Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Component baseret systemer

Student Name: **Mads Wrang Sigsgaard**

Student Exam number: **4118544**

GitHub User: **DenFlyvendeGris**

Youtube Video: **https://youtu.be/UGUmsBgANQs**

GitHub Repo: **https://github.com/DenFlyvendeGris/AsteroidsFX**

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Abstract

The problem which is represented in this report is about the modularity structure of the asteroids game and how we can utilize component-based concepts to create an easily changeable and modular structure. The way we want to achieve this is by successfully using some of the SOLID principles and specifically open/closed principle and Singular responsibility. We also use testing to understand our architectural foundation in our code which can reembrace the modularity of our project. In the end we concluded that using more of the SOLID principles could be of use to enhance modularity and easily changeable project, as well as creating more in-depth tests to cover a larger code functionality. However, our project has created a foundation for modularity and easily changeable code with our implemented Component based principles which covers our requirements and creates possibilities for extensions in the future.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Indholdsfortegnelse

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Introduction

The asteroid game is a well know game that was created back in 1979 and had a great impact on the software industry as it had simple yet challenging gameplay. This arcade game has since been recreated a lot of times with different spins on the gameplay and visuals. Apart from gameplay and visuals, it has also been created with different software structures. The simplicity of the game makes it a perfect practice to try different software structures like component-based software engineering (CBSE), which is what I have tried with the asteroids game. To create a CBSE it is essential to create a project that is modular and easily changeable, in other words it's important to have it be closed to modification and open to extension. In this report I will describe how I have used two principles from the SOLID principles, which are closed to modification while open to extension and single responsibility. These two principles are the foundation of the project being modular and easily changeable and is a perfect fit for a component-based architecture. The way it impacts the game is how it can run even with some components missing, which means the game will still run even without an asteroid or enemy, or even a player. The way to go about this project is to start with requirements, so that is what we will do.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Requirements

The requirements are an important step in creating a modular asteroids game. This is how the requirements turned out throughout the process of creating the application.

Functional Requirements		
ID	Title	Description
1	Player Module	The function of a player module is to draw the player on the world map, and give the player logic for movement
2	Enemy Module	The function of the enemy module is to spawn multiple enemies throughout the game runtime, and give them movement like the player, but they need some sort of movement pattern, and that is created by randomizing their movement.
3	Asteroid Module	The asteroid module is there to create asteroids and give them a state which is Large, Medium, or small. This is used in collision module to either split an asteroid or delete it.
4	Bullet Module	The bullet module is creating a bullet and removes them when they are outside of the height and width of the screen. Both the player and enemy use the bullet module
5	Common Module	The common module starts the game and draws entities on the world from the startup point, it is also the module that has all the entities and has the logic for deletion from the world.
6	Collision Module	The collision module is responsible of the collision logic. The entity class has onCollision logic that is used in Player, Enemy and Asteroid to respond in different ways depending on what entities collide with each other

Non-Functional Requirements		
ID	Title	Description
1	adding and removing of components	Enemies, player, and asteroids can be removed and added without recompilation of the program.
2	Modular	The program should be easily changeable by creating components that have its own responsibility
3	Nondependent	The program should be able to run even with components missing, like enemy, player, asteroids and so on.

4	Platform-agnostic	The program should run the same on different computers with different specs. This is done with delta time.
5	Remove/Swap Modules run time	The program should be able to handle removal or swapping of modules without having to recompile.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Analysis

In the component-based Asteroids game we have a player, enemy, asteroid, bullet, collision, common, commonBullet and Core which are all modular modules. These modules are all replaceable and extendable, they each contain their own logic with their own responsibility. By having the modules have their own responsibility we assure a modular codebase and by following the principle of open to extension, closed to modification we have a project that should be able to run even with some modules missing. The system should create the world and draw the polygons onto the map which is called the world, and these polygons are used as everything, which means enemy, player and asteroids are all made up of polygons.

The interfaces in the game contain components that has methods which are used in the classes, you may see an interface as a contract that enforces behavior. For example, in the player class we have a method called collision which is inheritance from the entity class, this method is created to control what happens when colliding and if the player is colliding with anything. To check this, we must check throughout the game's runtime, which means we call and use the method in the Collider class which uses the IPostEntityProcessingService which lets us call a method called process, and we can use this method to check whatever we want in the game's runtime.

The other interfaces we have are IGamePluginService, IEntityProcessingService and BulletSPI. The different things they do are:

BulletSPI: Created a bullet and draws them for both the player and enemy.
IGamePluginService: this interface has two methods called start and stop.
IEntityProcessingService: This interface has a method called process which gets called from the main class and iterates over all entities.

Actor: Player	Action: The player gets hit by the enemies' bullets
Description:	The player gets hit by an enemy's bullet
Preconditions:	The player is trying to shoot as many asteroids as possible to get the best high score while also avoiding the enemies' bullets
Postconditions:	The player has lost and has his score displayed.
Summery:	The player got hit by a bullet that triggered the destroyed function which deleted the player figure from screen and therefore triggered the "end" screen.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Actor: Player	Action: The player hits a newly spawned asteroid
Description:	The asteroid gets hit by a bullet which makes it split.
Preconditions:	The player is shooting after asteroids to get points, the smaller the asteroid the bigger the points.
Postconditions:	The asteroid got hit by the players bullet and split into two
Summery:	The player hit an asteroid which splits until it is too small to split, ramping up bigger and bigger points.

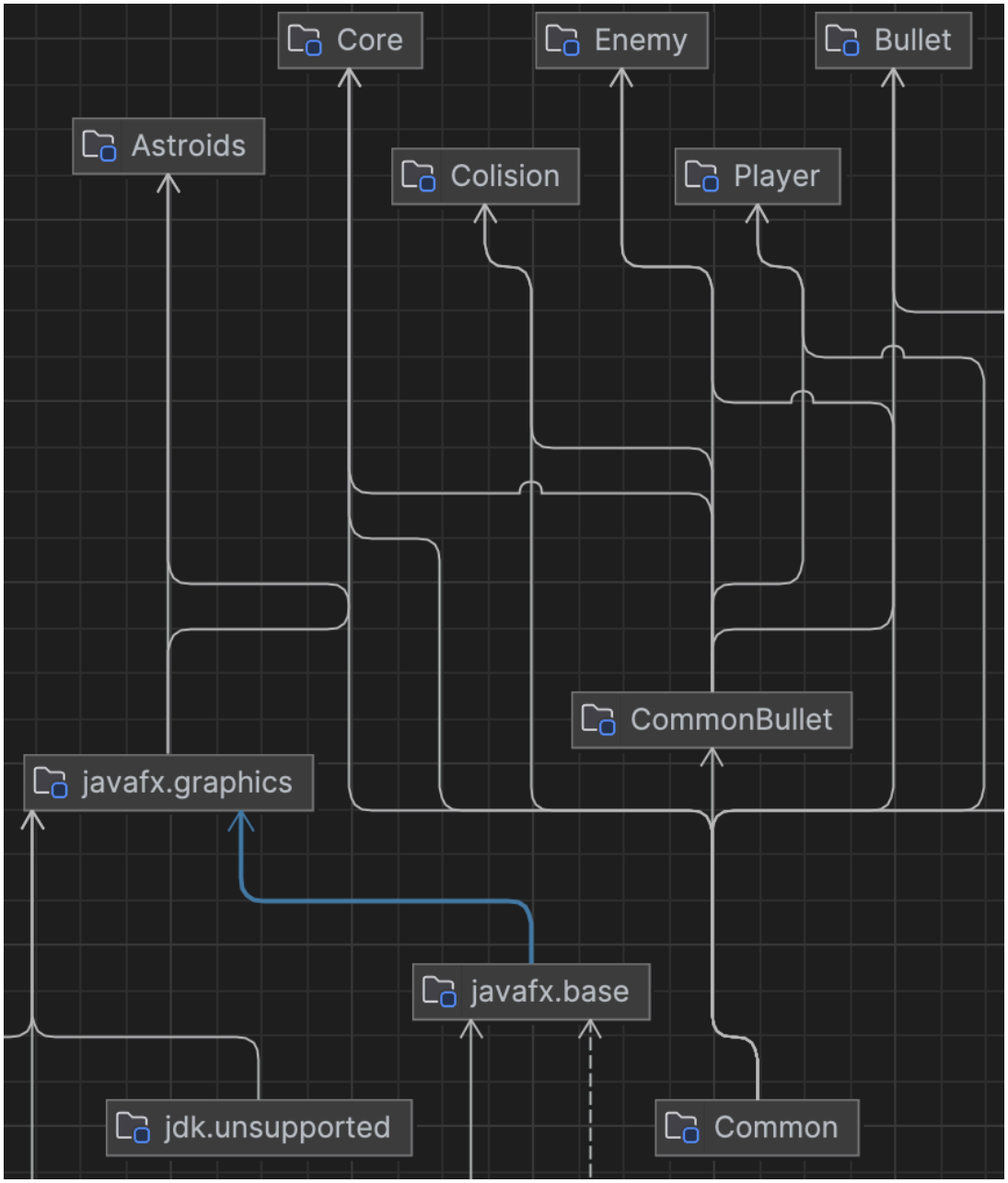
Actor: Enemy	Action: Shoots after the Player but ends up hitting the asteroid
Description:	The enemies have random movement and is just a nuisance for the player and they are trying to kill the player. One accidentally hit an asteroid
Preconditions:	The enemies are spawning randomized in random numbers and shoots in random directions.
Postconditions:	The enemy hit an asteroid which makes the asteroid split.
Summery:	Enemies' bullets do affect the asteroids splitting effect. But does not give points to the player

Actor:	Action: Asteroid hits enemy/player
Description:	The Player/Enemy have not paid attention to the asteroid and gets hit
Preconditions:	The player/enemy is alive and shooting
Postconditions:	The player/enemy got hit and got destroyed by the asteroid without it affecting the asteroid
Summery:	The asteroid can kill both player and enemy without sustaining damage

Actor:	Action: enemy moves outside screen and player tries to follow
Description:	The enemy tries to escape by running outside the screen and the player tries to follow
Preconditions:	The enemy is touching the side of the screen, and the player is right behind it

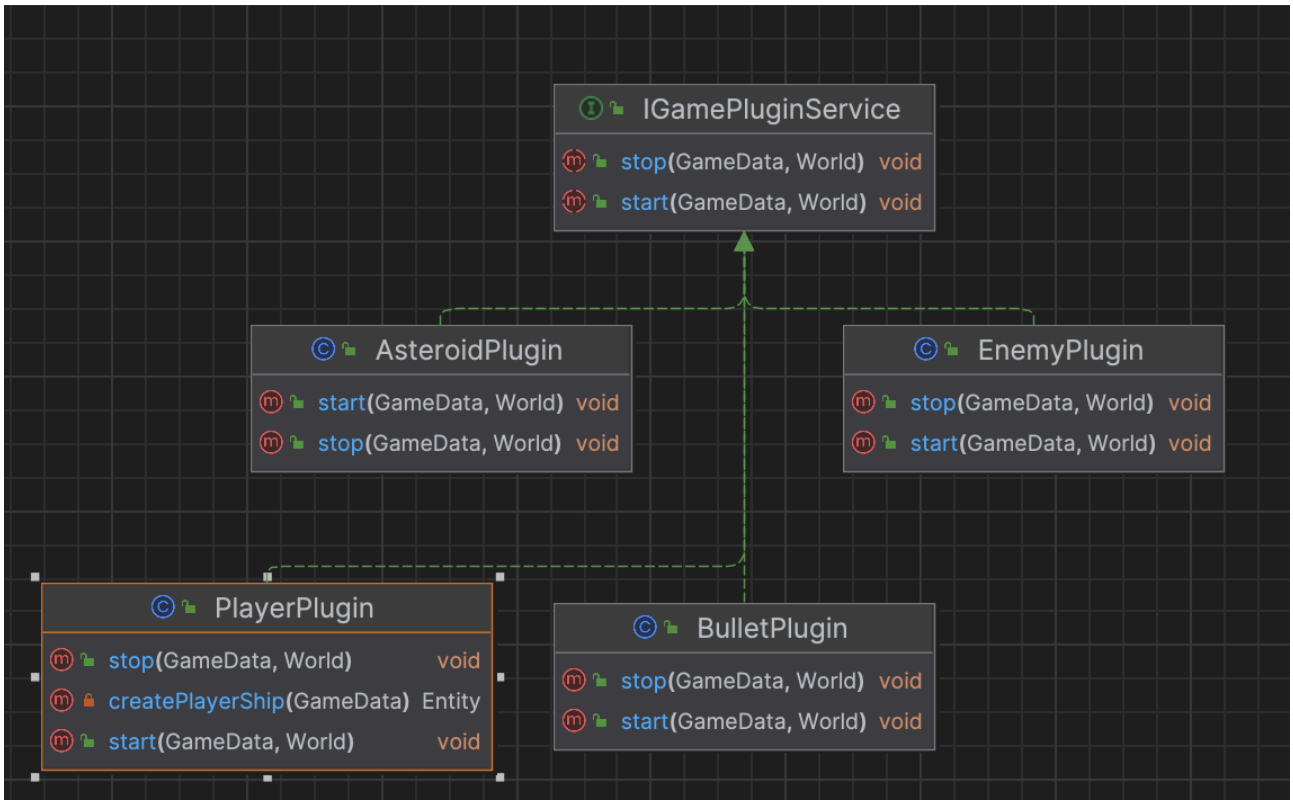
Postconditions:	The enemy gets destroyed by the update function and the player gets stopped at the last possible pixel inside the screen width and height
Summery:	The enemies can get destroyed by moving outside the screen and the player just gets stopped from moving further.

With the packages of our asteroid application, we are looking at how the different aspect all point to a common package which contains the essential data which all other packages share, here is an example:



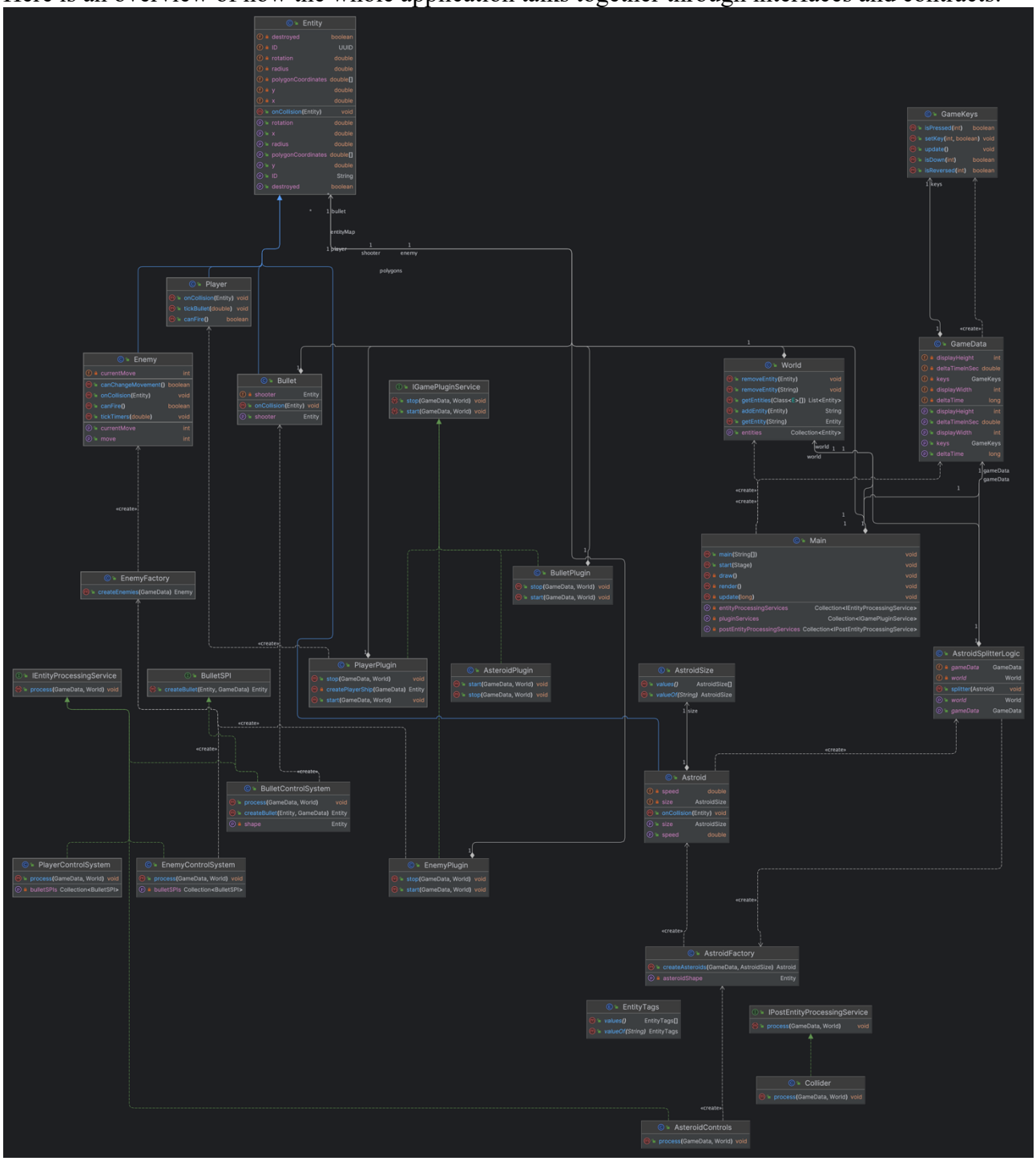
We can see that Common holds data which all packages use, the common package contains important data like, World class, entity class and GameData class. These are the essential building blocks for modularity in the system, without these we cannot draw anything in the game, even though we might be able to start it.

If we look at interfaces and how they operate instead of looking at the packages, we can see how every plugin uses the IGamePluginService. The plugins are using this interface because of the start and stop method, which they override to instantiate their respective classes. So, if we focus on EnemyPlugin on the right, we know that it overrides the start method. What then happens when the start method is called is for us to decide.



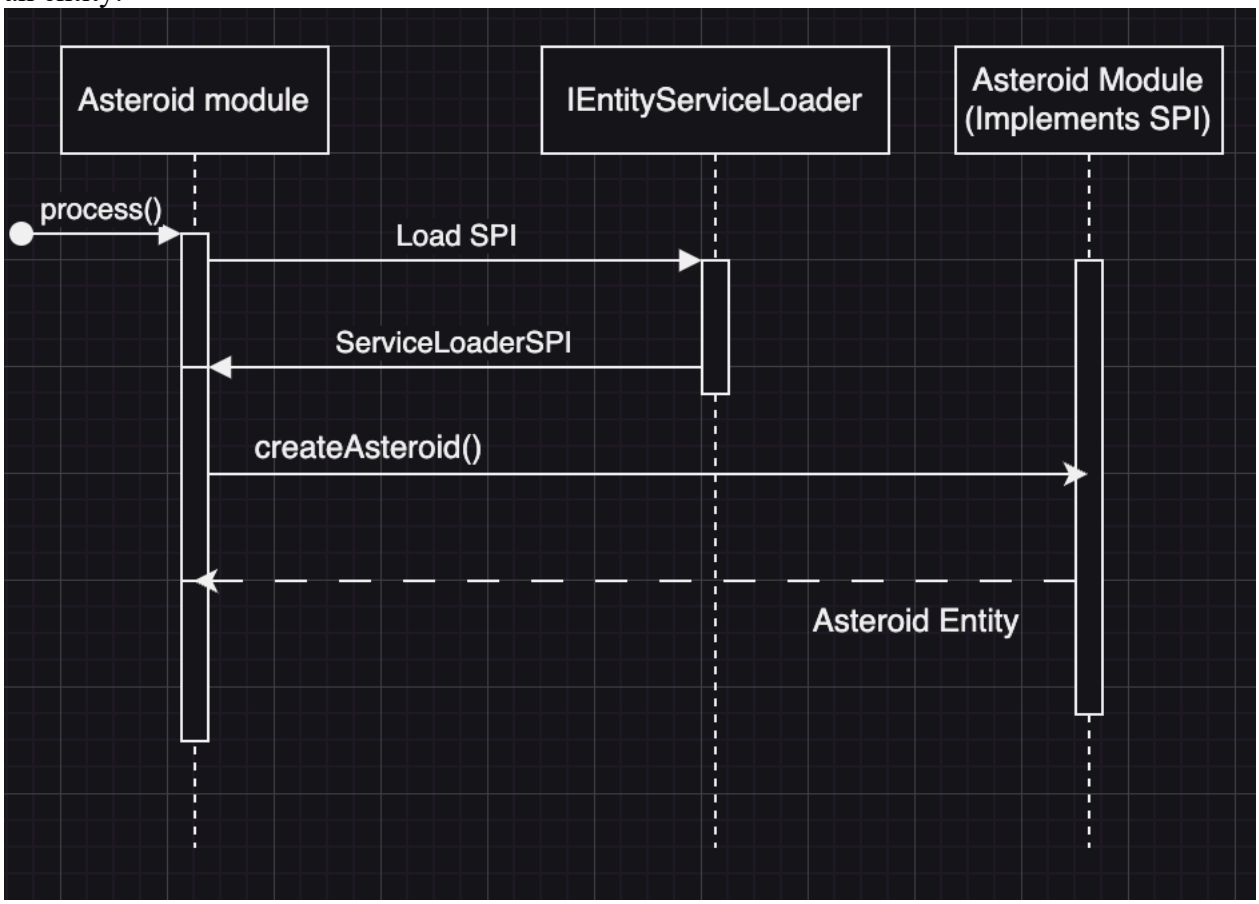
Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Here is an overview of how the whole application talks together through interfaces and contracts:



Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Here we have an example of how the modules communicate with the service loader and implements an entity.

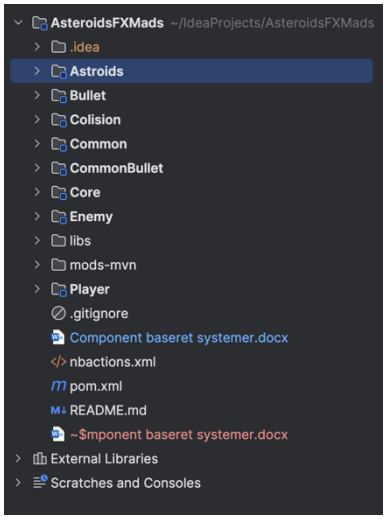


Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

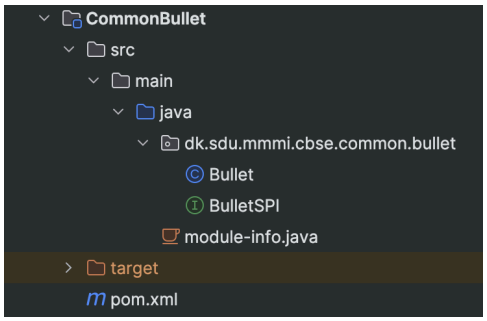
Design

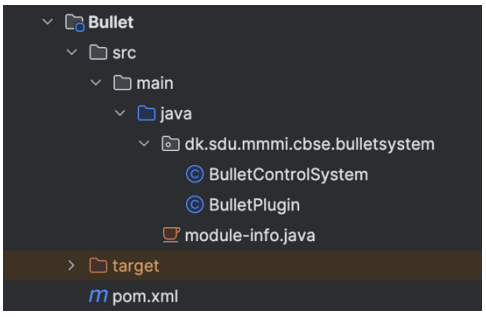
The design project structure is created so it can be as modular as possible. We have created a component-based project which has a “single responsibility principle“ and “Open to extension while closed to modification” from the SOLID principles. Using single responsibility principle means that a class should only have one job, this makes the code cleaner and more readable. Using the Open to extension and closed to modification principle is important for having a modular project, because we want to have a codebase which can be extended upon without having to alter its core implementations.

Here we have the example of the project structure, which is represented as single responsibility:



This helps us fulfill the requirement of having a modular project that can have spread out codebase which makes it easily understandable.

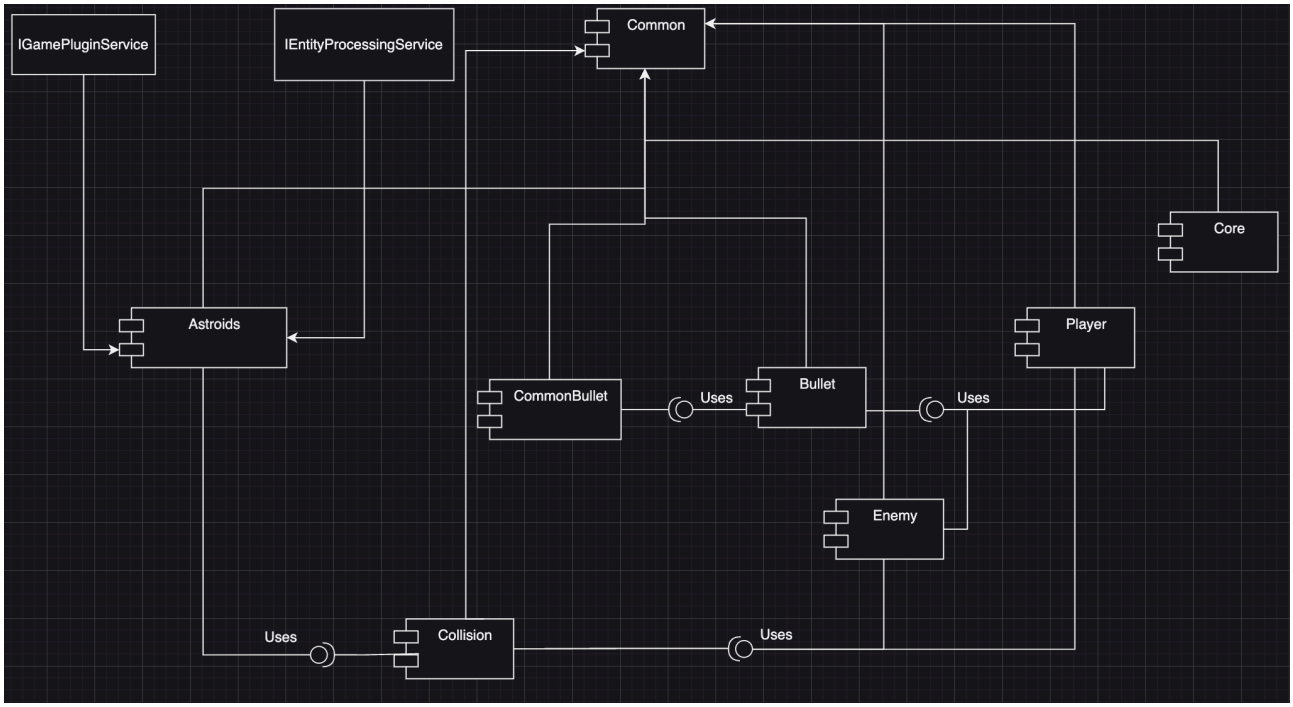




If we look inside of our commonBullet and Bullets modules we can see the single responsibility principle. A common problem in structures like this when they become big enough can be name conventions. If we have two classes or packages that has the same name, we have the problem of “split packages”, to fix this we create module layers to our service Loader and this in theory create two different layers of instantiation which means they get to be instantiated in an encapsulated environment.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

We can take another look at the components that we have in our project and how they talk with each other. It is worthy to note that if we can delete any component (except Common) and have the application still run. For simplicity's sake I will not write how the services is used in all the components, since it will create an unreadable UML diagram, so I have showed only how the services talk with Asteroids.



All the components are dependent on Common for the application to run. While everything else can be removed without having trouble running the application. We can see how the components work together to build logic in the game. As an example, we can look at how Asteroids uses Collision to collide, while the Player and Enemy also uses Collision, but the Player and Enemy components also both uses bullets and Bullet uses CommonBullet. This is an example of how every component works together in the project, while remaining modular and having a single responsibility.

We are using component contracts throughout the whole system, and these contracts lets us uphold a certain code standard so we know the code cannot run without the specified contract being fulfilled. While these contracts are being fulfilled the code will run and we have both pre and post conditions for this, and an example can be found if we look at our IEntityProcessingService interface. The conditions could look like this:

Pre-conditions:

- The gameData parameter must not be null.
- The world parameter must not be null.
- The world must contain at least one entity to process.

Post-conditions:

- The gamedata instance may be modified to reflect changes made during entity processing.
- The world state may be modified to reflect the changes that can be made during entity processing.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

The designing phase of a modular project is crucial like any other project, but in this case, it is important to have single responsibility on each class. The designing phase is crucial for how the implementation phase will proceed and we have a standard to upheld for the project to be modular.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Implementation

This chapter is regarding how the design process got implemented in the code with the standard kept high. This chapter will focus on how the service providers were implemented and used, and how that contributes to having a contract. The focus of how SPI was used will be explained through collision detection, collision handling and how asteroids react differently with collision than enemy and player.

The importance of modularity is coming from the way the serviceLoaders are implemented. As an example, we use IPostEntityProcessingService in the Collider class to use the process method which runs in the runtime:

```
Mads Wrang Sigsgaard
public class Collider implements IPostEntityProcessingService {

    1 usage  Mads Wrang Sigsgaard
    @Override
    public void process(GameData gameData, World world) {
        for (Entity entity: world.getEntities()) {
            for (Entity againstEntity: world.getEntities()) {
                if (entity.getID().equals(againstEntity.getID())) {
                    continue;
                }

                if (entity instanceof Bullet && ((Bullet) entity).getShooter().getID().equals(againstEntity.getID()))
                    continue;
                if (againstEntity instanceof Bullet && ((Bullet) againstEntity).getShooter().getID().equals(entity.getID()))
                    continue;

                double distance = Math.sqrt(Math.pow(entity.getX()-againstEntity.getX(), 2) + Math.pow(entity.getY()-againstEntity.getY(), 2));

                if (distance <= entity.getRadius() + againstEntity.getRadius()) {
                    entity.onCollision(againstEntity);
                }
            }
        }
    }
```

In this method we are checking for every entity in the world for the one colliding, and we run through the entities again to find the ones it is colliding with. If an entity gets hit by bullets it will be destroyed but that logic is not being created here, this is merely a collision detector to know what an entity has been colliding with. The “distance” is important to know when two entities are colliding and what we get is what we call Euclidean distance. Imagine a two-dimensional space where we have two entities on the plane with each an x-axis and a y-axis. To find the distance between the two entities we take the coordinates for both entities (x, y) and find the difference on their x value then squares it, we do the same with the y value. Now we have two squared values which we add them together and take the square root of the two. The result we get is the distance from the middle of the entities to one another and this is what we call the Euclidean distance.

We use this Euclidean distance to see if the radius from the entities is colliding. For this to work we have added a onCollision method to Entity because we need it on every entity in the game:

```
1 usage  4 overrides  Mads Wrang Sigsgaard
public void onCollision(Entity other) {
    System.out.println("Collision not implemented for " + getID() + "!");
}
```

We override this method in each entity to create our own logic for each, here is how we have done it in the asteroid class:

```
1 usage  Mads Wrang Sigsgaard
25      @Override
26      public void onCollision(Entity other) {
27          if (other instanceof Astroid) {
28              double x1 = getX();
29              double y1 = getY();
30
31              double x2 = other.getX();
32              double y2 = other.getY();
33
34              double deltaX = x1 - x2;
35              double deltaY = y1 - y2;
36
37              double magnitude = Math.sqrt(deltaX * deltaX + deltaY * deltaY);
38              deltaX = deltaX / magnitude;
39              deltaY = deltaY / magnitude;
40
41
42              this.setRotation(Math.toDegrees(Math.atan2(deltaY,deltaX)));
43
44              return;
45          }
46          new AstroidSplitterLogic().splitter( astroid: this);
47      }
48  }
```

If we look at the code here, we will quickly realize it is the same math we used to find the Euclidean distance, however, we use it differently this time since we use it to make two asteroids bounce off each other in different directions. It is not only different directions, but also exactly opposites of their trajectory.

The question is how the dependencies are accessed by the classes, and the answer can be found in the module-files.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

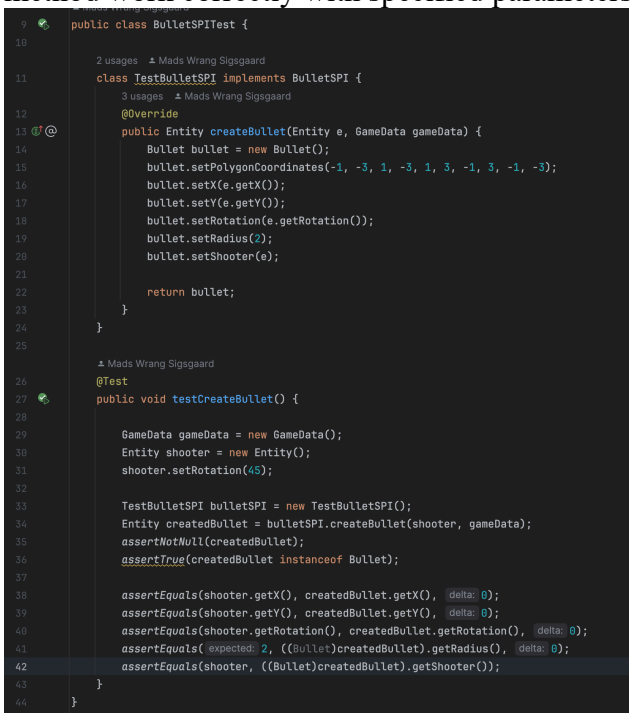
```
import dk.sdu.mmmi.cbse.astroidsystem.AsteroidControls;
import dk.sdu.mmmi.cbse.astroidsystem.AsteroidPlugin;
import dk.sdu.mmmi.cbse.common.services.IEntityProcessingService;
import dk.sdu.mmmi.cbse.common.services.IGamePluginService;

module Astroids {
    requires Common;
    requires javafx.graphics;
    exports dk.sdu.mmmi.cbse.astroidsystem;
    provides IGamePluginService with AsteroidPlugin;
    provides IEntityProcessingService with AsteroidControls;
}
```

We are enforcing our way to a reliable dependency since the module asteroids here needs to specify what it requires, and we must export our module for other modules to gain access to the data. This is a very important structure for upholding our open to extension while closed to modification principle. This is how we are enforcing a reliable dependency and a strong encapsulation in our project.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Test

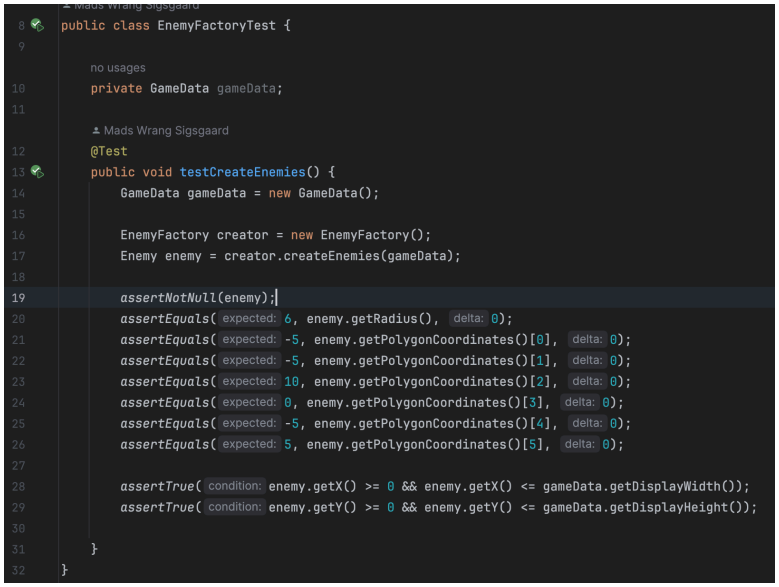
The test has been created to ensure the functionality of different entities work properly. By testing our components in isolation, we can ensure that our components are behaving accordingly, regardless of their dependencies. Testing is an important part of coding since it creates insight into the functionality of the individual components. With unit tests we have verified that the components performed as intended, for example we have created unit tests that verified that the bullet creation method work correctly with specified parameters. Here is the example:



```java
public class BulletSPITest {

    2 usages  Mads Wrang Sigsgaard
    class TestBulletSPI implements BulletSPI {
        3 usages  Mads Wrang Sigsgaard
        @Override
        public Entity createBullet(Entity e, GameData gameData) {
            Bullet bullet = new Bullet();
            bullet.setPolygonCoordinates(-1, -3, 1, -3, 1, 3, -1, 3, -1, -3);
            bullet.setX(e.getX());
            bullet.setY(e.getY());
            bullet.setRotation(e.getRotation());
            bullet.setRadius(2);
            bullet.setShooter(e);

            return bullet;
        }
    }

    Mads Wrang Sigsgaard
    @Test
    public void testCreateBullet() {

        GameData gameData = new GameData();
        Entity shooter = new Entity();
        shooter.setRotation(45);

        TestBulletSPI bulletSPI = new TestBulletSPI();
        Entity createdBullet = bulletSPI.createBullet(shooter, gameData);
        assertNotNull(createdBullet);
        assertTrue(createdBullet instanceof Bullet);

        assertEquals(shooter.getX(), createdBullet.getX(),  delta: 0);
        assertEquals(shooter.getY(), createdBullet.getY(),  delta: 0);
        assertEquals(shooter.getRotation(), createdBullet.getRotation(),  delta: 0);
        assertEquals( expected: 2, ((Bullet)createdBullet).getRadius(),  delta: 0);
        assertEquals(shooter, ((Bullet)createdBullet).getShooter());
    }
}
```

This is a simple test to check if the bullet is created and is there. A good reason to have tests such as this could be the safety net you have when having changed anything to an entity. By having a test that runs through correctly with good criteria, we can safely change our bullet implementation and then run the test again to check if we have made good changes. Just to shot another example we check here if an enemy can be created accordingly with the correct polygonCoodinates, which means the way it looks.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024



```java
public class EnemyFactoryTest {

    private GameData gameData;

    @Test
    public void testCreateEnemies() {
        GameData gameData = new GameData();

        EnemyFactory creator = new EnemyFactory();
        Enemy enemy = creator.createEnemies(gameData);

        assertNotNull(enemy);
        assertEquals( expected: 6, enemy.getRadius(),  delta: 0);
        assertEquals( expected: -5, enemy.getPolygonCoordinates()[0],  delta: 0);
        assertEquals( expected: -5, enemy.getPolygonCoordinates()[1],  delta: 0);
        assertEquals( expected: 10, enemy.getPolygonCoordinates()[2],  delta: 0);
        assertEquals( expected: 0, enemy.getPolygonCoordinates()[3],  delta: 0);
        assertEquals( expected: -5, enemy.getPolygonCoordinates()[4],  delta: 0);
        assertEquals( expected: 5, enemy.getPolygonCoordinates()[5],  delta: 0);

        assertTrue( condition: enemy.getX() >= 0 && enemy.getX() <= gameData.getDisplayWidth());
        assertTrue( condition: enemy.getY() >= 0 && enemy.getY() <= gameData.getDisplayHeight());

    }
}
```

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Discussion

The problem we have tried to solve is how we can make a modular and easily changeable project. In the foundation we used two principles from the SOLID principles, the open/closed principle and Singular responsibility principle. With these principles as our foundation, we were able to remove/swap modules without recompilation and still have a functional game. This was also one of the non-functional requirements. Our other requirements have also been fulfilled by creating specific logic for each module which has a singular responsibility while working together to create the game we know as asteroids. The game could be further improved upon by creating more features which would change the gameplay, but the structure of the projects stands as is and it is easy to extend upon with multiple ideas.

The games approach to addressing the identified essential problems regarding the module updates and adaptability is highly effective. Our structure is not perfect and could be improved upon by following all the SOLID principles, other than open/close and Single responsibility principles. This has been considered however it remains uncertain if I have used other principles other than the two earlier described. If I choose to continue with evaluating the project and work on it, a refinement of how to implement all the SOLID principles would be a priority. It would create a robust and great modular structure.

Testing the infrastructure is an important task when creating any project, and our project is no different. We highly depend on tests to check if our logic works and if we have implemented it correctly, just because our code works doesn't mean it is optimal. Testing could be a revelation of how the thoughts behind our code are different from our actual intention. This also means that comprehensive testing verifies the correctness of our code while also guarding against regressions as we extend our codebase.

Mads Wrang Sigsgaard
Component Based Software Engineering (CBSE)
4. Semester, 2024

Conclusion

Upon revision of the analysis, design and implementation we were successful in creating a structure that allows us to modify our project easily. By using components that have implemented singular responsibility and open/closed principle, we created a foundation which coherent with our functional and non-functional requirements. This was revised by creating tests for our entities and methods to keep the structure and coding standard high. Our test was important for validating the correctness and reliability of our implementation, this helps us to ensure that the system remains stable and maintainable.

This means that our program follows the component-based concepts which gives us the possibility to update the asteroids project without having to recompile. However, there is a lack of SOLID principles for the structure to be better suited as a modular and easily changeable project, this should be revised upon in further work with the project.

Furthermore, for the future of the project, besides the SOLID principles, more extensive tests should be created to have a bigger grasp of the code functionality. With this said the project is setup for extensions to be made wherever it would fit, which is the bare minimum for our modular project.

In conclusion, the project effectively follows component-based concepts while providing the flexibility we want to update the asteroids project without having to recompile. The structure of our project meets our current requirements and lays a solid foundation for further extension.