Component Based Systems

Abstract

Introduction

A picture containing cloud, outdoor, sky, rock

Description automatically generatedThe 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 : 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 product; 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 the project 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 trends of multi-repository projects, multi-environment pipelines through CI and extensibility/reuse of previously build applications for new and modern use cases, the monolith becomes less appealing.

The lack of modularity breaks the open/closed principle not at the level of code, but at the system level, since a developer is forced to modify the existing classes 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 their own communication structure 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 commit to 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:

A is a subtype of C.

A unary function called X that transforms A into B.

**:** function Xis a subtype function Y**.**

**Contravariance:** : 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:** 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:**  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 arguments, returns, 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-1 matches conveniently with A-2, since they use a shared library that will make it easier to produce an architecture where dependencies only go inwards towards the shared library.

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, an analysis of how they interrelate and what entities should be involved with the game. 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.

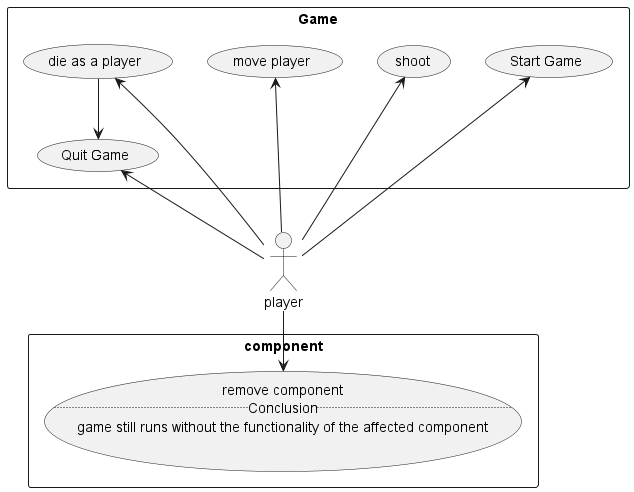


Figure : use-case diagram divided into two categories of use-cases. Component and Game related use-cases.

Diagram of a diagram of a clean architecture

Description automatically generated with low confidenceDesign

Figure : 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 is not only great to create applications in since it leaves the elements the elements that usually change often in the “*irresponsible*” outer layer but also places the entities in the middle where they are kept responsible, which reflects the fact that all of 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.

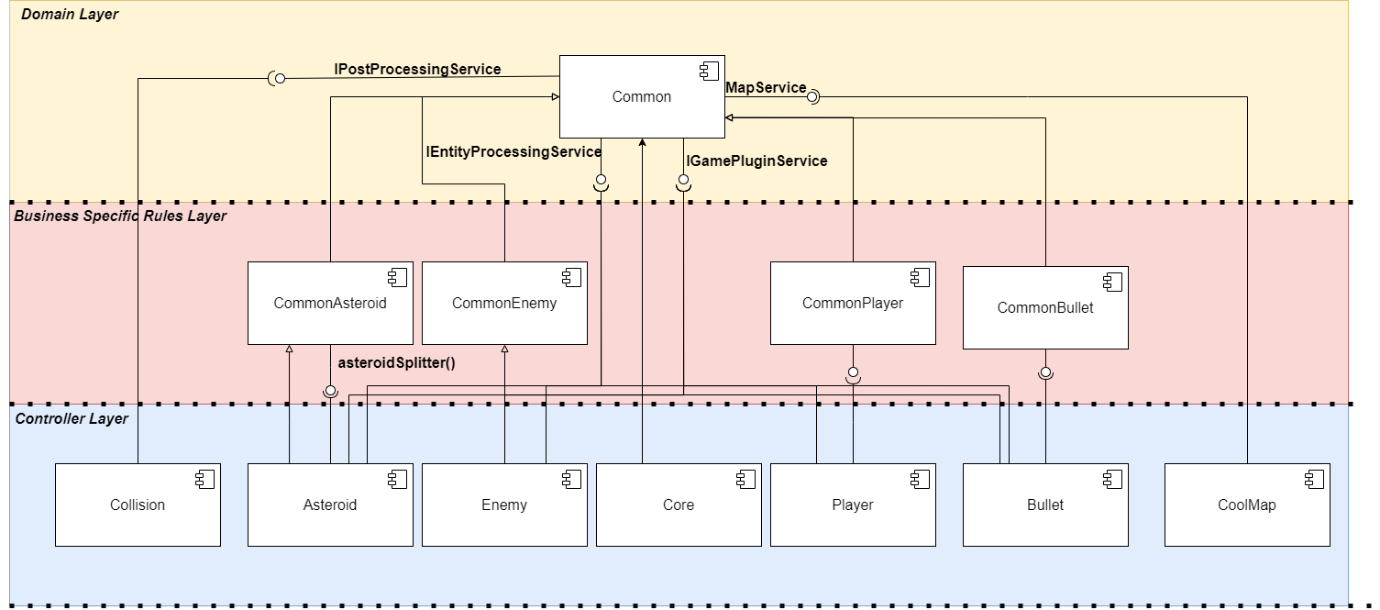


Figure : The architecture of the portfolio project modelled with the standards of UML. A larger example can be found in the appendix.

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 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 health 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. Without going into depth about the theory degrees of abstraction and instability since this is not a course focused on maintenance, the positional instability of the Common component should evaluate to , the Business Specific Rules components should evaluate to , and the controller components evaluate to These values can be evaluated by using the formula where fan is the dependency deriving from a class.

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 architecture. As Robert C. Martin wrote in Clean Architecture3:

“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 written in this report, as it reflects the source code. Few parts of source code were overwritten by future labs, if they are 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.

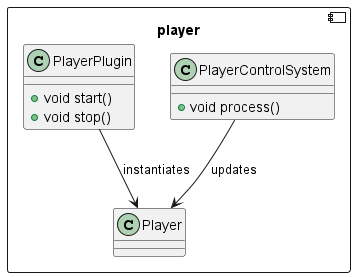
Each of the components Collision, Asteroid, Enemy, Player, and Bullet is implemented by separating the concerns of their object definition, instantiation and usage from each other. An example of this would be Player.

Figure :Classes of the player Component

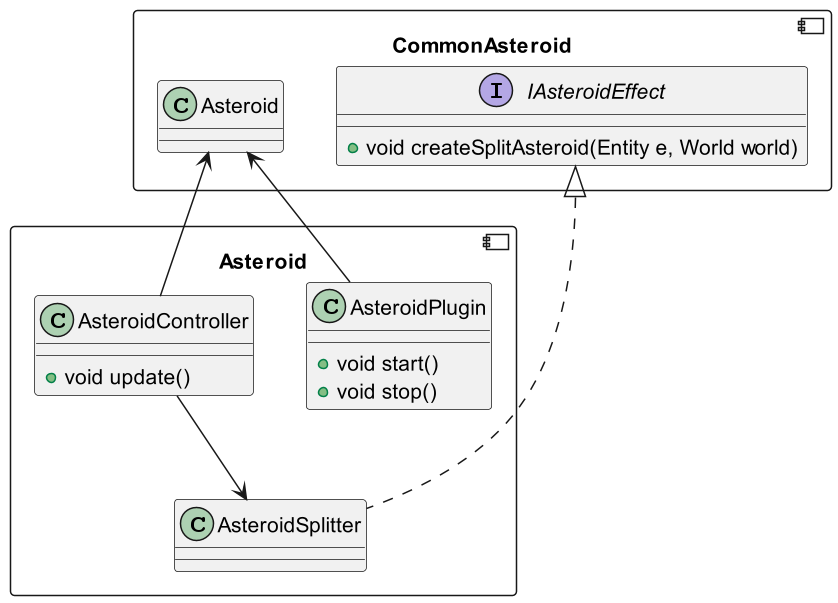
The Player class defines the type-definition of Player, PlayerPlugin instantiates the player and PlayerControlSystem uses the player objects of the game. One can find the same pattern in creational design patterns such as factory and builder patterns. This proves very flexible when other components need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

Figure : Classes of the Asteroid parts of the application

Test

Discussion

Conclusion

Appendix

[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.