Component Based Systems

Student Name: Joakim Leed [jolee18]

Student Exam Number: 4107152

GitHub User: OthelloEngineer

GitHub Repo: https://github.com/OthelloEngineer/component-oriented-portofolio

Abstract

The conventional monolithic architecture model has significant shortcomings with the frequently changing use-cases and requirements in the software development lifecycle. To improve the flexibility and maintainability of the software, Component Oriented Programming was introduced. To showcase the flexibility of Component Oriented Programming, the classic “Asteroids” game was re-created using Component Oriented Design principles and Programming techniques such as Service Locator and Dependency Injection patterns which in this project was respectively implemented with Java’s native ServiceLoader and Java Spring. The application was easy to extend and decoupled by using SPI’s described with pre- and post-conditions. The reliance on contracts, as encouraged by the Dependency Inversion Principle, lead to seamless mocking of the interfaces when testing, which was attributed by the high cohesion and low coupling of the code base.

Introduction

The monolith is a simple, frequently used application architecture that allows for fast development, easy networking and simple build process; however, it comes with downsides too. The monolithic structure as shown in the *figure 1* below from the movie *“2001: A Space Odyssey”* is tall, magnificent and smoothly polished. It does what it’s made for well; being a tall, menacing statue of the looming human evolution.

Figure 1: The Monolith from “2001: A Space Odyssey”

In software, it is important that developed applications do their job well and satisfy the business requirements. The monolith suits this purpose in the cleanest and most

optimized fashion; however, it lacks essential features that software needs in the world of enterprise applications, that is flexibility. The smooth edges and faces of the monolith make it nearly impossible to have it reshaped into a new structure that solves other problems that it was initially build for; it lacks modularity. When developers are building the project, they need to build it from bottom to top, which proves inefficient if the developers get smarter in the process of building the project. When they started building the middle, they realize that the bottom cannot support the monolith and they must start all over again. This in part reiterates the problems of the lack of modularity but also reveals that developers cannot build each part of the monolith independently. Especially with the new trends of multi-repository projects, multi-environment pipelines through CI and extensibility/reuse of previously build applications for new and modern use cases, the monolithic architecture model becomes less appealing.
The lack of modularity breaks the open/closed principle not only at the level of code, but at the design level, since a developer is forced to modify the existing classes instead of adding new files to its environment when requirements are changed or added. The source code is not closed for modification. Organisational Scaled Agile Frameworks such as *"Scrum at scale," "Nexus,* or *"SAFe,"* are limited in terms of efficiency when faced with an architecture that does not allow their teams to work independently of one another.

> *"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure." – Melvin E. Conway, known as Conway's law.*[1]

Forcing an organisation to build from a repository and architecture that does not reflect the communication structure of their organisation would be forcing the organisation to diverge from the natural tendencies of adhering to Conway's law, resulting in a burdensome and inefficient developing process.
One could argue that organising the code base into domains that reflect the responsibilities of the teams in the organization would solve this issue, and while that is true on paper, it does not solve the limitations of versioning and deployment. To solve these issues a new paradigm must be implemented. This paradigm is known as Component Oriented Programming.

Component Oriented Programming allows for the different teams to independently deploy, version and test the code that is specific to their responsibilities. The dependencies between the teams reflect the most natural conversational topic between the teams, that is the responsibilities of their transactions, in other words contracts: "I expect you to do X when I give you Y." Object Oriented Programming languages usually has a rich type-system that includes a type that specifically paraphrases this conversation in terms of code; interfaces. An interface can act as a contract since it defines a set of methods with declared inputs and outputs that its implementations must conform to in order to be a valid reference of that type.
The contract can be furtherly elaborated with questions such as: "I expect you to have these criteria in order before I initiate this transaction;" preconditions and "I expect you to have these criteria in order after our transaction is over;" postconditions. A further step can be taken towards the paradigm of Design by Contract where such contracts would be elaborated with contravariance, covariance and invariance, which

explore and defines the limitations of inheritance of the inputs, outputs and functions themselves. Without going into examples these can briefly defined as:

$\boldsymbol{A \leq C}$: A is a subtype of C.

$\boldsymbol{X(A \rightarrow B)}$: A unary function called X that transforms A into B.

$\boldsymbol{Y(A \rightarrow B) \leq X(C \rightarrow B)}$**:** function Y is a subtype function X**.**

Contravariance: $(\boldsymbol{A \leq C}) \Rightarrow \boldsymbol{X(C \rightarrow B) \leq Y(A \rightarrow B)}$: if A is a subtype of C, then a function X with input C that returns B must be a subtype of the function Y with input A and returns B.

Covariance: $(\boldsymbol{B \leq D}) \Rightarrow \boldsymbol{X(C \rightarrow B) \leq Y(C \rightarrow D)}$: if B is a subtype D, then B must be a valid return type of functions that return D; likewise, functions that return B is a subtype of functions that return D.

Invariance: $\boldsymbol{X(A \rightarrow B) \leq Y(C \rightarrow D) \Leftrightarrow A = C \land B = D}$: if and only if function X is a subtype of function Y, then type A must be equal to type C and type B must be equal to type D.

To highlight the benefits that Component Oriented Programming can provide, this project will show how it is possible to take an old game like Asteroids and create it using a Component Oriented Programming approach. The player will be able to control the player and fight different enemies and asteroids. The game will be run with Dependency injection and service locator patterns, to highlight and analyse the differences of the tools and the game will show the strengths of contracts and multi-repository projects.

Requirements

The product of this project will be an accumulation of seven different labs and result in a system where each component communicates through interfaces with defined SPI-contracts that describe the parameters, return values, pre- and post-conditions of the methods within the interfaces. Some of the requirements have been defined in the project description but a few others have been added to the requirements specification in order to showcase an understanding of proper component-oriented design.

Functional Requirements

Name	Description
P-1	The game must have a player component
P-2	The player of the game must control the player component
P-3	The game must stop when the player is dead
P-4	The player will be able to shoot enemies and asteroids in the game
E-1	The game must have an enemy component
E-2	The enemy must be able to kill the player
E-3	When shot, the enemy will take damage
E-4	When colliding with an asteroid, the enemy must take damage.

W-1	The game must have a weapon component
W-2	The weapon must shoot bullets
W-3	The weapon component must be used by the player
M-1	The game must have a map component
M-2	The player, asteroids and enemies must move around on the map component
M-3	The map component will provide a soundtrack for the game
M-4	The map will provide an asset for the background.
S-1	The game must have an asteroid component
S-2	The asteroid must be split when taken damage

Non-functional requirements

Name	Description
C-1	Player, Enemy and weapon components must be implementing service provided interfaces
C-2	Without recompilation, a component can be removed from the execution without affecting the other components
A-1	The application must follow strict architecture
G-1	The application must be a product of all java labs.

Analysis

The game will need a game engine to render all the assets of the entities from the plugins, that is Player, Asteroids and Enemy. For the player to be able to control the player, the inputs must be gained from either the game-engine or a plugin that can measure device inputs from another API. The enemies as well as the Asteroids will spawn immediately in the game and follow a straight path, since there are no requirements for AI, thus spending time constructing an AI for the enemies would gain no value to the project.

a shared library that will make it easier to produce an architecture where dependencies only go inwards towards higher abstraction levels, which can solve A-1.

Since there are shared use-cases between the enemies, asteroids and player, i.e. initializing and updating their state, it would be beneficial to create two flexible interfaces that make provide enough information for use-cases to be successfully implemented but not so much that the information will be overwhelming and difficult for them to use. The map component will not interact with the other entities so it should have its own interface. The game will be played on a computer and executed with the terminal. C-2 will be tested by either deleting or moving the jars from the logical directory wherein the modules are gathered to another unspecified location.

With all requirements discussed, a model of how they interrelate and what entities should be involved with the game can be drawn. A use-case diagram will help visualize the features and capabilities of the application.

As visualized in figure 2, there are mainly two different categories of features that need to be implemented. The first is the features of the game where all the features that make the game fun to play, the other category consists of requirement C-2.

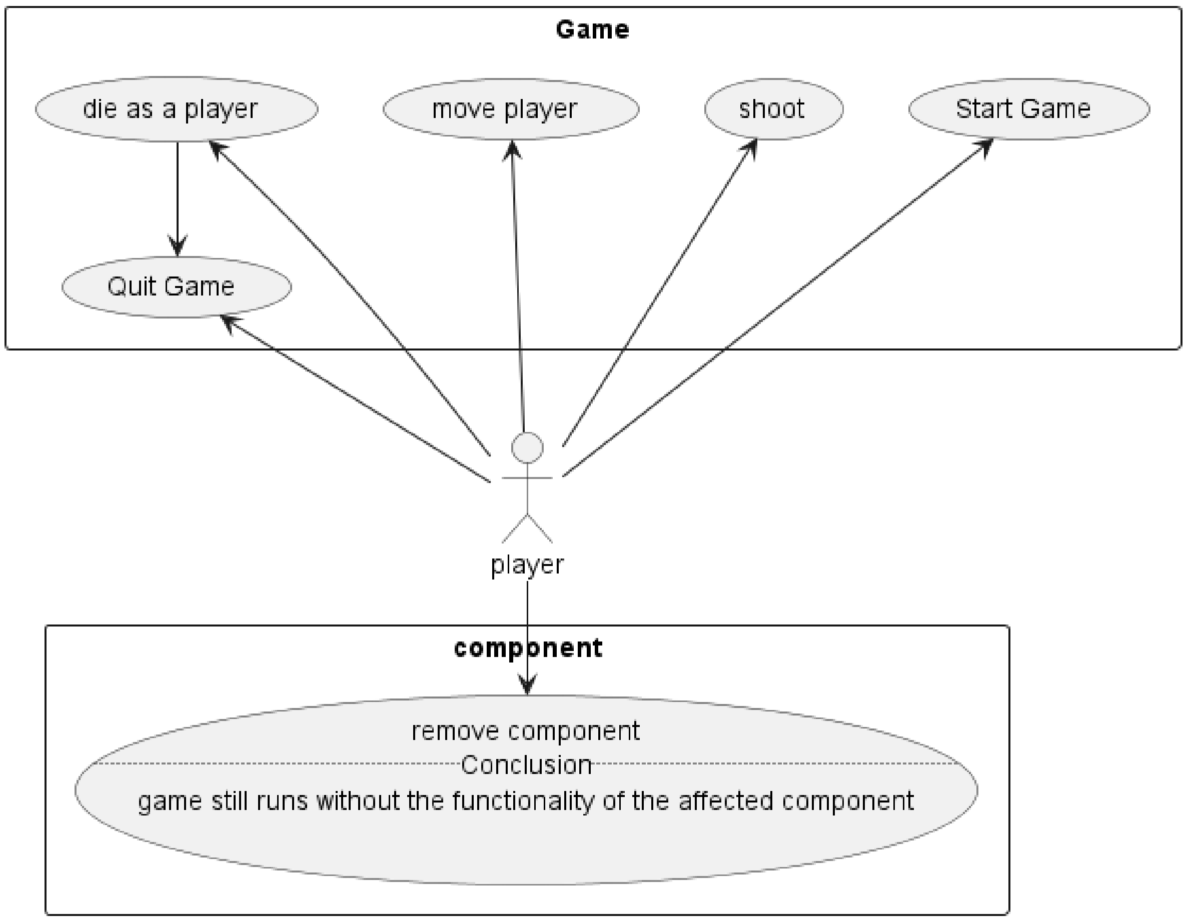


Figure 2: use-case diagram divided into two categories of use-cases. Component and Game related use-cases.

Design

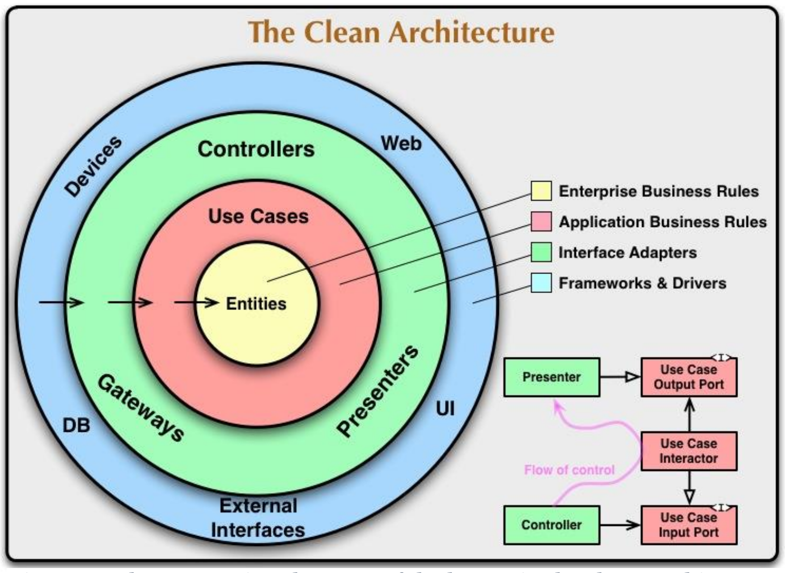


Figure 3: The conventional names of the layers in the clean architecture model

Requirement A-1 restricts the directions of the dependencies and demands that the architecture is split up into abstraction levels. These abstraction levels will be derived from the design paradigm of *"clean architecture"* where the layers on inwardly order are called; Infrastructure/UI, controllers/presenters, use-cases, Entities.
This model leaves the elements the elements that usually change often in the "*irresponsible*" outer layer and places the entities, that should rarely change, in the middle where they are kept responsible, which reflects the fact that all the components in the systems tend to use them. This project will a simplified version of the clean architecture. The Entities layer will be a domain layer, where all elements of a generic game will be defined such as Entity, Colour definitions and keys.
The layer surrounding the domain layer will be the *business specific rules* where details of the specific game will be found. For example, an interface for requirement S-2 will be defined here. By creating this layer, the components will only be exposed to the information that they choose to be exposed to by their module descriptor. The outer layer will be the controller layer, where the specific implementations of the Entities, post processors and such are defined. The controllers will be able to know of each other indirectly through their defined components in the business specific rules layer. An enemy would be able to call a "getPosition()" method on a Player Entity by using the entity defined in CommonPlayer defined in the business specific rules layer without depending on a component on the same abstraction level as itself. This is a great use of the Dependency Inversion Principle since it allows for switching and changing the specific implementation of the player in the controller level without affecting the Enemy depending on its abstractions. It also makes testing easier as Enemy can mock the Player module without breaking requirement A-1. This is not to say that the clean architecture model is the only way to solve such issues. As demonstrated in my semester project, I solved this by not creating a business specific layer with CommonX components, but instead creating an event driven architecture for the component-based game, where they all communicated through a common event *"registry"* defined in the domain layer through an EventBroker. An event-driven architecture is also a very valid approach to creating games. The greatest benefit of an

event driven architecture is its flexibility that keeps the developers flexible when they are unsure of how the components are going to relate to each other. The disadvantage is that it adds complexity to the run time state of the game, and it requires a great logging system to find bugs through the large data stream that the event broker manages. In the semester project, I integrated Log4J to manage the logging through a LoggingService SPI.
The code from the semester project can be seen below.

```
private final Map<EventType, Set<EventProcessor<? extends Event>>>
subscribers = new EnumMap<>(EventType.class);

public <T extends Event> void publish(T event){
    events.put(event.eventType, event);
    for (EventProcessor<? extends Event> subscriber :
subscribers.get(event.eventType)) {
        ((EventProcessor<T>) subscriber).handleEvent(event);
    }
}
```

In this project, however, the requirements are well-defined and a clear vision of how the application will evolve, thus the former approach is more suitable.
Highlighting the abstractions levels and the components, the resulting architecture will look like this.

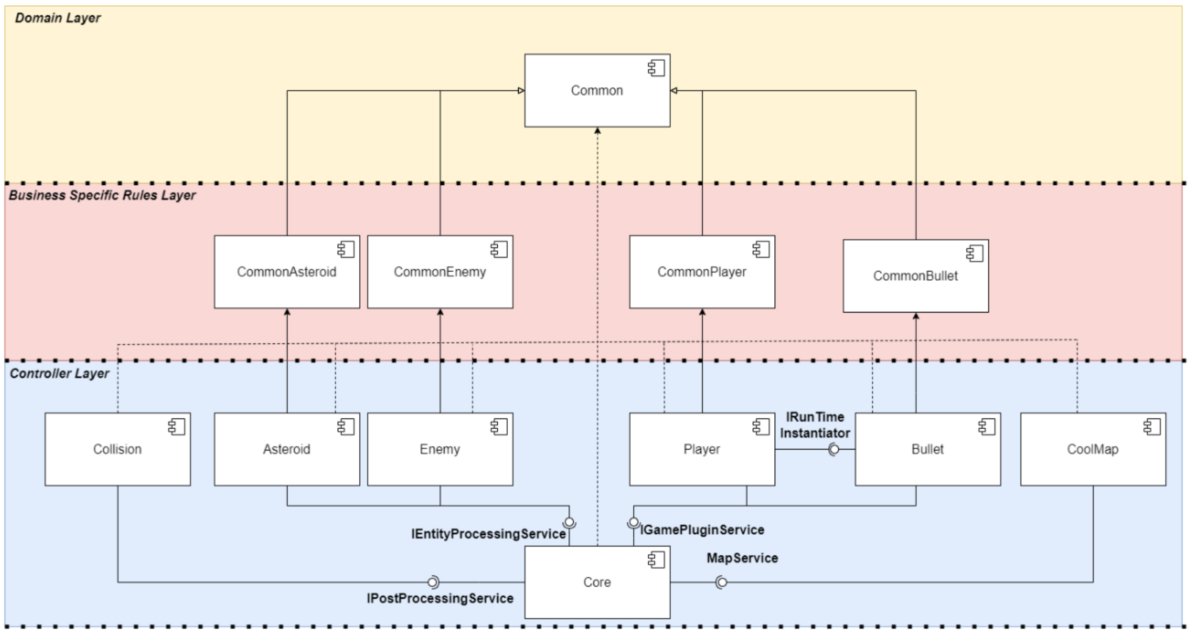


Figure 4: The architecture of the portfolio project modelled with the standards of UML. A larger example can be found in the appendix.

An interesting distinguishment between the relationship of controllers and common can be made when inspecting for instance Collision and Asteroid. Asteroids might be used by the Player component to keep track points of specific players, which means it needs to know about the asteroids. Collision on the other hand is a post processor that no other components should know about, thus it does not have a representation on the business specific rules layer. The Player's position is important for the enemies if they were to implement an AI and Players might get points for killing enemies. The components on the Business Specific Rules Layer creates extensibility for new features in the game, if that is desired when the project matures. That is the true strength of software architecture; it makes the application able to easily change in all the places that it is expected to change. The areas of the codebase in the application that is expected to change the most are the concrete implementations of Entity and of the SPI's since they offer the most logic to the application, which is almost guaranteed to change when a related feature needs to be changed. With logic targeted as a prone area to change, it is vital to the maintainability of the application that little to no logic or at least no application specific logic is defined in the Common component, since that could risk affecting all of the components in the application when requirements change. Without going into depth about the relationship between degrees of abstraction and instability since this is not a course focused on maintenance, the positional instability of the Common component should evaluate to 0, the Business Specific Rules components should evaluate to ≤ 0.5, and the controller components evaluate to 1. These values can be evaluated by using the formula $\frac{fan_{out}}{(fan_{in}+fan_{out}} = I_{instability}$ where a fan is a dependency either pointing in towards or out from a component.

Another essential part of software architecture is that it should not leak implementation details. Whether a service locator or a dependency injection framework will be used is an implementation, so it is absent from the architectural design. As Robert C. Martin wrote in Clean Architecture[3]:

> *"A good architect pretends that the decision has not been made, and shapes the system such that those decisions can still be deferred or changed for as long as possible. A good architect maximizes the number of decisions not made."*

The addition of the Business Specific Rules Layer maximizes the number of decisions not made, which makes the application easily extensible for new features which is one of the expected changes.

Implementation

The implementation of the portfolio project was developed through seven iterations that is the labs of the course. It is the accumulative product that will be described in this report, as it reflects the final source code. Few parts of source code that were overwritten by future labs will be mentioned, since they were related to the instantiation of SPI implementations.
The game logic will be skipped, since it does not solve the primary goal of the project that is component-oriented programming.
Each of the components Collision, Asteroid, Enemy, Player, and Bullet is implemented by separating the concerns of their object definition, instantiation and usage from each other. An example of this would be Player.

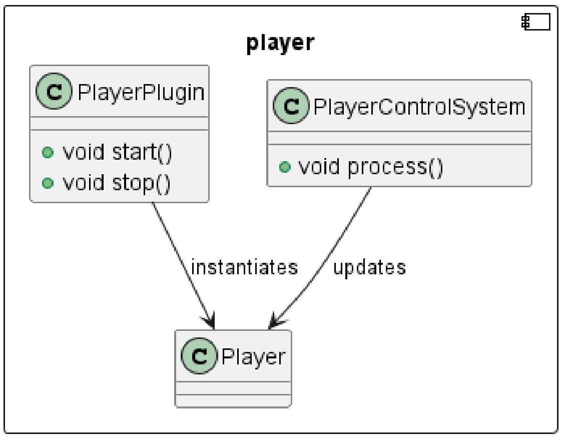


Figure 5:Classes of the player Component

The Player class defines the type-definition of Player, PlayerPlugin instantiates the player and PlayerControlSystem uses the player objects of the game. One can find the same pattern in creational design patterns such as factory and builder patterns. This proves very flexible when other components require awareness of the objects in the class, by storing the type-definition in the business specific rules layer instead of the controller layer.

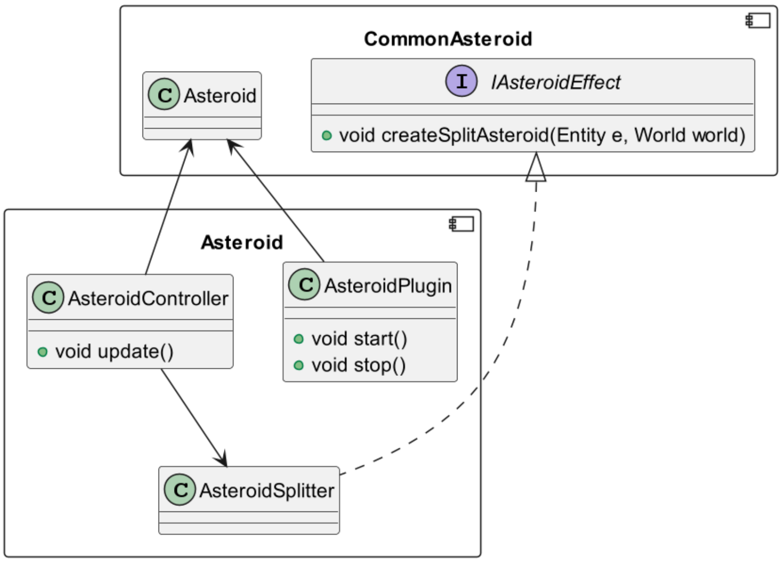


Figure 6: Classes of the Asteroid parts of the application

The methods provided by the SPI's of the application was documented thoroughly through explanation of their parameters, pre- and postconditions. This was done to ensure that the side-effects of the implementations were kept within non-breaking constraints as parameters and return types only tell a portion of the story of an interface. Below are the SPI contracts of the Common component listed.

IEntityProcessingService	
Operation	void process(GameData gameData, World world);
Description	The process method is called every refresh of the game state. The plugin will update its objects through this loop. The order in which plugins are being called is **NOT** guaranteed.
Parameters	GameData stores the metadata about the game's state, such as display dimensions of the view port and most notably the difference of time since last method call with getDelta(). World stores the entities of the game, specific entities of the game can be retrieved with their class signature with the getEntities(Class) method.
Pre-conditions	gameData must not be null. world must not be null. gameData.getDelta() must not return a negative float value. World contains all the entities that need update in this game loop. The events within GameData must represent the events from the former loop if not an offset must be included in the plugins to negate the non-idempotent nature of the system.
Post-conditions	World has been updated properly with delta (time) considered. No alteration has occurred in gameData excluding the potential addition of events. All entities that the plugin is responsible for has been updated, added or removed. Errors that might emerge within the method body must be handled.

IGamePluginService	
Operation	void start(GameData gameData, World world);
Description	The method is called in the beginning of the game. The initial objects that the plugin is responsible for will be instantiated and put into the world object.
Parameters	GameData stores the metadata about the game's state, such as display dimensions of the view port and most notably the difference of time since last method call with getDelta(). World stores the entities of the game, the entities can be added by using the addEntity() method.
Pre-conditions	gameData must not be null. world must not be null. The dimensions of gameData must represent the dimensions of the viewport.
Post-conditions	The initial objects that the plugin is responsible for have been instantiated and added the world object. No alteration has occurred in gameData excluding the potential addition of events.

	Errors that might emerge within the method body must be handled.

IGamePluginService	
Operation	void stop(GameData gameData, World world);
Description	The method is called in the end of the game. The objects that the plugin is responsible for will be found in world and removed from the world object.
Parameters	GameData stores the metadata about the game's state, such as display dimensions of the view port and most notably the difference of time since last method call with getDelta(). World stores the entities of the game, the entities can be added by using the addEntity() method.
Precconditions	gameData must not be null. world must not be null. World must contain all the objects that the plugin is responsible for and remove them
Post-conditions	The object that the plugin is responsible for managing has been removed from the world object. No alteration has occurred in gameData excluding the potential addition of events. Errors that might emerge within the method body must be handled.

IPostEntityProcessingService	
Operation	void process(GameData gameData, World world);
Description	The process method is called every refresh of the game state. The order in which plugins are being called is **NOT** guaranteed, but it can be assured that the methods of IEntityProcessingService has been called before this function, hence the Post in the name.
Parameters	GameData stores the metadata about the game's state, such as display dimensions of the view port and most notably the difference of time since last method call with getDelta(). World stores the entities of the game, specific entities of the game can be retrieved with their class signature with the getEntities(Class) method.
Pre-conditions	gameData must not be null. world must not be null. The methods of IEntityProcessingService have been called before this function.
Post-conditions	The effect of the postprocessor has been carried out. No alteration has occurred in gameData excluding the potential addition of events. Errors that might emerge within the method body must be handled. The state of world reflects the post-processing changes made during the current game loop.

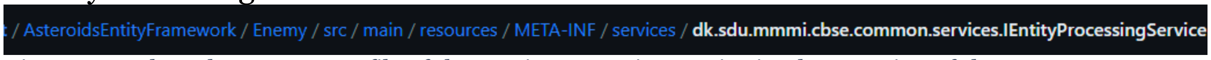
MapService	
Operation	String getMap();
Description	The Mapservice is responsible for giving the file path to a background image that will appear in the of the game.
Return value	A string pointing to an image file
Pre-conditions	The GameEngine must have access to the same files as MapService The Image File extension must be of format .png or .jpg
Post-conditions	The returned value must be a valid path directly to the image file. If the path does not exist, the error must be handled.

MapService	
Operation	String getSoundTrack();
Description	The Mapservice is responsible for giving the file path to a soundtrack that will be played while the game. The soundtrack will be looped through indefinitely.
Return value	A string pointing to an audio file
Pre-conditions	The GameEngine must have access to the same files as MapService The Image File extension must be of format .mp3 or .wav
Post-conditions	The returned value must be a valid path directly to the audio file. If the path does not exist, the error must be handled.

In the first lab, the Core component had direct dependencies to the specific implementations in order to instantiate them. In later labs it was replaced by the ServiceLoader which is a ServiceLocator that was introduced in Java 8. The ServiceLoader made use of dependency inversion to move the control of specific implementation instantiation from the source code to the environment in which the application runs in. The application-level control of which objects to instantiate can be regained if markers are placed inside of the implementations which the applications themselves can be filter through, but this negates the flexibility that the ServiceLocator introduces and consequently its entire purpose is defeated. This would be a code-smell, specifically it will point towards a violation of the Interface Segregation Principle. The implementation details should be hidden from the user of the implementation, in order to keep the code base flexible to change, extensions, and especially to keep the codebase easy to test.

In the first labs where the ServiceLoader was used, it found the implementations of the SPI's through a predefined folder path, with a predefined package name that corresponds to the FQCN, i.e. Fully Qualified Class Name, provided interface name of implementation.

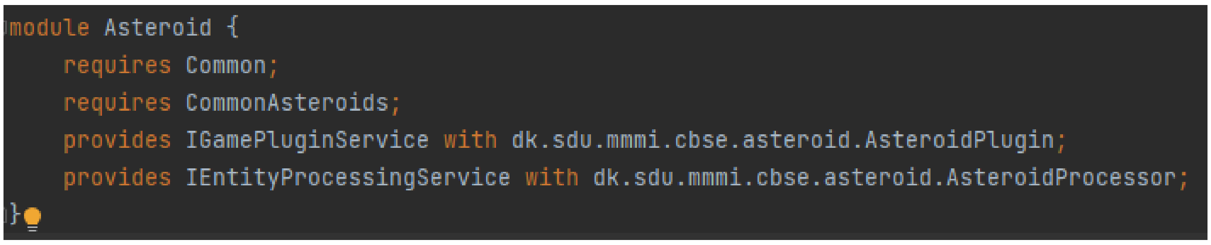
An example of this would be the EnemyControlSystem that implements the IEntityProcessingService



/ AsteroidsEntityFramework / Enemy / src / main / resources / META-INF / services / dk.sdu.mmmi.cbse.common.services.IEntityProcessingService

Figure 7: Path to the META-INF file of the IEntityProcessingService implementation of the EnemyComponent. This can be found in a different repository than is linked on the front page, since the implementation is overwritten by future labs[4].

"dk.sdu.mmmi.cbse.playersystem.EnemyControlSystem" is the content of the file in figure 8. This is later removed in favour of the module-info files from JPMS introduced by the *"java jigsaw project"*, that can explicitly expose provided implementations with the *"provides … with …"* keywords. An example from the code base can be seen below of figure 9.



```
module Asteroid {
    requires Common;
    requires CommonAsteroids;
    provides IGamePluginService with dk.sdu.mmmi.cbse.asteroid.AsteroidPlugin;
    provides IEntityProcessingService with dk.sdu.mmmi.cbse.asteroid.AsteroidProcessor;
}
```

Figure 8: Module-info.java file of the Asteroid Component

The FQCN of the SPI's are no longer needed in the exposure of the implementations to the ServiceLocator, which creates a more flexible, easier to use system that does not expose its bowels to its implementations.

To showcase how the Spring Framework can instantiate objects and introduce inversion of control, the intra-component relations of the Core component was instantiated with Spring. Game instantiated an ImplementationLocator @*component* class through @*Autowire* and the Main class instantiated a Game object through a *"configuration"* class of Game called GameConfig. The Main class discovered the configuration class by registering it and scanning the packages of the Main Component for the configuration class and Game class. A more in-depth explanation of how this was carried out can be found in appendix.

The key feature of the Spring framework is Inversion of Control, IoC. By defining the functionality of the application and not how the different sections of the code base are referring to each other, the management of the object lifecycles has been transferred from the developer to the framework[5]. This diverges from traditional programming practices where the developer is also responsible for managing the lifecycles and dependencies of the objects in the application, which is prone to result in tightly coupled and hard-to-maintain code without solid coding principles and design discipline. When the developer only focuses on writing the core functionality of the application, the development speed will naturally be increased while Spring ensures that the codebase will remain modular and flexible.

Spring, however, does have a niche annotation- and XML-based syntax that one must be well-versed in to gain the advantageous velocity that it provides. Spring also has fantastic libraries for instance one that introduce Aspect Oriented Programming features such as interceptions for logging, sessions and auth. Its disadvantage is that if one needs in-depth control of the lifecycles of the java beans, one will quickly find themselves buried in heaps of XML-configurations.

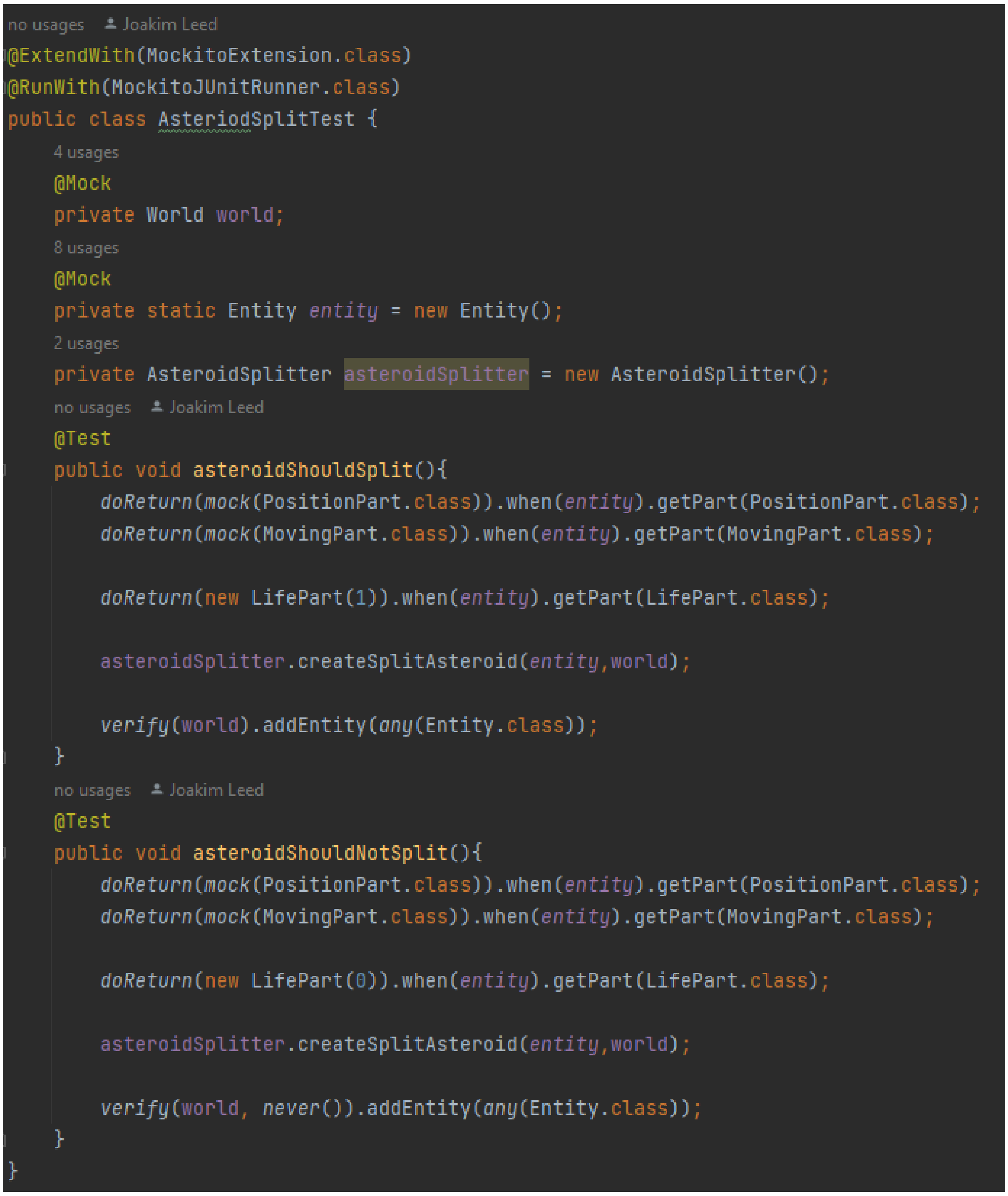
Spring requires runtime access to the components, since it uses reflection for its service discovery. Consequently, to integrate it with JPMS, the components must either *"exports"* or *"opens"* themselves to Spring. Since Spring only needs runtime access, "*opens*" would keep the exposure of the component to the minimum, which according to the *law of demeter,* also known as *principle of least knowledge*, is desireable.

LibGDX lacks a module descriptor. Consequently, to grant it a one it must be shaded. Shading is typically used to create "fat-jars" or in other terms "Uber-jars" that include not only the bytecode of the application but also the bytecode of its dependencies. This can prevent split-packages by renaming packages, simplifying the deployment of the jars, prevent dependency versioning issues where changes were not backwards compatible, etc. Shading should be used with care as it bloats the size of the application.

Test

It is a common assumption that applications build with service locators are in nature harder to test[4,5], since the dependencies in question managed by Spring can easily be replaced by mocks and the configurations can easily be swapped out. A well-designed system should however not rely on specific implementations but contracts that can easily be mocked by a good mocking framework. The testing of the asteroids application has not faced any of aforementioned issues. This might be in part be caused by the focus on contracts that leak no implementation details and in part be caused by none of the tests having a wider scope than the cohesions of the components themselves given that they are unit tests. When creating tests with an application-level scope such as regression tests, specific implementations might be needed to simulate the real application behaviour that will be deployed in the production environment.

The unit tests of the application were implemented using a mocking framework called Mockito. Specifically, the BDD library of the Mockito framework. On the next page are the tests of the AsteroidSplitter.



```
no usages  Joakim Leed
@ExtendWith(MockitoExtension.class)
@RunWith(MockitoJUnitRunner.class)
public class AsteriodSplitTest {
    4 usages
    @Mock
    private World world;
    8 usages
    @Mock
    private static Entity entity = new Entity();
    2 usages
    private AsteroidSplitter asteroidSplitter = new AsteroidSplitter();
    no usages  Joakim Leed
    @Test
    public void asteroidShouldSplit(){
        doReturn(mock(PositionPart.class)).when(entity).getPart(PositionPart.class);
        doReturn(mock(MovingPart.class)).when(entity).getPart(MovingPart.class);

        doReturn(new LifePart(1)).when(entity).getPart(LifePart.class);

        asteroidSplitter.createSplitAsteroid(entity,world);

        verify(world).addEntity(any(Entity.class));
    }
    no usages  Joakim Leed
    @Test
    public void asteroidShouldNotSplit(){
        doReturn(mock(PositionPart.class)).when(entity).getPart(PositionPart.class);
        doReturn(mock(MovingPart.class)).when(entity).getPart(MovingPart.class);

        doReturn(new LifePart(0)).when(entity).getPart(LifePart.class);

        asteroidSplitter.createSplitAsteroid(entity,world);

        verify(world, never()).addEntity(any(Entity.class));
    }
}
```

Figure 9: The tests of the AsteroidSplitter implemented with Mockito.

All dependencies from other components than the component itself is mocked in order to isolate the testing scope to the test itself. The specific values that are in play in the tests are explicitly defined with the *"doReturn()"* method of the to highlight which values are important for the conditions in the tests.

To test and record requirement C-2, a video was recorded with the removal of the asteroids and enemy components without re-compilation.
https://youtu.be/eipAlkhi_ko

Discussion

All the requirements have been fulfilled. Perhaps exiting the game with a status code 1 is not an optimal, from a UX perspective, solution for handling player death but it ultimately stops the game when the player is dead. Further Heads-Up Display, HUD, elements would be great features to implement gameplay wise, however gameplay is not the focus of this project.

A higher testing coverage would introduce more resilience into further development of the game and all requirements except S-2 are verified, personally by me, the author of the project, and not any formal tests, which is naturally not optimal. Testing whether the application has a "Player" component would require unorthodox testing such as pattern matching names of files or testing whether a CommonPlayer Component has an implementation, which can only be done by a separate testing component, sort of resembling a sidecar in a Kubernetes deployment.

Merging the weapon and bullets components could be interpreted as not fulfilling the requirement since the Weapon component does not shoot objects of a separate Bullet component. However, it is never specified whether Bullet should be a component, so it could also be the weapon itself.

The current spring implementation in the Core component is quite unorthodox and the IntelliJ is certainly unhappy with the, if I may say so myself, witty solution that was intentionally left.

The CommonPlayer was never implemented, since it wouldn't bring extra value to the product in terms of requirements and the potential of using the Business Specific Rules abstraction layer was already showcased with CommonBullet and CommonAsteroid.

Conclusion

Through the use of Spring, Java's native ServiceLoader and JPMS the classic asteroids game was recreated with the Component Oriented Programming paradigm. The application is closed to modification and open to future extensions with the addition of a Business Specific Rules abstraction layer that intermediates the relationship between the components adhering to strict architecture. The architecture was designed with low coupling and high cohesion in mind which was highly regarded when architectural trade-offs were made. The game exhibits expected gameplay features such as player movement, shooting, asteroids that split and enemies that can kill the player.

The codebase was easy to test by mocking the interfaces with Mockito although larger proportions of the codebase should be tested in the future.

References

[1] M. Conway, "Committees Paper," Mel Conway's personal website, 2023. [Online]. Available: https://www.melconway.com/Home/Committees_Paper.html

[2] R. C. Martin, "The Clean Architecture," The Clean Code Blog, Aug. 13, 2012. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[3] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River, NJ, USA: Prentice Hall, 2018.

[4] OthelloEngineer, Github Repository Feb. 27, 2023. [Online]. Available: https://github.com/OthelloEngineer/component-based-1st-

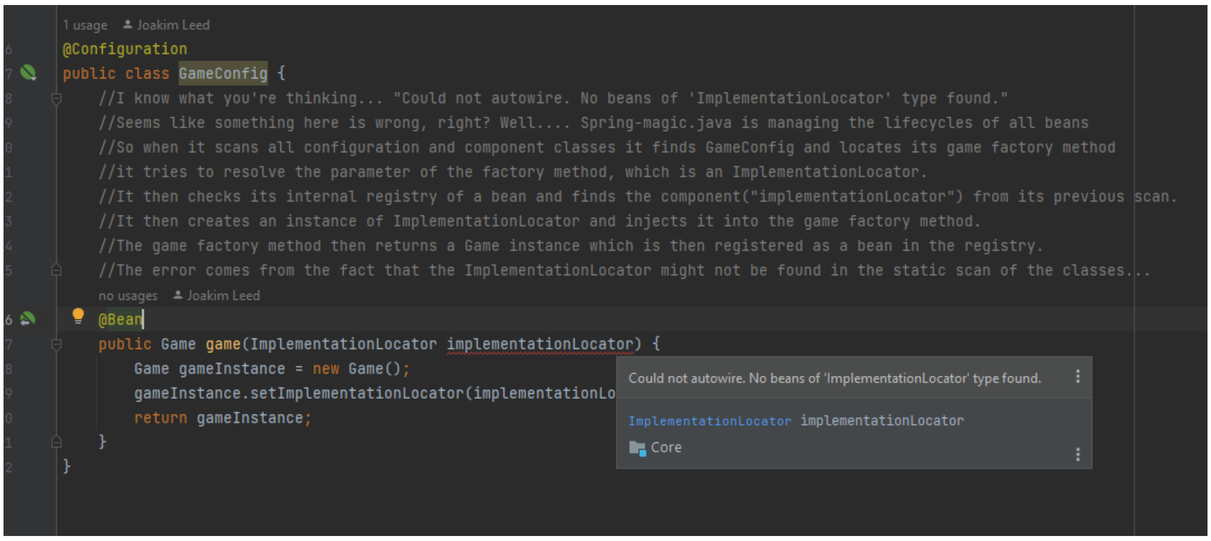
assignment/blob/main/AsteroidsEntityFramework/Enemy/src/main/resources/META-INF/services/dk.sdu.mmmi.cbse.common.services.IEntityProcessingService

[5] Fowler Martin, Inversion Of Control June, 26, 2005 [Online] Available: https://martinfowler.com/bliki/InversionOfControl.html

[6] Hannen Scott, How I Learned to Stop Worrying and Love the Service Locator [Online] Available: https://scotthannen.org/blog/2018/11/27/stop-worrying-love-service-locator.html

Appendix

A witty comment thread about my solution to use Spring in the Main Component.



```
1 usage   Joakim Leed
@Configuration
public class GameConfig {
    //I know what you're thinking... "Could not autowire. No beans of 'ImplementationLocator' type found."
    //Seems like something here is wrong, right? Well.... Spring-magic.java is managing the lifecycles of all beans
    //So when it scans all configuration and component classes it finds GameConfig and locates its game factory method
    //it tries to resolve the parameter of the factory method, which is an ImplementationLocator.
    //It then checks its internal registry of a bean and finds the component("implementationLocator") from its previous scan.
    //It then creates an instance of ImplementationLocator and injects it into the game factory method.
    //The game factory method then returns a Game instance which is then registered as a bean in the registry.
    //The error comes from the fact that the ImplementationLocator might not be found in the static scan of the classes...
    no usages   Joakim Leed
    @Bean
    public Game game(ImplementationLocator implementationLocator) {
        Game gameInstance = new Game();
        gameInstance.setImplementationLocator(implementationLo
        return gameInstance;
    }
}
```

Could not autowire. No beans of 'ImplementationLocator' type found.

ImplementationLocator implementationLocator

Core