collisions |
| F9 | Scoring system microservice | The project must have a microservice application to handle the point score |

## Non-functional Requirements

The non-functional requirements describe features or quality attributes that are used to evaluate the performance and function of a system rather than its specific behaviour, which is described by the functional requirements.

The table of non-functional requirements below was based on the requirements for this course, as well as the assignments in the lab exercises.

|  |  |  |
| --- | --- | --- |
| **Non-functional requirements** | | |
| **ID** | **Title** | **Description** |
| FN1 | Unit testing | The game should have at least 1 unit test |
| FN2 | Modularity | It must be possible to remove non-common modules without recompilation |
| FN3 | Portability | The game should run similarly on hardware with varying specs  (Game elements should move at similar speeds on different hardware) |

# 3 Analysis

Analysis describes only **what** the system should do and not **how** it is done.

In analysis, you can come up with a rough draft of the interfaces and the entities of the game.

Furthermore, you should document use cases/gameplay, the object model using a UML class diagram and the communication between components with sequence diagrams.

## Technologies

The choice of technologies and their capabilities will be an important part of how the system is later implemented and will serve as a foundation for the project.

### JavaFX

The game will be built using the JavaFX client application framework. The JavaFX framework enables the creation of an application window and for graphics to be displayed within it. Furthermore, JavaFX can be used to obtain both mouse and keyboard inputs from the user.

Though more modern graphical frameworks exist and may be better suited for larger graphical Java applications, JavaFX has all the features needed for this project. Additionally having prior experience using the framework enables a swift start to development.

### JPMS

To adhere to the requirements for modularity, the project will be built using the Java Platform Module System. JPMS allows for strong encapsulation between modules, requiring that a module explicitly declares what packages it exports to other modules as well as a declaration of which other modules it itself is dependent on.

These declarations allow for fine control of which parts of the system can be used to create extensions in the future.

### Apache Maven

Maven is used as a build system for the project. Using Maven makes it easier to handle dependencies and with its capabilities for plugins it can be customized and extended with further functionality. One such plugin is ‘exec-maven-plugin’ from MojoHaus. This enables easy configuration of how the program is executed.

## System Use Cases

To explore the desired behavior of the system, use case specifications were made for some select cases. This allows for thought to be put into how the system should behave, without touching on how this behavior will be achieved.

This use case describes the main game loop, and some of the behavior that will run within it.

|  |  |
| --- | --- |
| Use Case Name | Game loop |
| Actors | Core, all present modules |
| Description | The core runs all its main loop functionality |
| Pre-condition | The core start method has been run |
| Post-condition | All main loop functionality has resolved |
| Path | * All processor classes are obtained using Java Service Loader. * All processor classes are run. * All post-processor classes are obtained using Java Service Loader. * All post-processor classes are run. |

This use case describes the player action of shooting a bullet.

|  |  |
| --- | --- |
| Use Case Name | Player shooting |
| Actors | Player, Bullet |
| Description | The player presses the shoot button and expects a bullet to fly forward from the player character |
| Pre-condition | The player module is present, and the bullet module is present |
| Post-condition | A new bullet object has been instantiated correctly |
| Path | * The player checks if enough time has elapsed since last firing. * The player obtains a new bullet object through an SPI. * The player adds the bullet into the world. * The player resets the cooldown timer. |
| Alternate path | * Not enough time has elapsed since last firing. * Nothing happens. |

This use case describes the splitting of asteroids upon collision.

|  |  |
| --- | --- |
| Use Case Name | Asteroid splitting |
| Actors | Asteroid, Other entity |
| Description | An asteroid collides with another entity and splits into two smaller asteroids |
| Pre-condition | The asteroid module is present, and the collision module is present |
| Post-condition | A collision event has happened, and all collision effects have been resolved |
| Path | * The collision system detects a collision and triggers collision events on the relevant entities. * The asteroid checks that the other entity is a bullet. * The bullet deletes itself. * The asteroid destroys itself, while at the same time creating 2 new smaller asteroids in its place. |
| Alternate path | * The collision system detects a collision and triggers collision events on the relevant entities. * The asteroid checks that the other entity is **not** a bullet. * The asteroid responds appropriately to the collision but does not split. |

This use case describes how enemies are spawned.

|  |  |
| --- | --- |
| Use Case Name | Enemy spawning |
| Actors | Enemy Processor |
| Description | A new enemy is created within the game world |
| Pre-condition | The enemy module is present |
| Post-condition | A new enemy object has been instantiated correctly |
| Path | * The enemy processor checks that the current number of enemies is less than desired. * The enemy processor spawns a new enemy of screen. |
| Alternate path | * The desired number of enemies are already present. * Nothing happens. |

## Module Communication

The communication between modules will be kept to a minimum, to retain as few dependencies as possible. However, some communication between modules will be needed. To enable this communication service provider interfaces (SPI’s) will be used. By defining certain SPIs within the core part of the system, every module will be allowed to use those SPI’s and retrieve objects from other modules using the Java Service Loader.

This also means that whatever behavior is defined within these service interfaces, will be the only behavior made available to other modules, enabling fine control over the inter-module communication.

Below is a rough sequence diagram displaying an example of how a module would use the Service Loader to get access to another module. Note that for this specific example, the SPI class would have to define the ‘createBullet()’ method.

A diagram of a program

Description automatically generated

## Object Model

Below can be seen a rough draft of the projects object model. The model describes three basic service interfaces that will allow for the core functionality of the system. Each of these will be used by the core class/module. As discussed in [Module Communication](#_Module_Communication) these will allow for inter-module communication. In this case it is intended to be used by the core module only to run the basic functionality of every other extending module.

The classes implementing these interfaces will be working with the Entity class, with the idea being that the Entity class will contain common data, such as position. In other words, they contain only their own state. The classes implementing the entity processing services will then be used as behavior classes which modify the state of the entities.

The core module will be able to obtain and run all these entity processors using the Java Service Loader.

Each module will also be able to extend the Entity class and add to its state data. However, this extra data will only be available within its own module.

A screenshot of a computer

Description automatically generated

# 4 Design

The design describes **what** the structure of the system should be to fulfill the requirements.

Document the architecture and abstractions of the system.

Design develops those abstractions into realizable components.

Describe and sketch the **component models** of the game using a UML component diagram, see [[UML]](https://mcas-proxyweb.mcas.ms/certificate-checker?login=false&originalUrl=https%3A%2F%2Fyoutu.be.mcas.ms%2FKQUGFFN4M90%3FMcasTsid%3D15600&McasCSRF=e3d0355ef58220a2ce832c3102938d848870c939fdd45a3fc240b2c2cf9cca75) .

The component contracts in the system must be described in terms of pre- and post-conditions.

Furthermore, the different elements of the game and how they are connected must be described.

## Data Structures

### Entity

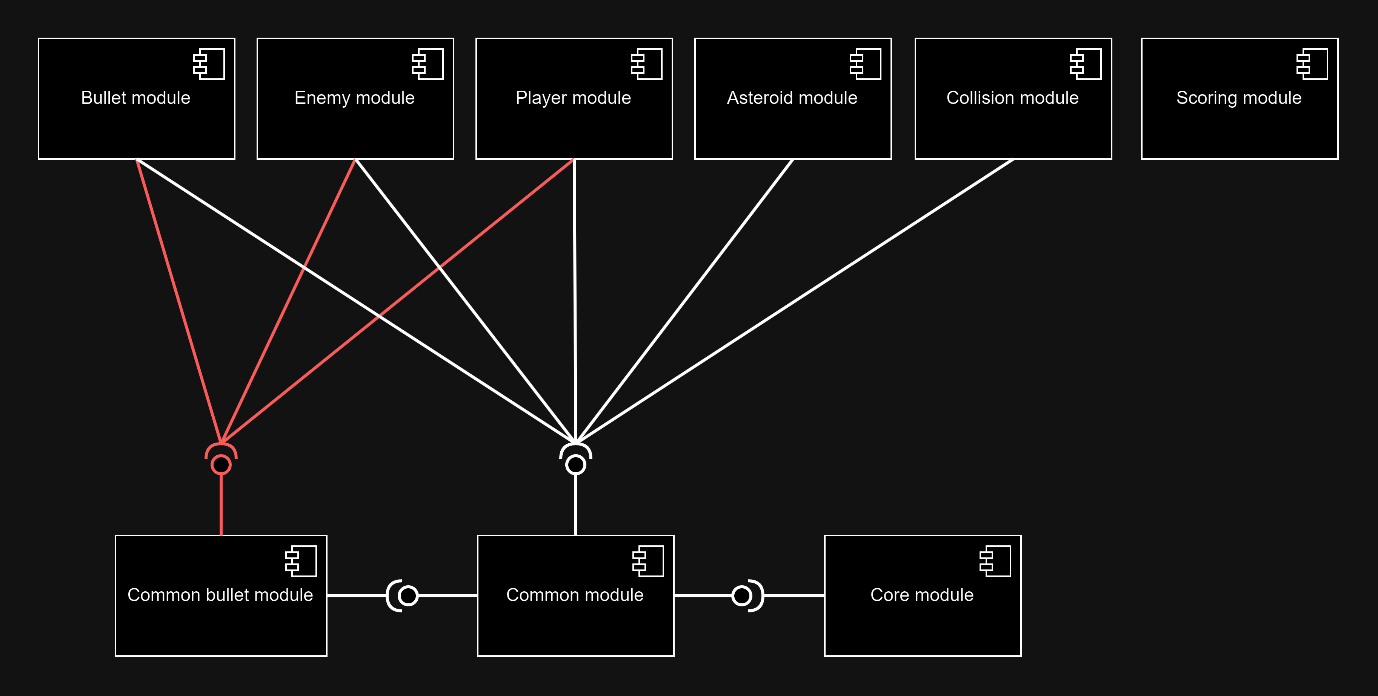
### World

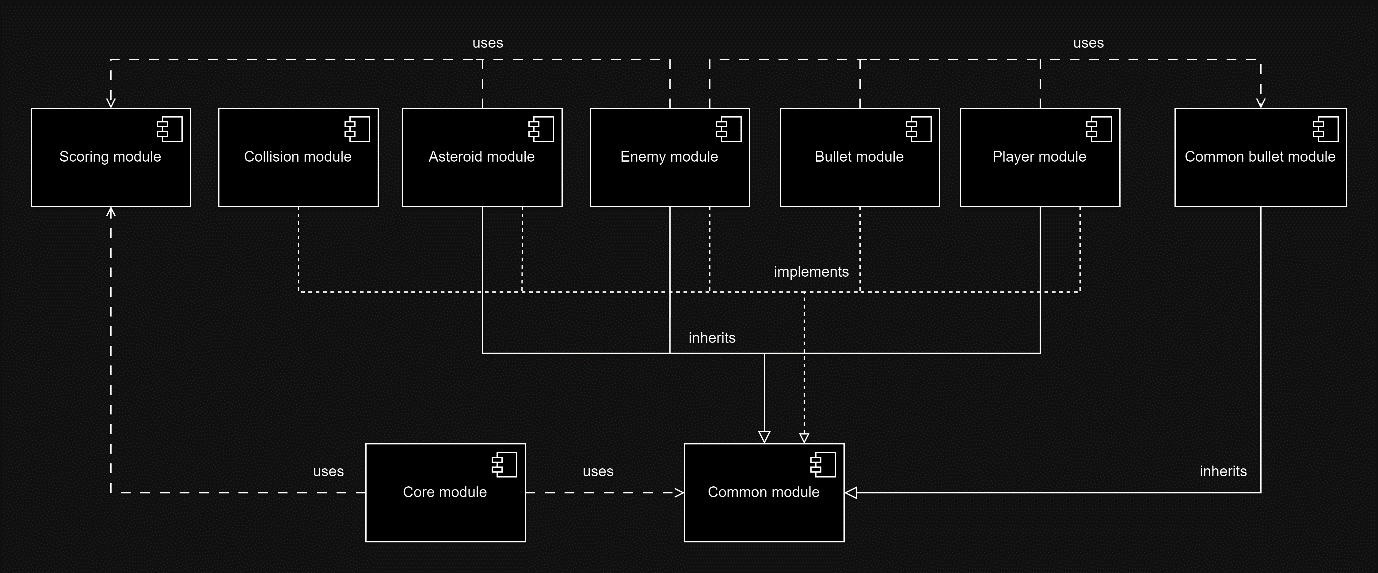
### GameData

### Vector

## Segmented Entities

## Component Model





## SOLID Principles

# 5 Implementation

In implementation, you document the implementation (code) of the components from design.

Describe the details of how the components are registered and accessed.

How are reliable dependencies and strong encapsulation enforced in your project?

What component models are applied and where in the source code?

Provide a descriptive explanation of each element in the implementation and provide arguments for your choices.

You should describe how you register components and how you access them.

Note, you are allowed to reuse the game logic, but you must implement the **[GameLab], [JavaLab]**, **[JPMSLabs]**, **[SpringLab],** **[TestLab]** and [MicroServiceLab] labs based on your own GitHub branches.

# 6 Test

Describe how experimental validation was performed through deployment of the game on top of the component container in a real setting.

Test the system's software-abilities such as dynamic updates using integration and unit test.

## Unit Testing

The project makes use of unit testing specifically for the collision system module. The test class consists of multiple test methods. Each method tests a different situation which could be encountered naturally within the collision system during gameplay. The unit tests ensures that the system behaves as intended and makes it easier to discover errors immediately when changes are made.

More modules and classes could benefit from the implementation of unit tests, though for the scope of this project, a single test class meets the requirements.

For larger projects with multiple developers, unit tests become vital to ensure correct code behavior across the whole system.

## Integration Testing

Manual integration testing was used for testing the functionality of the whole system, as well as validating that all non-common modules can be removed without recompilation or error. These tests were run as a sort of black box tests, meaning that only the result matter without interest in how the system produced these results.

To do this each module was removed one at a time after compilation, to test whether the system can run without each individual module. The same individual tests were run but reversed, only keeping one module at a time to test whether those modules can run in isolation without problems. The game was also tested with all modules, no modules and with the microservice for scorekeeping not running. The results of these tests can be seen in the table below.

|  |  |
| --- | --- |
| **Manual module testing** | |
| ‘/’ *Denotes that a module has been removed and all other modules are present*  ‘+’ *Denotes that only a single module is present, and all others have been removed* | |
| **Test configuration** | **Result** |
| Default (all modules present) | **No errors** |
| / All modules | **No errors** |
| / Player module | **No errors** |
| / Enemy module | **No errors** |
| / Asteroid module | **No errors** |
| / Collision module | **No errors** |
| / Bullet module | **No errors** |
| + Player module | **No errors** |
| + Enemy module | **No errors** |
| + Asteroid module | **No errors** |
| + Collision module | **No errors** |
| + Bullet module | **No errors** |
| Microservice not running | **No errors** |

This manual method of testing is tedious and inefficient and should not be conducted in this manner in projects larger than this. Instead, the integration testing should be done programmatically with an automated test system akin to the previously mentioned unit test.

## Portability Testing

To test the portability quality attribute, described in the non-functional requirements, manual testing was used on multiple computers. For this project this was tested qualitatively by simply looking at the game running on different hardware, to see if major differences would occur.

For more reliable and quantitative testing, the update rate of the program could be measured, while continuously moving an entity across the screen. By evaluating that the entity moves the same distance in a certain amount of time on multiple computers, despite differing update rates, it can be assured that the game is adjusting for different hardware speeds and running correctly.

# 7 Discussion

Discuss how well the game solved the identified essential problems (module updates etc.).

To which extent did your design meet the requirements?

# 8 Conclusion

First summarize the report.

Remember that you are summarizing the report for a reader that has read the introduction and the body of the report already and has a strong sense of key concepts and applied technologies.

Explain the potential impacts of your system in relation to the main issue.

Direct future work directions related to the main issue.

However, this should not be seen as an opportunity to develop new ideas in significant detail and should be clearly linked to the work described in your report.

# References

JPMS

<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>

USE CASE

<https://www.visual-paradigm.com/guide/use-case/what-is-use-case-specification/>

BLACK BOX

<https://www.imperva.com/learn/application-security/black-box-testing/>

SOLID

<https://www.baeldung.com/solid-principles>

MOJOHAUS PLUGIN

https://www.mojohaus.org/exec-maven-plugin/index.html