Software Technology - 4. Semester

Asteroids Project: Komponentbaserede systemer

May 02 2025

Course Code - T510035101

Written by:

Silas Jul Kierstein - Mail: sikie23@student.sdu.dk

Contents

1 Abstract

This report details the development of 'Asteroids', a 2D arcade game built using JavaFX, addressing the challenge of managing unique game entities in a flexible and maintainable way. A core principle of my game's architecture is the implementation of service-provided interfaces by components such as the Player, Enemy, and Weapon. With the use of Java's service-loaders, dependencies between certain components can be avoided, making them more self contained. With this approach the game becomes more modular, and makes future expansions simpler, such as modification, testing, and new components, without impacting unrelated parts of the system.

2 Introduction

In my game 'Asteroids', a players is challenged to survive in an alien infested space, steering a spaceship through an asteroid field. It uses sprites to draw visuals, and animations from a free asset-packs, made by Foozle [1].

Players control their ship's movement/firing, and with asteroids/enemies coming at you non-stop. It requires quick reflexes to navigate the ever-increasing onslaught of asteroids, their splitting fragments once shot, and hostile enemy spacecrafts. The goal of my game is to achieve the highest score, by shooting away, and destroying whatever you can, and outlast the relentless space environment, for as long as possible.

3 Requirements

Requirements are elicited from the report template and lab-descriptions from throughout the semester. The requirements has guided the development process, been a key factor in decision making, and is considered the standard in software industry, when it comes to creating software solutions that satisfies stakeholder demands.

3.1 Requirement specification

To keep track of requirements they were each put in a table containing ID, Description and a Priority, using the MoSCoW prioritization method.

The Must-have requirements, being the critical parts of the system, that the product cannot function without such as the 'Player'-component. Should-have's are important parts of the project, that should be implemented. Could-have's being desired features, but not vital for the game to be functional. the Won't-have requirements are initiatives that are recognized, but not prioritized for the development of the game (I don't have any for this project).

MoSCoW: Functional requirements		
ID	**Description**	**Priority**
F1	A 'Player' component	**Must**
F2	A 'Enemy' component	**Must**
F3	A 'Bullet'/'Weapon' component	**Must**
F4	A 'Asteroid' component	**Must**
F5	Rendering and drawing (core) components	**Must**
F6	'Player', 'Enemy' and 'Weapon' components implement service provided interfaces that allow the components to be updated and removed without recompilation.	**Must**
F7	Implement provided interfaces: IGamePluginService, IEntityProcessorService and IPostEntityProcessorService	**Must**
F8	Document service interfaces using Java doc	**Must**
F9	Asteroids move randomly	**Must**
F10	Implement a simple collision detection system based on Pythagoras and the provided IPostEntityProcessorService interface	**Must**
F11	When fired upon Asteroids should split into two smaller Asteroids and when small enough they should be destroyed	**Must**
F12	The player ship and enemy-ships should be destroyed when hit by each others bullets a certain number of times	**Must**
F13	Implement game using Java Platform Module System	**Must**
F14	Declare imports and exports in module-info.java files for each module	**Must**
F15	Write a unit test for one of the components	**Must**
F16	Implement the Core component from the Asteroids game using the Spring container and the Dependency Injection Component Model.	**Must**
F17	Instantiate the Game class and use Spring for Dependency injection of the IEntityProcessors and IGamePluginServices.	**Must**
F18	Integrate the Scoring MicroService in the AsteroidsGame using the Spring RestTemplate	**Must**
F19	Implement sprites for visuals	**Should**
F20	Create different types of enemies	**Should**
F21	Objects drawn out of screen are removed	**Should**
F22	Implement sprites for visuals	**Should**
F23	Use sprite animations for certain events like dying and shooting	**Should**
F24	Difficulty scales with time played	**Could**
F25	Player levels up after shooting enough enemies, and can choose one of three upgrades to his ship	**Could**
F26	Create random pickups dropped randomly by enemies	**Could**
F27	Use delta-time for smooth animations and movement	**Could**

Table 1: *Table of functional requirements.*

In a real world setting the must-have's above are not 'critical' parts of the system, as the game might run fine without them. But this is what is asked of me to do (can be compared to product owner demands), so they become must-have's.

The should- and could-have's, are my ideas and touches, that I think would improve playability and add character to the game. They are added to make it more engaging for a user, and for me to be more invested in developing the game as this is what I wish I could do. I did not include non-functional requirements as this is outside the scope of the project.

4 Analysis

With requirements defined and ideas in place, this section describes with the use of UML diagrams what the game should do, the primary components, sequences, and how they should connect (in theory).

4.1 Player interaction

The 'Player' should be a spaceship that is controlled with the use of the mouse and AWSD-keys. I used a use case diagram, to visualize player actions:

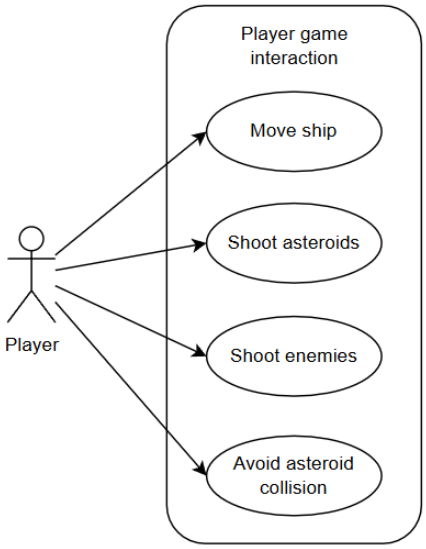

Figure 1: *Player interaction use-case diagram*

The use case diagram above shows the most important interactions between the Player and game systems. The Player is represented as an actor and game systems as actions. In the diagram, the

Player moves to dodge incoming threats. The diagram outlines what the player can do within the game without specifying how these actions are implemented.

4.2 Core classes

To describe the structure of the game, and how the 'Player', 'Asteroid', 'Enemy' and 'Bullet' should look like, i used a UML class diagram:

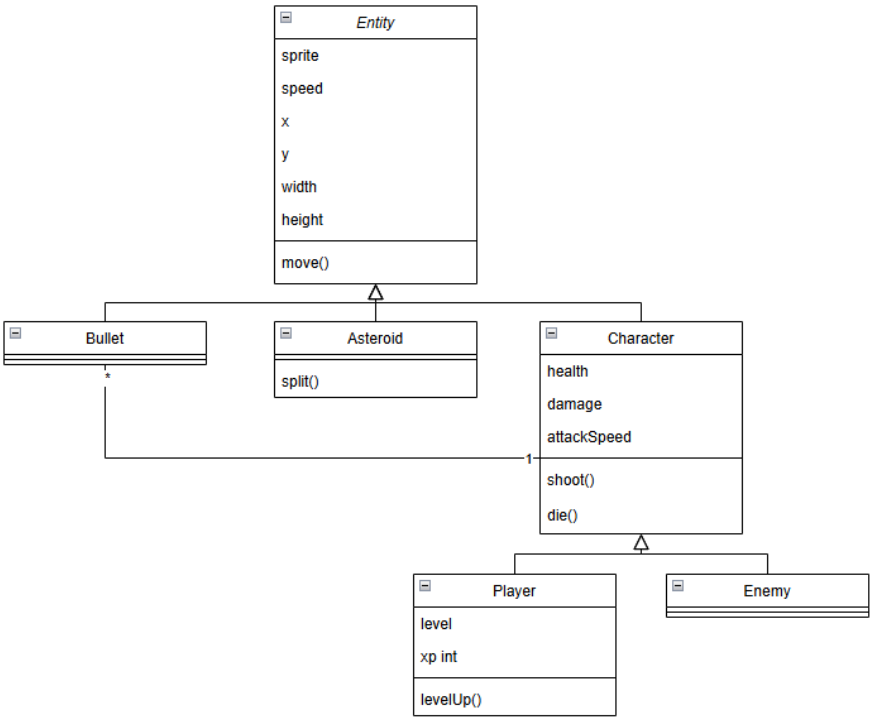

Figure 2: *UML class diagram of core classes*

This class diagram shows the most important classes that are fundamental to the game. Looking at the top is the abstract Entity class, which is a common parent of all game-objects. It centralizes shared characteristics like position (x, y), dimensions (width, height), and movement (move()).

From Entity, three specialized classes inherit: Bullet, Asteroid, and Character. The Bullet class is what is used by characters to shoot. The Asteroid class are the space rocks the Player has to shoot or dodge. When shot it will fragment if big enough using split(). The Character class describes 'intelligent' objects like the player's ship or enemies. Characters are unique as they have the ability to shoot() Bullets. Characters also have a die() method to handle their removal from play and trigger death animations.

4.3 Player firing bullet

From pressing the mouse, to seeing a bullet flying from the player, takes quite a few steps, especially when the player and bullet cannot have a dependencies on each other, as they would not fulfill the Must-have requirement **F6**. This sequence diagram shows the sequence that should happen for a player to fire a bullet to avoid this dependency issue:

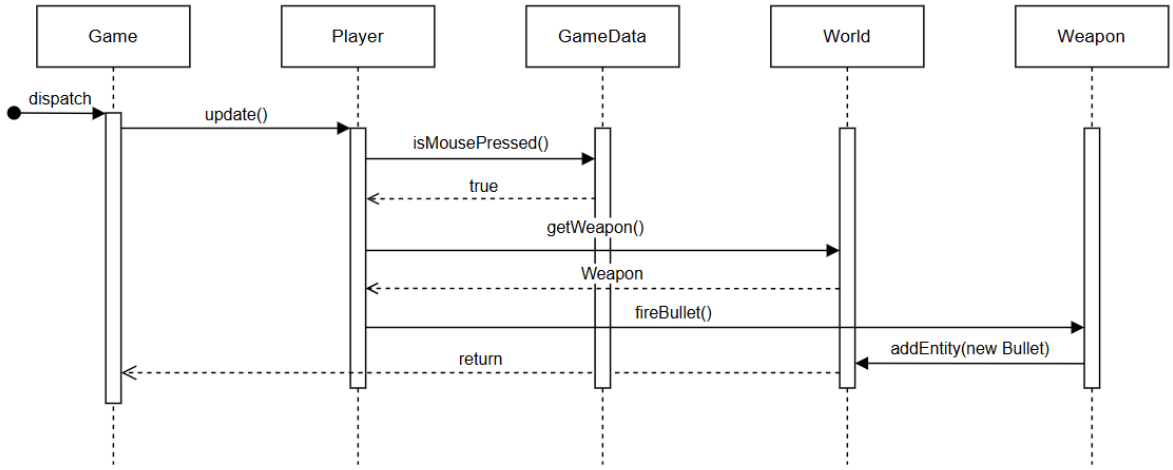

Figure 3: *Sequence diagram of player firing a bullet*

When a player fires a bullet in the game, it starts with the Game object initiating an update() call to the Player object. The Player then checks with the GameData object if the mouse input for firing is currently pressed. In this case when returning true, the Player calls the world object for the reference to the weapon object which has the ability to instantiate Bullets. With this approach dependencies between the Player and Weapon is avoided, as the Player interacts with the Weapon component through an abstraction implemented in the World class.

4.4 Spring & ServiceLoaders

For controlling components, accessibility and dependencies, different solutions vary in terms of modularity, flexibility, testability, and runtime performance. Different proposed solutions for this course are Spring and ServiceLoaders.

Spring

Spring is a popular framework in Java that can be used to handle dependency injection. Spring's context management system can be used to define, initialize and inject components or beans in a centralized, controlled and configurable way.

ServiceLoaders
Unlike Spring, ServiceLoaders requires no third-party dependencies as it's a part of the java.util package. ServiceLoaders works by discovering and instantiating implementations of a given interface or abstract class that are provided in the module-info file. This way, implementations of our services can dynamically be discovered and instantiated without requiring them in the module-info.

With both of these approaches, dependencies are reduced if implemented correctly, making it possible to remove components without recompiling. This is also one of the great benefits of microservices as you can create completely independent, self-contained modules.

5 Design

This chapter details how abstraction and contracts can be used to make the core modules self contained. As an example if the player wants to fire a bullet, it does not need to know how a bullet is fired, it just needs to know enough to fire the bullet. This can be done through contracts. If we have an interface IWeapon which has a fire() method. If the Player can get a reference to the Weapon as shown above in the sequence diagram [3], and the Weapon implements IWeapon, the components can interact with each other at a high level, without needing to understand any of the logic that makes it work. It is very similar to the Facade programming pattern, where you hide away complex logic in a class's methods and can use them without needing to understand how they work.

5.1 Components

To understand what the components and their abstractions should look like, the components are detailed in a component diagram:

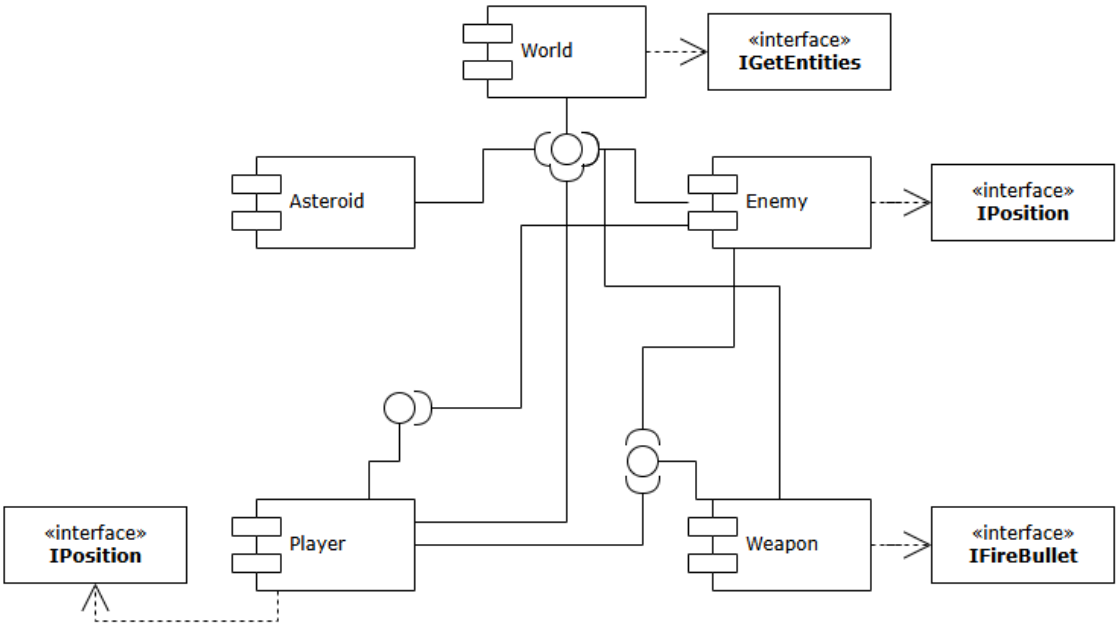

Figure 4: *Component diagram of core modules*

This component diagram shows the game's core components, which are the main parts of the game. Components display a "lollipop" (a circle connected by a line) this represents the component providing the specific service or functionality described by its referencing interface.

- **World:** World provides its functionalities described in IGetEntities, which should have a function that returns a list of all entities.
- **Player:** Needs getEntities() in its logic, to know if it is being hit by an enemy Bullet- or colliding with an Asteroid-Entity. It also uses the weapon to fire bullets.
- **Enemy:** Needs a Player reference to get the Players position when firing a bullet with the Weapon. Needs getEntities() to know when it's shot by the player.
- **Asteroid:** Needs getEntities() to know when it is shot and should split.
- **Weapon:** Needs getEntities() to add bullets to it (assuming getEntities() returns a reference to a list of entities `List<Entity>`)

5.2 Contracts

To clearly specify what these contracts should look like, and their pre- and post-conditions, the ones used in the component diagram above [4] are described here:

5.2.1 IPosition

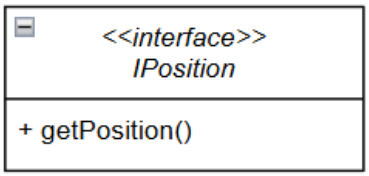

Figure 5: *IPosition interface UML class diagram*

This contract/abstraction is what allows other objects to get an Entity's position, which should be used by the Enemy and Player to get each others position.

Pre-conditions

- There must be an active entity in the game that implements this interface.
- The Entity must have a defined position it can return.

Post-conditions

- Entity's position is returned, including the Entity's X- and Y-Position.
- Entity's position remains the same, and is not affected by this method call.

5.2.2 IFireBullet

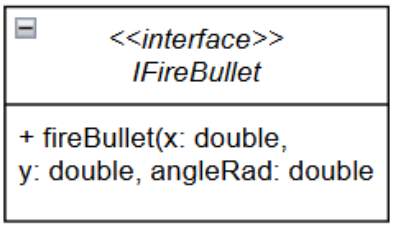

Figure 6: *IFireBullet interface UML class diagram*

This is used by the Enemy and Player to fire Bullets.

Pre-conditions

- There must be an implementation of a Bullet.

Post-conditions

- A Bullet is instantiated and appended to the list of Entities in the World object.
- The Bullet is instantiated in the correct position and flying in the given direction.

5.2.3 IGetEntities

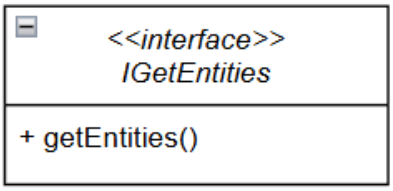

Figure 7: *IGetEntities interface UML class diagram*

This is used by all other core components to figure out if they are colliding with certain Entities. The Player could be colliding with an Asteroid. An Enemy could be colliding with a Player-Bullet and vice-versa.

Pre-conditions

- Variable is initialized as an empty list to avoid null reference errors if empty.

Post-conditions

- The list returned, contains all entities in the game.

6 Implementation

The game implements the design described in the previous chapter, but some logic and minor details has changed such as the World object returns a list of entities that matches the type provided in the argument, instead of returning a reference to the list. This decision was made so I don't have to sort through it inside the update logic, for each Entity's update logic.

6.1 Core components

Here i show and describe all the core components shown in the design-chapter [4].

6.1.1 World

The World class acts as a container for storing and accessing important game-related objects such as Entities, Player, Spawners and Weapon. This lets all other modules access them, as they are placed in the 'Common' module.

```
81      public ArrayList<Entity> getEntities(EntityType type) {
82          ArrayList<Entity> e = new ArrayList<>();
83          for (IGameObject gameObject : gameObjects) {
84              if (gameObject instanceof Entity entity) {
85                  if (entity.getType() == type) e.add(entity);
86              }
87          }
88          return e;
89      }
90
91      public boolean isColliding(Entity a, Entity b) {
92          return a.getColliderX() < b.getColliderX() + b.getColliderWidth() &&
93                  a.getColliderX() + a.getColliderWidth() > b.getColliderX() &&
94                  a.getColliderY() < b.getColliderY() + b.getColliderHeight() &&
95                  a.getColliderY() + a.getColliderHeight() > b.getColliderY();
96      }
97
98      public IPlayer getPlayer() {
99          return this.player;
100     }
101
102     public IWeapon getWeapon() {
103         return this.weapon;
104     }
105 }
```

Figure 8: *Snippet of the "World"-class*

For the getEntities() method, an enum was used to choose the type of entities you want the method to return. This way, if I want to check if a Player collides with an Asteroid, I can easily get all Asteroids, as all Entities has an EntityType. I did not use collision detection as described in requirement F10 [1]. This was done because I am used to using rectangular colliders in game-engines such as Unity, and I thought it would be fun to see if I could describe the logic for this kind of colliders in Java. With getter-methods such as getPlayer() and getWeapon(), all IGameObjects can in theory fire bullets if implemented in it's update method. This might not be ideal as you don't want to end up accidentally firing bullets from your Asteroid, but this was the easiest solution i could think of

(beside ServiceLoaders).

6.1.2 Player

In the asset-packs that were used for this game, there are different drawing of the Player ship (different conditions), so I put them in a map. This way i can access the different ships as needed:

```
32          // Ships
33          shipMap.put(PlayerCondition.PERFECT, new Sprite ("/player/pl100.png", this
                .width, this.height, this.scale));
34          shipMap.put(PlayerCondition.GOOD, new Sprite ("/player/pl75.png", this.
                width, this.height, this.scale));
35          shipMap.put(PlayerCondition.DAMAGED, new Sprite ("/player/pl50.png", this.
                width, this.height, this.scale));
36          shipMap.put(PlayerCondition.SHIT, new Sprite ("/player/pl25.png", this.
                width, this.height, this.scale));
```

Figure 9: *Snippet of the "Player"-class showing shipMap*

The Sprite class uses a Facade pattern that hides away the logic for loading a sprite-sheet, chopping it up and getting a list of sprites, converting them to a JavaFX-image, controlling animations, rotating images and scaling images. When drawing the Player, the getPlayerImg() method, is used and returns the Player image, depending on the amount of health the Player has left:

```
90          // draw ship
91          double[] pos = getCenterImagePos(this.width, this.height);
92          gc.drawImage(getPlayerImg(), pos[0] , pos[1], this.width*scale, this.
                height*scale);
```

Figure 10: *Snippet of the "Player"-class showing ship drawing logic*

For updating the player position, handling userIO and checking collision, I used an Update method that looks like this:

```
43      @Override
44      public void update(GameData gameData, World world) {
45          if (isDead) return;
46
47          this.angle = calcAngle(gameData.getMousePosX(), gameData.getMousePosY());
48          move(gameData, world);
49
50          // Firing logic
51          if (gameData.getMousePressed() && isLoaded()) {
52              fire(world, gameData);
53          }
54
55          // Enemy bullet collision
56          for (Entity bullet : world.getEntities(EntityType.ENEMYBULLET)) {
57              if (world.isColliding(this, bullet)) {
58                  onHit(bullet, world);
59              }
60          }
61      }
```

Figure 11: *Snippet of the "Player"-class showing update method*

First the angle to the mouse position is calculated. It is used in drawing the Player and the angle of the Bullet when firing. The Player fires when the mouse is pressed, and time between last fire is high enough (depends on fire-rate of the Player). Collision is checked for Bullets with type Enemy-Bullet, and when collision is detected, the onHit method is called causing the player to take damage, and removes the Bullet.

6.1.3 Asteroid

What is unique about the Asteroid is that it needs to split as specified in requirement F11 [1]. When it detects that it has been shot it, just like the Player, calls the onHit method. Here the Asteroid splits into two new Asteroids, if its size is above the minSize variable:

```
43      private void onHit(Asteroid hitAsteroid, Entity bullet, World world) {
44          world.removeGameObject((IGameObject) bullet);
45          if (hitAsteroid.size >= minSize) {
46              split(world);
47              world.removeGameObject(hitAsteroid);
48          } else {
49              triggerDeathAnimation();
50          }
51      }
52
53      private void split(World world) {
54          // Spawn 2 new asteroids in half size
55          Asteroid child1 = new Asteroid(this.x, this.y, this.angleRadians + Math.PI
                /8, (int) (this.size/2), this.speed, this.scale, this.player);
56          Asteroid child2 = new Asteroid(this.x, this.y, this.angleRadians - Math.PI
                /8, (int) (this.size/2), this.speed, this.scale, this.player);
57
58          world.addGameObject(child1);
59          world.addGameObject(child2);
60      }
```

Figure 12: *Snippet of the "Asteroid"-class showing splitting logic*

The Asteroid will only ever trigger its onDeath-animation if colliding with a player or it cannot split anymore. This clearly signals to the Player that the Asteroid is destroyed, and it won't split anymore.

6.1.4 Weapon

Weapon is loaded using IPluginService where it assigns itself to the world with setWeapon(new Weapon()). The Weapon implements IWeapon as shown in the Design chapter [6], and allows the Enemy/Player to fire bullets, by instantiating a new Bullet and adding it to the list of GameObjects:

```
43      public class Weapon implements IWeapon {
44
45      @Override
46      public void fire(World world, double x, double y, double speed, double
            angleRad, int damage, BulletType type, EntityType entityType, double scale
            ) {
47          Bullet bullet =  new Bullet(x,y,speed,angleRad, damage, type, entityType,
                scale);
48          world.addGameObject(bullet);
49      }
50  }
```

Figure 13: *Snippet of the "Weapon"-class showing fire() method*

This was a cheap way to avoid the dependency between Player/Enemy, but I don't think this is a very "clean" solution, and it would have been better to use a ServiceLoader to find the Bullet implementation, and instantiate it inside of the Player/Enemy.

6.1.5 Enemy

As I decided to implement more than one type of Enemy (shooter and melee), I also needed to implement spawn logic for when they should spawn, and where they should spawn. This is all handled inside the EnemySpawner-class.

Each type of Enemy extends the abstract class 'Enemy'. The 'Melee' is a simple enemy that "seeks" the player and damages him when they collide. The 'Shooter' is a type of enemy that randomly moves around the screen, making it harder to it. It also dies with one shot, but it deals a lot of damage to the Player, so dodging this is critical. It was inspired by a noisy annoying fly - it flies around and "annoys" the Player by trying to kill him. Enemies cannot move through each other as I implemented a separationforce that separates enemies if they get too close (force is applied in the move() method):

```
43      protected double[] calcSeperationForce(World world) {
44          // Separation force to prevent enemies from stacking
45          double separationX = 0;
46          double separationY = 0;
47          double minDistance = this.width-10;
48
49          for (Entity other : world.getEntities(EntityType.ENEMY)) {
50              if (other == this) continue;
51
52              double dx = this.x - other.getX();
53              double dy = this.y - other.getY();
54              double distance = Math.sqrt(dx * dx + dy * dy);
55
56              if (distance < minDistance && distance > 0) {
57                  // If an enemy gets too close we apply the force
58                  separationX += dx / distance;
59                  separationY += dy / distance;
60              }
61          }
62          return new double[] {separationX, separationY};
63      }
```

Figure 14: *Snippet of the "Enemy"-class showing separationforce*

6.2 Accessing & registering components

All components' compiled Jar-files and Artifacts from dependencies, are copied into a folder named mods-mvn, as shown in the example-project [2]. This is where the ServiceLoader looks for implementations and where the project gets run from using the command 'mvn exec:exec'. When the project starts, the JavaFX application is first configured, and then the start() method is run on all found implementations of the IPluginService, using java.util ServiceLoader:

```
43          // Start IPluginService implementations
44          pluginServiceList().forEach(plugin -> plugin.start(world));
```

Figure 15: *Snippet of the "Game"-class*

6.3 Labs

In this section i explain how i implemented the labs throughout the course.

6.3.1 GameLab

In the GameLab I had to implement F2-, F4-, F7-, F8-, F9-, F10-, F11- and the F12-requiremnt [1]. I noticed in the Asteroids-Example [2], that Entities were updated using pre- and post-processing services, but drawing was done by by getting all enemies from world, so i thought: "Why use Services for processing/updating, but not for drawing?". So in my implementation I used an update method defined in IGameObject to update my GameObjects, as this seemed more logical to me:

```
43          // Update and draw all gameobjects
44          for (IGameObject gameObject : world.getGameObjects()) {
45              gameObject.update(gameData, world);
46              gameObject.draw(gameData, gc, world);
47          }
```

Figure 16: *Snippet of rendering logic*

The implementation of the components are roughly described above [6.1], but the source code can be viewed on my GitHub [B] from the 'main' branch.

6.3.2 JPMS

In the JPMS-lab the task was to implement requirements: F13, F14 [1]. I was already using module-info.java, so i kept working on core game mechanics. In JPMS-lab3 we are taught about how JPMS handles split-packages (different modules with the same package names), and how to resolve related

errors. The easiest option is to just rename one of the packages, or to merge them if possible. Another option is to use a JPMS Module Layer. I tried to follow the example, but after hours of recieving the same error over and over, I resigned in order to remain sane.

6.4 SpringLab

In the SpringLab, the Spring framework is used to handle dependency injection, and to initialilize components/beans. The full implementation can be seen in the repository [B] in the branch 'Spring'. Since only one ServiceLoader was used for this project my 'ModuleConfig' is very simple:

```
43 @Configuration
44 class ModuleConfig {
45
46     public ModuleConfig() {
47     }
48
49     @Bean
50     public Game game() {
51         return new Game(pluginServiceList());
52     }
53
54     @Bean
55     public List<IPluginService> pluginServiceList(){
56         return ServiceLoader.load(IPluginService.class).stream().map(ServiceLoader
               .Provider::get).collect(toList());
57     }
58 }
```

Figure 17: *Snippet of Spring implementation*

The 'AnnotationConfigApplicationContext' is then loaded in the 'Main' class using the 'ModuleConfig' to extract my 'Beans', and instantiate the 'Game' object.

6.4.1 TestLab

In this lab i wrote tests using JUnit and Mockito to test the isColliding() method located in the 'World' class. I Mocked the 'Entity' class that are passed as arguments in the isColliding() method. This way I can isolate the logic that is being tested without having to worry about the Entity causing an error or returning faulty values. The full test is shown in appendices [18]

6.4.2 MicroServiceLab

Instead of enforcing strong encapsulation with contracts/abstraction and ServiceLoaders, this lab focuses on using microservices. I created a rest-api like the one shown in the example project [2]. To

keep track of the score, I created a 'ScoringClient' with methods to add points to a score, and one to update a JavaFX Label placed in the upper left corner of the window. The 'ScoringClient' can be seen in appendices [19]. The full implementation can be seen in the branch 'Microservices'.

7 Test

For developing the game a lot of manual testing was used. This is because the feel of the game, an animation or player-movement is very difficult to describe/define with Unit-tests. When making a game the best way to test if you have created the desired outcome, is to play it. However if something can be automated with tests such as collision detection, it should, as these tests are cheap, as they require no user and usually run very fast.

When testing and debugging the collision detection, I created a boolean variable 'isTesting' (Game-class) that if true will draw all Entities' colliders, making it easy to see if a bullet gets removed correctly the exact moment they collide. This was the same approach I took with testing animations: Try something -> see if it works -> repeat (until it works). This is generally the approach I take with most development including most of this game.

For testing the removal of components, compiled Jar-files was also very simple. If a variables from a component that never got loaded or had no implementation is being still being accessed (Player tries shooting with a non-existing Weapon), the CMD does a very good job of letting you know, especially when the game loop runs 144 times a second. These issues were fixed by checking edge-cases. For example making sure that the Player is not null before trying to access it in the Enemy class.

8 Discussion

The game solves the majority of the requirements I specified [1]. However I know I could have fulfilled a lot more Must-have requirements, if I had focused more on implementing what was asked of me instead of what I wanted to do myself. An issue I encountered was prioritizing my own vision of what I thought the game should look like, and what the demands were. What was asked of me, being the assignments in the Labs, and my own vision being the ideas I had for the game, like visuals and unique enemy types. Despite me focusing on implementing what I thought would be fun for the project I still implemented the labs (not all to full extend).

As the requirements are clearly defined it becomes easy to run the game or take a look at my code, to see if a requirement is fulfilled. Ideally I would have a real user in a realistic setting, and perform a User Acceptance Test with high-level definitions of how tests should be carried out, to determine if a requirement is fulfilled. When carrying out the tests myself, I also try to adopt a user mindset and forget that I am a developer, providing a more realistic result. I went over each requirement and

marked it as green for fully implemented, yellow for partially implemented and red for not implemented:

MoSCoW: Functional requirements		
ID	**Description**	**Priority**
F1	A 'Player' component	**Must**
F2	A 'Enemy' component	**Must**
F3	A 'Bullet'/'Weapon' component	**Must**
F4	A 'Asteroid' component	**Must**
F5	Rendering and drawing (core) components	**Must**
F6	'Player', 'Enemy' and 'Weapon' components implement service provided interfaces that allow the components to be updated and removed without recompilation.	**Must**
F7	Implement provided interfaces: IGamePluginService, IEntityProcessorService and IPostEntityProcessorService	**Must**
F8	Document service interfaces using Java doc	**Must**
F9	Asteroids move randomly	**Must**
F10	Implement a simple collision detection system based on Pythagoras and the provided IPostEntityProcessorService interface	**Must**
F11	When fired upon Asteroids should split into two smaller Asteroids and when small enough they should be destroyed	**Must**
F12	The player ship and enemy-ships should be destroyed when hit by each others bullets a certain number of times	**Must**
F13	Implement game using Java Platform Module System	**Must**
F14	Declare imports and exports in module-info.java files for each module	**Must**
F15	Write a unit test for one of the components	**Must**
F16	Implement the Core component from the Asteroids game using the Spring container and the Dependency Injection Component Model.	**Must**
F17	Instantiate the Game class and use Spring for Dependency injection of the IEntityProcessors and IGamePluginServices.	**Must**
F18	Integrate the Scoring MicroService in the AsteroidsGame using the Spring RestTemplate	**Must**
F19	Implement sprites for visuals	**Should**
F20	Create different types of enemies	**Should**
F21	Objects drawn out of screen are removed	**Should**
F22	Implement sprites for visuals	**Should**
F23	Use sprite animations for certain events like dying and shooting	**Should**
F24	Difficulty scales with time played	**Could**
F25	Player levels up after shooting enough enemies, and can choose one of three upgrades to his ship	**Could**
F26	Create random pickups dropped randomly by enemies	**Could**
F27	Use delta-time for smooth animations and movement	**Could**

Table 2: *Table of fully/partailly/not implemented functional requirements.*

I count F7 as partially implemented, as I use the IGamePluginService for starting all implementations, but I used a different solution for updating/processing. F7 is counted as partially implemented as I did implement a collision detection system, but I did not use Pythagoras for the implementation, but instead used Axis-Aligned Bounding Box (AABB) collision detection. F17 is counted as partial for the same reasons as for F7, I only have one of the three interfaces implemented in my game. As you can see I have all my should-have requirements implemented, which should not be the case as must-have's should have been given higher priority. The could-have requirements were other fun ideas I would have loved to do, but i did not find the time to finish.

9 Conclusion

This report set out to address my implementation of a Asteroids game made in JavaFX, and developing software that enforces strong encapsulation. In doing so, I demonstrate how ServiceLoders and contracts/abstractions provides a robust and flexible solution. Through analysis of the game using structural and behavioral diagrams, I developed a game that makes it possible to remove/change components without recompilation. In my analysis and design I focused on creating clear abstractions and defining component contracts, which were then brought to life through implementation and validation using manual testing and Unit-tests. The system's architectural choices has a big impact on modularity and dependencies, highlighting the importance of architectural decisions as they can become harder to change/refactor as the code-base grows. Looking ahead, future work should prioritize weighing the pros and cons of major design decisions, which would help prevent regret down the line.

Appendices

A Youtube demo

Demo video

B Project repository

Project repository

C Test

```
0  package com.asteroids.common.data;
1
2  import com.asteroids.common.gameObjects.Entity;
3  import org.junit.jupiter.api.*;
4  import org.mockito.Mockito;
5
6  import static org.junit.jupiter.api.Assertions.*;
7  import static org.mockito.Mockito.when;
8
9  class WorldTest {
10
11     private World world = new World(1000, 1000);
12
13     // Create and return a mocked Entity
14     private Entity mockEntity(double x, double y, double width, double height) {
15         Entity entity = Mockito.mock(Entity.class);
16         when(entity.getColliderX()).thenReturn(x);
17         when(entity.getColliderY()).thenReturn(y);
18         when(entity.getColliderWidth()).thenReturn(width);
19         when(entity.getColliderHeight()).thenReturn(height);
20         return entity;
21     }
22
23     @Test
24     @DisplayName("Should detect collision when entities overlap fully")
25     void testIsColliding_fullOverlap() {
26         Entity entityA = mockEntity(0, 0, 10, 10);
27         Entity entityB = mockEntity(5, 5, 10, 10);
28         assertTrue(world.isColliding(entityA, entityB), "Entities should be
               colliding with full overlap");
29     }
30
31     @Test
32     @DisplayName("Should detect collision when entities touch on an edge (minimal
           overlap)")
33     void testIsColliding_minimalOverlap() {
34         Entity entityA = mockEntity(0, 0, 10, 10);
35         Entity entityB = mockEntity(9, 9, 2, 2); // Overlaps with 1 pixel in X and
                Y
36         assertTrue(world.isColliding(entityA, entityB), "Entities should be
               colliding with minimal overlap");
37     }
38
39     @Test
40     @DisplayName("Should detect collision when one entity is contained within
           another")
41     void testIsColliding_contained() {
42         Entity entityA = mockEntity(0, 0, 20, 20);
43         Entity entityB = mockEntity(5, 5, 5, 5);
44         assertTrue(world.isColliding(entityA, entityB), "Entity B should be
               colliding as it's inside A");
45     }
46
47     @Test
48     @DisplayName("Should not detect collision when entities are separated
           horizontally")
49     void testIsColliding_separatedX() {
50         Entity entityA = mockEntity(0, 0, 10, 10);
51         Entity entityB = mockEntity(11, 0, 10, 10); // Separated by 1 pixel
               horizontally
52         assertFalse(world.isColliding(entityA, entityB), "Entities should not be
```

D ScoringClient

```java
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import javafx.scene.control.Label;

public class ScoringClient {
    private HttpClient client = HttpClient.newHttpClient();
    private Label scoreLabel;

    public ScoringClient(Label scoreLabel) {
        this.scoreLabel = scoreLabel;
    }

    public void addScore(int amount) {
        String apiUrl = "http://localhost:8080/score?point=" + amount;

        try {
            // Build request
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(URI.create(apiUrl))
                    .GET()
                    .build();

            // Send request
            HttpResponse<String> response = client.send(request, HttpResponse.
                BodyHandlers.ofString());

            // Handle Response
            scoreLabel.setText("Score: " + response.body());
            return;
        } catch (Exception e) {
            System.err.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
        }
        scoreLabel.setText("Score: " + -1);
    }

    public static void main(String[] args) {
        ScoringClient scoringClient = new ScoringClient(null);

        scoringClient.addScore(10);
    }
}
```

Figure 19: *Snippet of 'ScoringClient'-class*

10 Bibliography

References

[1] Foozle. *Asset-Packs*. URL: (https://foozlecc.itch.io/).

[2] sweat-tek. *Asteroids example*. URL: (https://github.com/sweat-tek/AsteroidsFX).