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 |

**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 |
| A-2 | The application must be a product of all java labs. |

Analysis

Design

Implementation

Test

Discussion

Conclusion

Appendix

[1] Author(s), "Title of Web Page," Publisher or affiliated organization, day month year. [Online]. Available: URL