Component Based Systems

Student name: Oliver Riis-Vestergaard Møller (olmoe22)
Student Exam Number: 190402706
GitHub User: Oliverxll
GitHub Repo: https://github.com/Oliverxll/AsteroidsFX

Abstract

Traditional monolithic software architectures struggle to adapt to the dynamic and evolving requirements of modern development. To overcome these limitations, modular design approaches, such as Component-Oriented Programming (COP), offer enhanced flexibility and maintainability. In this project, we recreated the classic "Asteroids" game using pure Java and the Java Platform Module System (JPMS) to illustrate the advantages of COP. The project also explored the use of Java Module Layers to handle split packages, and investigated the potential of swapping out the core with a Spring framework to dynamically load jar files as components, as well as integrating a Spring Boot microservice. By leveraging patterns like Service Locator and Dependency Injection—implemented via Java's ServiceLoader—we achieved a modular and extensible application structure. The use of Service Provider Interfaces (SPIs) with clearly defined contracts ensured high cohesion and low coupling. This modular approach not only facilitated easier testing through interface mocking but also significantly improved the overall adaptability and robustness of the codebase.

Introduction

Traditional monolithic software architectures, much like the monoliths from which they derive their name, stand tall and imposing with smooth, unyielding surfaces and sharp edges. The term "monolith" originates from the mid-19th century French "monolithe," itself derived from the Greek "monolithos," meaning "single stone." Just as ancient monoliths symbolize solidity and permanence, monolithic software architectures have been characterized by their rigid, inflexible structures. However, in the rapidly evolving landscape of modern software development, this rigidity has become a significant drawback.

Forcing an organization to build software from a monolithic architecture, which does not reflect its communication structure, contradicts the principles of Conway's Law. This law suggests that systems designed by organizations will mirror the communication structures of those organizations. Ignoring this natural tendency results in a burdensome and inefficient development process. While organizing the codebase into domains reflecting team responsibilities may seem like a solution, it does not address the limitations of versioning and deployment. A new paradigm, known as Component-Oriented Programming (COP), offers a way forward.

Component-Oriented Programming allows different teams to independently deploy, version, and test the code specific to their responsibilities. The dependencies between teams are naturally defined by their interactions and contractual obligations: "Do THAT when I give you THIS." Object-Oriented Programming (OOP) languages often include rich type systems with interfaces, which act as contracts by defining methods with specific inputs and outputs. These interfaces ensure that implementations conform to expected behaviors.

Contracts in COP can be further elaborated with preconditions and postconditions: "I expect these criteria to be met before I initiate this transaction," and "I expect these criteria to be met after the transaction is complete." This aligns with the paradigm of Design by Contract, which uses contravariance, covariance, and invariance to define the limitations of inheritance for inputs, outputs, and functions.

To illustrate the benefits of Component-Oriented Programming, this project recreates the classic game "Asteroids" using pure Java and the Java Platform Module System (JPMS). The project explores Java Module Layers to handle split packages and investigates the potential of integrating a Spring core to dynamically load jar files as components, alongside a Spring Boot microservice. Patterns like Service Locator and Dependency Injection, implemented via Java's ServiceLoader, are utilized to achieve a modular and extensible application structure.

The use of Service Provider Interfaces (SPIs) with clearly defined contracts ensures high cohesion and low coupling, facilitating easier testing through interface mocking. This modular approach significantly enhances the adaptability and robustness of the codebase, showcasing how COP can effectively replace traditional monolithic architectures in modern software development.

Requirements

2.1 Functional requirements

Functional Requirements	
F-01	Game has Player Component.
F-02	Game has Enemy Component.
F-03	Game has Weapon Component.
F-04	Game has Core Component.
F-05	Game has Map Component.
F-06	Game has Bullet Component.
F-07	Game has Asteroid Component.
F-08	Game has Collision Component.
F-09	Game updates time between frames every cycle.
F-10	Entities should be updated each update cycle.
F-11	Entities are able to move in all 2D-axis within the game window.
F-12	Player moves on arrow keys.
F-13	The player is not be able to move outside the game window.
F-14	Asteroids and Enemies always move forward.
F-15	Asteroids and Enemies should be continue their movement from the opposite side they cross on the game window.
F-16	Enemies turn randomly.
F-17	Entities have a health attribute.
F-18	Asteroids split when damaged.
F-19	Asteroids, Enemies, and Player are removed when dead.
F-20	Entities have a modifiable move and turn speed.
F-21	Entities are colorable.
F-22	Enemies and Player can shoot a weapon.
F-23	Weapons can shoot once every x interval.
F-24	Player shoots when space is pressed.
F-25	Enemy shoots every x.
F-26	Asteroid-Player and Enemy-Player collision kills player.
F-27	Bullet-Asteroid collision damages Asteroid when shooter is Player.
F-28	Bullet-Enemy collision damages Enemy when shooter is Player.
F-29	Bullet-Player collision damages Player when shooter is Enemy.

2.2 Non-functional requirements

Non-Functional Requirements	
NF-01	Player, Enemy and Weapon components must implement service provided interfaces.
NF-02	Without recompilation, a component can be removed from the execution without affecting the other components.

2.3 Interface Contracts

In a component-based game architecture, interfaces play a crucial role in defining contracts for how different components interact with each other. They define the methods that a component must implement, ensuring a consistent way for components to communicate.

2.3.1 IEntityProcessingService

This interface defines the contract for components that process entities in the game. It has a **process(GameData gameData, World world)** method that is called every frame to update the behavior of the entities. The order of the entities being processed is not guaranteed.

2.3.2 IPostEntityProcessingService

This interface is similar to **IEntityProcessingService**, but its **process(GameData gameData, World world)** method is called after every frame. This can be used for any cleanup or finalization tasks that need to be performed after the main entity processing has been completed. This is most often the place for games to handle collisions before the render of the next frame.

2.3.3 IGamePluginService

This interface defines the contract for plugins that add functionality to the game. It has two methods: **start(GameData gameData, World world)** and **stop(GameData gameData, World world)**. The start method is called when the game starts and is responsible for initializing and adding entities to the world. The stop method is called when the game ends and is responsible for removing entities from the world.

2.3.4 MapSPI

This interface defines the contract for creating and retrieving the game map. It has a **getMap()** method that returns an ImageView object representing the game map. This allows for flexibility in the implementation of the game map, as different implementations can provide different map images.

2.3.5 WeaponSPI

This interface defines the contract for creating weapons in the game. It contains a **createWeapon(Entity owner)** method that creates a new weapon for a given entity, which is typically the player or an enemy. The Entity owner parameter represents the entity that will own the weapon.

2.3.6 BulletSPI

This interface defines the contract for creating bullets in the game. It has a **createBullet(Entity e, GameData gameData)** method that creates a new bullet for a given entity. This allows for flexibility in the implementation of bullets, as different implementations can provide different bullet behaviors and characteristics.

Analysis

This project encompasses two primary use cases: the ability to seamlessly remove a component while ensuring the game continues to function without the removed component's functionality, and the interactive experience of the player within the game.

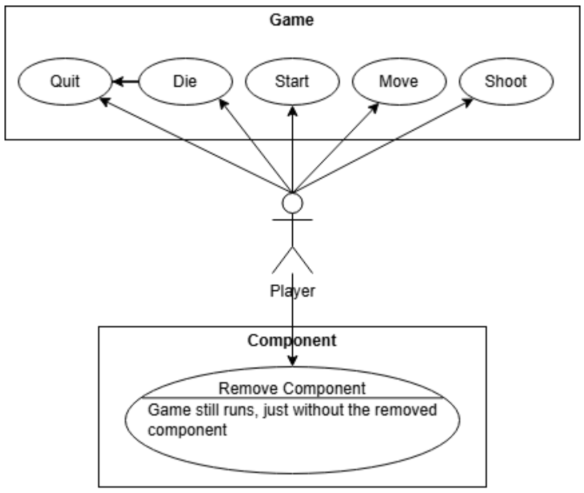


Figure 3.1: Depiction of the two use cases. Component-related and Game-related.

Given the shared use cases among Asteroids, Enemies, Player, Weapon, and Bullet, it could be important to define a common object and establish a contract for how these objects should be loaded, removed, and updated. Creating a base object from which all these entities derive would be an effective approach.

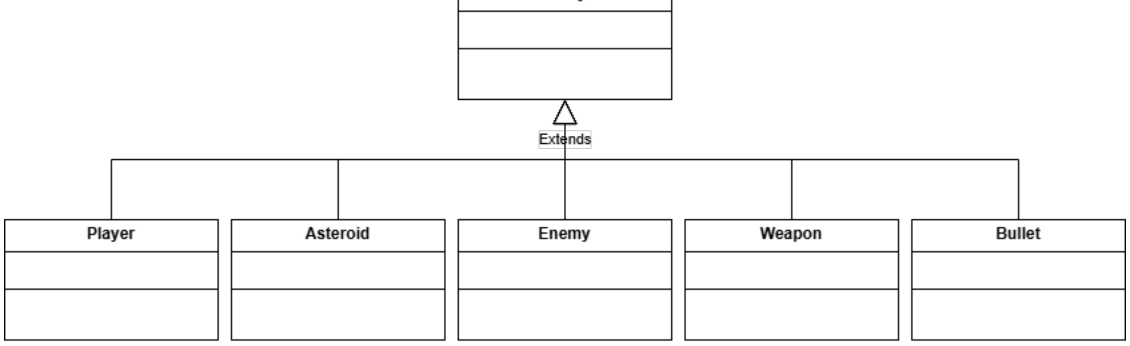


Figure 3.2: One common derivative.

Additionally, this approach allows for the extension of the Entity's functionality. Through polymorphism, these functionalities can be utilized within the game update loop via the established contract, as needed.

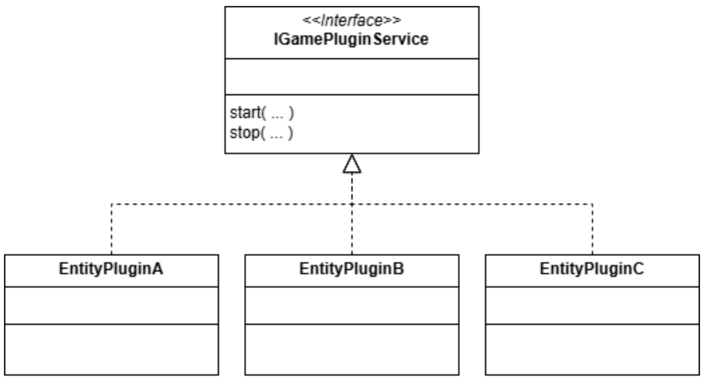


Figure 3.3: One common contract for loading/unloading plugins.

The common contract for instantiating and destroying all entities, as illustrated in Figure 3.3, should be defined by an interface to be implemented by the plugins of the concrete entities.

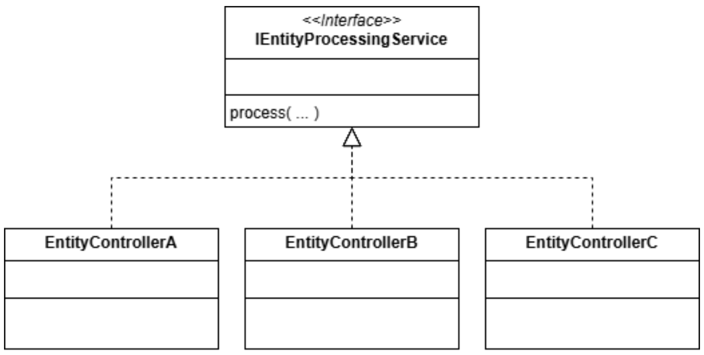


Figure 3.4: One common contract for updating entities.

The common contract for updating all entities, as illustrated in Figure 3.4, should be defined by an interface to be implemented by the controllers of the concrete entities.

This concept should be extended with the IPostEntityProcessingService, which is to be utilized to determine collisions between entities after game logic has been executed.

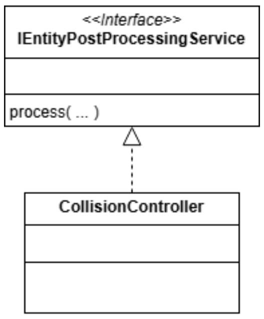


Figure 3.5: Post processing for collisions

Moving to the more specialized components, Bullet, Map, and Weapon, which do not share commonalities with other components, each should have their own contracts.

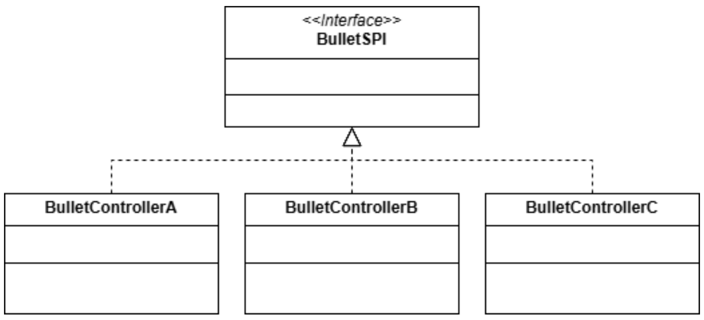


Figure 3.6: Diagram of BulletSPI

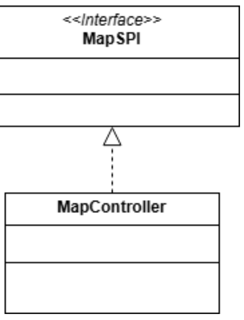


Figure 3.7: Diagram of MapSPI

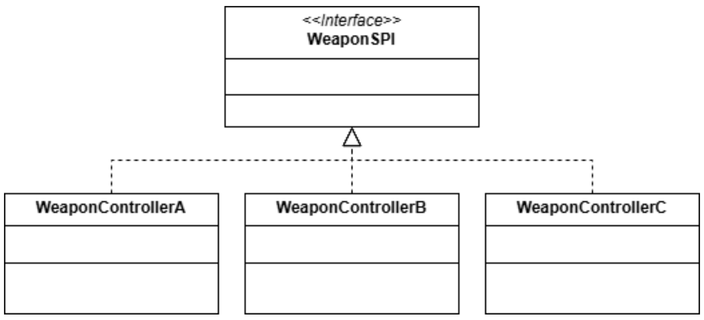


Figure 3.8: Diagram of WeaponSPI

These interfaces would allow for the extension of functionality within their respective domains, enabling the creation of various types of weapons, the loading of different maps, and the shooting of different kinds of bullets.

Keeping the contracts in mind, utilizing Java's Service Locator to find implementations of these contracts provides the most effective separation of concerns. The sequence diagram below illustrates how this process should unfold:

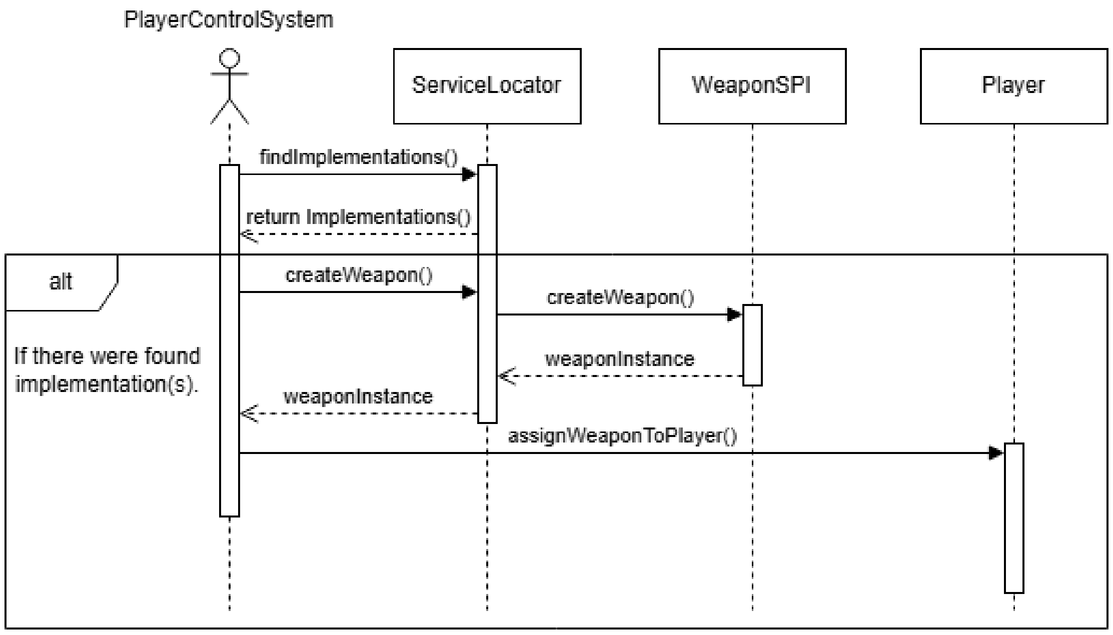


Figure 3.9: Sequence diagram of PlayerControlSystem creating a weapon for the player to use.

Although not all alternate paths are depicted, if no implementations are found, the process will simply not execute the weapon creation step.

Design

The design of the system focuses on structuring the application to fulfill both the functional and non-function requirements. To achieve this, the architecture must be divided into clear abstraction levels, ensuring minimal dependency between components, thereby promoting flexibility and maintainability. Non-Function requirement NF-02 mandates that the architecture supports the removal of components without a recompilation, while NF-01 directs the implementation through Service Provided Interfaces (SPIs).

4.1 Architecture and Abstractions

The architecture of the game is built on a layered approach, adhering to the principles of Clean Architecture. This design philosophy emphasizes the separation of concerns, with each layer having distinct responsibilities and minimal dependencies on the other layers. The key layers in this architecture are:

1. **Entities Layer:** Contains the core business logic and entities such as Asteroid, Player, Enemy, Weapon, Bullet, and Map.
2. **Use Cases Layer:** Manages the application-specific business rules and coordinates the flow of data between the entities and the interfaces.
3. **Interface Adapters Layer:** Facilitates communication between the use cases and external systems or interfaces, such as the UI or external libraries.
4. **Frameworks and Drivers Layer:** Encompasses the details of the frameworks and tools used, such as the Java Platform Module System (JPMS) and Service Locators.

4.2 Component Models

The component models of the game are designed to be modular and interchangeable, adhering to the non-functional requirements. The primary components include:

- Asteroid Component
- Bullet Component
- Enemy Component
- Map Component
- Player Component
- Weapon Component

Each of these components implements specific service interfaces, allowing them to be independently developed, tested, and replaced. The interactions between these components are managed through defined contracts, ensuring high cohesion and low coupling.

4.3 UML Component Diagram

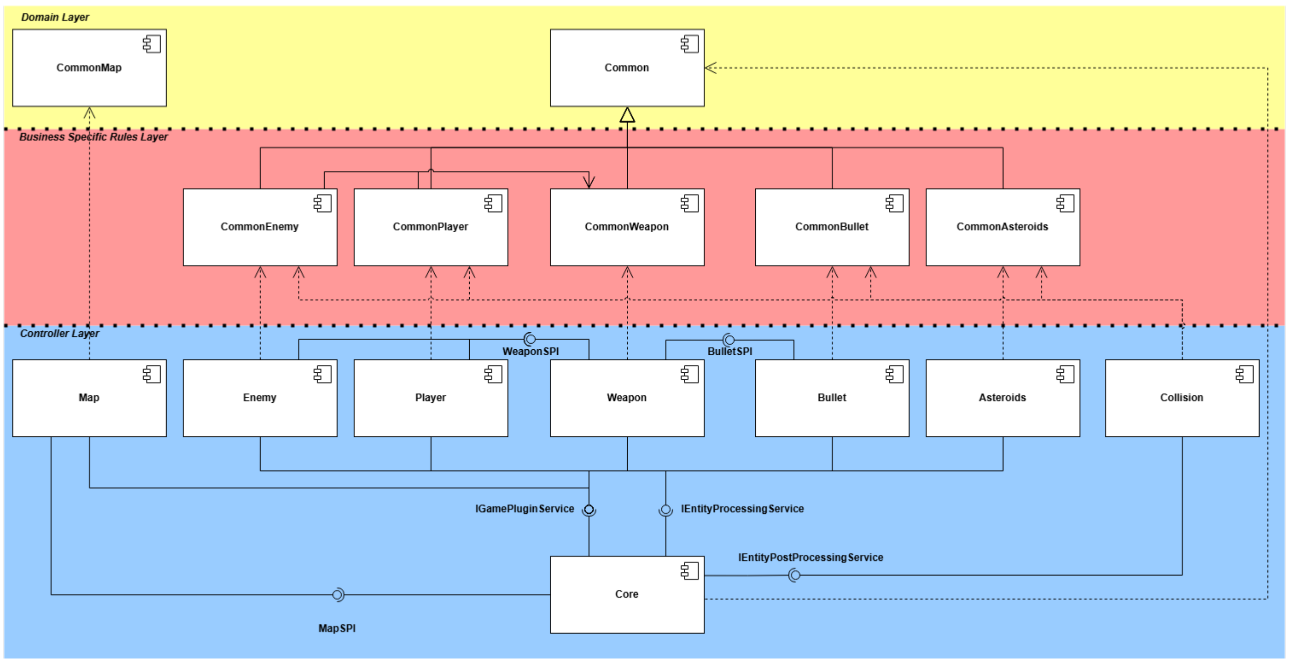


Figure 4.1: Architecture of the projects modules.

4.4 Component Contracts

The contracts for each component are defined in terms of pre-conditions and post-conditions, ensuring that the components adhere to the expected behavior and can be tested independently.

IGamePluginService	
Operation	void start(GameData gameData, World world)
Description	The method is called when the game starts. The initial objects that the plugin is responsible for will be instantiated and put into the world.
Parameters	GameData stores the metadata about the game's state. World stores the instantiated entities of the game.
Pre-conditions	*gameData* isn't null. *world* isn't null. *gameData* contains viewport dimensions.
Post-conditions	All *entities* has been processed in the *world*.

Operation	void stop(GameData gameData, World world)
Description	The method is called when the game ends. Objects created by the plugins are responsible for deconstructing them as well.
Parameters	GameData stores the metadata about the game's state. World stores the instantiated entities of the game.
Pre-conditions	*gameData* isn't null. *world* isn't null. *world* contains *entities* that the plugin is responsible for.
Post-conditions	All *entities* that the plugin is responsible for, are removed from *world*.

IEntityProcessingService	
Operation	void process(GameData gameData, World world)
Description	The method is called every frame and is used to update the behaviour of entities. The order of the entities being processed is **NOT** guaranteed.
Parameters	GameData stores the metadata about the game's state. World stores the instantiated entities of the game.
Pre-conditions	*gameData* isn't null. *world* isn't null. *world* contains *entities* that has a controller that implements *IEntityProcessingService*
Post-conditions	All *entities* has been processed in the *world.*

IEntityPostProcessingService	
Operation	void process(GameData gameData, World world)
Description	The method is called **AFTER** every frame and is used to update the behaviour of the entities. The order of the entities being processed is **NOT** guaranteed.
Parameters	GameData stores the metadata about the game's state. World stores the instantiated entities of the game.
Pre-conditions	*gameData* isn't null. *world* isn't null. *world* contains *entities.*
Post-conditions	All *entities* has been processed in the *world.*

MapSPI	
Operation	ImageView getMap()
Description	The method is called when setting the map background.
Parameters	
Pre-conditions	*GamePlugins* have been started. An image at the load location exists.
Post-conditions	A ImageView object is created and returned.

WeaponSPI	
Operation	Weapon createWeapon(Entity owner)
Description	The method is called when creating a weapon is needed. The order of the entities being processed is **NOT** guaranteed.
Parameters	Owner is the entity who is going to own the weapon, used for creating bullets.
Pre-conditions	*owner* isn't null.
Post-conditions	A new *weapon* instance is created, added to the *world*, and returned.

BulletSPI	
Operation	Entity createBullet(Entity shooter, GameData gameData)
Description	The method is called when creating a bullet is needed.
Parameters	GameData stores the metadata about the game's state. World stores the instantiated entities of the game.
Pre-conditions	*shooter* isn't null. *gameData* isn't null.
Post-conditions	A new *bullet* is created and added to the *world.*

Communication between modules is handled using Service Provider Interfaces (SPIs) to maintain loose coupling and high cohesion. The modules are designed to interact through these interfaces rather than direct references, ensuring flexibility and modularity. Figure **??** illustrates the relationships between the system's modules, highlighting how controllers utilize their respective Common modules and are processed in the Core and interact with each others through SPIs.

Bullet Module

- **Description:** Manages bullet entities within the game.
- **Dependencies:** Requires **Common** and **CommonBullet** modules.
- **Communication:** Provides implementations for **IGamePluginService**, **BulletSPI**, and **IEntityProcessingService**.

Asteroid Module

- **Description:** Manages asteroid entities.
- **Dependencies:** Requires **Common** and **CommonAsteroids** modules.
- **Communication:** Provides implementations for **IGamePluginService** and **IEntityProcessingService**.

Enemy Module

- **Description:** Manages enemy entities.
- **Dependencies:** Requires **Common**, **CommonEnemy**, and **CommonWeapon** modules.
- **Communication:** Uses **WeaponSPI** and provides implementations for **IGamePluginService** and **IEntityProcessingService**.

Player Module

- **Description:** Manages the player entity.
- **Dependencies:** Requires **Common**, **CommonPlayer**, and **CommonWeapon** modules.
- **Communication:** Uses **WeaponSPI** and provides implementations for **IGamePluginService** and **IEntityProcessingService**.

Map Module

- **Description:** Manages the game map.
- **Dependencies:** Requires **Common** and **CommonMap** modules.
- **Communication:** Provides implementations for **IGamePluginService** and **MapSPI**.

Weapon Module

- **Description:** Manages weapons used by players and enemies.
- **Dependencies:** Requires **Common**, **CommonWeapon**, and **CommonBullet** modules.
- **Communication:** Uses **BulletSPI** and provides implementations for **WeaponSPI**, **IEntityProcessingService**, and **IGamePluginService**.

Implementation

The implementation of the portfolio project was developed through 6 days and the iterations from the labs of the course. It is the final product implementation that will be in this chapter. There are some of the labs that sent the project on side quests to explore Java's Module Layer for handling split packages, swapping out the Core component with a Spring Application core, and incorporating a Spring Boot microservice.
The game logic is not something worth taking a deep dive into, as it does not solve the primary goal of the project, that is component-oriented programming.

Each component—Asteroid, Bullet, Collision, Enemy, Map, Player, and Weapon—is implemented with a clear separation of concerns regarding their definition, instantiation, and usage. This approach ensures modularity and flexibility within the system.
For example, the Enemy component is structured as follows:

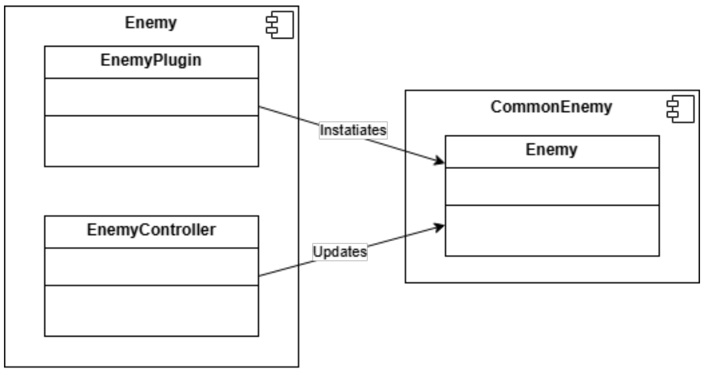


Figure 5.1: Classes of Enemy

Enemy Class: Defines enemy attributes and behaviors.
EnemyPlugin: Manages the creation and destruction of enemy entities.
EnemyControlSystem: Updates enemy states each game cycle.

This design mirrors creational design patterns such as Factory and Builder patterns, allowing for flexible and maintainable code. By maintaining the type definitions within the business-specific rules layer, rather than the controller layer, the architecture supports better modularity and easier updates. This separation ensures that each component can independently handle its own responsibilities while remaining aware of other necessary components through well-defined interfaces.

5.1 Component Registration and Access

The components in this project are registered and accessed using Java's Service Provider Interface (SPI) and ServiceLoader. This mechanism allows dynamic discovery and loading of service implementations, promoting loose coupling and high modularity.

5.1.1 Registration

Each component module contains a module-info.java file where services are provided and used. For example, the Enemy module registers its services as follows:

```
module Enemy {
    requires javafx.graphics;
    requires Common;
    requires CommonEnemy;
    requires CommonWeapon;

    uses dk.sdu.mmmi.cbse.common.weapon.WeaponSPI;

    provides IGamePluginService with EnemyPlugin;
    provides IEntityProcessingService with EnemyControlSystem;
}
```

Figure 5.2: Enemy module-info registration.

5.1.2 Access

The ServiceLoader class is used to load implementations of the registered services dynamically. For example, the PlayerControlSystem accesses weapon services as follows:

```
if(gameData.getKeys().isDown(GameKeys.SPACE)) {
    if (player.getWeapon() != null) {
        // Logic for shooting
    } else {
        getWeaponSPIs().stream().findFirst().ifPresent(
                spi -> {
                    // Logic for attaching weapon to Player
                }
        );
    }
}

private Collection<? extends WeaponSPI> getWeaponSPIs() {
    return ServiceLoader.load(WeaponSPI.class)
                        .stream().map(ServiceLoader.Provider::get)
                        .collect(toList());
}
```

Figure 5.3: PlayerControlSystem accessing WeaponControlSystem through the WeaponSPI from CommonWeapon.

5.2 Enforcing Reliable Dependencies and Strong Encapsulation

Reliable dependencies and strong encapsulation are enforced through careful module design and adherence to clean architecture principles. An example of this can be seen in aforementioned Figure 5.2 where a clear outline of which dependencies are required by the Enemy module.

5.2.1 Module Boundaries:

Each module defines clear boundaries for what is exposed and what is hidden. For instance, **module-info.java** ensures only necessary packages are exported, while internal packages remain inaccessible to other modules.

5.2.2 Interface-Based Design:

Interfaces define contracts for services, ensuring that implementations can be easily swapped without affecting other parts of the system. This is evident in how **IGamePluginService** and **IEntityProcessingService** are used throughout the system.

5.2.3 Dependency Injection:

Dependency injection, facilitated by the ServiceLoader, ensures that dependencies are provided at runtime, promoting decoupling and testability.

5.3 Component Models and Source Code

5.3.1 Plugin Model:

Each game component (e.g., Enemy, Player, Weapon) follows a plugin model where the Plugin interface is implemented to manage the lifecycle of game entities.

```java
public class EnemyPlugin implements IGamePluginService {
    // Vars

    @Override
    public void start(GameData gameData, World world) {
        // Logic for instantiating when game starts
    }

    public Entity createEnemy(GameData gameData) {
        // Logic for creating the instances
    }

    @Override
    public void stop(GameData gameData, World world) {
        // Logic for removing the instances
    }
}
```

Figure 5.4: Plugin Model

5.3.2 Processing Model:

The processing of entities is handled by classes implementing IEntityProcessingService, which updates entity behavior each frame.

```java
public class EnemyControlSystem implements IEntityProcessingService {

    // Vars

    @Override
    public void process(GameData gameData, World world) {
        for (Entity entity : world.getEntities(Enemy.class)) {
            // Logic for processing all entities of type Enemy
        }
    }
}
```

Figure 5.5: Processing Model

5.3.3 Service Provider Interface (SPI) Model:

Specialized components such as WeaponSPI, Map, and BulletSPI define contracts for getting the loaded map, creating weapons and bullets, allowing for extensible and interchangeable implementations.

```java
public interface WeaponSPI {
    void createWeapon(Entity owner);
}
public interface MapSPI {
    ImageView getMap();
}
public interface BulletSPI {
    Entity createBullet(Entity e, GameData gameData);
}
```

Figure 5.6: SPI models

5.4 Side Quests

5.4.1 Spring Application Core

Throughout the project, the labs sent the project on a few side quests, this one is where the Java Core is swapped with a Spring Application Core to manage dependency injections, showcasing the power of component-oriented programming.

The main class initializes the Spring context and starts the game, injecting the necessary dependencies:

```java
    public class Main extends Application {

    public static void main(String[] args) {
        launch(Main.class);
    }

    @Override
    public void start(Stage window) throws Exception {
        AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(ModuleCo

        for (String beanName : ctx.getBeanDefinitionNames()) {
            System.out.println(beanName);
        }

        Game game = ctx.getBean(Game.class);
        game.start(window);
        game.render();
    }
}
```

Figure 5.7: The Spring Application main

The **Game** class is modified to not extend JavaFX application and to receive its dependencies via constructor injection. This setup allows Spring to manage the lifecycle and injection of these components, ensuring clean separation of concerns.

```java
public class Game {

    // Vars

    Game(   List<IGamePluginService> gamePluginServices,
            List<IEntityProcessingService> entityProcessingServiceList,
            List<IPostEntityProcessingService> postEntityProcessingServices,
            List<MapSPI> mapServices) {
        this.gamePluginServices = gamePluginServices;
        this.entityProcessingServiceList = entityProcessingServiceList;
        this.postEntityProcessingServices = postEntityProcessingServices;
        this.mapServices = mapServices;
    }

    // Rest of the class
}
```

Figure 5.8: The modified Game Class

To facilitate Spring's dependency injection, we need a configuration class (**ModuleConfig**) that defines how beans are created and wired together:

```java
@Configuration
class ModuleConfig {

    public ModuleConfig() {
    }

    @Bean
    public Game game(){
        return new Game(getGamePluginServices(), getEntityProcessingServices(), getPostEntityPro
    }

    @Bean
    public List<IEntityProcessingService> getEntityProcessingServices(){
        return ServiceLoader.load(IEntityProcessingService.class)
                            .stream().map(ServiceLoader.Provider::get)
                            .collect(toList());
    }

    @Bean
    public List<IGamePluginService> getGamePluginServices() {
        return ServiceLoader.load(IGamePluginService.class)
                            .stream().map(ServiceLoader.Provider::get)
                            .collect(toList());
    }

    @Bean
    public List<IPostEntityProcessingService> getPostEntityProcessingServices() {
        return ServiceLoader.load(IPostEntityProcessingService.class)
                            .stream().map(ServiceLoader.Provider::get)
                            .collect(toList());
    }

    @Bean
    public List<MapSPI> getMapServices() {
        return ServiceLoader.load(MapSPI.class)
                            .stream().map(ServiceLoader.Provider::get)
                            .collect(toList());
    }
}
```

Figure 5.9: ModuleConfig for Spring Application

5.4.2 Split Packages

One of the tasks the project faced involved resolving the issue of split packages by using Java Module Layers. This was necessary because the lab mandated that identical packages and classes existed in two modules, creating conflicts.

The Main class initializes a module layer for dynamically loaded plugins, resolving the issue of split packages by clearly defining module boundaries and ensuring that each module is self-contained. The modules that contain these identical packages and classes are placed into a plugins folder.

```java
public class Main extends Application {

    // Vars

    private static ModuleLayer layer;

    public static void main(String[] args) {
        Path pluginsDir = Paths.get("plugins"); // Directory with plugins JARs

        // Search for plugins in the plugins directory
        ModuleFinder pluginsFinder = ModuleFinder.of(pluginsDir);

        // Find all names of all found plugin modules
        List<String> plugins = pluginsFinder
                .findAll()
                .stream()
                .map(ModuleReference::descriptor)
                .map(ModuleDescriptor::name)
                .collect(Collectors.toList());

        // Create configuration that will resolve plugin modules
        // (verify that the graph of modules is correct)
        Configuration pluginsConfiguration = ModuleLayer
                .boot()
                .configuration()
                .resolve(pluginsFinder, ModuleFinder.of(), plugins);

        // Create a module layer for plugins
        layer = ModuleLayer
                .boot()
                .defineModulesWithOneLoader(pluginsConfiguration, ClassLoader.getSystemClassLoad

        launch(Main.class);
    }

    // Rest of the class
}
```

Figure 5.10: Module Layer implementation

5.4.3 Spring Boot Microservice

The last task for the project was to implement a microservice into the project for tracking the player's score, the integration is handled by both the game client and a Spring Boot-based REST API. The game client sends HTTP requests to this REST API to get and update the score.

The **ScoringSystemApplication** is a Spring Boot application that provides REST endpoints for managing the player's score.

```java
@SpringBootApplication
@RestController
@CrossOrigin
public class ScoringSystemApplication {

    private int score = 0;

    public static void main(String[] args) {
        SpringApplication.run(ScoringSystemApplication.class, args);
    }

    @GetMapping("/score/get")
    public int getScore() {
        return this.score;
    }

    @PutMapping("/score/add/{score}")
    public int addToScore(@PathVariable(value = "score") int score) {
        this.score += score;

        return this.score;
    }

    @PutMapping("/score/set/{score}")
    public int updateScore(@PathVariable(value = "score") int score) {
        this.score = score;

        return this.score;
    }
}
```

Figure 5.11: Scoring System Spring Boot REST API application

The **Main** class in the game client uses **HttpClient** to communicate with the scoring microservice. This involves sending HTTP requests to retrieve the score for a HUD element, that is *Text score*.

```java
public class Main extends Application {

    //Vars

    private Text score;
    private HttpClient client = HttpClient.newHttpClient();

    // Class logic

    @Override
    public void start(Stage window) throws Exception {
        // Start logic

        HttpRequest requestGetScore = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:8080/score/set/0"))
                .PUT(HttpRequest.BodyPublishers.ofString(""))
                .build();

        // Retrieves the score for amount of asteroids destroyed.
        try {
            HttpResponse<String> responseGetScore = client.send(requestGetScore, HttpResponse.Bo
            score.setText("Destroyed asteroids: " + responseGetScore.body());
        } catch (IOException | InterruptedException e) {
            throw new RuntimeException(e);
        }

        // Rest of start
    }

    // Class logic

    private void update() {
        // Update loop logic

        HttpRequest requestGetScore = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:8080/score/get"))
                .GET().build();

        // Retrieves the score for amount of asteroids destroyed.
        try {
            HttpResponse<String> responseGetScore = client.send(requestGetScore, HttpResponse.Bo
            score.setText("Destroyed asteroids: " + responseGetScore.body());
        } catch (IOException | InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    // Rest of class
}
```

Figure 5.12: Modified Main Class

Here is the **CollisionDetector** code integrated with the scoring microservice:

```java
public class CollisionDetector implements IPostEntityProcessingService {

    HttpClient client = HttpClient.newHttpClient();

    @Override
    public void process(GameData gameData, World world) {
        // two for loops for all entities in the world
        for (Entity entity1 : world.getEntities()) {
            for (Entity entity2 : world.getEntities()) {

                // Collision logic

                // CollisionDetection
                if (collides(entity1, entity2)) {

                    // Collision logic

                    // Bullet collisions.
                    if (entity1 instanceof Bullet) {
                        if (((Bullet) entity1).getShooter() instanceof Player) {
                            if (entity2 instanceof Asteroid) {
                                entity2.setHealth(entity2.getHealth() - 1); // Asteroid handled
                                ((Asteroid) entity2).setHit(true);

                                incrementScore(1);
                            }

                        // Rest of collision logic

                    }
                }
            }
        }
    }

    // Class logic

    private void incrementScore(int i) {
        HttpRequest requestAddToScore = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:8080/score/add/" + i))
                .PUT(HttpRequest.BodyPublishers.ofString(""))
                .build();
        try {
            client.send(requestAddToScore, HttpResponse.BodyHandlers.ofString());
        } catch (IOException | InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 5.13: Modified Collision Class

Test

In software development, testing ensures the reliability and quality of the application. When dealing with service locator patterns and dependency injection frameworks like Java Platform Module System, it's crucial to devise effective strategies for both unit and integration testing. Below, we'll discuss different integration testing strategies applicable to the Asteroids game application, highlighting their pros and cons.

6.1 Unit Testing

In the project JUnit and Mockito was utilized for unit testing functionality of the components. There were produced Unit Tests for the Player module and its functionality. The tests for the Player component focused on its core functionalities, including movement, rotation, health management, and interactions with the game world. Below is a breakdown of each test and its purpose.

6.1.1 Before Each

Before each test, the **setUp** method initializes the **PlayerControlSystem**, **GameData**, **World**, and **Player** objects. Mocks are created for **GameData**, **World**, and **Player** using Mockito.
The **GameKeys** object, which tracks key presses, is also mocked and set up to be returned when **getKeys()** is called on **gameData**.
This setup ensures that each test starts with a consistent and controlled environment.

```java
@BeforeEach
public void setUp() {
    playerControlSystem = new PlayerControlSystem();
    gameData = mock(GameData.class);
    world = mock(World.class);
    player = mock(Player.class);
    GameKeys gameKeys = mock(GameKeys.class);
    when(gameData.getKeys()).thenReturn(gameKeys);
    when(world.getEntities(Player.class)).thenReturn(List.of(player));
}
```

Figure 6.1: Setup before each test

6.1.2 Movement

```
@Test
public void testProcessPlayerMovesUp() {
    when(player.getHealth()).thenReturn(1);
    when(gameData.getKeys().isDown(GameKeys.UP)).thenReturn(true);
    when(player.getRotation()).thenReturn(0.0);
    when(player.getMoveSpeed()).thenReturn(1);
    when(gameData.getDeltaTime()).thenReturn(1.0);
    double initialX = 0.0;
    double initialY = 0.0;
    when(player.getX()).thenReturn(initialX);
    when(player.getY()).thenReturn(initialY);

    playerControlSystem.process(gameData, world);

    verify(player, times(1)).setX(anyDouble());
    verify(player, times(1)).setY(anyDouble());
}
@Test
public void testProcessPlayerMovesLeft() {
    when(player.getHealth()).thenReturn(1);
    when(gameData.getKeys().isDown(GameKeys.LEFT)).thenReturn(true);

    playerControlSystem.process(gameData, world);

    verify(player, times(1)).setRotation(anyDouble());
}
@Test
public void testProcessPlayerMovesRight() {
    when(player.getHealth()).thenReturn(1);
    when(gameData.getKeys().isDown(GameKeys.RIGHT)).thenReturn(true);

    playerControlSystem.process(gameData, world);

    verify(player, times(1)).setRotation(anyDouble());
}
```

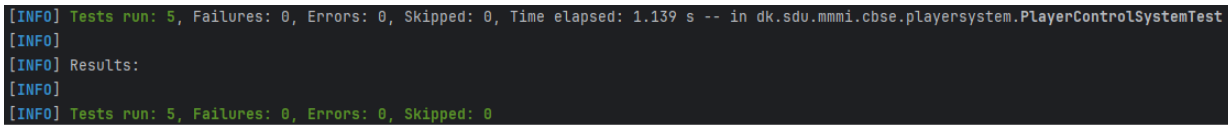
Figure 6.2: Player Movement Tests

6.1.3 Damage and Death

```java
@Test
public void testPlayerTakesDamage() {
    // Assume the player's initial health is 3
    when(player.getHealth()).thenReturn(3);

    // Simulate the player taking 1 damage
    when(player.getHealth()).thenReturn(2);

    // Assert that the player's health has decreased to 2
    assertEquals(2, player.getHealth());
}

@Test
public void testProcessPlayerHealthZero() {
    when(player.getHealth()).thenReturn(0);
    playerControlSystem.process(gameData, world);
    verify(world, times(1)).removeEntity(player);
}
```

Figure 6.3: Player Damage and Death Tests

6.1.4 Verification of test passes



```
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.139 s -- in dk.sdu.mmmi.cbse.playersystem.PlayerControlSystemTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Figure 6.4: Screen print of terminal

The primary reason for the lack of integration tests in this project, is the limited time available before the project deadline, this is the authors own shortcoming and has nothing to do with the actual 4 months of time that was available to finalize the project. The project's focus might have been on developing core functionalities and ensuring that the system works as expected in a real-world scenario. Another part is, that the Test lab never mentioned the production of an integration test, only producing tests for one of the components, as a result, integration testing have been deprioritized.

Despite this, manual testing was done when ensuring that modules can be removed and the application can be run again without a recompilation. The following modules have been confirmed personally by me, the author:

Asteroids	Works
Bullet	Works
Collision	Works
Enemy	Works
Map	Works
Player	Works
Weapon	Works

Table 6.1: Table of Modules that are able to be removed without recompilation.

A video showcasing this, can be found here: https://youtu.be/KG-zCLTcGcM

6.2 How integration tests could have been produced:

Integration Testing:
Goal: Verify the interaction between different components and ensure they work together as expected.
Tools: Use JUnit for the test framework and Mockito for mocking dependencies.
Approach:

- Set up a test environment that mirrors the production setup as closely as possible.
- Write tests to check the integration between different services, such as the weapon creating bullets through BulletSPI.

6.2.1 Using Maven to Share and Unpack Tests

Maven can attach test classes to each module, allowing shared tests across multiple modules. Tests can then be included or excluded during the unpacking process.

Pros	Cons
Both unit and integration tests shared in one step.	Needs a workaround cause of MDEP-732.
Shared tests will be packed into dependent artifact. Prevent changes.	Class duplication makes dependent test-jar bigger unnecessarily.
Shared tests will be packed into dependent artifact. Prevent changes.	
Inclusion and exclusion supported during unpack.	

6.2.2 Using *maven-surefire-plugin* and *maven-failsafe-plugin* with *dependenciesToScan*

These plugins have the *dependenciesToScan* configuration attribute, allowing them to scan listed dependencies for test classes to include and run.

Pros	Cons
Easy to separate unit- and integration test sharing.	Configuration must be duplicated when unit- and integration tests are shared.
Shared tests wont copied, no class duplication.	Delusive [ERROR] messages in logs but build is success.
Responsible plugins will handle dependent test classes	Possible configuration changes time to time, version to version.

6.2.3 Using *build-helper-maven-plugin*

Pros	Cons
None	Works only with local folders. External artifacts not supported.
	Folders are hardcoded. Maintenance issues.

[1]
Although no integration tests were produced due to time constraints, a well-structured approach to testing can be implemented. Starting with unit tests followed by integration, will ensure the application is robust and reliable. Leveraging tools like JUnit and Mockito will facilitate the creation of effective tests that validate the application's functionality and interactions.

Discussion

All the project requirements have been successfully fulfilled. Although the game does not exit when the player dies, this was not explicitly stated as a requirement. From a user experience perspective, it might be preferable to leave the game running even after the player's death, as it gives the player an opportunity to reflect on their performance or restart the game. This approach aligns with the general practice in game design where players can have a moment to review their score or game over screen before exiting or restarting.

While numerous additional gameplay features could be implemented, the focus of this project was not on expanding gameplay but on meeting the core requirements and demonstrating the architecture and design principles.

A higher testing coverage would enhance the resilience of the game, facilitating further development and maintenance. To ensure robustness and detect potential issues early, a Continuous Integration (CI) pipeline could be set up. This pipeline would automate the testing process, including the removal of components and verifying the game's execution without failures. The process would involve running a script that gradually removes components one by one, except those starting with "Common" or named "Core," through pattern matching. The game would then be compiled and executed to ensure it runs correctly before merging new features into the main branch. While some might view this as over-engineering for a small game, it is crucial for demonstrating the benefits of CI pipelines in enhancing project reliability and streamlining future development efforts.

Each of the lab exercises has been completed as described, ensuring that all specified objectives and requirements have been met. This structured approach highlights the project's successful completion.

Conclusion

In this report, we detailed the development and implementation of a modular game system, using principles of Component Oriented Programming to recreate the classic Asteroids game. Through the integration of Java's native ServiceLoader, and the Java Platform Module System (JPMS), we built an application that is robust and extendable. The project sucesfully explored how the core of the game can be swapped with a Spring Application core, how Split Packages are handled with Java's Module Layer, and integrating a microservice into a component-oriented programming project. The architecture was designed to be closed to modification yet open to future extensions, facilitated by a Business Specific Rules abstraction layer that intermediates component interactions. This design ensures low coupling and high cohesion, key principles that guided our architectural decisions.

The game successfully incorporates expected features such as player movement, shooting, asteroid splitting, and enemies that interact with the player. Each component of the system adheres to strict architectural standards, promoting maintainability and scalability. The modular approach allows for easy updates and the addition of new features without disrupting existing functionality.

The potential impact of this system is significant, demonstrating how a well-architected, modular approach can simplify game development and future-proof the application against evolving requirements. By implementing a Continuous Integration pipeline, future development can be streamlined, ensuring that new features and updates are integrated seamlessly and tested rigorously.

Future work should focus on increasing the testing coverage and incorporating more sophisticated gameplay features to enhance the user experience. The established architecture provides a solid foundation for continuous improvement and expansion, ensuring that the game can adapt and grow in response to user feedback and technological advancements. By implementing a Continuous Integration pipeline, future development can be streamlined, ensuring that new features and updates are integrated seamlessly and tested rigorously.

In conclusion, this project showcases the successful application of modern software engineering principles to create a flexible, scalable, and maintainable game system. The use of modular architecture and microservices sets a strong precedent for future development, ensuring a resilient and adaptable platform for ongoing enhancements.

Bibliography

[1] Testing java multi-module maven project. [Online]. Available: https://stackoverflow.com/questions/65344318/sharing-integration-test-in-multi-module-maven-project