ATiNS

Technical Academy of Informatics

in Applied Sciences

Faculty of Informatics

Dawid Kaluża

Analysis of architecture solutions used in game development with Unity3D

Thesis: master's

The field of study: Informatics

The field of specialization: Programming

The thesis developed under the supervision of:

PhD Eng Leszek Grocholski

Wrocław 2024

Abstract

This thesis presents a comparative analysis of two distinct architectural approaches within the Unity3D game engine, aimed at enhancing the understanding of how architectural choices can impact the development process and final product in game development completing its goal. This approach enables a focused comparison of factors such as flexibility, maintainability, performance, and their appropriateness for various game genres. The analysis will then be meticulously analyzed and compared across both implementations of the project. Through the examination of a Match-3 game scenario, implemented under two different architectural strategies, key insights have been gained regarding the trade-offs between simplicity and complexity in game architecture, ensuring that the chosen architecture aligns with both the immediate and long-term goals. This thesis underscores the necessity for continued research to enhance our understanding and implementation of effective game architectures.

Keywords

Unity3D, Game Development, Software Architecture, Model-View-ViewModel (MVVM), Dependency Injection (DI), Architectural Patterns, Game Engines, Simulation, Comparative Analysis, Scalability

Contents

Introduction

With the rapid growth in game engine technology and the ease of access to game publishing and monetization platforms, the landscape of game development has transformed significantly.

Problem definition

As game engines like Unity3D get better and it becomes easier to release and earn from games, we see more games being made than ever before [1]. Unity3D helps make all sorts of games, from mobile to big console games. But with so many different games, developers face a challenge: picking the best way to organize their game projects.

Addressing this challenge is crucial for maintaining a game's quality, performance, and ability to evolve. This thesis aims to explore and compare various architectural frameworks within Unity3D to identify the most suitable ones for different types of games.

The representation of game architecture, especially as it pertains to Unity, is markedly limited in academic literature. To address this gap, the study will incorporate insights from related fields outside of game development, supplemented by the author's practical experience in the industry.

Goal of the thesis

The goal of this thesis is to evaluate and compare different architectures in game development using the Unity3D engine to determine the most effective architectural models based on project needs.

We will initially delve into Unity3D’s architectural offerings, followed by a methodology to analyze and compare these frameworks. The comparison will focus on factors such as flexibility, maintainability, performance, and their appropriateness for various game genres. The ultimate objective is to provide developers with insights that will aid them in selecting the optimal architecture for their projects, thereby improving game development outcomes.

This approach intends to make the decision-making process easier for developers, ensuring that their games are not only well-structured but also scalable and easier to update.

1. Introduction to games

The history of games stretches far back into human history, well before the advent of computers. Games have been a fundamental part of human culture and society, serving as sources of entertainment, education, and social interaction across different civilizations. It wasn't until the 1960s and 1970s, with the creation of games like Spacewar! [2] on early computers, the foundation for the modern video game industry was laid.

This rich history of games, evolving from simple board and physical games to complex electronic formats, underscores the intrinsic human desire for play, competition, and challenge. It set the stage for the explosion of digital and video gaming that would follow in the late 20th century, transforming games into a dominant form of entertainment and a significant cultural and economic force.

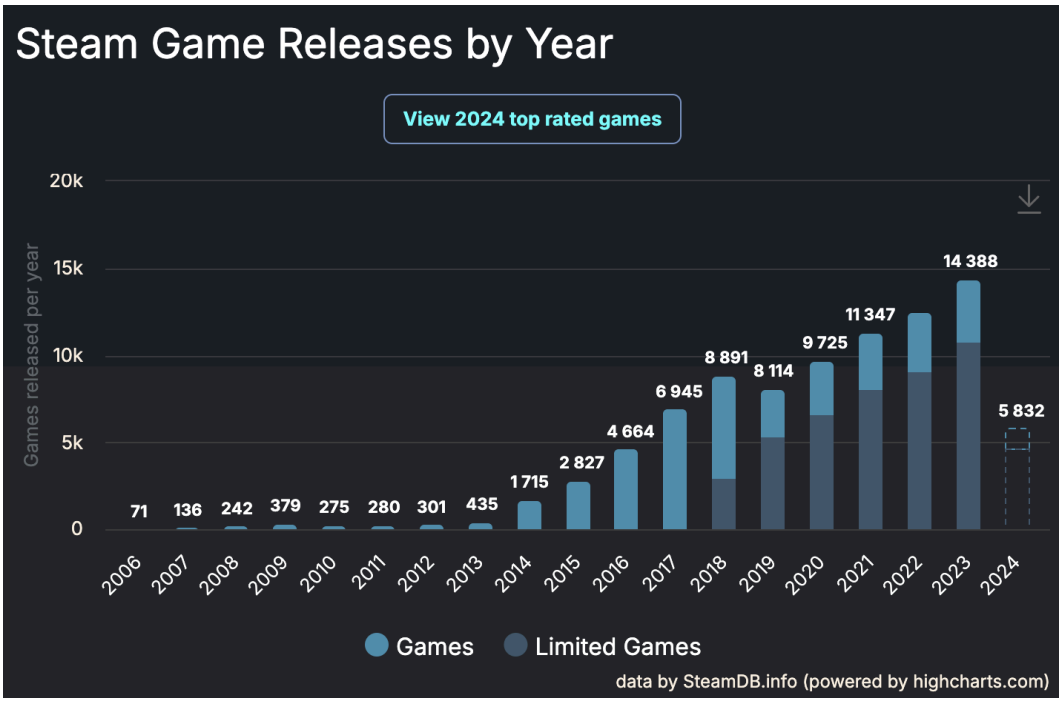

Figure 1.1. Steam game releases. The number of game releases is growing at a fast pace. (Source: [1])

The accessibility of game development tools, such as Unity3D and Unreal Engine, has democratized game creation, enabling both independent developers and large studios to produce games. The industry's revenue streams have diversified, including direct game sales, subscriptions, in-game purchases, and advertising. With the advent of digital

distribution platforms like Steam, the PlayStation Network, and the Xbox Live Marketplace, developers have direct access to global audiences [3]. The game industry's growth is further propelled by esports and live streaming platforms like Twitch, which have turned gaming into a spectator sport, creating new opportunities for engagement and monetization [4].

2. About author and industry work

As a Computer Science graduate student with ambitions to contribute to game development, my academic and professional journey in the game industry has been deeply rewarding. Prior to my current academic pursuit, I dedicated five years to the game industry, with a focus on developing mobile free-to-play games. This period included extensive work at Ten Square Games [5], where I specialized in game development and independently explored game design through many personal projects. My experience peaked with my role as a Senior Unity Engineer, highlighting a key area of my expertise.

This practical experience afforded me a detailed understanding of game architecture and exposed me to the complexities and challenges of crafting scalable game architectures. My involvement in projects ranging from those hampered by inadequate structure to those flourishing due to their robust architecture has profoundly shaped the perspective and methodological approach of this thesis.

This thesis aims to meld theoretical knowledge with practical application, informed by the valuable lessons from my industry experience. My goal is to enrich the academic community's comprehension of architectural strategies within Unity. By leveraging my background, I aspire to provide practical insights and a thorough analysis on selecting the appropriate architecture for game development projects.

3. Game engine comparison

Creating a game requires many systems. The more complex the game, the more systems it can require. Examples of such systems are: resource and file management, handling human interface devices, tools for debugging and profiling or rendering loop [6]. More in-depth analysis of those systems in Chapter 5.

Game engines are essential in the game development process for several reasons, streamlining the creation of games and enabling developers to focus on crafting unique gameplay experiences without getting bogged down in the technical intricacies of game programming [26].

Several major game development companies use proprietary, internal game engines designed specifically for their needs. These engines are not available to the public and are tailored to support the unique workflows, art styles, and performance requirements of the company's projects. Since those game engines are private architecture analysis cannot be used therefore this work will focus only on publicly available ones.

3.1. Unity

Unity is a cross-platform game engine developed by Unity Technologies [7], widely used for creating video games, simulations, and other interactive content. It provides a comprehensive set of tools for game development, from initial design to final deployment. Unity supports 2D and 3D game development, offering features like an intuitive visual editor, a powerful physics engine, advanced animation capabilities, and support for multiple scripting languages, primarily C#.

Figure 3.1. Unity logo (Source: Unity Technologies [7])

It stands out for its ability to deploy games across more than 25 platforms, including mobile devices (iOS, Android), desktops (Windows, macOS, Linux), web browsers, consoles (PlayStation, Xbox, Nintendo Switch), and VR/AR devices. Unity also offers a vast Asset Store, where developers can purchase or sell assets and tools to enhance their game development workflow.

Unity's pricing structure includes free tiers for lower revenue games and paid licenses for bigger studios, catering to a range of developer needs. Additionally, Unity offers services like cloud-based collaboration tools, analytics, and monetization options, enhancing its versatility for developers from indie creators to large studios.

3.2. Unreal Engine

Unreal Engine, developed by Epic Games [8], is a game engine renowned for its ability to create visually stunning and immersive 3D experiences. It excels in delivering high-fidelity graphics, making it a preferred choice for AAA games and cinematic content. Unlike Unity, which supports both 2D and 3D development, Unreal Engine is particularly focused on advanced 3D graphics, leveraging its powerful rendering capabilities.

Unreal Engine provides broad platform support, similar to Unity, including deployments to PC, consoles, mobile devices, and VR/AR platforms. However, it is especially known for pushing the limits of graphical fidelity across these platforms.

Epic Games offers Unreal Engine under a unique business model where it's free to use for development, with revenue sharing applied when games exceed a specific income threshold. This model is attractive for developers at all levels, from indies experimenting with new game concepts to large studios developing blockbuster titles.

3.3. Godot

Godot Engine [9] is an open-source game engine known for its flexibility and ease of use, supporting both 2D and 3D game development. Unlike Unity and Unreal Engine, which offer free tiers and paid subscriptions based on revenue, Godot is completely free and open-source, with no revenue caps or subscription fees. This makes it particularly

attractive for indie developers and educators who seek full access to game development tools without financial constraints.

Godot features a unique scene system and a scripting language called GDScript, designed to be easy to learn for beginners while offering powerful features for advanced users. This stands in contrast to Unity's use of C# and Unreal Engine's Blueprint system and C++, providing a distinct development experience.

While Godot Engine is a promising competitor to Unity and Unreal Engine, it currently does not offer the same level of support and stability as these established engines. Godot is particularly popular among small indie game developers, owing to its open-source nature and ease of use for projects that may not demand extensive long-term support. Godot is continuously evolving, with contributions aimed at improving its stability and expanding its functionality, making it increasingly suitable for a wider range of projects, including those with more ambitious scopes.

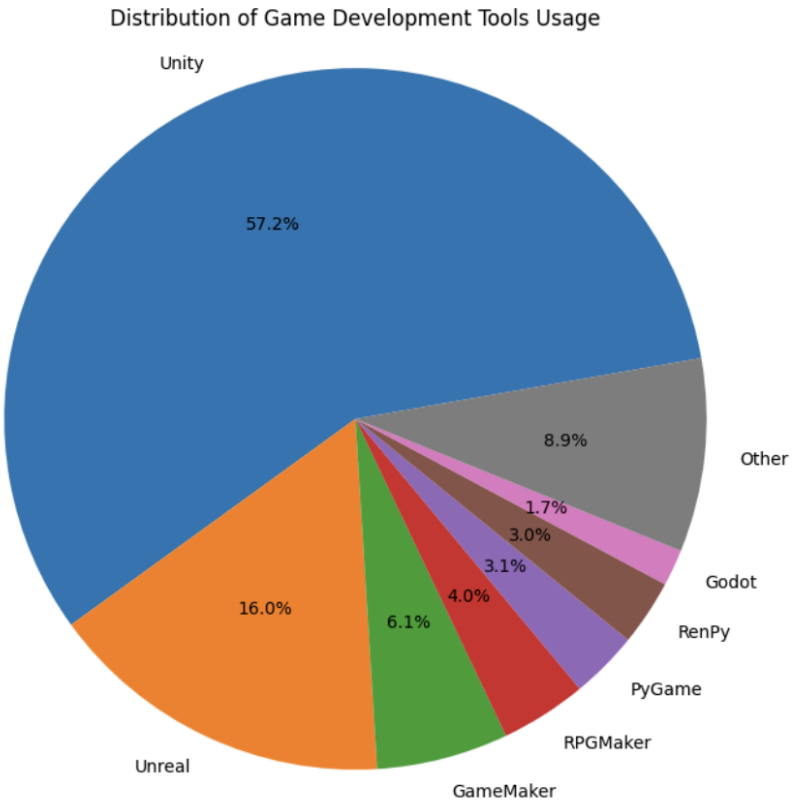

Figure 3.2. Game engines distribution on Steam (Source: own work based on data from steam.db [16])

3.4. Others

Beyond Unity, Unreal Engine, and Godot, several other less popular game engines are worth mentioning, each with unique features and target audiences like CryEngine [10], RPG Maker [11], Construct [12]. All those engines have significantly smaller coverage in created games but still can be widely represented in some niche genres.

3.5. Conclusion

While there are many different public game engines available, the scope of this thesis necessitates the selection of a single engine for in-depth analysis. After careful comparison, a decision was made to focus on the Unity3D game engine. It has most games created on platforms like itch.io [13] or steam [14] and a wide range of genres, platforms and skill levels. The broader reach of Unity makes it easier to recruit developers making it in turn even more popular among people willing to learn it [15].

While the focus is on Unity3D, it's important to acknowledge that many principles of game development are not exclusive to a single engine. The foundational concepts, design patterns, and architectural strategies discussed in this work possess a degree of universality. Although specific implementation details may vary across different engines, the underlying principles remain applicable. With appropriate adaptations, the insights and methodologies outlined in this thesis can be applied to other game development environments, offering a broad relevance to the field of game design and development.

4. Introduction to Unity

As previously mentioned, Unity is a powerful and versatile game engine developed by Unity Technologies [7]. The focus of this chapter will be to introduce the basics of working with Unity, primarily focusing on the scripting and structural foundations that are crucial for understanding the proposed architectures.

Unity's core structure revolves around several key concepts that dictate how developers construct and organize their projects within the engine. These foundational elements include Scenes, Game Objects, and Components, all of which interact to form the backbone of any Unity project.

4.1. Unity Editor

The Unity Editor serves as the interface to the Unity Engine, enabling interaction with all its functionalities. It offers several essential views that facilitate various aspects of game development, including:

- **Hierarchy**: Displays the layout of all objects in the active scenes, organizing them in a tree structure for easy navigation and management.
- **Scene**: Provides a free-camera view of the world, allowing developers to navigate and edit the game environment in real time.
- **Game**: Shows the rendered view of the scene as it would appear to the player, enabling real-time preview and testing.
- **Inspector**: Displays properties of the selected object, allowing for detailed configuration and customization of game objects and their components.
- **Project**: Organizes all project files and assets in a structured manner, making it simple to manage and access resources.
- **Console**: Captures logs, warnings, and errors, assisting in debugging and optimization processes.

These views are crucial for the game development process. While there are many more editor windows available, and developers can create custom ones tailored to their projects, the ones mentioned above are among the most common and important for efficient workflow.

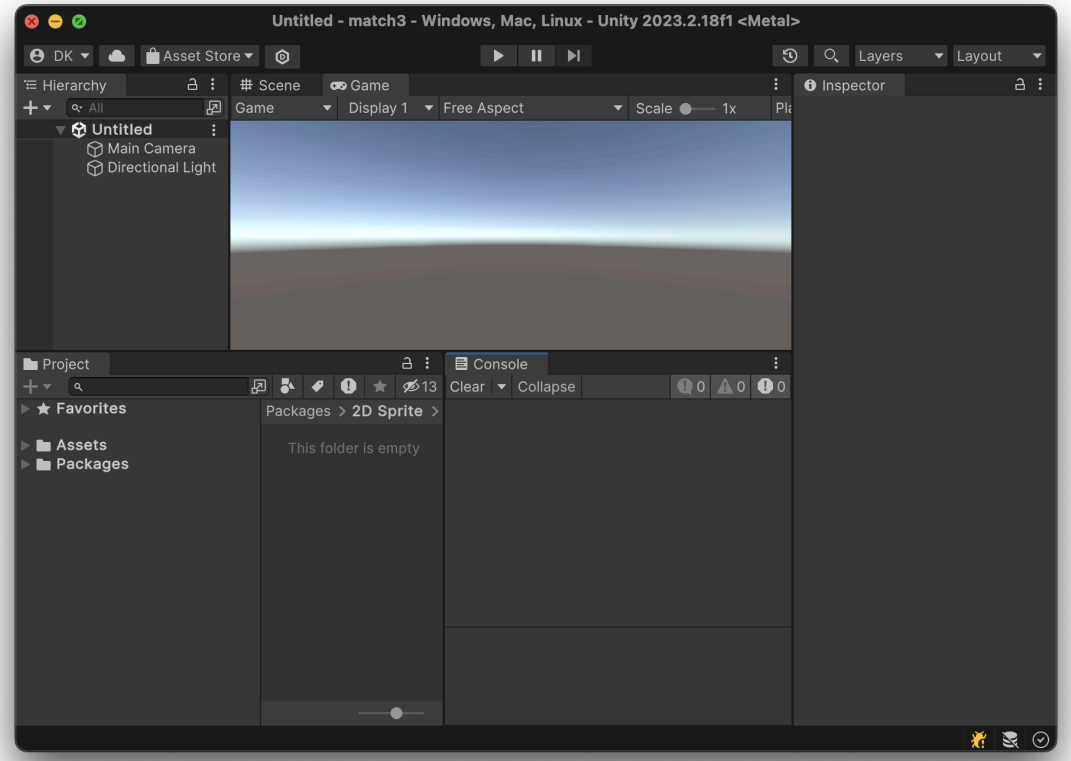

Figure 4.1. Empty project inside Unity Editor with all the views. (Source: own work based on Unity Editor [7])

4.2. Scenes

Scenes provide a high-level organization of the game's elements, functioning as containers that can hold a structured hierarchy of GameObjects. A project must contain at least one scene to run.

A common practice is to utilize each scene as a separate level of a game, where each scene encompasses the relevant environment, characters, and other elements. This approach allows for dividing the game world into manageable segments to optimize loading times and performance. Unity's SceneManager facilitates the easy switching between scenes.

However, while using scenes is generally convenient, improper use can lead to significant issues. A frequent mistake among new developers is embedding the game's business logic directly within a scene. This approach can result in losing the state of managers used in a previous scene when transitioning to a new one. These and other issues associated with

scene management will be discussed in greater detail in the analysis section of the thesis, specifically in Chapter 10.

4.3. GameObjects

In Unity, the GameObject class represents any entity that can exist within a Scene. GameObjects are a foundational concept, serving as the primary building blocks required by every Component in the game. By themselves, GameObjects lack any inherent logic; they function as containers for Components, which define their behavior and characteristics as presented on Figure 4.2.

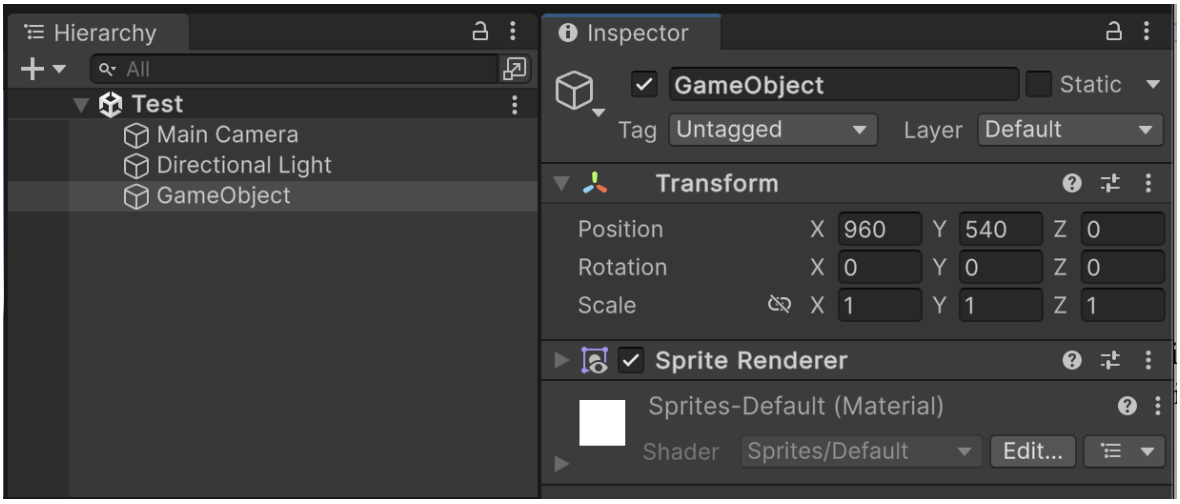

Figure 4.2. "GameObject" on a Scene "Test" with a "Transform" and "SpriteRenderer" (Source: own work)

Every GameObject is intrinsically linked to a Transform Component, indicating that each GameObject must have a Transform. This requirement ensures that every GameObject occupies a specific location in the scene. From a usability perspective, this setup is convenient, as it simplifies the positioning and organization of entities within the game world. However, it also introduces a peculiar constraint: even non-physical entities, such as business logic managers that do not interact with the game's physical space, must be placed within the 3D world-space. This requirement can lead to some non-intuitive design decisions, especially when dealing with abstract game logic that doesn't naturally fit within a spatial context.

4.4. Components

Components are central to Unity's implementation of the Composition over Inheritance design pattern. This approach enables the creation of modular Components that can be reused across different GameObjects. Unity comes equipped with a variety of fundamental Components, such as Transform, Light, Camera, Canvas, Image, Audio Source, and even Video Player.

A particularly crucial type of Component is the MonoBehaviour. Serving as the base class for all custom Components, MonoBehaviour allows developers to define new behaviors and logic. By creating scripts that extend MonoBehaviour, developers can implement customized functionality. For example, a script could be devised to periodically move an object from left to right. This system of Components and GameObjects facilitates a flexible and powerful way to construct game elements, promoting reusability and modularity in game development.

The architecture discussed herein focuses on how to properly structure and organize Components to develop a game effectively. MonoBehaviours form the cornerstone of a programmer's work, serving as the base implementation for custom behavior in Unity, thus becoming crucial in the proper structuring of the game's architecture. Even when employing Dependency Injection [17] frameworks to decouple from Unity's core functionalities and work with pure C# classes, an entry point initiated within a MonoBehaviour is still required. This underscores the pivotal role MonoBehaviours play in bridging the gap between Unity's environment and the broader C# programming landscape, ensuring that regardless of the abstraction level, all game logic eventually integrates with Unity's ecosystem.

4.5. Serialized Fields

Each Component in Unity is designed with limited direct interactions with other objects. By default, every Component can access its own GameObject and Transform, and it has the capability to use the GetComponent() method to search for additional components attached to the same GameObject.

To facilitate more explicit behavior customization of Components, Unity introduced the concept of Serialized Fields. Serialized Fields allow developers to link objects to a Component directly from the Unity Editor's Inspector view. For instance, an Image component might have a Serialized Field where a developer can assign the Sprite it should display. This feature enhances the usability and flexibility of Components, enabling developers to configure their properties and relationships within the Unity Editor, streamlining the development process.

4.6. Example of usage

To illustrate how this entire system functions, an example is most effective. Let's say we want to add a cube object to the scene view. The steps would be as follows:

- Add a new **Scene**. Unity automatically adds "Camera" and "Directional Light" objects, which are essential for rendering the scene.
- Create a new **GameObject** and name it "Cube". Unity automatically attaches a "Transform" **Component**, necessary for positioning the object within the scene.
- Add a "Mesh Filter" **Component**.
- In the "Mesh Filter" **Serialized Field** titled "Mesh," select Unity’s default "Cube" Mesh.
- Add a "Mesh Renderer" **Component**.
- In the "Mesh Renderer" **Serialized Field** "Materials," choose Unity’s default "Lit" Material.

Following these steps, the Game view should display the rendered cube as in Figure 4.3. This process showcases Unity's component-based architecture, emphasizing the ease of assembling and configuring game elements through the Unity Editor.

Even a straightforward task like rendering a placeholder cube might seem complex at first glance. This example relies on Unity's default assets, such as Materials, Meshes, Camera, and Light. If one were to add all these elements manually, the process could easily extend to twenty additional steps.

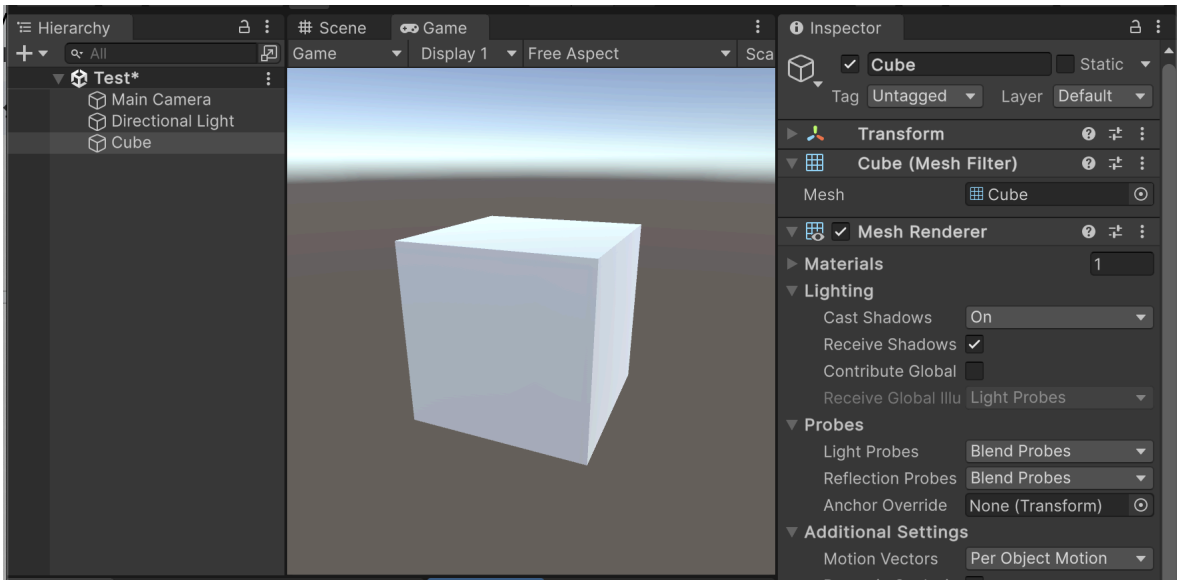

Figure 4.3. Adding a 3D Cube to a scene in Unity (Source: own work)

This scenario highlights Unity's structured approach to game development. Beginning with a **Scene**, developers add **GameObjects**, to which **Components** are attached and their **Serialized Fields** are set. These foundational elements are integral to any game developed in Unity. Consequently, any architecture within Unity must conform to these basic building blocks that constitute the engine's core framework.

5. Categorization of games architectural needs

The gaming industry is vast and multifaceted, encompassing a wide range of offerings from simple browser games and indie PC titles to AAA productions and constantly updated mobile games [3]. The type of game significantly influences the architectural approach one should adopt. Each game genre prioritizes different aspects of game development, while certain elements may be less relevant or entirely absent in others. This diversity mandates a tailored approach to architecture, ensuring that the chosen framework aligns with the specific needs and focus areas of the game being developed.

5.1. Based on platform

The gaming landscape extends beyond just the games themselves; a platform for playing these games is essential. While it might seem that targeting the PC or console market alone would simplify game development, in reality, a significant portion of gaming revenue—over half—originates from the mobile platform, encompassing Android and iOS devices.

The advent of cross-platform game engines like Unity is diminishing the traditional emphasis on platform-specific development. While there are still games designed for specific platforms, many publishers aim to maximize their audience by releasing games across multiple platforms simultaneously.

This approach does blur the lines between games designed for specific platforms, but it also introduces complexities for developers, who must ensure compatibility across diverse systems. The distinction between game categories remains evident, particularly in the AAA segment, where the size and quality of games often make them unsuitable for mobile devices. Conversely, mobile games are typically designed for short, accessible play sessions, not requiring the startup time of a PC or console. However, cloud computing solutions like Nvidia GeForce Now [18] and Android emulators such as BlueStacks [19] are bridging these gaps, allowing AAA games to be played on mobile devices and mobile games on PCs.

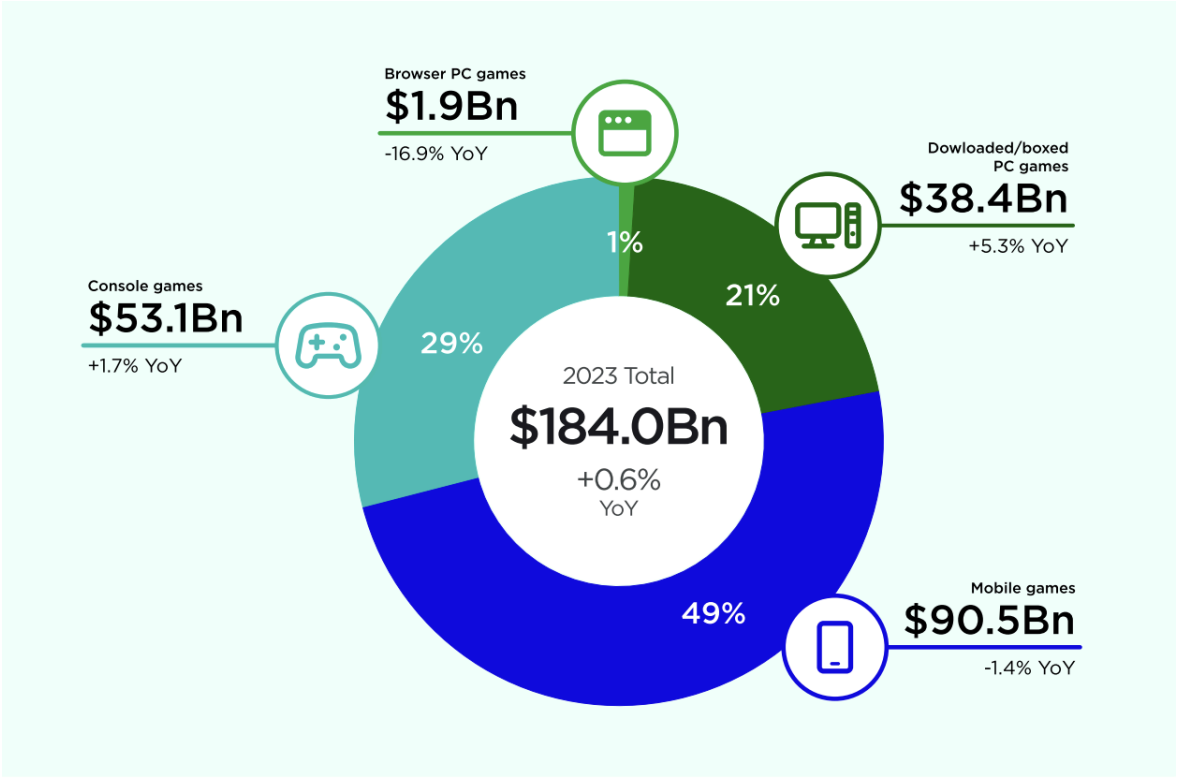

Figure 5.1. Games platforms by revenue and trends for 2023 (Source: NewZoo report for 2024, [3])

From an architectural perspective, the choice of platform is becoming less critical compared to factors like performance optimization and rendering techniques. Developers now focus more on creating versatile architectures that can adapt to the requirements of different platforms, ensuring a consistent gaming experience across the board.

5.2. Based on project size

When categorizing games by size, the architectural requirements become increasingly apparent. Small indie games might opt for a simplified structure by primarily utilizing solutions provided by Unity, though this approach is generally not recommended. In contrast, larger games cannot afford to overlook the importance of deliberate architectural design.

Games can be differentiated by size based on the sheer number of features and systems they incorporate, not the physical size on disk. The greater the complexity and quantity of these systems, the more connections between them will likely be established. Without proper organization of these connections, developers risk creating tightly coupled systems, which can lead to an increase in potential bugs and maintenance challenges.

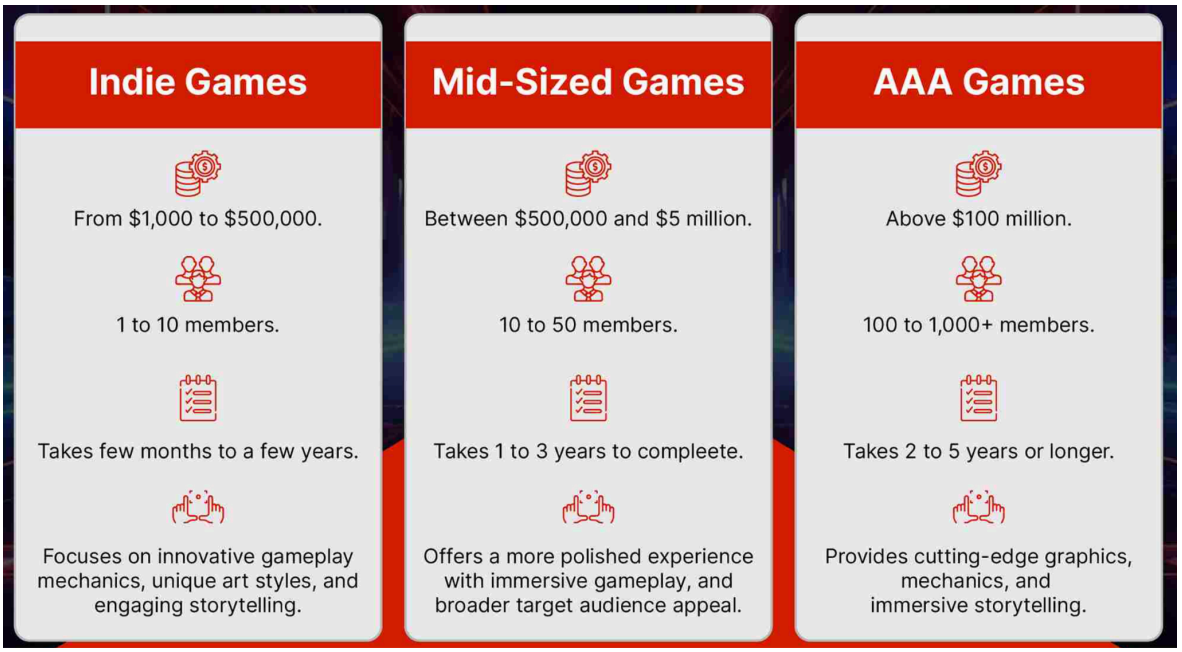

Figure 5.2. Project size categorization based on project size (Source: InvoGames, How much does it cost to make a video game? [50])

5.3. Based on lifecycle

The application lifecycle is a crucial determinant of the architectural approach required for game development. This lifecycle encompasses the various stages a game undergoes, from the initial concept to its release and subsequent maintenance. The complexity of the architecture needed is often inversely related to the game's lifecycle [20] length.

At one end of the spectrum are games developed during game jams, events typically lasting around 48 hours with the objective to deliver a complete game within this timeframe. The lifecycle of such games rarely extends beyond the duration of the jam, and post-event maintenance is uncommon. Consequently, the architecture for these projects tends to be straightforward and focused on rapid development.

On the other end are the so-called evergreen titles, which continue to be developed and enhanced many years post-release. An example of an evergreen title is Candy Crush Saga [21], launched on April 12, 2012. It has consistently been updated with new features over the years, maintaining its status as one of the top-earning games globally. Sustaining this level of maintenance and ongoing development necessitates a well-planned and flexible architecture.

Positioned between these extremes are various other types of games. AAA titles, for instance, may require several years to develop and might receive updates for many years following their release. Similarly, even smaller indie games often undergo significant post-launch enhancements, sometimes doubling in content and functionality. Achieving such post-launch growth and ensuring timely updates hinge on the foundational architecture of the project, underscoring the importance of thoughtful architectural planning in game development.

5.4. Based on genre

Another crucial aspect influencing architecture is the game's genre [22]. While factors such as platform, size, and lifecycle may dictate the robustness of the architecture, the genre determines its specific requirements. Strategy or simulation games, for example, necessitate patterns that support the simultaneous display of thousands of objects. Action games demand high performance and advanced visual effects, whereas UI-heavy games require efficient management of displayed windows.

This distinction underscores how game development diverges from traditional software development. While some design patterns from software engineering can be adapted for game development—such as MVC or MVVM [23] for UI-heavy games, DI [24] for component interaction, or Singletons for global access—there are patterns unique to game development. These include the Entity Component System (ECS) [25] for flexible game object composition and the Game Loop for driving game updates and rendering. These game-specific patterns address the unique challenges of game development and will be explored in more detail in Chapters 6 and 7.

6. Software architecture

The availability of tools such as IDEs, compilers, debuggers, online resources, and even AI has empowered developers to create software swiftly. However, making a program work is only part of achieving success. The scale and complexity of software impose a need for ongoing maintenance and necessitate frequent interaction with existing code. The more challenging aspect of software engineering often lies in this latter part: maintenance.

6.1. Architectural need justification

Creating software involves two critical aspects: fulfilling the business goal and maintaining the integrity of the project's architecture. Focusing exclusively on one aspect over the other is not advisable. A project overly concentrated on architecture may fail to succeed if it doesn't address a real-world problem, rendering it unnecessary. Conversely, a project solely driven by its business goal might achieve short-term success but faces sustainability challenges over time [27].

Case studies reveal that projects often encounter a dilemma where, despite an increase in the development team's size, the rate at which new features are introduced continues to decelerate [28]. This scenario indicates that while the costs of building and maintaining the software escalate, the additional expenses predominantly contribute to resolving issues. Such situations can highlight several problems that were not adequately addressed during the development process.

Initially, development might commence hastily with a minimal number of developers, likely neglecting proper code structuring, design, or architectural considerations. Subsequently, following the project's success, no efforts are made to rectify this oversight, with all attention diverted to rolling out new functionalities. Although it's crucial to keep the software's business objectives and problem-solving capabilities in mind, this thesis will primarily concentrate on architectural considerations.

6.2. Software architecture outside of games

Software architecture involves design decisions that shape a project's structure and behavior, enabling teams to ensure the system meets essential requirements like

modifiability, availability, and security [29]. It's beneficial across various stages of system development, whether creating new systems, enhancing existing ones, or modernizing outdated ones. Effective architecture not only organizes the project for all stakeholders but also facilitates time and cost efficiencies, and identifies potential issues early in the development process.

Unlike physical architecture, from which its name derives, software architecture exhibits a "recursive" nature. This implies that systems with a defined architecture are built using lower-level components, each with their own architectures, cascading down to the machine code level. For instance, high-level architectural patterns such as Layered Architecture, Microservices Architecture, or Client-Server Architecture can be implemented in conjunction with one another. A Client-Server system might employ a Layered Architecture for its client side and a Microservices Architecture for its server, illustrating the flexible and nested nature of software architectural decisions.

6.3. Architecture of games

Given this thesis's focus on game development, particularly within the context of Unity, delving into architectural patterns or frameworks outside of gaming might lead to confusion. Game development operates distinctly from traditional software programming environments [30]. In conventional software development, teams often comprise roles centered around product and project management, such as product owners, project managers, analysts, designers, developers, and testers. A User Experience designer might also be part of the team if the project demands visual design work.

Conversely, game development encompasses these foundational roles while also integrating numerous specialized positions due to its emphasis on graphics and gameplay mechanics. A typical game development team includes 2D and 3D graphic designers, animators, game designers, and live-ops designers. Additionally, there are often roles that, while not part of the core team, must collaborate closely with it, such as those in localization, support, audio, and other areas, reflecting the multidisciplinary nature of game development. This is also reflected in the game architecture, where developers are tasked not only with creating a scalable and maintainable framework but also with enabling non-coders to modify and contribute new content to the game. If every aspect of the game required direct programmer intervention for addition and configuration, development

timelines would significantly extend. To address this, game engines offer built-in editors equipped with inspectors and various tools that must be integrated into the architecture. These features are essential for streamlining the development process, allowing for a broader range of team members to directly influence game content and mechanics.

7. Case study architecture analysis

Algorithms are detailed sequences of steps or instructions designed to perform a specific task and can be analyzed and compared based on time or space complexity. In contrast, architecture or design patterns act as wireframes or blueprints for solving common challenges in software development. As a result, it's difficult to analyze architecture without considering its implementation [31]. To address this issue in the thesis, a hands-on approach will be employed: **a single application**, specifically a game, will be **conceptualized and then developed using two distinct architectural strategies**. Minimizing variables is crucial for conducting a meaningful analysis [32]. By simplifying the factors involved, this approach enables a focused comparison of how different architectural patterns impact the development process, as well as the final product's performance, maintainability, and scalability.

The drawback of this analysis is that it operates under certain assumptions about the problem, necessitating that readers view the conclusions not as definitive answers but as guidelines for identifying a suitable architectural approach for a specific game. If a single architecture could address all scenarios effectively, there would be no need for multiple solutions or analysis. Reality, however, presents a variety of unique use cases, each demanding a customized solution. This was discussed in more detail in Chapter 5. Here, the focus will narrow to a single case study to minimize complexity and provide a clear illustration of the principles involved.

7.1. Game design

Given that the thesis primarily analyzes architectural patterns, the underlying game design will be simplified to concentrate on these aspects. Therefore, the core game mechanic chosen for study is that of a **Match-3 game** [34].

Match-3 games involve swapping adjacent items (commonly gems, stones, or fruits) on a grid to form a line of three or more of the same item, which then clears those items from the board and allows new ones to fall into their place [35]. The objective often includes achieving a certain score, completing tasks within a limited number of moves, or overcoming specific challenges. This mechanic's simplicity makes it an ideal candidate for

focusing on architectural decisions without the added complexity of intricate game design elements.

7.2. Game specification

Starting with a common core mechanic, such as that of a Match-3 game, allows for the integration of new functionalities to forge more sophisticated solutions [36]. The aim of incorporating additional systems is to enrich the dependency graph and rigorously evaluate the architecture's capacity to handle complexity. A Match-3 game provides a solid foundation for extensive extensibility. Systems like achievements, player statistics, level progression and selection, and rankings are prime examples of features that can enhance the game's depth. Implementing these systems, even in a basic or mock form, facilitates a thorough assessment of the underlying architecture, ensuring it can support both the core gameplay and the intricate web of interconnected systems that constitute the meta layer. This approach not only tests the architecture’s robustness but also its flexibility and scalability, crucial attributes for accommodating future expansions or modifications [37].

Figure 7.1. Example of Match-3 game, Candy Crush Saga (Source: Candy Crush Saga on Google Play Store [33])

Examples of those systems can be:

- **Achievements**: This system tracks and rewards players for completing specific milestones or challenges within the game. For instance, completing a level without losing a life or achieving a certain score threshold.
- **Statistics**: Captures and analyzes player behavior and performance metrics, such as average scores, most-used power-ups, or levels with the highest failure rates. This data can inform game balance adjustments and personalized player feedback.
- **Level Selection and Progression**: Allows players to navigate through different stages or levels of the game. This system can include a map or menu where players can select levels, view their progress, and unlock new levels by completing the previous ones.
- **Rankings**: Compares player scores or achievements on a leaderboard, fostering competition and engagement by displaying player rankings globally or among friends.

Visualizing these systems and their interactions can be facilitated through a dependency graph, a diagram that illustrates how different game components depend on each other. For example, the level progression system might depend on the statistics system to unlock levels based on certain performance metrics, and the achievements system might rely on the rankings system to award achievements for reaching specific ranks.

The most critical aspect of this work is that these systems will **simulate a real-life scenario** of such a game, where design goals evolve and adapt over time. This simulation allows for the modification and addition of new features while minimizing the creation of bugs.

7.3. Game architectures

A pivotal, if not the most critical, decision in this work is the selection of the architectural approach. The project will be realized through two distinct methodologies: a basic implementation and a more sophisticated, structured architecture. The basic implementation will entail creating the game in Unity without leveraging external packages or employing advanced techniques. In contrast, the advanced solution will be developed using robust design patterns, adhering closely to established principles and standards.

For a Match-3 game with a complex meta layer involving systems like stats, progression, and rankings, great benefit can be gained from an architecture that supports modularity, scalability, and ease of maintenance. Architectural patterns that would help are for example:

- Entity Component System (ECS) [25],
- Model-View-Controller (MVC) pattern [23] and
- Dependency Injection (DI) [24].

Given the complexity and the variety of systems proposed for the Match-3 game, a combination of MVC for managing game states and UI interactions, along with ECS for handling game logic and entity interactions, could provide a robust foundation. This hybrid approach leverages the strengths of both architectures, offering scalability and maintainability. Additionally, integrating Dependency Injection can further improve modularity and testability, particularly as the game evolves. The finer details of the architecture as well as examples of usages in the next chapters.

8. Architecture evaluation metrics

Given the project's assumptions, technology choices, and detailed specifics, it's equally crucial to carefully select and understand the metrics for measuring and comparing the two proposed approaches. Choosing inappropriate metrics could negate all the effort invested [31].

When evaluating a software artifact, such as a game source code, it is possible to assess attributes related to any representation or description of a problem or solution, which typically fall into two primary categories: structure and behavior. Additionally, attributes pertaining to the development process or methodology can also be measured, focusing on aspects such as quantity (size) and complexity. [51]

8.1. Areas of measurement

In game development, the emphasis placed on different focus areas can vary depending on the specific requirements and context of each project. While the significance of these areas may differ across scenarios, their core objectives generally remain consistent across diverse projects. Important areas such as user experience, art style, and animations will not be covered in this discussion. Instead, the focus will be on areas that are particularly relevant to the role of a Unity Developer and form a crucial part of the programmer's responsibilities. Below is a detailed description of each selected focus area in the context of this thesis:

1. **Performance:** Addresses the efficiency and smoothness with which the game operates on targeted devices. Performance is critical as it directly impacts the player's experience and satisfaction, influencing factors such as frame rates, load times, and overall responsiveness.
2. **Development:** Focuses on the creation of the software, including all necessary components for its functionality. This area deals with topics such as the speed of development, scalability of the software, and how functionalities are implemented to meet project requirements.
3. **Quality:** Encompasses the overall reliability and the game's ability to operate without defects. It focuses on the quality of the code and the robustness of solutions, emphasizing how resistant to errors the software is. This includes rigorous testing procedures and effective bug management strategies.

4. **Cooperation:** Highlights the collaborative nature of game development, which involves multiple disciplines beyond programming. This focus area examines how aspects of game design can be delegated to non-programmers, such as designers and content creators, and how these roles can modify elements without always relying on developer intervention.

These areas are integral to the role of a Unity Developer and play a significant part in the successful development and maintenance of a game. Each area is targeted to ensure a well-rounded approach to building, testing, and improving game software within a collaborative development environment.

8.2. Proposed measures

For each of the areas discussed above some metrics are needed. Grouping of proposed measures for each area that will be used to compare the implementations:

- **Performance Metrics**
    - **Frame Rate and Frame Time:** Measure smoothness and responsiveness of gameplay. Higher frame rates and lower frame times indicate better performance.
    - **Load Time:** Evaluates how quickly game scenes and assets load. Faster load times contribute to better user satisfaction and less waiting.
    - **CPU, GPU and Memory Usage:** Important for determining the efficiency of the game. Lower CPU and memory usage signify a more optimized game that can run on a wider range of hardware.
- **Development Metrics**
    - **Development Time:** The total time taken from project or feature start to completion. Shorter development times can indicate more flexible architecture or ease of use.
    - **Lines of Code (LOC) Count**: this measure in separation is not that useful but in a comparative analysis can help determine the size of solution as well as reusability. In this case the focus will be on the Logical Lines Of Code (LLOC) which don't include empty lines or brackets or imports and only count the singular logical operations lines.

    - **Dependency Management Metrics**: proposed by R. C. Martin in Clean Architecture [27] metrics for architecture dependencies. It includes the measure of component stability and abstractness.
    - **Number of Reusable Components:** Higher numbers suggest a more modular and scalable architecture, allowing for easier updates and expansions.
    - **Modules Cohesion**: Weak cohesion indicates poor design. Metrics defined by Chidamber and Kemerer like LCOM or CBO [54].
- **Quality Metrics**
    - **Bug Count and Bug Density:** Measures the number of bugs relative to the size of the codebase. Lower counts and densities indicate higher code quality and stability.
    - **Bug Fixing Time:** Time dedicated to resolving bugs. Compared to the development time can indicate maintainability of the project.
- **Cooperation Metrics**
    - **Design Flexibility**: Evaluate the number of design elements configurable without code access, such as UI components like icons or animation speeds.
    - **Inspector content creation**: Track functionalities that allow non-programmers to create or modify content directly through the Unity Inspector, enhancing collaborative potential.

8.3. ISO/IEC norms

The ISO/IEC standards related to software and systems engineering, developed by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), encompass a series of international guidelines and norms. These standards are designed to ensure quality, reliability, and efficiency throughout the development, maintenance, and documentation of software and systems. Utilizing ISO/IEC norms is essential for achieving consistency and standardization in measurements across different projects. Within the context of a thesis, these norms enhance credibility and trust by providing a recognized framework for evaluation.

These norms address a broad spectrum of aspects such as functionality, reliability, usability, efficiency, maintainability, and security. Particularly relevant to this thesis are the ISO/IEC 25000 series, known as "Software product Quality Requirements and Evaluation

(SQuaRE)." Of special interest is ISO/IEC 25023, "Measurement of system and software product quality," [38] which offers specific measures for assessing the characteristics defined in the product quality model.

The ISO/IEC measures we will be focusing most on are:

1. MMD-G-1 for Modifiability
2. MMD-G-2 for Modifiability
3. UOP-G-1 for Operability
4. MRE-S-1 for Reusability
5. MMO-S-1 for Modifiability
6. PRE-S-1 for Replaceability

Measurements from the categories outlined above will be systematically collected both during and after each phase of the implementation process. This approach ensures that data is captured in real-time, providing insights into the dynamics of each development stage, as well as after completion to assess final outcomes. The collected data will then be meticulously analyzed and compared across both versions of the project. This comparative analysis will help identify which implementation is more efficient, robust, and better suited to meet the project's objectives in terms of performance, quality, scalability, maintainability, and flexibility. Such an evaluation will be instrumental in determining the strengths and weaknesses of each architectural approach, guiding future decisions and improvements in the development process.

8.4. Analysis software

For the collecting of measures a software solution will be used. The first and the most important one will be Jetbrains Rider IDE [52] which provides many tools for software analysis as well as access to plugins like “Qodana” for static analysis or “CognitiveComplexity” or the complexity metrics. While Rider is a widely used IDE for Unity game development it also serves as a great tool for code statistics and analysis.

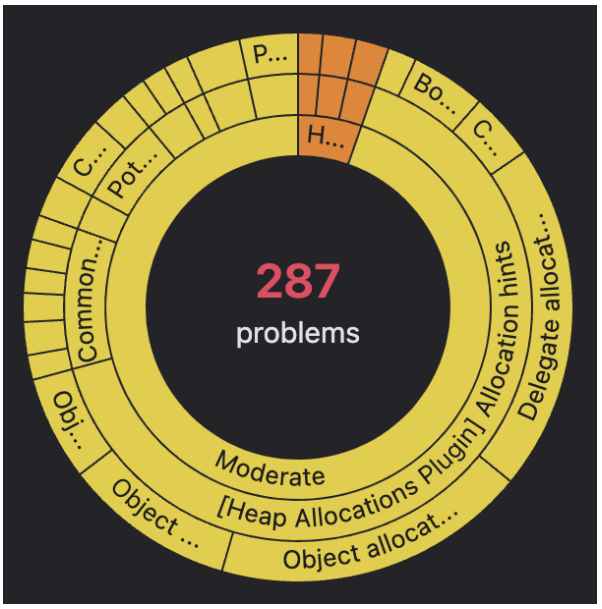

Figure 8.1. Jetbrains Qodana analysis results. It can be used to investigate issues with code. (Source: own work with use of Jetbrains Quodana [52])

The next tool that was meant to be used for collecting measures was SonarQube [53] from SonarSource. Unfortunately after many tries this software was not supporting Unity projects with multiple assembly definitions on which the architecture was relying heavily.

Instead of SonarQube a better suited program was used: NDepend [55], more focused on C# projects and solutions. A commercial tool that provides a comprehensive suite of code metrics, including but not limited to those defined by Chidamber and Kemerer. It offers detailed reports and visualizations of software metrics.

8.5. Metrics analysis

The raw measurements alone may not yield actionable insights. To effectively differentiate between solutions, a metric analysis is essential. Each category of product aspects can be weighted according to its relevance to the specific use case. For instance, a small solo developer's game might prioritize development speed and solution simplicity over scalability or maintainability. These latter aspects are typically more crucial for large projects, where hundreds of developers, as well as non-developers, must collaborate efficiently.

9. Proposed game systems

For a game to be successful, numerous systems must interact seamlessly to ensure proper functionality. The complexity and number of these systems largely depend on the game's genre and target audience. Furthermore, each component of the implementation can be treated as a separate case study, allowing for a more detailed and granular comparison. This approach goes beyond merely comparing the final products; it enables ongoing monitoring and analysis throughout the development process. By evaluating each system individually and in relation to others, we can identify strengths, weaknesses, and areas for improvement in real-time, enhancing the overall quality and effectiveness of the game development cycle.

In this chapter, we will select and explore the most critical systems that contribute to the proposed game, discuss their implementation, and lay the groundwork for the experiments that will be analyzed in subsequent chapters. This foundation will enable a thorough understanding of how each system impacts the overall functionality and appeal of the game, setting the stage for more detailed evaluations to follow.

9.1. Match-3 mechanic

As mentioned in Chapter 7.1 the basic game design mechanic will be Match-3. Creating a successful Match-3 game involves a blend of engaging gameplay mechanics, intuitive user interfaces, captivating visuals, and robust programming. Essential components and features you need to include in a Match-3 game [35]:

- **Board and Chips**: The board serves as the base on which the game is played, and the chips or tiles are the elements players manipulate. These need to be dynamically generated and placed on the board.
- **Tile Matching**: The fundamental mechanic where players swap adjacent tiles to create a line of three or more identical items, which then clear from the board.
- **Objectives**: Each level should have specific goals, such as reaching a score, clearing specific types of tiles, or dropping items to the bottom of the board.
- **Power-ups and Combos**: Include special tiles that clear larger sections of the board, such as bombs, line clears, or color wipes, which players can create by matching four or more tiles or forming specific patterns. Those can be given as a reward for harder matches or used as a monetization basis.

- **Move Limits/Time Limits**: Some levels could be time-based, requiring players to achieve objectives within a given time, while others might limit the number of moves.

While seemingly simple, the Match-3 mechanic offers extensive flexibility. Adjusting the chip generator settings, modifying the board layout, and introducing new elements or level objectives can create thousands of unique challenges. Designing more difficult levels also allows for the incorporation of power-ups, which can serve as a primary monetization element. For instance, in the Match-3 game "Flip This House," [39] which I worked on professionally, this mechanic paved the way for more loosely coupled elements such as house decoration and a narrative backdrop. However, the most significant advantage of adopting such a versatile mechanic lies in the diverse scenarios it enables. Level selection and progression provide excellent opportunities to evaluate the flexibility of the game's architecture.

9.2. User Interface

The User Interface (UI) serves as the primary interaction layer between the game and the player, functioning as a communication tool through which players interact with the game and receive feedback on their actions. While UI might be a secondary feature in some games, others may rely heavily on it. In this thesis, the emphasis will be placed on UI implementation, rather than User Experience (UX) or visual aesthetics.

Unity offers robust UI capabilities through its Canvas component, which allows for the display of UI elements such as images, texts, buttons, and layout components. This creates a foundational toolset for UI development. However, the flexibility in how UI is implemented can vary according to developers' needs, presenting both advantages and potential challenges.

A simple approach involves adding UI elements directly into the scene and toggling their active status as needed. While straightforward, this method can lead to issues such as increased scene loading times, potential merge conflicts, and decreased reusability. For smaller projects, these drawbacks may be negligible, but they can introduce significant risks, such as tight coupling and dependency management issues, which may lead to bugs even in small-scale games.

A more structured method involves using a window stack architecture, with Unity's Prefabs serving as the basis for each window. These can be loaded from disk or a remote server. This approach, while more organized, does not address issues of tight coupling, which may necessitate a more complex architectural pattern such as MVC (Model-View-Controller) or MVVM (Model-View-ViewModel) [23] to separate the graphical interface, data, and business logic effectively.

The UI in this project will be developed as a basic, scalable system that can accommodate additional functionalities over time. The game will initially feature a simple UI, with complexity increasing as more windows and features are integrated, testing the system's scalability and flexibility. Optional functionalities to be explored include the ability to test different views for the same logic, sourcing views from external locations, and dynamically changing scenes within the game. This approach will allow for an in-depth analysis of the UI implementation's effectiveness in handling expanding game complexity.

9.3. Level selection and progression

In content-driven games like Match-3, it is crucial to maintain a steady progression of levels that increase in difficulty. Players should be gradually introduced to new mechanics, objectives, and game elements to enhance their experience. To meet these requirements, the game should feature a level selection system where players can choose from a list of unlocked levels, and a level progression system where successfully completing one level unlocks the next. Additionally, to reduce the burden on level designers, mechanisms to introduce level repetition and progression barriers should be implemented. One common approach is a "lives system," where each loss results in a lost heart that replenishes over a specified period. Additionally, implementing a star system can significantly enhance content density by adding replayability to levels as players strive to achieve the maximum number of stars.

The inclusion of progression necessitates a saving mechanism to ensure that when players return to the game, they do not have to start over. The game should remember how many levels the player has completed and the number of points they have earned up to that point. This functionality not only preserves player progress but also enhances player engagement and retention by providing a seamless continuation of gameplay across sessions.

9.4. Statistics and achievements

To fully integrate the internal elements of the Match-3 core with the surrounding meta layer, incorporating an achievements system would be highly effective. This system could track and calculate player statistics, such as the ratio of moves to matches or the overall win rate, by collecting data from individual levels and aggregating it into meaningful statistics. Achievements are a classic feature that appeals to completionist players, providing clear goals and rewarding players for reaching specific milestones.

Each achievement could be associated with specific in-game rewards, such as currencies or power-ups, further motivating players to engage with the game and strive to complete these challenges. Additionally, the achievements system could be expanded to include tiered rewards that increase in value as the challenges become more difficult, enhancing player retention and encouraging continuous gameplay. Integrating this system not only enriches the player's experience but also adds depth to the game, making it more engaging and rewarding.

9.5. Rankings

To incentivize mechanics like scoring and stars and to introduce a competitive layer, implementing a ranking system is an excellent feature. In such a system, all players are evaluated based on their performance statistics from a previous period and assigned a place in the rankings. This functionality effectively drives user engagement and improves Key Performance Indicators (KPIs) since competition is often a strong motivator in the gaming industry [46].

Rankings can take various forms. They can be global, allowing worldwide competition; national, for players within the same country; or friends-only, where players can invite their friends to compete directly against each other, enhancing the social aspect of the game. Rankings can also be structured periodically, such as in a league system where the top players advance to a higher division, and those at the bottom may drop to a lower league. This system groups players with similar skill levels, fostering fair and stimulating competition.

Additionally, rankings can be integrated with a reward system, where top-ranking players receive rewards such as in-game currency or consumables like power-ups. This not only rewards skilled play but also encourages continuous participation and progression within the game.

9.6. Other functionalities

The genre of games is complex and expansive, offering myriad possibilities for adding functionalities that can complicate a game's structure and broaden its scope. However, for the purposes of this thesis, the systems and features already proposed delve deeply enough into the complexities of game architecture to enable a thorough comparison and analysis of different architectural approaches.

Here are some ideas for additional features that could be considered for future expansion or subsequent studies [36]:

- Power-ups - consumable helpers for the Match-3.
- Narrative storyline - characters with stories and dialogues creating background for the game.
- Meta-game elements - elements like home or garden renovation and decoration.
- In-game store - monetization element that can offer new levels or power-ups.
- Live-operations - in-game events, seasons and challenges that extend the basic game loop.
- Business intelligence - gathering of data for improving the content.
- Player profile - option to lookup other player profiles and socialize.
- Clans and clubs - social elements to connect players and engage them by additional club content.

And many more ideas from already existing games or even new mechanics that are not yet present in the games.

10. Case studies implementation

Utilizing the "Multiple Case Studies" methodology [40], this thesis will prepare and analyze several experiment scenarios to explore a broad spectrum of potential applications. Due to the limited scope of this thesis, only a select few cases will be examined. Nonetheless, these cases are anticipated to provide a broader perspective on potential applications and a holistic understanding of the subject matter.

10.1. Implementation environment

For this project, both implementations will utilize Unity version 2023.2.18f1 on a MacBook Air M1 with 16GB RAM. Each version of the game will be profiled and tested using a standalone build on the same machine. For clarity, the implementations will be referred to as P1 for the first implementation and P2 for the second.

The first implementation (P1) will utilize only the most basic features of Unity, minimizing the use of external packages. The code structure will be straightforward, resembling entry-level code found in online resources and tutorials.

The second implementation (P2) will adopt a more sophisticated, state-of-the-art approach, similar to the architectures used in large-scale live projects within the industry. P2 will be set up with external packages that enhance its architectural components, including:

- **Dependency Injection** (DI): Utilizing VContainer [41] to manage dependencies more efficiently.
- **Model-View-ViewModel** (MVVM): Creating simple window management to facilitate a clean separation of the UI from the business logic.

Both implementations will be developed within the same Unity project but will be separated by assembly definitions, ensuring that there are no references between them, which makes them inaccessible to each other. Each implementation will have its own starting scene and startup script. Both versions will share access to the same base libraries and a common code assembly definition, which will not be included in the analysis to avoid skewing the results.

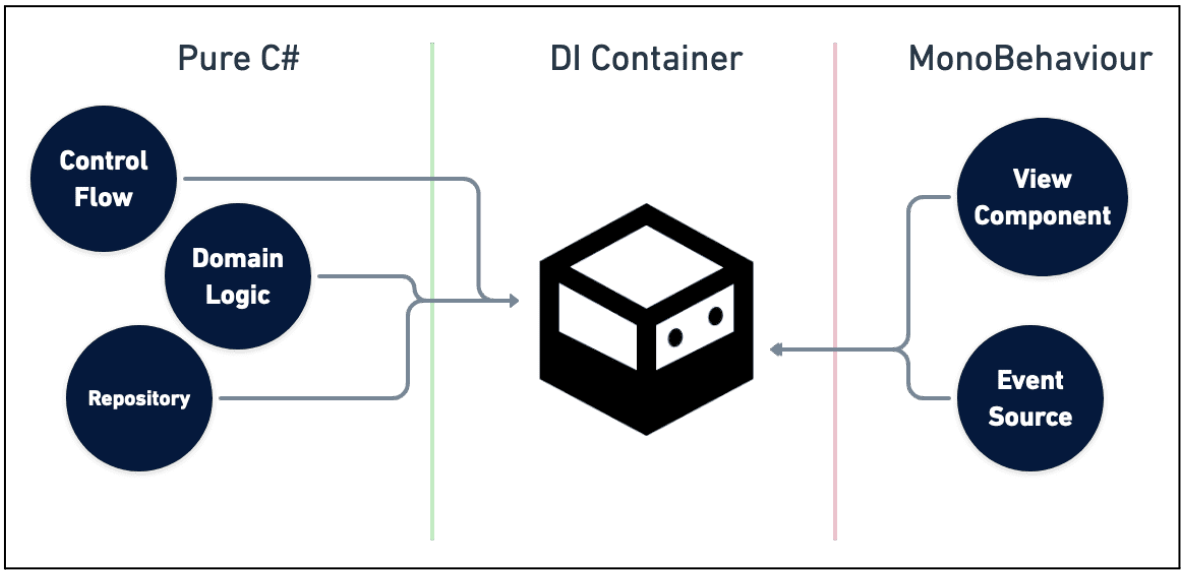

Figure 10.1. DI idea in Unity (Source: VContainer [41])

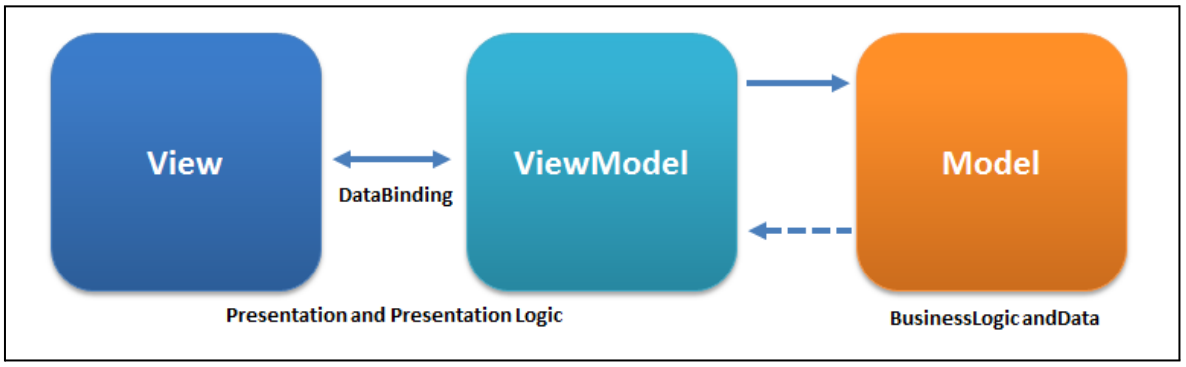

Figure 10.2. Overview of the ModelViewViewModel architecture (Source: By Ugaya40 - Own work)

To maintain the integrity of the comparative analysis, each project will contain only the code that is unique to that version. This approach ensures that duplicating common functionalities does not influence the outcome of the results. This setup helps to isolate the effects of the architectural differences between the two implementations. Projects will also share the assets like textures, materials, fonts and animations.

10.2. List of scenarios

The implementation process will be structured into distinct steps, each representing a small increment in the development of the game. These steps, or case studies, mirror the iterative nature of game development, which can range from a solo developer creating a small game

to a large team of several hundred people executing the design specifications of an enterprise-level project. This approach effectively simulates a real-life environment where the full scope of the project and its requirements are not initially known and are incrementally incorporated into the product based on evolving needs and priorities.

For guidance during implementation, the description from Chapter 9 will be utilized as a Game Design Document (GDD) [42], which will serve as the primary resource outlining the intended structure and functionality of the game.

The comprehensive list of scenarios includes:

- S1: Setup project
- S2: Add base Match-3 Mechanic
- S3: Add Scoring for Matches
- S4: Add Matched Chips Count Objective
- S5: Add Move Limit
- S6: Add Gameplay HUD
- S7: Add Level List
- S8: Add Level Selection
- S9: Add Player Progression
- S10: Add Player Statistics
- S11: Add Achievements
- S12: Add Simple Ranking

Each of these steps provides a succinct overview of the tasks that need to be accomplished. Detailed explanations and implementation specifics for each step will be elaborated on throughout this chapter. As the implementation of the project progresses, each scenario will be thoroughly introduced and analyzed. These scenarios will each reveal new aspects of the project, starting from the initial setup and basic systems and extending to complex metagame features. Every scenario will be developed in two different versions and subsequently analyzed to compare their functionalities and architectural differences.

Following the completion of the implementation phase, a testing phase will commence. The Quality Assurance (QA) process will be employed to identify any bugs and verify that both versions deliver the same functionalities, despite their different implementation approaches. This ensures that, although the versions may be built differently, they will

meet the same requirements and provide a consistent experience from the end-user’s perspective.

10.3. Project setup

The first scenario is **S1: Setup project**. The goal is to create a new project and set it with required scenes and create an environment for next steps. The project began with an empty Unity Template, which provided only an empty scene and access to core functionalities. Subsequently, three folders were created: 'Common,' 'P1,' and 'P2.' These were designated for storing common resources and for implementation-specific resources for versions one and two, respectively. Each implementation was assigned its own scene, both named identically. An assembly definition file was also added to each to segregate code access and prevent naming collisions.

With the setup for implementation P1 complete, we then prepared for P2, which required establishing a Dependency Injection (DI) framework. This step introduced a clear initial disadvantage for P2 in terms of setup complexity. For this thesis, VContainer was chosen as the DI framework. VContainer is a lightweight DI library tailored for Unity that prioritizes high performance and ease of use. It is designed to be fast and minimize garbage collection, which is crucial for performance-critical applications like games.

VContainer [41] supports various forms of injection including constructor, property, and method injections. It also integrates seamlessly with Unity's lifecycle, offering features such as automatic registration and resolution of components aligned with Unity's events. Being open-source and well-regarded within the Unity community, VContainer provides excellent documentation and a lightweight architecture, making it an ideal choice for developers new to game development who are setting up their first DI framework.

While other frameworks like StrangeIoC, Zenject (Extenject), or Reflex were considered, VContainer's combination of performance focus, comprehensive documentation, and community support made it the preferred choice for this project.

The integration of additional packages is efficiently managed by Unity's Package Manager. This process involves adding a new registry for OpenUPM and extending the registry scope to include the VContainer's scope. The new package must also be added to the

assembly definition's list of references. Once the package is successfully integrated into the project, the next step is to create a project context, establishing a global context named GameContext that resolves only once and serves as the singular starting point for the game.

The decision to enhance the project's architecture with additional elements caused Implementation P2 to take longer, introducing a new dependency within the package manager and an extra component in the scene. Although the process of adding the package is straightforward, it can present challenges for a beginner programmer. A significant issue is the necessity to understand the basics of the inversion of control principle, and the time-consuming nature of selecting an appropriate framework can also pose difficulties.

In this step, the measurable factors are limited to the additional development time required to integrate the VContainer into P2, and the incorporation of new non-core APIs from the VContainer. These measures help quantify the overhead introduced by adopting an external Dependency Injection framework in the project.

10.4. Match-3 mechanic implementation

The next scenario is **S2: Add base Match-3 mechanic**. For both implementations, the Match-3 mechanic will utilize a simple Scriptable Object Architecture, leveraging Unity's Scriptable Objects for the board settings and chip configurations. This approach promotes modularity and extensibility directly from the Unity inspector, facilitating easier adjustments and scalability in game development.

Despite both implementations utilizing the same Match-3 mechanic, they will require different approaches to gameplay integration. P1 is currently at a basic stage with only an empty scene created, necessitating the implementation of a starting point for the game. In contrast, P2 has a Dependency Injection system already configured, which serves as a ready-made foundation for injecting gameplay elements into the scene. This setup in P2 potentially streamlines the process of integrating game mechanics compared to P1, where more groundwork is needed to establish the gameplay framework.

The Match-3 mechanic is encapsulated as a separate feature that can be initiated by invoking the method:

```
UniTask SetupBoard(BoardSettings settings)
```

UniTask [43] is Unity's adaptation of the async/await pattern, enabling asynchronous, non-blocking execution of code. While C# natively provides a similar implementation using `Task`, UniTask is optimized for Unity's single-threaded environment, offering a more efficient and simpler approach for asynchronous operations within Unity. Having a UniTask as a return type tells the compiler that this method can and should be awaited and can run asynchronously.

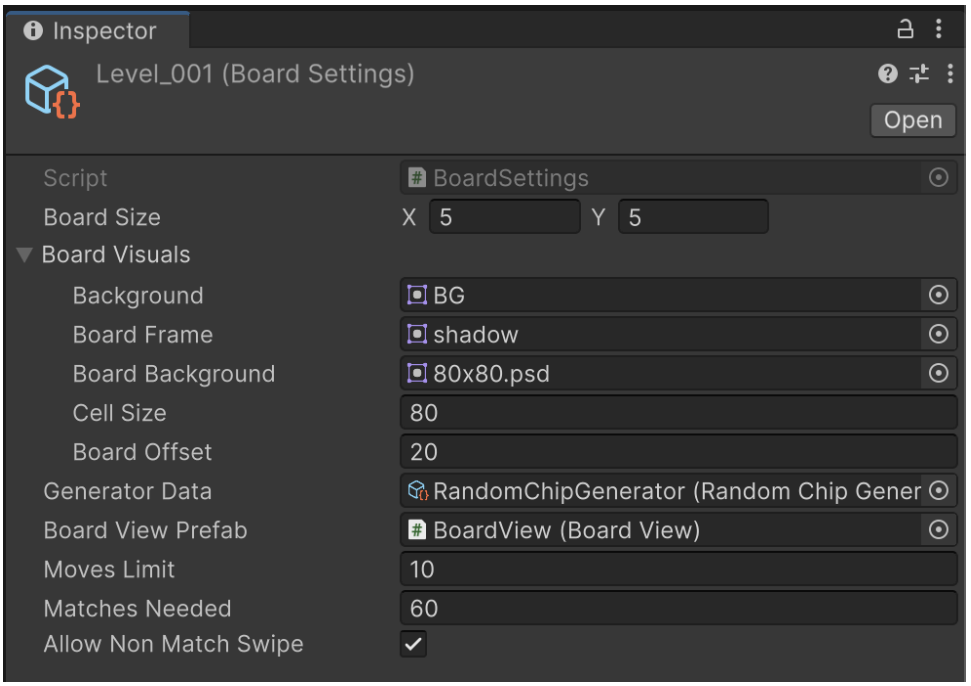

Figure 10.3. Example configuration for the Match-3 board settings (Source: own work)

BoardSettings is a Scriptable Object visible in Figure 10.3 that contains all the necessary data to initialize a Match-3 game. It stores details such as the board size and layout, chip generator logic, and visual elements like chip size and the aesthetics of the board or background graphics. This structured approach ensures that all relevant game settings are easily managed and modified.

By invoking this method, a match in the game is initiated. The interesting aspect of the implementation is how and where this method should be called. In implementation P1, a new empty GameObject has been added to the scene, to which a custom MonoBehaviour component named `GameManager` is added. This `GameManager` creates a new instance of Match3 and starts the game using settings from a serialized field reference. Currently, without a level selection system, the level can only be selected from the Unity Inspector, and the build contains only this one setting.

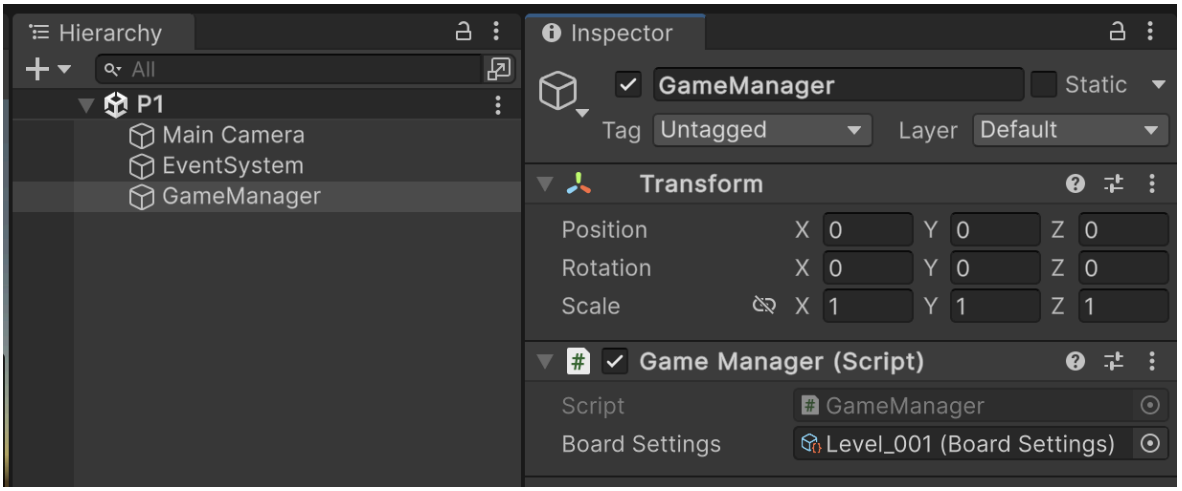

Figure 10.4. The initial project setup for P1 (Source: own work)

For version P2, the setup is similar, but instead of a `GameManager`, there is a `GameBoot` component added to the new empty GameObject. Unlike `GameManager`, `GameBoot` does not directly create a new instance of Match-3. Instead, it receives the Match-3 instance via Dependency Injection provided by a previously established `GameContext`. Like in P1, the level in P2 is also referenced through the inspector, and `GameBoot` initiates the game.

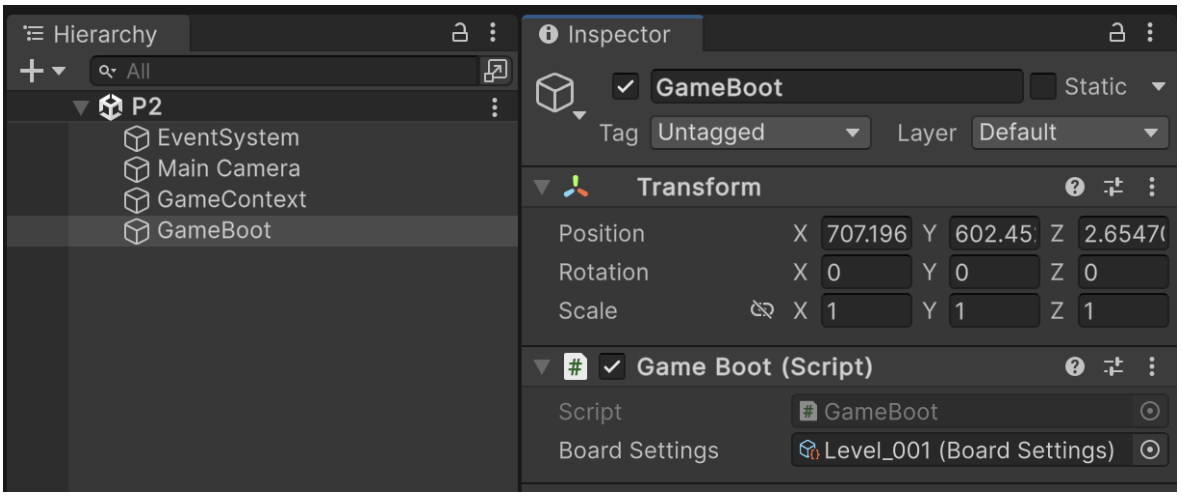

Figure 10.5. The initial project setup for P2 (Source: own work)

While the differences between P1 and P2 might appear minimal and subtle, they are significant and will be more relevant in subsequent chapters. The key distinction involves the ownership of the Match-3 instance. In P1, `GameManager` owns the Match-3 instance—it creates it and stores it in a private variable. In contrast, in P2, the ownership is

held by the Dependency Injection context, which maintains the Match-3 instance. At this stage, the differences don't significantly impact performance because the game is still early in development with only one system introduced. However, P1 has a slight advantage due to its fewer component requirements (no need for `GameContext`) and simpler code, achieved by using the "new" keyword instead of registering a new type within a context.

It's important to note that at this stage, the game lacks specific limitations or objectives. The logic for scoring, checking objectives, or determining win conditions should not be part of the core mechanic logic. This separation is crucial because each level might feature different difficulties or objectives. For instance, achieving a one-star reward might require completing the level in less than 30 moves, whereas earning three stars might require 15 moves or fewer. Separating this logic from the implementation of the game mechanics will result in a better overall architecture. In the context of this thesis, keeping the objective logic separate enhances the analysis by preventing the encapsulation of this logic within the common code, thus allowing each implementation to fully leverage its architectural strengths.

To accommodate this separation, the Match-3 mechanic utilizes C# "Action" delegates that are invoked at critical moments. One such event is triggered when a move is made—whether a player swipes tiles to match them or activates a special chip like a bomb—essentially, any user action that results in a change in the board state. Another event is triggered when a match is made, which occurs whenever one or more chips are cleared from the board as a result of three or more aligned tiles, a special effect that removes adjacent tiles, or all tiles of a specific color, with the potential for further extension to include more elements. The details surrounding objectives and scoring will be further developed in upcoming chapters, where UI and progression mechanics will be integrated and made available for use.

10.5. Objectives and scoring

The upcoming scenarios, S3, S4, and S5, will introduce fundamental functionalities central to the Match-3 game mechanics, focusing on creating a minimal viable product that mirrors a realistic scenario for analysis. Specifically:

- **S3: Add Scoring for Matches** - This scenario will implement a basic scoring system for matches made, essential for any Match-3 game.

- **S4: Add Matched Chips Count Objective** - This will introduce objectives based on the number of chips matched. Initially, the objectives will be simple, such as achieving a set number of matches.
- **S5: Add Move Limit** - This scenario will incorporate a move limit, restricting the number of moves a player can make, which adds strategic depth to gameplay.

While additional features like a star rating system, combos, boosters, special block challenges, and a “lives” system could enhance engagement and monetization, they will not be included in the initial implementation. These features are mentioned as potential expansions but are beyond the scope of the current minimal viable product focus. This approach ensures that the essential Match-3 mechanics are developed and analyzed effectively before considering more complex features.

As discussed in the previous part, the core game uses the observer pattern to notify about moves and matches done by the player. In P1 the GameManager is already holding the Match3 instance so the simplest approach would be to include the scoring and objective mechanics also inside there. In P2 the instance of Match3 is injected via DI so instead of changing the GameBoot better idea from the architectural point of view would be to create a ScoringSystem and ObjectivesSystem.

BoardSettings now includes parameters such as the move limit and the required number of matches to win the level. If the player exhausts all moves, the game concludes with a loss. Conversely, achieving the required matches results in a win, with the player earning a bonus of 100 points for each unused move. Each matched chip scores 10 points. Currently, the game's state, including wins, losses, and scores, is output to the console via debug logs, as no UI has been implemented yet.

For P1, all game logic resides within the `GameManager`, creating a straightforward yet increasingly complex main class. The foundational elements remain consistent across both implementations. However, in P2, no new code was added to the `GameBoot`. Instead, the logic for scenarios S3, S4, and S5 has been delegated to specialized systems: `ScoringSystem` and `ObjectivesSystem`, both of which are instantiated via Dependency Injection and respond to game events. This structure leads to a significant increase in lines of code (LOC); P1 contains a single script with 76 LOC, whereas P2 involves eight scripts totaling 290 LOC. Excluding boilerplate elements like imports and

namespace declarations, the effective LOC is 55 for P1 and 174 for P2—a more than threefold increase. Despite the larger codebase, what P2 offers over P1 is the abstraction and modularity of its systems, as the Dependency Injection (DI) facilitates the substitution of implementations without impacting other parts of the project. Additionally, P1 integrates all scenario requirements directly into the main project class, whereas P2 abstracts these details, exposing fewer methods and fields.

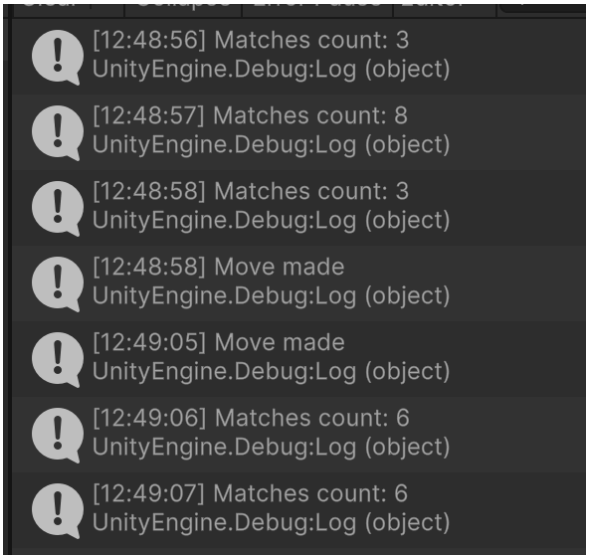

Figure 10.6. The Match-3 game progress visible from the Unity console level as no UI is yet present. (Source: own work)

During this development phase, the first bugs emerged. In P2, an issue arose during the implementation of S4, where the Dependency Injection framework required a new factory [44] for objectives. Thanks to well-documented code and explicit exception handling, this problem was quickly resolved. In P1, a bug was encountered during the implementation of S5 where the game-end condition was erroneously checked multiple times consecutively, disrupting the scoring system for unused moves. This error, although more challenging to detect as it required playing through the entire game under specific conditions, was eventually identified and corrected.

10.6. Window system - UI

Currently, the game's functionality is limited to endlessly playing the core Match-3 mechanics. There is no display to show game progress, such as the number of matches made, the number of moves the player has used, or the current score outside of the editor-only log console. Furthermore, essential UI components like a starting screen, level

selection, and win or lose screens are absent. For the game to function properly and offer a complete user experience, it is crucial to implement a way for players to interact with these elements.

In this phase, the requirement is **S6: Add Gameplay HUD** (Heads-Up Display) for the Match-3 game, along with implementing other features that will facilitate future enhancements. A HUD is crucial in video games for presenting essential information transparently on the screen, allowing players to see important data without diverting their attention from the main gameplay area. This concept originated in aviation, where pilots needed quick access to critical information without obstruction to their field of view.

For our Match-3 game, the HUD will include:

- **Moves Left Counter**: Displays the number of moves the player has remaining.
- **Count of Matches Needed to Win**: Shows how many more matches are required to complete the level.
- **Score Display**: Keeps track of the player's current score.

Each component of the HUD should dynamically update in response to changes in the underlying values. The forfeit button, in particular, should trigger an immediate game loss, enabling players to opt out quickly whenever they choose. This setup ensures that all critical game information is readily accessible and effectively enhances player engagement and control.

While this requirement mentions only one "window" of the game the codebase should be prepared for next windows as those will follow shortly after. The simplest implementation of such a system can be done with adding the windows to a scene and then having a script to enable and disable windows on demand. With this approach no complex systems outside of the Unity simplest functionalities are needed. While being the simplest and fastest to develop this approach can have many shortcomings in more sophisticated usages, but since the P1 focuses on the immediate gain and the functionalities are satisfactory for this stage it is an acceptable solution that can be often seen in games.

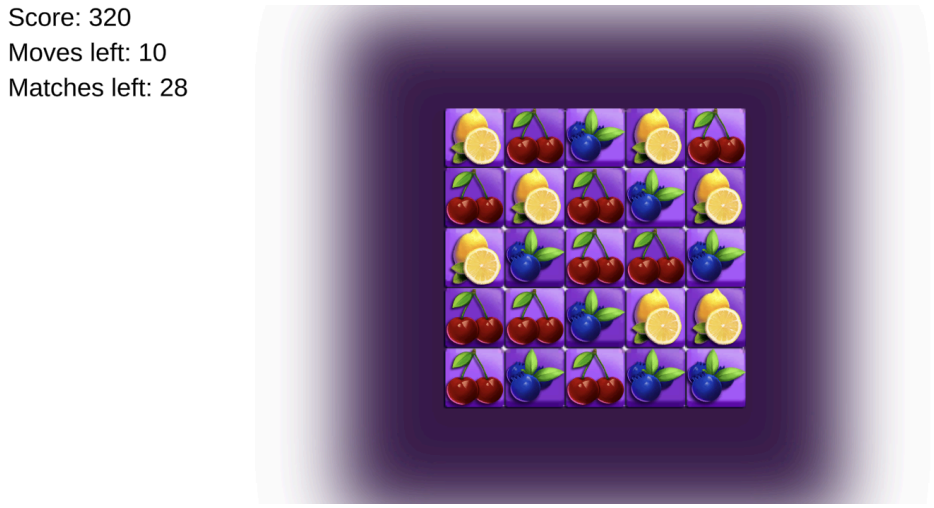

Figure 10.7. Match-3 board created from the settings in Figure 10.3 with simple HUD in the top left corner (Source: own work)

P1 will feature a `WindowManager` with a serialized list of windows references, visible in Figure 10.8. Opting for a simple implementation does not necessarily imply a compromise in quality. Effective abstraction is achieved through a `Window` base class and a dictionary that allows access to windows by type, enhancing scalability. This design means that adding a new window involves merely adding a reference within the manager.

```
public class WindowManager : MonoBehaviour
{
    [SerializeField] private List<Window> _windows;
    private readonly Dictionary<Type, Window> _typeToWindow = new();

    private void Awake(){...}

    private void RegisterWindows(){...}

    public T OpenWindow<T>() where T : Window{...}
}
```

Figure 10.8. WindowManager implementation for P1. Manager takes a list of serialized windows and registers them to then open by type. (Source: own work)

One limitation of this system is that only one window can be active at a time, but this can be addressed by designing the windows to be fullscreen, which generally fits the typical gameplay HUD requirements. A more significant issue, which relates more broadly to the global architecture rather than the window system itself, involves the updates to the HUD values. Currently, updating the HUD values necessitates either changing the private fields

of the `GameManager` to public or requiring the `GameManager` to proactively update the HUD whenever its values change. This situation highlights a potential area for improving the interaction between the game's core logic and its user interface, ensuring that the system remains robust while maintaining encapsulation and minimizing dependencies.

For P2, the goal is to invest more time into the foundational architecture to better accommodate future enhancements. With this in mind, the architecture proposed for this version is a simplified Model-View-ViewModel (MVVM) pattern. Since Unity's UI system based on Canvas is relatively new and Unity continues to experiment with new approaches—like the UI Toolkit, which is more akin to web-based UI—there are no widely accepted implementations of the MVVM pattern in Unity yet.

A custom simplified implementation will be developed, drawing heavily on the Observer Pattern. In this setup, the ViewModel will monitor changes in the Model, and any modifications will trigger the bound View to update accordingly. This approach ensures that the UI remains responsive and accurately reflects the current state of the game data, enhancing the maintainability and scalability of the system as new features and complexities are introduced.

The implementation for P2 started with the Observable pattern [45], utilizing the `IObservableProperty<T>` interface in place of plain data types like `int`. Each `IObservableProperty<T>` has a `Value` and a subscription mechanism via a custom `DisposableSubscription`, which wraps a C# `Action` and returns an `IDisposable`. This disposable object is later stored and managed within the subscriber's disposables list.

```
public interface IObservableProperty<T>
{
    DisposableSubscription<T> OnValueChanged { get; set; }
    T Value { get; }
    IDisposable InvokeAndSubscribe(Action<T> observer);
}
```

Figure 10.9. Observer pattern implementation interface for P2. Used as a baseline for MVVM architecture (Source: own work)

While P1 occasionally faces issues with updating values consistently across all relevant locations, P2 introduces the challenge of properly disposing of subscriptions. Here, Dependency Injection proves beneficial as it automatically handles the disposal of dependencies. Implementing the `IDisposable` interface on a ViewModel and adding subscriptions to a `CompositeDisposable` list simplifies the management of disposables.

```
public class WindowSystem
{
    private readonly IObjectResolver _container;

    public WindowSystem(IObjectResolver container)
    {
        _container = container;
    }

    public T OpenWindow<T>() where T : ViewModel
    {
        var viewModel = Activator.CreateInstance<T>();
        _container.Inject(viewModel);
        viewModel.Show();
        return viewModel;
    }
}
```

Figure 10.10. WindowSystem implementation for P2. Instead of registering windows from references it creates a ViewModel which then Instantiates the View. (Source: own work)

In terms of the MVVM architecture:

- **Model**: Acts as the data source, e.g., the score from `ScoringSystem` or win conditions from `ObjectivesSystem`.
- **View**: A basic class with serialized fields for UI elements and a `Dispose` method for cleanup.
- **ViewModel**: Manages the view instance, binding model properties to UI element references.

This architecture is encapsulated within the `WindowSystem`, Figure 10.10, which provides a method to open windows in a manner similar to P1. Windows prefabs are loaded from resources, which may introduce a slight overhead in window opening times

but results in a faster initial load of the starting scene. This setup does not support reusing ViewModels without complex partial disposals and unsubscribes, potentially complicating the architecture further. Performance concerns arising from this approach could be addressed in future improvements.

Additionally, due to the lazy creation of objects in Dependency Injection, a `GameplaySystem` was added to P2, Figure 10.11. Instead of directly starting the Match-3 game, `GameBoot` now initializes the `GameplaySystem`, which is responsible for initiating gameplay and displaying the HUD. This separation of responsibilities contrasts with P1, where the `GameManager` increasingly accumulates functions over time.

```
public class GameplaySystem : IDisposable
{
    private readonly Match3 _match3;
    private readonly WindowSystem _windowSystem;
    private GameHUDViewModel _hud;

    public GameplaySystem(Match3 match3, WindowSystem windowSystem,
        ScoringSystem scoringSystem, ObjectivesSystem objectivesSystem){...}

    public async UniTask StartGame(BoardSettings boardSettings)
    {
        await _match3.StartGame(boardSettings);
        _hud = _windowSystem.OpenWindow<GameHUDViewModel>();
    }

    private void OnGameEnded(bool successful){...}

    public void Dispose(){...}
}
```

Figure 10.11. Implementation for GameplaySystem in P2. It presents the injection process, starting the game and showing HUD. (Source: own work)

Summary of this step:

- **P1**: Remains straightforward, now consisting of four classes and adding 44 LOC. Encountered two minor bugs related to UI updates and window registration, both quickly fixable.

- **P2**: Expanded to include twelve new classes, totaling twenty, with almost 300 additional LOC. Faced three bugs: one related to an unregistered subscription causing a

memory leak, and two linked to Dependency Injection lazy resolving, including a scoring system registration timing issue and a null reference exception due to a race condition in UI objectives.

At this stage, P2 is more resource-intensive in terms of development time and codebase size. Although P1 has some architectural shortcomings and might require substantial modifications to meet future requirements, its simplicity and speed of implementation are advantageous. P2 offers a more robust API that could facilitate future expansions without duplicating code. Nonetheless, the game remains incomplete, and it is too early to draw definitive conclusions about the overall effectiveness of each approach given the project's ongoing nature.

10.7. Level selection and progression

The current game setup begins automatically upon entering play mode, and once the objective is met, the board vanishes, leaving behind an empty white screen. Such behavior is not suitable for the final game version. A fundamental requirement for a playable game mechanic includes a starting screen with a play button, allowing players to initiate the game and return to this screen upon completion. Additionally, offering just one level is insufficient for maintaining player engagement. Players should have the ability to progress through multiple levels, with each subsequent level unlocking upon the completion of the preceding one.

To address these requirements, the following scenarios will be implemented:

- **S7: Add Level List** - Instead of featuring only a single level, the game will be designed to support multiple levels listed in a sequence. This setup allows for the addition of diverse challenges and prolongs the gameplay experience.
- **S8: Add Level Selection** - The levels introduced in S7 will be displayed in a user interface, allowing players to select any level to start playing. This feature will enhance player autonomy and engagement by providing a variety of levels to choose from at their discretion.
- **S9: Add Player Progression** - Levels will initially be locked except for the first one. Completing a level (level n) and winning it will unlock the subsequent level (level n+1). Furthermore, a new button will be added to enable players to quickly

start playing the highest unlocked level without navigating through the level selection screen.

Upon completion of these steps, the game will begin to resemble a full-fledged Match-3 game, equipped with a starting screen, a level selection interface, and a logical progression system for the players as in Figure 10.12. These enhancements aim to create a more engaging and cohesive gaming experience that encourages continuous play.

For the P1 implementation, substantial modifications were necessary within the `GameManager` class. Initially, the game did not feature level selection; levels were directly initiated and tied to the Unity object's lifecycle. The simplest enhancement involved replacing the single level reference with a list of all levels, utilizing Unity's `PlayerPrefs` as a persistent storage layer for the current level index. `PlayerPrefs` allows for data storage within the application's cache, maintaining persistence until either the application is uninstalled or the cache is manually cleared. This mechanism enables the persistence of the current level index and the highest unlocked level index across sessions.

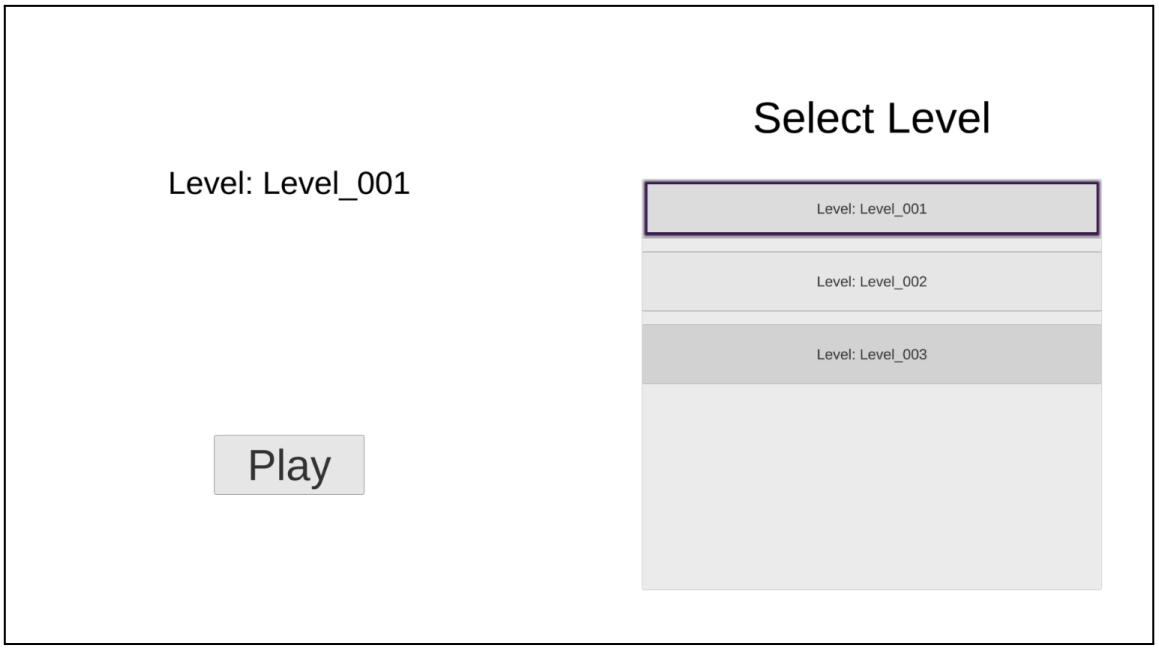

Figure 10.12. The Starting screen UI view. With the level selection on the right and option to start the game on the left. (Source: own work)

Now, instead of directly starting the Match-3 mechanic, the game will open with the `StartScreen` UI, which includes a button to play and select a level. While the UI remains consistent across both versions, in P1, without the observable pattern available,

every change must be manually propagated. To maintain class encapsulation and hierarchy, GameManager retains the responsibility to initiate the Match-3 game but does so through a C# Action method passed down, simplifying communication.

In contrast, P2 leverages the existing `GameplaySystem` and Dependency Injection (DI) to manage levels and progression through a new system called `LevelProgressionSystem`. This system manages the linkage between the list of levels, defined in a Scriptable Object, and player progression. Unlike P1, where implementation specifics are exposed, P2 conceals the logic, displaying only the list of all levels, unlocked levels, and the current level. This system extensively utilizes `IObservableProperty<T>` for properties like the current level or unlocked status, facilitating easy updates through subscriptions.

The UI in P2 builds upon the previously established structure by adding new views and view models. Notably, the implementation of the P2 view required only 26 lines of code (LOC) and a single external reference, whereas P1 required 41 LOC and four external references. Moreover, P2 maintains a clear separation between business logic and the view, whereas in P1, these aspects are intermixed, which could lead to potential issues.

Despite these differences, P1 required slightly less time and fewer overall lines of code for these steps and exhibited fewer bugs overall. P1 encountered four minor bugs related to missed references and typos in `PlayerPrefs` keys, and issues with updating values. P2 encountered three bugs, but these were more challenging to resolve due to complexities in the WindowSystem and inconsistencies in `PlayerPrefs` keys.

Overall, the game is becoming functional, with players now able to progress and select levels. While there are still gaps in user experience, such as animations or tutorials, the game is progressively nearing its initial scope of functionality.

10.8. Statistics and achievements

Now that a basic version of the game is operational, it's time to incorporate additional "helper" features. These functionalities, such as player statistics and achievement systems, add extra layers to the game, providing players with reasons to replay levels and engage in

competition. These systems, while standard in the gaming industry, also serve as robust tests of the game architecture's resilience.

The upcoming requirements include:

- **S10: Add Player Statistics** - This will involve collecting and storing various player statistics.
- **S11: Add Achievements** - Utilizing the collected statistics as a basis, this feature will reward players with achievements for reaching specific milestones.

Proposed statistics include:

1. Number of Matches Done.
2. Number of Moves Done.
3. Number of Wins and Losses.
4. Win to Loss Ratio.
5. Total Score Gained.
6. Total Time in Game.

Figure 10.13. Statistics screen (Source: own work)

Maintaining a concise list of statistics simplifies testing and provides insights into the integration of additional metrics in the future. Alongside these statistics, a simple interface will be added to display them, which should update dynamically, such as showing time

increasing while the window is open. It's important to note that these statistics should continue to be updated and preserved even if the player quits the game. This ensures that all player progress and achievements are accurately recorded and maintained across sessions, enhancing player engagement and satisfaction with the game's continuity and reliability.

For the achievements, the proposed list includes:

1. Win 1 Level.
2. Win 10 Levels.
3. Do 100 Matches.
4. Do 1000 Matches.
5. Gain a Total of 10,000 Points.
6. Play for 5 Minutes.

Figure 10.14. Achievements screen (Source: own work)

While this list is intentionally limited to streamline development and testing, the achievement system should be designed with scalability in mind, allowing for easy addition of new achievements. Similar to the statistics display, the achievements interface will need a dynamic update mechanism, including checkboxes to show whether an achievement has been completed. This will enhance user interaction by providing immediate feedback on their progress within the game.

The `GameManager` in the first implementation has already exceeded one hundred lines, and adding further functionality directly to it would significantly impact its readability. Continuously extending `GameManager` with new features like statistics and achievements could become unmanageable. Additionally, managing the UI through callbacks passed throughout various methods could further complicate the architecture. To maintain a reasonable quality level in this implementation, it is crucial to separate these functionalities.

There are several straightforward solutions to address this. One approach involves making fields within `GameManager` public and using singletons or accessing components via the Unity hierarchy. However, to preserve encapsulation, converting these fields into read-only properties that are externally accessible might be more appropriate. This approach, though, would require that new managers also be implemented as `MonoBehaviours`, which presents its own set of challenges. These issues will be discussed in more detail in the last section of this chapter, and the implementation will proceed with the proposed solution for now.

P1 is currently facing a significant challenge in managing the statistics system effectively. The problem stems from the fact that updates of values are only summarized per game and manually adjusted across the entire file, complicating accurate tracking. One approach considered was to duplicate the logic for calculating values on the manager's side. However, this method would lead to redundant code and require updates in multiple places whenever changes occur, making it less desirable due to increased complexity and maintenance effort.

Another potential solution was to check for changes each frame. This method, while somewhat inefficient—since most values only need updating every few seconds, and the Update method runs every frame—was deemed justifiable for statistics that require constant updates, such as the 'time' statistic. Although this approach involves minimal additional CPU overhead, the effort to continually check for changes could be extensive and unnecessarily complex.

The chosen solution to address this issue is to directly notify the stats manager each time a statistic changes. This method is straightforward and ensures that all statistics are kept

up-to-date in real-time. However, it does carry the risk of human error, potentially leading to overlooked updates and resulting discrepancies in the data recorded. Despite these risks, notifying the stats manager directly remains the most efficient and effective method to maintain accurate statistics, as it avoids the complications of redundant coding and the intensive resource use of constant frame checks.

P2 will maintain the style established in previous implementations by using a `StatsSystem` that consolidates various game values into a centralized location. While the View and ViewModel classes for the statistics are straightforward and can be implemented quickly, the challenge lies in developing an appropriate Model class. The abstraction introduced in earlier steps for objectives necessitates additional checks and subscriptions to ensure accuracy. Furthermore, to maintain proper encapsulation, each private `ObservableProperty<T>` is exposed as a get-only `IObservableProperty<T>`, which prohibits value modification and results in additional boilerplate code, thereby increasing the size of the codebase.

For the achievements system, P1 will adopt a similar approach to the one used for statistics. A new `AchievementsManager` will be implemented, linked to the `StatsManager` and containing a list of achievements. However, the absence of an observer pattern in P1 necessitates manual updates to the view, which complicates maintaining synchronicity between the model and the User Interface. In both implementations the state of a player's achievements will be stored in a custom object that will be serialized to JSON and stored inside `PlayerPrefs`.

P2 has a significant advantage in that all fields in the stats system are already well-prepared and integrated, making their use in the achievements system simple and efficient. The main challenge is ensuring that the achievements are represented in a manner that facilitates easy addition of new items without encountering the problems seen in the objectives abstraction, such as overly complex dependencies and maintenance issues. This requires careful design to balance flexibility and simplicity in the system architecture.

With the addition of new features like Achievements, the P2 implementation is beginning to appear more advantageous. Until now, the simpler P1 approach was faster to develop, easier to understand, and caused fewer issues. However, as the systems for P2 are now

established and in place, much of the work revolves around leveraging these existing systems. The Achievements feature marked the first instance where implementation in P2 required less time than in P1, demonstrating the benefits of a more integrated system as the project's complexity increases. While the overall architecture of P2 may seem more complex, examining individual components like Achievements reveals that they are less complicated to manage compared to P1.

Concurrently, P1 has started to face challenges with the updatability of the achievements state. Since the values are not subscribable, it is difficult to track when changes occur. One potential solution could be to convert the statistics fields into properties and add a subscription mechanism within the setter method. However, this approach would complicate data serialization with JSON, which is currently used for state storage, potentially requiring the implementation of a facade layer similar to that in P2. To maintain simplicity and avoid extensive refactoring at this stage, an alternative solution has been adopted: adding an Update method to refresh the state of the window each frame in search of changes. While this method introduces some performance overhead— which will be quantified in subsequent sections—it represents the simplest solution under the current circumstances.

10.9. Rankings

Rankings are an excellent feature for fostering competition among players within a game. They allow players to compare their scores and vie for the top spot. Proper implementation of rankings in a game typically involves multiplayer capabilities where numerous players submit their statistics to a server, which then stores the data in a database. This data is subsequently retrieved by players to display the rankings. While our implementation for requirement **S12: Add Simple Ranking** won't delve into complex aspects such as setting up a server and database, we will implement the skeletal structure of the client side using mock data.

To retrieve and send data in a typical game setup, HTTP requests are used. Unity provides `UnityWebRequest` for handling such requests. The ranking logic needs to address several key considerations:

1. **When to send player data to the server?**
    - The data should be sent frequently enough to ensure accuracy but without overwhelming the server. Optimal frequency will depend on the game's pace and the significance of the rankings.
2. **When to request ranking data?**
    - Data retrieval should occur on demand, such as when opening a ranking window, and should be limited to prevent excessive server load, ideally no more than once per minute.
3. **How to display the ranking?**
    - The UI for displaying rankings can be straightforward, similar to how statistics and achievements are presented, ensuring consistency and ease of use for players.

By focusing on these areas, we can create a functional and efficient ranking system within the game's current architecture, using mock data to simulate server interactions and provide a realistic user experience.

The implementation of the ranking system will be structured into two main components: a networking layer and a ranking layer. The networking layer will handle communications, and after a brief delay, it will return a list of mocked ranking entries to simulate server interactions. Once the networking setup is established, the focus will shift to the ranking layer, where the game's business logic processes the received data.

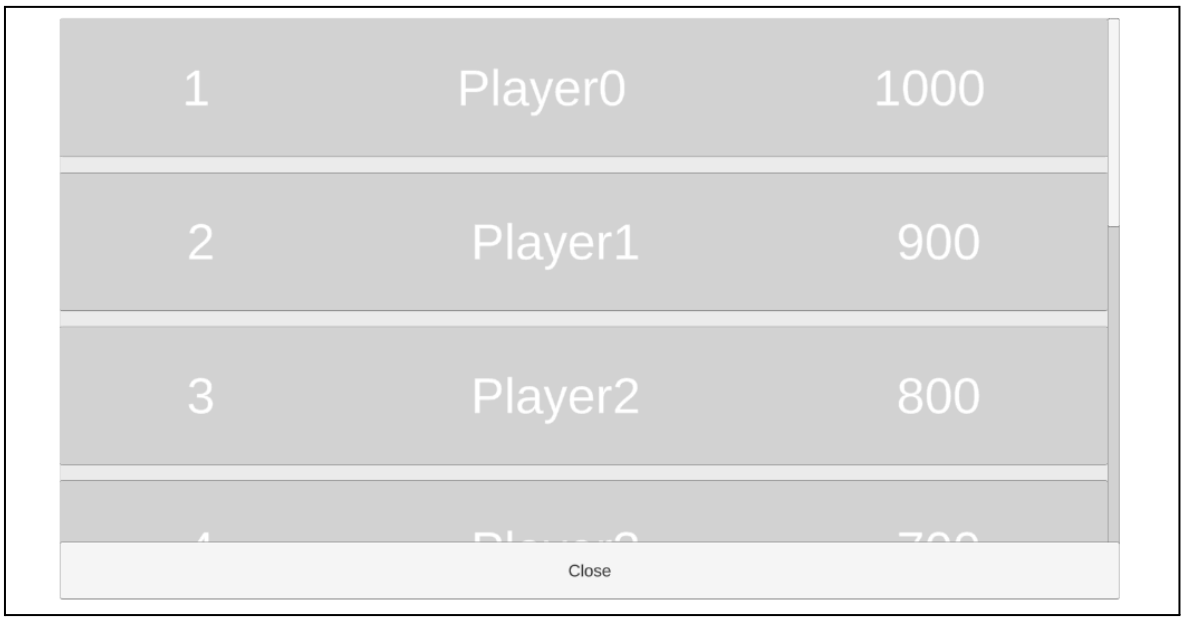

Figure 10.15. Ranking screen (Source: own work)

The final step in the implementation involves integrating the User Interface. Building on the foundational architecture already established, the UI will display the ranking data effectively. Both P1 and P2 implementations will follow a similar approach to previous features, ensuring consistency and leveraging existing structures. This methodical progression ensures that each component integrates smoothly, providing a seamless experience in displaying and updating rankings within the game.

Other than the way of displaying the view and dependencies management both solutions are very similar. What starts being very tiresome in the P1 is the need to connect all the dependencies manually via hierarchy view. This often leads to bugs where some reference is not plugged correctly. With this implementation there were two bugs, one connected to a new ranking window not being added to the window manager windows list and another to game manager lacking reference to ranking manager so both related to the inspector references.

11. Projects analysis

With the design scope completed and all functionalities integrated, it is now time to compile the target builds and perform experiments for the proposed methodology metrics. Using the measures and software defined in Chapter 8 both implementations will be scrutinized and compared.

11.1. Performance analysis

The first experiment will be an analysis of performance of the games to assess the impact of the chosen architectures. Utilizing the Unity Profiler to analyze framerate, frame time, and the CPU/GPU load, along with the Memory Profiler to evaluate RAM usage and Garbage Collector (GC) allocations which significantly influence application behavior.

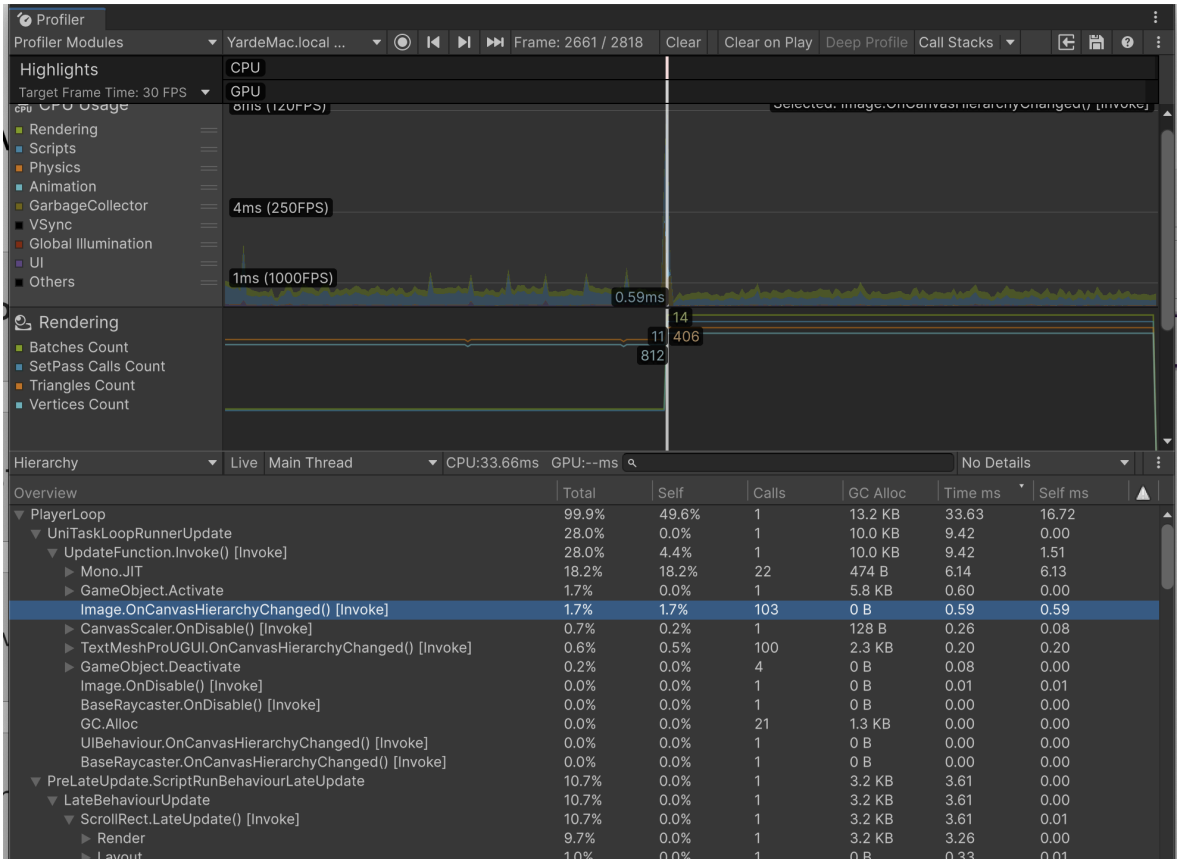

Figure 11.1. The view of a Unity Profiler 2023. Horizontal line represents game frames, vertical line represents value depending on measure, ex. frametime for CPU usage. (Source: own work from Unity Profiler)

As both implementations are housed within a single project but separated into different assemblies, two builds will be generated. The first build will only include the P1 scene, and the second will only include the P2 scene. Unity's code stripping feature ensures that unused parts of the project are not included in the build, optimizing the final output. To mitigate the slower build times typically associated with initial builds and to benefit from incremental builds, a preliminary build containing both versions was created, followed by individual builds for each version. The build process for P1 took 7 seconds and resulted in a file size of 144.4 MB. The build time for P2 was nearly identical, with the file size being less than 100 KB larger, a difference of less than 0.1%.

For performance analysis, measurements of frame time in milliseconds and GC allocations in MB will be provided for each frame that deviates from the target. Since the desired target framerate is 60 FPS, the acceptable frame time is around 16.6 ms. Given the relatively small size of the game, it should consistently meet the target framerate when idle. Unity attempts to maintain the target framerate by delaying the next frame if the previous one completes faster than the target frame time, ensuring smooth gameplay.

Table 11.1. Table of highest impact frames for performance with their frametime and GC allocations for both implementations. (Source: own work)

Frame Action Description	P1 frametime [ms]	P1 GC Size [MB]	P2 frametime [ms]	P2 GC Size [MB]
Scene Initialization	87.92	0.21	4.18	0.21
Game Start	18.05	1.31	18.56	1.42
Stat Window Open	50.32	1.6	34.87	1.5
Achievement Window Open	34.11	1.9	27.59	1.8
Ranking Window Open	49.08	2.3	27.1	2.2
Match-3 Game Start	58.45	3.3	60.93	2.7

Over a sample of 600 frames, only 6 frames failed to meet the target framerate, as detailed in Table 11.1. These deviations were typically associated with resource-intensive actions triggered by the user, such as starting the game or opening a window. The variations between results were generally minor and could be attributed to random fluctuations in the

engine's performance. After several additional runs, the results remained consistently at a similar level. Results visible in Figure 11.2 for P1 and Figure 11.3 for P2, the horizontal axis represents consecutive frames, vertical axis represents frametime in [ms].

One notable observation was the scene initialization times. In P2, all windows are loaded dynamically from resources, leading to **quicker initialization times**. In contrast, P1 includes all windows directly within the scene, resulting in **significantly longer loading times**. Although this delay is initially masked by Unity's splash screen, adding more windows in P1 could potentially degrade the user experience. Additionally, window transitions in P1 are somewhat more resource-intensive as they require a complete rebuild of the scene's render structure whenever changes occur.

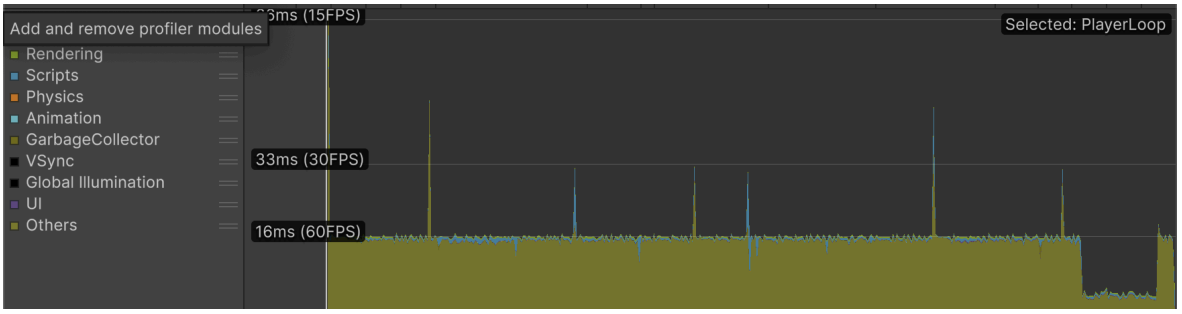

Figure 11.2. CPU usage in a P1. Game is running at almost constant 60 FPS with 2 frames of drops below 30 FPS. (Source: own work from Unity Profiler)

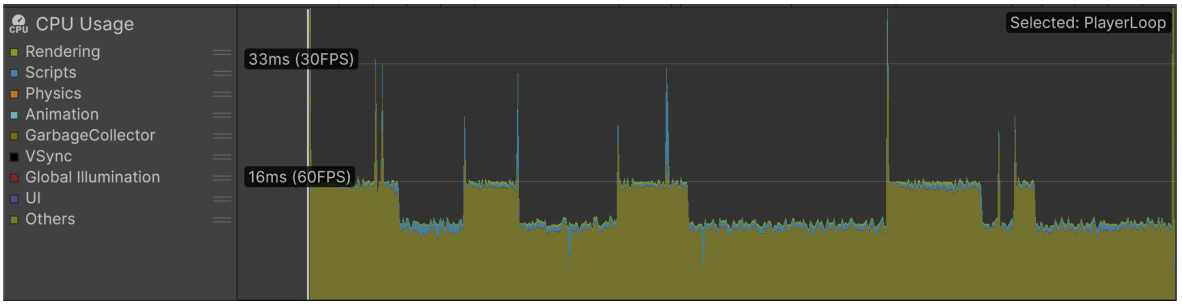

Figure 11.3. CPU usage in P2. The game is running running most of the time at 90 FPS with some sections dropping to 60 FPS and 1 frame of drop below 30 FPS (Source: own work from Unity Profiler)

Figure 11.4. Memory usage in P1. Garbage collector memory allocated. (Source: own work from Unity Profiler)

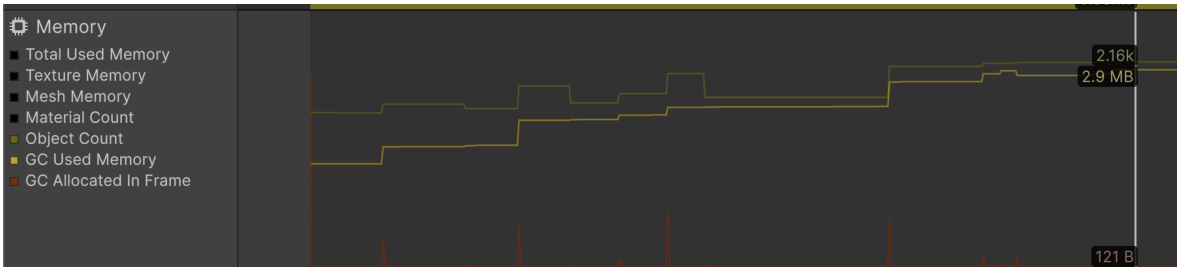

Figure 11.5. Memory usage in P2. Garbage collector memory allocated. (Source: own work from Unity Profiler)

For the Garbage Collector Allocations (GC Alloc) visible in Figure 11.4 for P1 and Figure 11.5 for P2, the horizontal axis represents consecutive frames, vertical axis represents frametime in [MB]. The measure is important as higher allocation means more RAM usage as well as memory fragmentation.

In summary, the performance of both implementations is commendably high, with even the lowest 1% of frame rates remaining above 10 FPS. **Importantly, this experiment reveals that the overhead introduced by Dependency Injection and the additional classes required for the MVVM pattern is negligible.** Instead, more substantial performance impacts may arise from other design choices, such as the method of window management. From an architectural standpoint, no significant issues were detected in either implementation, indicating robustness in both approaches.

11.2. Development results analysis

With all systems implemented and performance metrics collected, it is now time for a detailed analysis and summary of the development outcomes. We'll begin with a high-level overview using a class diagram to illustrate the architecture.

In the case of P1, all classes inherit from MonoBehaviour, meaning they are integrated into Unity's lifecycle and player loop. Consequently, these classes must be placed on a scene and attached to a GameObject to function properly. This setup introduces several challenges:

- **Scene Management:** The inability to unload a scene without destroying its managers or adding extra logic to prevent destruction on load can complicate gameplay in larger or 3D games where different elements of the world are placed in separate scenes. Additionally, this can cause issues when attempting to restart the game.
- **Dependency Management**: All dependencies are hard-coded references assigned manually in the Unity inspector. This prevents the use of interfaces and abstract classes to manage dependencies, reducing flexibility. Furthermore, renaming any referenced items can unassign these references, requiring manual reassignment each time.
- **Adding New Dependencies**: Incorporating a new dependency involves a two-step process where a `[SerializedField]` must first be added to the class, and then it must be manually linked in the inspector. This contrasts with Dependency Injection (DI) systems, where only one change is needed.
- **Performance concern**: Performance tests indicated that having everything on the initial scene increases the overhead associated with scene loading.

The class diagram in Figure 11.6 reveals that the `GameManager` in P1 assumes extensive responsibilities. It acts as the starting point of the game and manages scoring, objectives, running the Match-3 game logic, and updating UI components like the `GameHUD` and `StartScreen`. It also oversees player progression. However, since there is no need to reference the `GameManager` externally, all connections are one-way, which slightly simplifies interactions but at the cost of flexibility and scalability.

For P2, the class diagram Figure 11.7 reflects a more complex structure due to the multitude of systems, views, and view models it contains. At the core of this architecture is the `ProjectContext`, which acts as the main container for dependency injection and serves as the initiation point for all systems. Although in the class diagra there is no abstraction layer applied for simplicity, the DI framework allows for binding concrete

types to contract types, enabling easy substitution of implemented types without the need to alter their usages. In the diagram, dependencies are not depicted as direct associations because the types are injected from the context rather than being directly referenced.

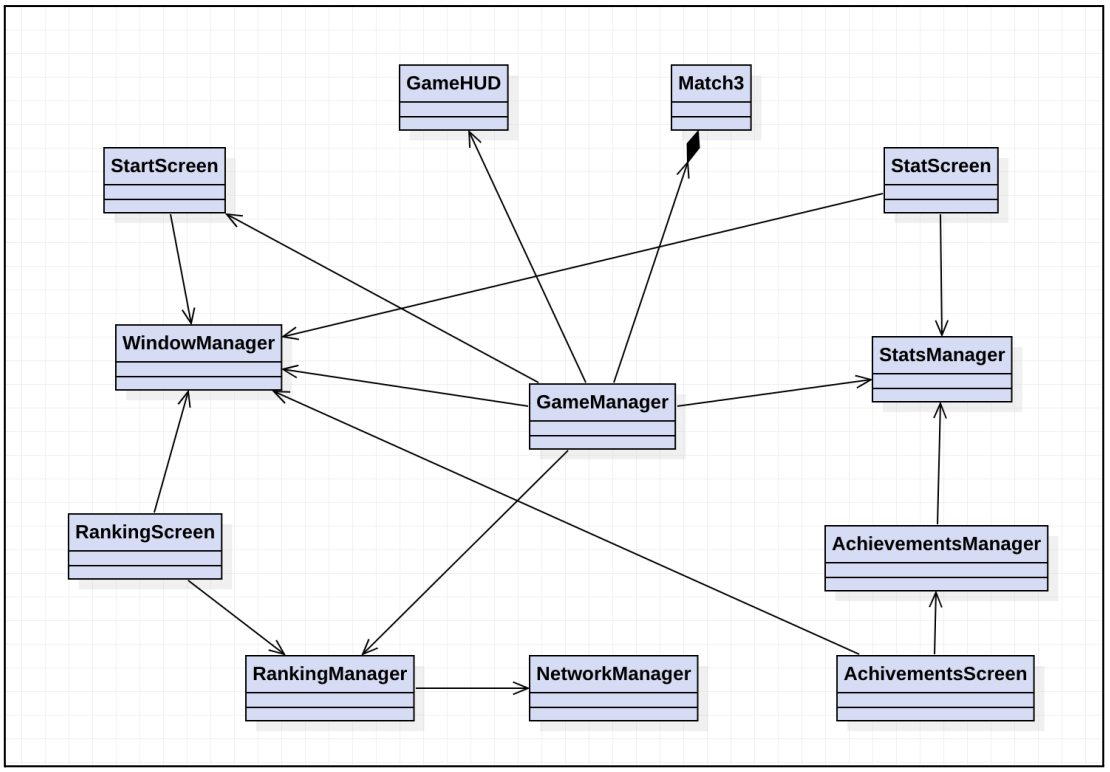

Figure 11.6. Simplified class diagram for implementation P1. (Source: own work)

The game begins with `GameBoot`, which is where the `ProjectContext` registers the systems. However, these systems are created lazily, meaning they are not instantiated until they are specifically resolved. This approach is particularly beneficial in larger games where some systems may only be needed later, optimizing resource utilization. `GameBoot` relies on `WindowSystem` to display the starting view. `StartScreenViewModel` is dependent on both `ProgressionSystem` and `GameplaySystem`, subscribing to their methods to display levels and initiate the game. This architecture is more linear and modular, with each component being accessible only as needed.

However, this structure also introduces challenges. The complexity of dependencies can make the project more difficult to navigate and increases the risk of creating cyclic dependencies, which are not allowed in VContainer and would require refactoring to resolve.

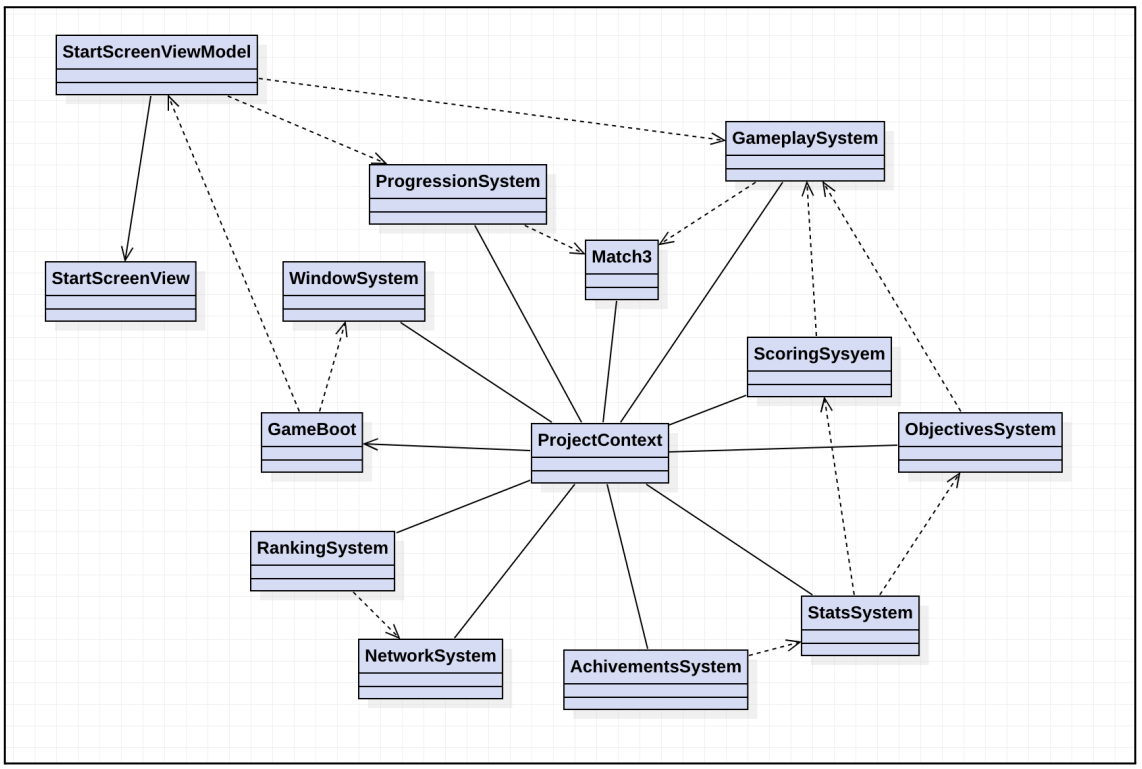

Figure 11.7. Simplified class diagram for implementation P2. (Source: own work)

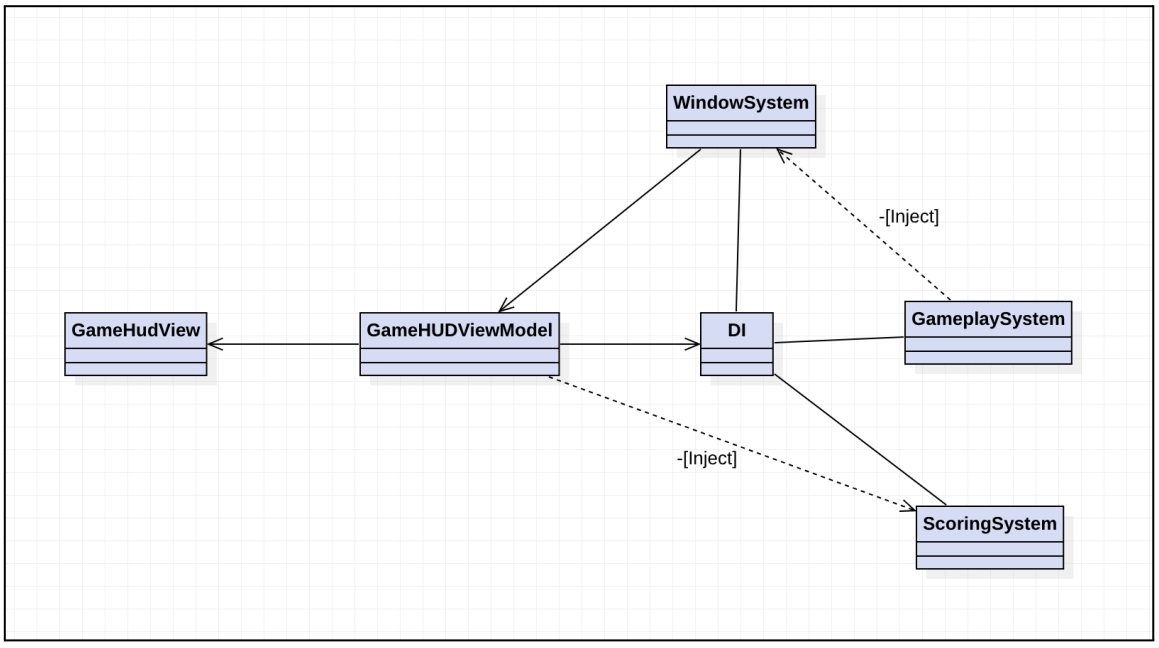

Figure 11.8. Class diagram for implementation of the MVVM architecture in version P2. (Source: own work)

The class diagram also includes a simplified depiction of the MVVM structure used in the UI components. Each UI component follows a similar pattern where a `ViewModel` is driven by the `WindowSystem` upon a request from an external system. The `ViewModel`

subscribes to updates from a `Model` and manages a `View`. For example, the `GameHUDViewModel` is triggered by the `GameplaySystem` through the `WindowSystem`. It subscribes to updates from the `ScoringSystem` and manages the `GameHUDView`.

After examining the overall class structure, it's now an opportune time to delve deeper into the development metrics gathered during implementation. A key metric to consider is the implementation time. As illustrated in Figure 11.9, P2 consistently required more time for completion across nearly every step of the implementation process. This pattern highlights the complexity and the level of detail involved in P2's architectural approach, reflecting its comprehensive and robust framework compared to P1.

Reviewing the aggregated development time as shown in Figure 11.10, it's clear that the incremental differences in time taken for each scenario implementation in P2 accumulate to a significant total difference. Overall, P2 required an additional 102 minutes to implement, which corresponds to a **35.6% increase in time** compared to P1. It's important to note that these measurements do not include time spent reading documentation or designing the architectural structure. Given my prior experience with this type of architecture, it's reasonable to anticipate that the time difference could be even more substantial for a programmer without similar experience. This highlights the complexity and potentially steeper learning curve associated with P2's approach.

The next metric of interest is the count of lines of code (LOC) required for each implementation. It's important to note that the LOC metric excludes any code that is part of external packages, such as VContainer in the case of P2, ensuring that the measure reflects only the code directly authored for the project. Additionally, the measure used here is the **Logical Lines of Code (LLOC)** which focuses on the effective part of each class, excluding imports, namespaces, empty lines, and lines containing only brackets. This approach is adopted to avoid discrepancies that could arise from different code formatting styles or coding and naming conventions, which might otherwise skew the results. By maintaining a consistent coding style across both versions, we ensure that the difference in LOC reflects the actual amount of code a programmer must write and manage, providing a clear comparison between the two implementations.

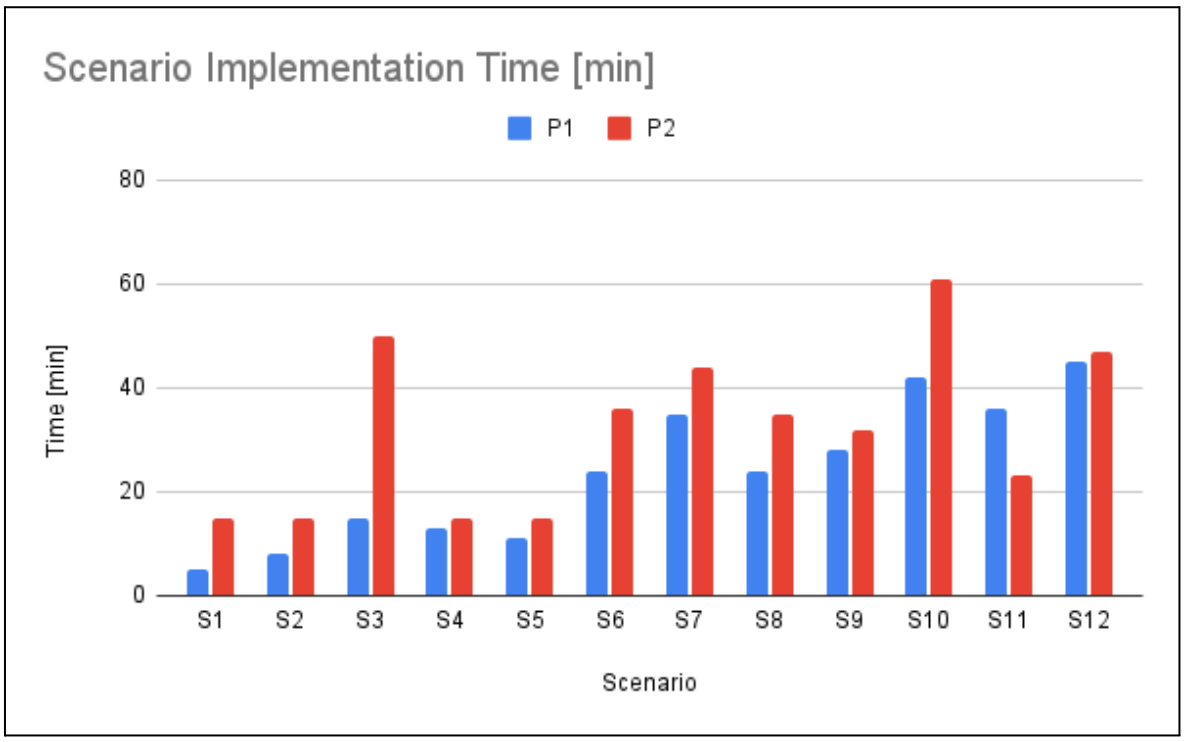

Figure 11.9. Chart of measured development time for both implementations grouped by the scenarios. (Source: own work)

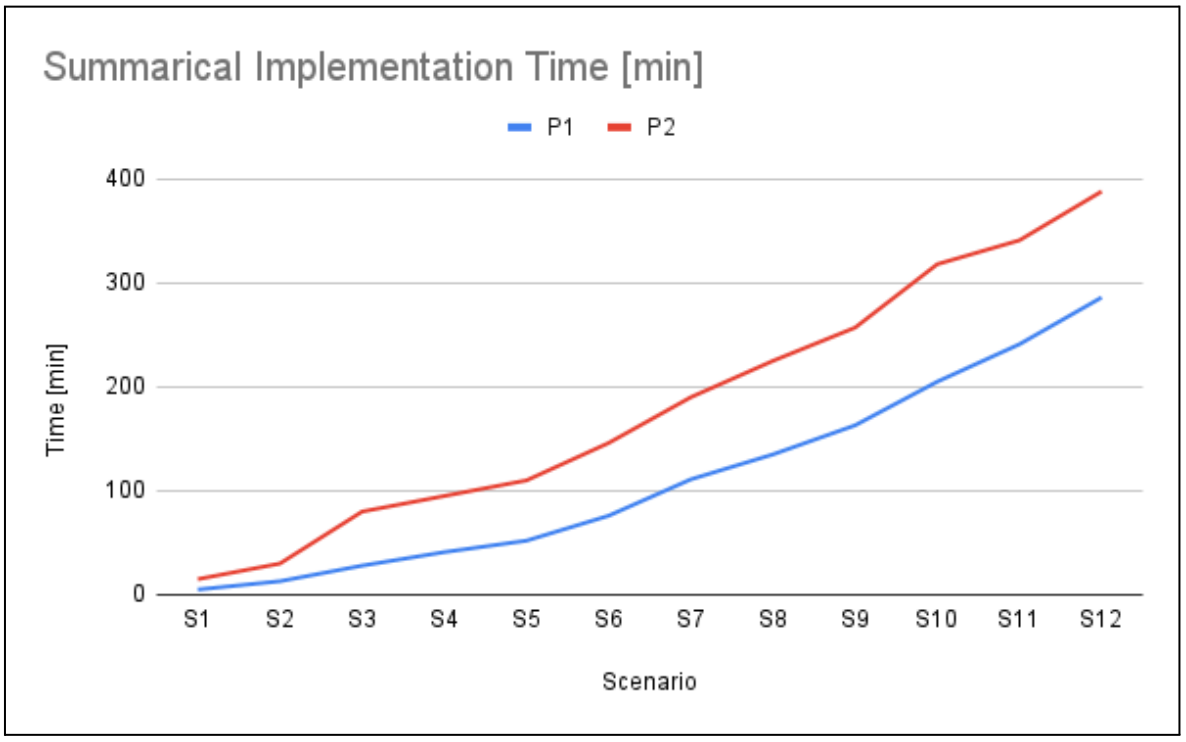

Figure 11.10. Comparison of aggregated development time. (Source: own work)

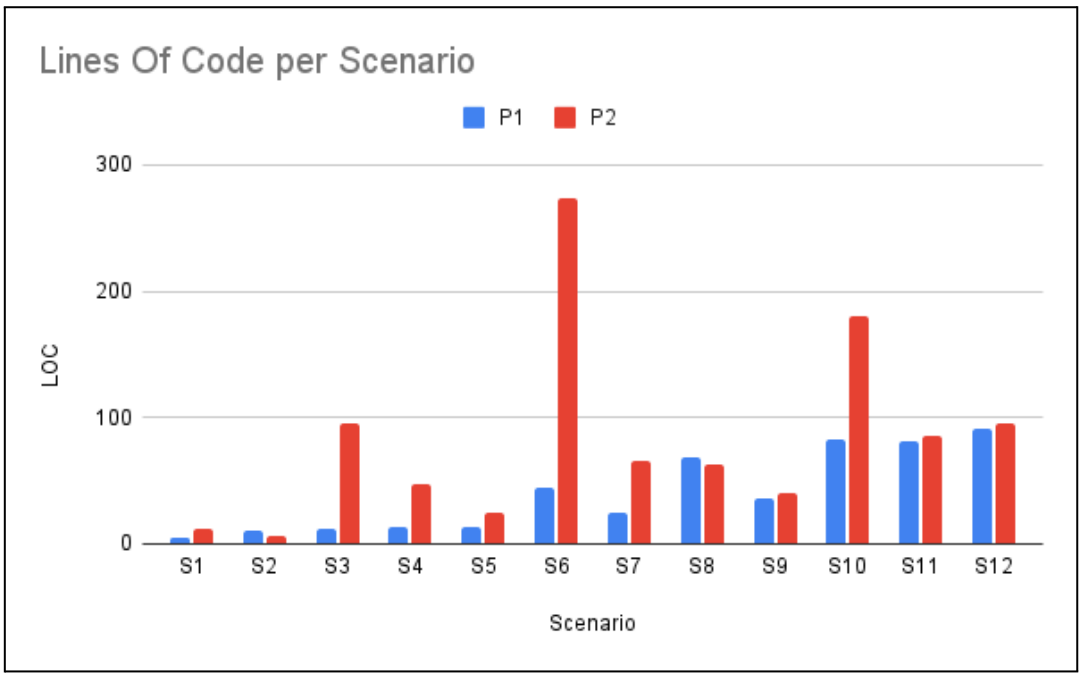

Figure 11.11. Lines of code required for each scenario implementation. (Source: own work)

With the increase in development time, there has also been an increase in the amount of code required. While some scenarios necessitated a similar amount of code between the two implementations, others, particularly S3, S6, and S10, had a more significant impact on P2. Looking at the total lines of code (LOC) for both versions as shown in Figure 11.12, we see a substantial increase: **P2 contains twice as many lines of code as P1.**

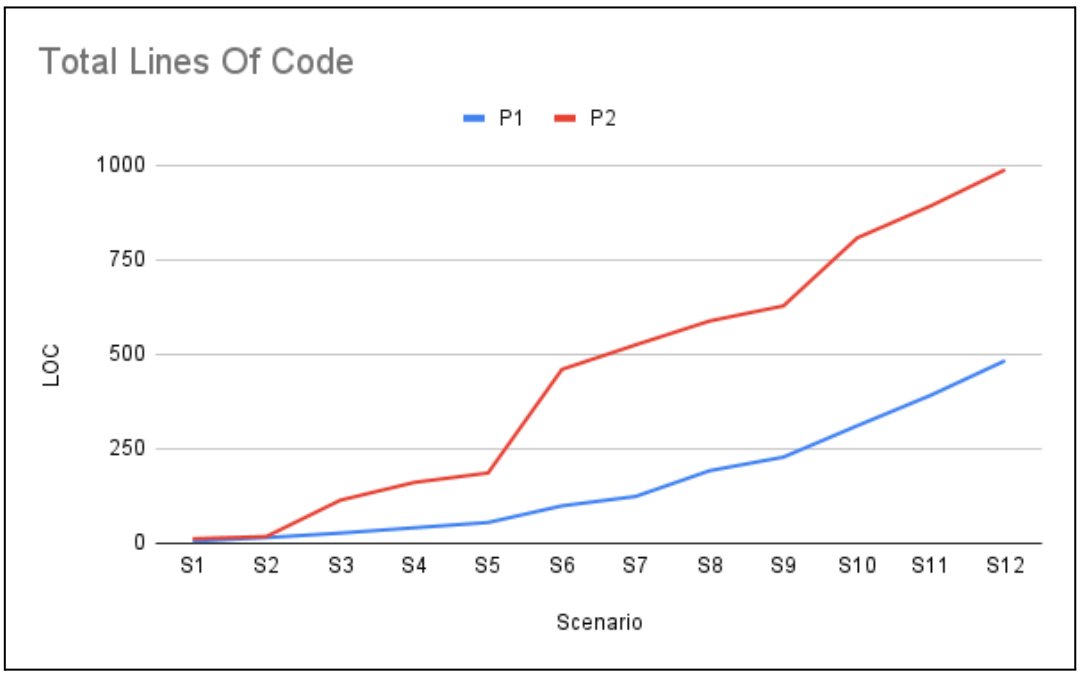

Figure 11.12. Total lines of code added in P1 and P2 to meet the requirements (Source: own work)

It's important to note that having more lines of code does not inherently indicate a problem with either implementation; however, it does increase the overall complexity of the codebase. The crucial factor to consider beyond the mere number of lines is the usability of each solution. This aspect, which involves evaluating how effectively the code supports modifications, scalability, and maintenance, will be the focus of the subsequent sections. This evaluation will help determine the practical implications of the increased code volume in P2.

The Dependency Management Metric, described in Chapter 10, assesses module stability and abstractness by treating each functionality as a distinct module. This metric incorporates:

- Instability (I), which quantifies a module's susceptibility to change by calculating the ratio of efferent (outgoing) to total couplings (incoming and outgoing). Lower values indicate greater stability.
- Abstractness (A), measured by the ratio of abstract classes to total classes, indicating how generic a module is.
- Distance from Main Sequence (D), calculated as $D = |A+I-1|$, evaluates the balance between stability and abstractness, aiming for minimal deviation from the Main Sequence to avoid overly rigid or abstract modules.

Using this measure we can evaluate the overall quality of the dependency management of the project.

Table 11.2. Table with dependency management metrics grouped by the functionality implementation part (Source: own work)

Functionality	**P1**			**P2**		
	I	***A***	***D***	***I***	***A***	***D***
Match-3	0.5	0	0.5	0.33	0.4	0.07
Progression	0.5	0.2	0.3	0.5	0.5	0
Statistics	0	0	1	0.4	0.33	0.27
Achievements	1	0	0	0.5	0.5	0
Ranking	1	0	0	0.66	0.5	0.16

The dependency management data from Table 11.2 shows that P1 mostly avoids abstraction except in the case of window management. Conversely, P2 actively utilizes abstraction, often achieving moderate values. P1's dependency architecture, centered around a global `GameManager`, highlights its instability. P2, using Dependency Injection (DI) to manage dependencies, maintains a semi-stable state for most classes, aligning it closer to the Main Sequence line. This suggests that **P2 manages dependencies more effectively, contributing to a more stable and maintainable architecture**.

Now let's assess the reusability and stability of the game components. Considering that the game is newly developed to meet initial requirements, an important factor is its adaptability—the flexibility of the code structure to accommodate changes. One way to measure adaptability is by examining the number of public methods in classes and their usages. A lower number of public methods typically indicates better encapsulation of the code.

For the UI, P1 implementations have a total of 12 public methods with 12 usages, which suggests that UI components are frequently accessed externally and often require manual updates from outside calls. In contrast, P2 is limited to 4 public methods with 4 usages, all internal within the view system. This difference highlights a tighter encapsulation and less dependency on external interactions within P2's architecture.

Regarding reusability, P2 demonstrates a more robust design by utilizing a binding system where four methods are used 12 times, along with base classes for View and ViewModel, which are reused 7 and 5 times respectively. Additionally, the Observable pattern in P2 is implemented in a total of 23 places, underscoring its high reusability. In P1, however, there is little evidence of component reusability—most elements are used only once. The exception is the Window class, which currently serves merely as a placeholder to group all windows within a single list but does not yet offer functional value.

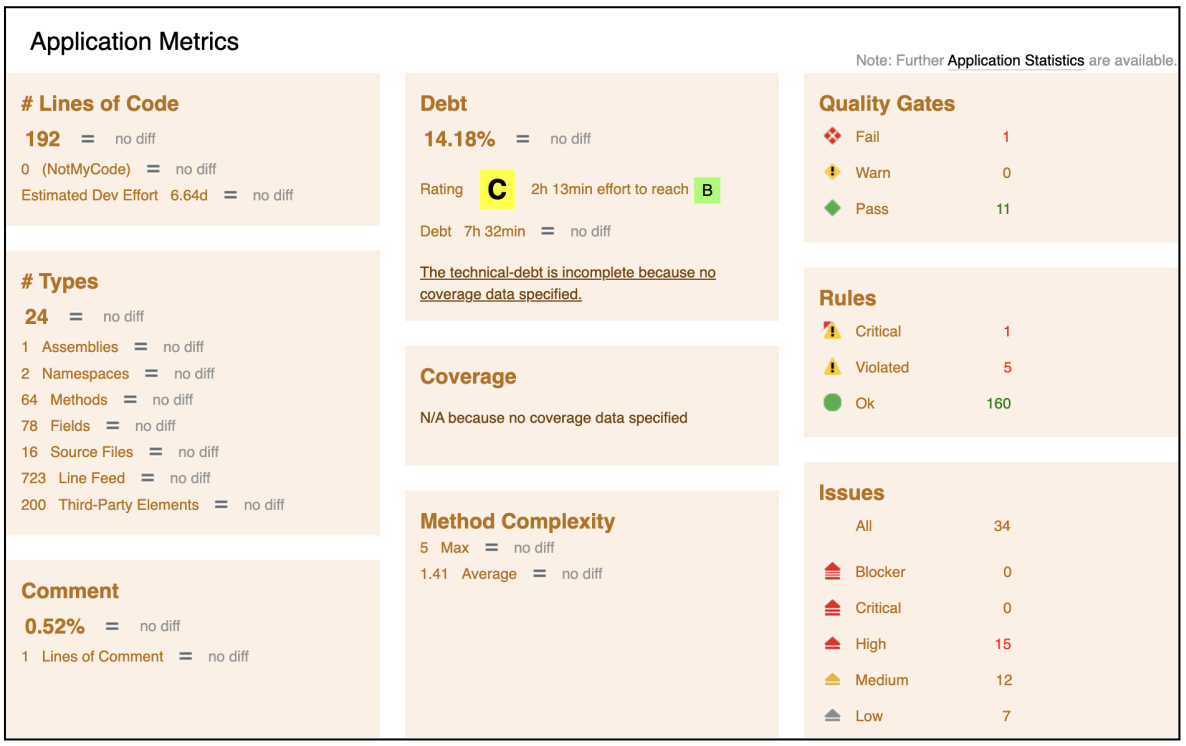

Figure 11.13. Analysis of the P1 assembly definition project. (Source: own work based on NDepend [55])

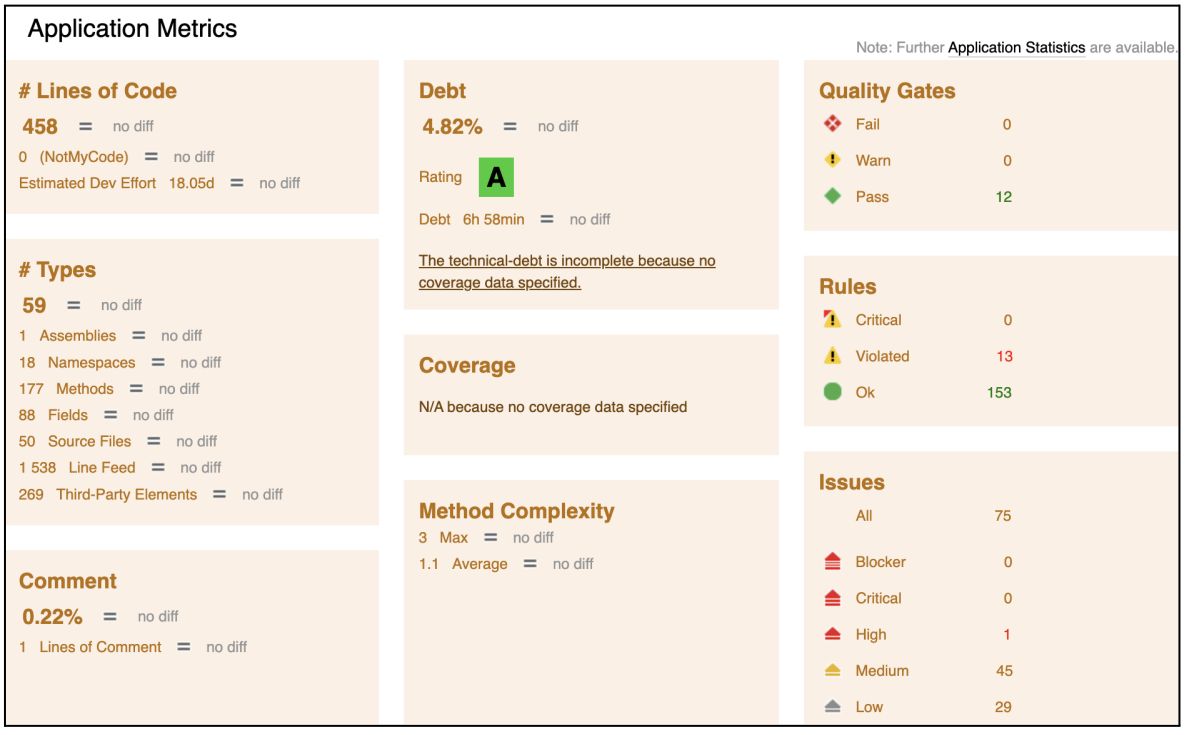

Figure 11.14. Analysis of the P2 assembly definition project. (Source: own work based on NDepend [55])

The use of NDepend [55] to analyze projects P1 and P2 has highlighted significant differences between the two implementations. According to Figure 11.13, P1 has a technical debt rating of 14.18% and an average complexity score of 1.41, but it failed to meet the Quality Gates set by NDepend due to cyclic dependencies. Conversely, as shown in Figure 11.14, P2 achieved a lower debt rating of 4.82% and a reduced average cognitive complexity of 1.1, successfully passing all Quality Gates checks. This demonstrates P2's superior structural integrity and lower maintenance cost.

The detailed statistics in Table 11.3 highlight that despite P2 having more classes and methods, its average method complexity is lower. This difference is further accentuated by its use of an abstraction layer; P2 leverages a Dependency Injection (DI) framework to abstract implementation details behind interfaces, unlike P1, which relies solely on concrete types. This approach is also reflected in other statistics, notably in Table 11.2, demonstrating P2's effectiveness in managing complexity through modularization.

The analysis using tools like NDepend shows that P2, despite having more lines of code, exhibits lower cyclomatic complexity. This indicates that P2 effectively divides functionality into smaller, more manageable modules. In contrast, P1 tends to consolidate extensive functionality into fewer classes, potentially complicating understanding and maintenance.

Table 11.3. Comparison of application statistics for P1 (on the left) and P2 (on the right) (Source: own work based on NDepend [55])

Stat	P1			P2		
	# Occurrences	Avg	StdDev	# Occurrences	Avg	StdDev
Properties on interfaces	0 interfaces	0	0	10 interfaces	1.5	2.01
Methods on interfaces	0 interfaces	0	0	10 interfaces	2.2	1.83
Arguments on methods on interfaces	0 methods	0	0	22 methods	0.18	0.39
Public properties on classes	24 Classes	0.08	0.28	49 Classes	1.31	2.14
Public methods on classes	24 classes	1.71	0.89	49 classes	3.24	2.39

Arguments on public methods on classes	41 methods	0.46	0.83	159 methods	0.43	0.97
IL instructions in non-abstract methods	68 methods	31.01	42.09	224 methods	17.31	29.04
Cyclomatic complexity on non abstract Methods	68 Methods	2.56	3.1	224 Methods	1.85	2.26

This analysis reveals that **P2's architecture not only supports better code encapsulation but also enhances the reusability of its components, making it more adaptable for future expansions or modifications compared to P1 on the cost of development time and overall project complexity.**

11.3. Quality Assurance testing

Now that all implementations are complete and performance tests have been successfully conducted, with no further changes to the code anticipated, it is the appropriate time to ensure the stability and quality of the game. This stage involves rigorous testing to meet several acceptance criteria, aimed at identifying any issues that were not directly addressed during the development phase. Examples of such acceptance criteria include "I can start a Match-3 game" and "After making a move, I see the move counter go down." While this thesis primarily focuses on comparing architectural approaches rather than in-depth software testing, the outcomes of this process will be briefly summarized.

All identified issues will be documented, and the time required to fix them will be allocated to the respective scenario implementation. Most bugs encountered are related to the update of values during transitions, especially when a window is reset after it has already been displayed, which can lead to visible changes in the displayed values. Fortunately, no major bugs were discovered, indicating that the implementations are relatively stable and function as intended. This phase is crucial for confirming the reliability of the game before it is considered ready for release or further deployment.

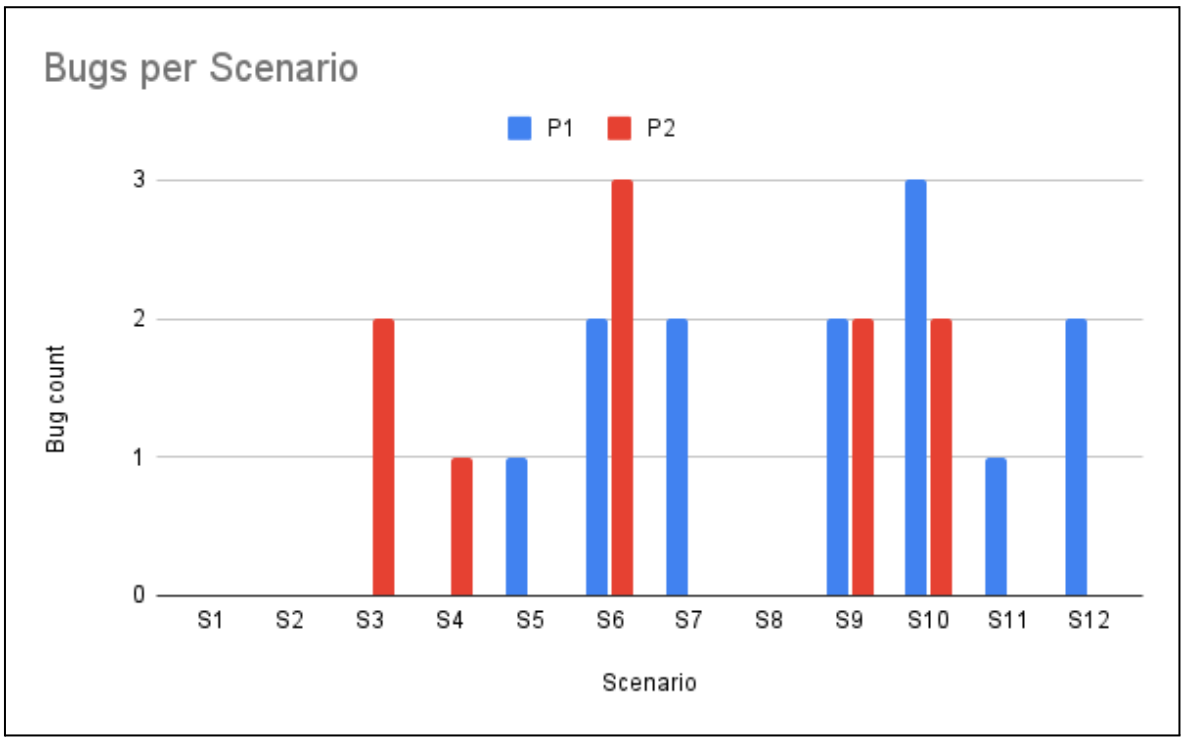

Figure 11.15. Number of bugs introduced in each scenario implementation for P1 and P2 (Source: own work)

In the implementation of P1, 13 bugs were discovered, while in P2, only 10 bugs were identified. However, the mere count of bugs does not provide a full understanding of their impact, as the severity and implications of each bug can vary significantly. For instance, a minor visual glitch in a single frame is considerably less disruptive than a game-crashing exception that demands extensive hours to resolve.

To provide a clearer context for these figures, additional data has been presented in graphs from Figure 11.15. These graphs include the time spent on bug fixing compared to the total implementation time, offering a more detailed perspective on how bug resolution contributes to the overall development effort in both implementations. This comparison helps to highlight not just the frequency of bugs but also the effort required to address them, thereby giving a more comprehensive view of the stability and robustness of each implementation.

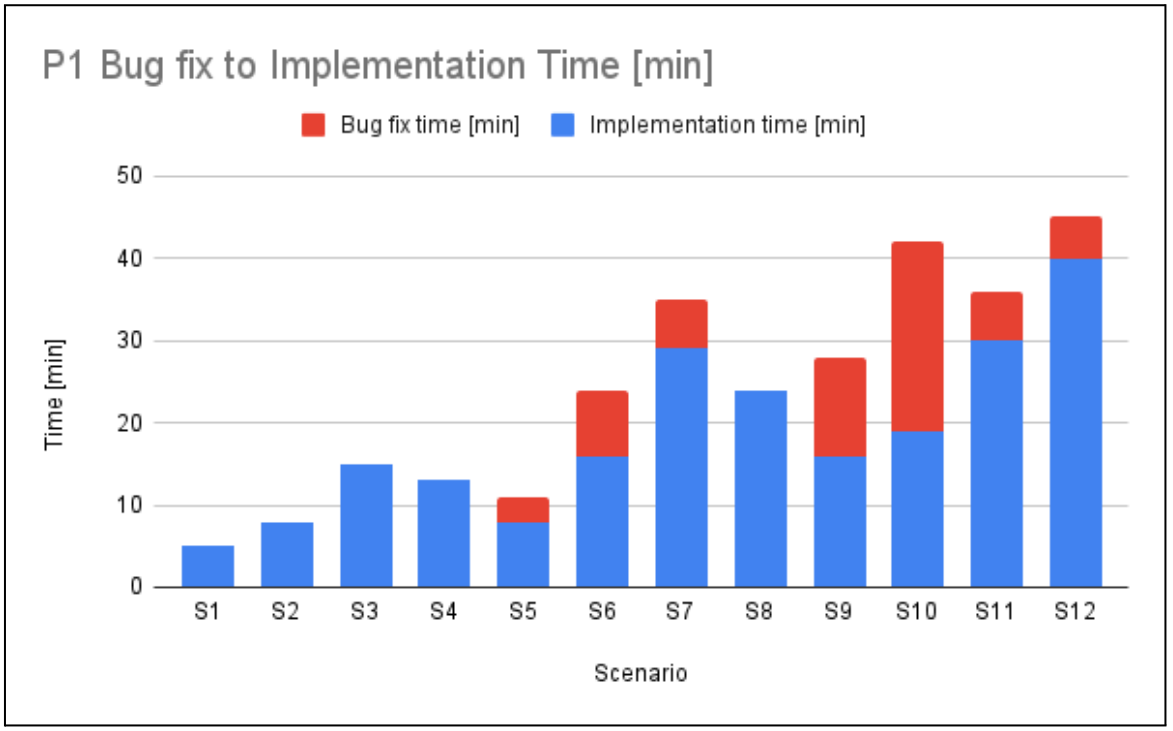

Figure 11.16. Time spent working on the implementation and big fixing for P1 (Source: own work)

The total time spent on fixing bugs was 63 minutes for P1 and 82 minutes for P2. This indicates that although P1 had a higher number of bugs, they were generally less severe and could be resolved quickly. Common issues in P1, such as missing references, were typically straightforward to identify and fix, often taking less than a minute each.

However, more significant challenges arose during scenarios S9 and S10, where the implementation of progression and statistics systems required a minor refactor. This was necessary to properly expose all the relevant values without creating excessive tight couplings, which added complexity and required more time to address effectively.

P2 encountered issues related to dependency management and the observable pattern. The Dependency Injection framework used in P2 helps prevent cyclic dependencies and manages dependencies lazily. This feature, while useful, led to unexpected problems during S3, where the systems for objectives and scoring were resolved too late, resulting in inaccurate counts. Although the issue was relatively straightforward to identify, debugging took longer than anticipated due to the complexity of the dependency injection setup.

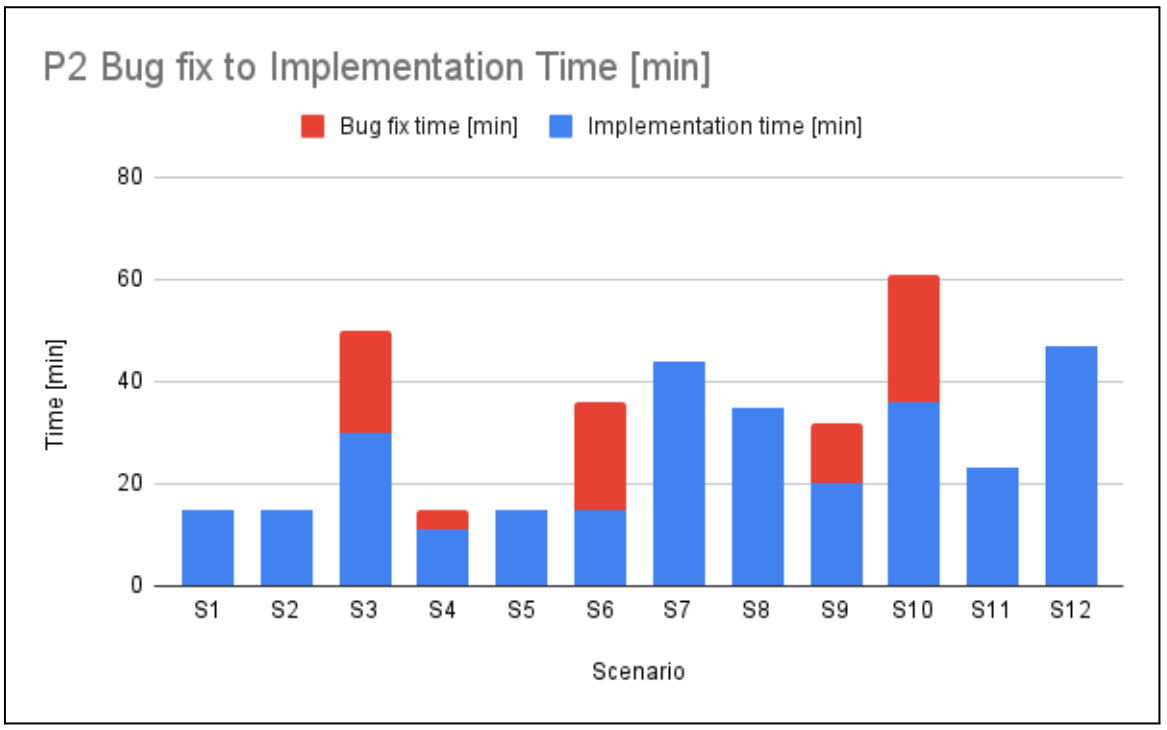

Figure 11.17. Time spent working on the implementation and big fixing for P2 (Source: own work)

Additionally, scenarios S6 and S10 highlighted challenges associated with the observable pattern. While this pattern is effective for managing updatable values, it introduces the risk of failing to unsubscribe or not properly disposing of all subscriptions. These oversights can lead to subtle bugs that manifest as memory leaks over time, making them particularly difficult to detect and resolve. Fortunately, having a robust initial structure for the pattern can mitigate some of these risks, but it requires careful consideration and meticulous management to ensure that subscriptions are handled correctly. This adds an extra layer of complexity when deciding on the implementation of such patterns in a project.

11.4. Cooperation support analysis

The most significant distinction between P1 and P2 regarding collaboration with other specializations, such as game designers or artists, lies in their use of Prefabs and Scriptable Objects. In P1, all elements are directly placed on a scene, whereas in P2, all views are organized as separate prefabs and all configuration settings are managed through separate Scriptable Objects.

The practice of placing everything directly on a scene in P1 often leads to git merge conflicts when multiple team members edit the same scene simultaneously. This approach

might be manageable for a solo developer, but in a larger team setting, adjustments would be necessary to streamline collaboration and version control. In terms of features that are fully configurable from the inspector level, there is no difference between the two versions, maintaining uniformity in how properties can be adjusted directly within Unity's editor.

11.5. ISO/IEC 25000 measurements

To properly determine the comparison with use of industry standards collected measures will be also used to calculate the applicable ISO/IEC measures. Full comparison available in Table 11.4. Since values don't hold a uniform scale and sometimes a smaller number is better while in other cases a bigger number is preferred, a color coding was introduced. In those measures the implementation P2 scores overall higher results. However since the limited size of the produced software the count of modules may be unsatisfactory to decide which solution

Table 11.4. Table for measured ISO/IEC norms (Source: own work)

ID	Name	Definition	P1 value	P2 value
MMD-G-1	Modification Complexity	X = A / B A = Work time spent to modify [min] B = Number of modifications	8.5 = 36 / 4	6.33 = 19 / 3
MMD-G-2	Modification success rate	X = 1 – B/A A = number of troubles within a certain period before modification B = number of troubles in the same period after modification	0.66 = 1 - 1 / 3	0.25 = 1 - 1 / 4
UOP-G-1	Operational consistency	X = 1 – A / B A = number of operations that behave inconsistently B= total number of operations that behave similarly.	0.79 = 1 - 4 / 19	0.96 = 1 - 1 / 23
MRE-S-1	Conformance to coding standards	X = A / B A = Number of conforming modules to coding rules B = Total number of developed software modules	1 = 5 / 5	1 = 11 / 11
MMO-S-1	Localization degree of correction impact	X = A / B A = Number of failures emerged after failure is resolved by modification during specified period B = Number of	0.46 = 6 / 13	0.5 = 5 / 10

		resolved failures		
PRE-S-1	Functional inclusiveness	X = A / B A = number of functions which produce similar results as previously and where changes have not been required. B = number of tested functions which are similar to functions provided by another software to be replaced.	0.75 = 6 / 8	0.5 = 5 / 10

The data presented in Table 11.4 indicate that P2 generally performed better across most metrics, particularly those related to code modification and the consistency of operations. This improvement is attributed to the more structured architecture of P2, which supports enhanced management and adaptability in software development processes.

11.6. Summary

With the comparative analysis of two architectural solutions finished the key findings of the case study include:

- **Development Time and Complexity**: Implementation P2, which utilized advanced architectural patterns and Dependency Injection, took longer to develop and resulted in a more complex codebase compared to the simpler, faster P1 implementation.
- **Bug Analysis and Maintenance**: P1 had more bugs, but they were less severe and easier to fix compared to P2. This indicates that while P2's architecture was more complex, it potentially led to more challenging issues.
- **Performance and Scalability**: Both architectures performed well in terms of gameplay with negligible differences in build sizes and performance metrics. However, P2 offered better scalability and flexibility due to its modular design.
- **Reusability and Modularity**: P2 excelled in reusability and modularity through its use of dependency injection and clear separation of concerns, which could handle future game expansions and modifications more efficiently.
- **Quality and Usability**: The study highlights the trade-offs between development speed and architectural robustness. P2's structured approach was better suited for larger, more complex projects requiring long-term maintenance and scalability.

While P1 is suited for smaller, less complex projects with shorter lifecycles, P2's sophisticated architecture is recommended for larger projects where future expansion and maintainability are critical.

11.7. Recommended architecture based on usage

Choosing the right architecture for Unity3D game development hinges on the project's scale, team size, and specific requirements. For small projects or solo developers, a straightforward architecture like P1 is recommended due to its quick setup and ease of use, making it perfect for prototypes and indie games. Medium to large projects should adopt a more sophisticated architecture like P2, which provides enhanced maintainability and scalability, suitable for teams needing extensive collaboration. Large or enterprise-level projects may require a hybrid or an advanced version of P2 to accommodate high scalability and complex game mechanics effectively. For projects focused on dynamic content creation, a P2-like architecture with enhanced modularity is ideal, enabling efficient updates and expansions. Selecting the appropriate architecture is crucial for optimizing development processes and ensuring that the game can support both current needs and future growth.

It's important to note, however, that while robust architecture is beneficial, it's not always a prerequisite for success. Games like Vampire Survivors [47], which may lack sophisticated architectural design, have still achieved significant success. Conversely, there are numerous instances where projects with well-planned architectures have incurred millions in costs without ever successfully launching. This highlights that while architecture is a critical factor, it should be balanced with other aspects of game development.

11.8. Future directions

Although this thesis provides a comprehensive examination of game architecture within Unity3D, it only begins to explore the vast complexities involved. Future research could expand upon this work by integrating more systems and further implementations into the existing projects analyzed here. Additionally, considering the diverse requirements of games from different genres and scales, subsequent studies could focus on completely new implementations tailored to these varied needs. Another promising avenue is to deconstruct the architecture into smaller, more manageable components and conduct detailed analyses

of specific elements, such as UI implementation. The field of game architecture, especially as documented in academic literature, remains underexplored and ripe for deeper investigation. This thesis underscores the necessity for continued research to enhance our understanding and implementation of effective game architectures.

An interesting direction would be an investigation of each solution code smells [48] as well as potential technical debt [49] created by it.

12. Conclusions

This thesis has provided a detailed comparative analysis of two distinct architectural approaches within the Unity3D game engine, aimed at enhancing the understanding of how architectural choices can impact the development process and final product in game development completing its goal. Through the examination of a Match-3 game scenario, implemented under two different architectural frameworks, key insights have been gained regarding the trade-offs between simplicity and complexity in game architecture.

P1, with its straightforward approach, proves effective for smaller teams or solo developers where fewer integration points reduce the complexity and potential for merge conflicts. This approach emphasizes rapid development and ease of understanding, making it suitable for projects with less demanding architectural needs.

On the other hand, P2's use of advanced design patterns like MVVM, Dependency Injection, and the separation of concerns through Prefabs and Scriptable Objects caters well to larger teams and projects with a need for scalability and maintainability. While P2 involves a steeper learning curve and initially higher development times, its structured approach reduces long-term maintenance challenges and facilitates better collaboration among team members from different specializations.

If there were a single best architecture, every project would adopt it. However, each solution has its pros and cons, making the decision dependent on the developers. The start of each project should be preceded by a careful investigation of its specific needs.

Ultimately, the choice between these architectural models should be guided by the specific requirements and context of the project, including team size, project complexity, and future maintenance and scalability needs. This thesis underscores the importance of careful architectural planning in game development, ensuring that the chosen architecture aligns with both the immediate and long-term goals of the project.

Bibliography

[1] "Game release stats," SteamDB. Available: https://steamdb.info/stats/releases/. Accessed on: May 9, 2024.

[2] S. Russell, M. Graetz, and W. Wiitanen, Spacewar!, [Video game]. Cambridge, MA, USA: MIT, 1962.

[3] Newzoo, "Global Games Market Report," Newzoo, Amsterdam, Netherlands, 2023.

[4] M. Johnson, "The esports industry predicts 2023," Esports Insider, London, UK, 2023. [Online]. Available: https://esportsinsider.com/2023/01/esports-industry-predicts-2023 [Accessed: May 9, 2024]

[5] "Ten Square Games," Ten Square Games. [Online]. Available: https://tensquaregames.com/ . [Accessed: May 9, 2024].

[6] J. Gregory, Game Engine Architecture, 3rd ed. Boca Raton, FL, USA: CRC Press, 2018.

[7] "Unity," Unity Technologies. [Online]. Available: https://unity.com/. [Accessed: May 9, 2024].

[8] "Unreal Engine," Epic Games. [Online]. Available: https://www.unrealengine.com/. [Accessed: May 9, 2024].

[9] "Godot Engine," Godot Engine. [Online]. Available: https://godotengine.org/ [Accessed: May 9, 2024].

[10] "CryEngine," Crytek. [Online]. Available: https://www.cryengine.com/. [Accessed: May 9, 2024].

[11] "RPG Maker," Kadokawa Corporation. [Online]. Available: https://www.rpgmakerweb.com/. [Accessed: May 9, 2024].

[12] "Construct," Scirra Ltd. [Online]. Available: https://www.construct.net/. [Accessed: May 9, 2024].

[13] Itch.io, “Most used Engines,” *itch.io*, 2024. https://itch.io/game-development/engines/most-projects [Accessed: May 9, 2024].

[14] "Steam," Valve Corporation. [Online]. Available: https://store.steampowered.com/. [Accessed: May 9, 2024].

[15] A. Penzentcev, "Architecture and implementation of the system for serious games in Unity3D," Master Thesis, Brno, May 2015 .

[16] steam.db, “Technologies,” steam.db, 2024. https://steamdb.info/tech/ [Accessed: May 9, 2024].

[17] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley, 2002.

[18] "NVIDIA GeForce NOW," NVIDIA Corporation. [Online]. Available: https://www.nvidia.com/en-us/geforce-now/. [Accessed: May 9, 2024].

[19] "BlueStacks," BlueStack Systems, Inc. [Online]. Available: https://www.bluestacks.com/. [Accessed: May 9, 2024].

[20] S. Aleem, L. F. Capretz, and F. Ahmed, "Game Development Software Engineering Process Life Cycle: A Systematic Review," Journal of Software Engineering Research and Development, vol. 4, no. 6, pp. 1-30, Nov. 2016, doi: 10.1186/s40411-016-0032-7.

[21] King, Candy Crush Saga, [Video game]. Stockholm, Sweden: King Digital Entertainment, 2012.

[22] S. M. Doherty, J. R. Keebler, S. S. Davidson, E. M. Palmer, and C. M. Frederick, "Recategorization of Video Game Genres," in Proc. of the Human Factors and Ergonomics Society Annual Meeting, vol. 62, no. 1, pp. 2099-2103, 2018. doi: 10.1177/1541931218621473

[23] C. Anderson, “The Model-View-ViewModel (MVVM) Design Pattern.” Pro Business Applications with Silverlight 5. Apress, Berkeley, CA. (2012). doi: 10.1007/978-1-4302-3501-9_13

[24] N. Parviainen, "Dependency Injection in Unity3D," Bachelor thesis, JAMK University of Applied Sciences, Jyväskylä, Finland, Mar. 2017.

[25] T. Härkönen, "Advantages and Implementation of Entity-Component-Systems," Bachelor thesis, Dept. Inf. Technol., Tampere Univ., Tampere, Finland, Apr. 2019.

[26] A. M. Barczak and H. Woźniak, "Comparative Study on Game Engines," Studia Informatica. Systems and Information Technology. Systemy I Technologie Informacyjne, no. 1-2, pp. 5-24, 2019. doi: 10.34739/si.2019.23.01

[27] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston, MA, USA, pp. 21-43, 145-161: Prentice Hall, 2017. ISBN: 978-0-13-449416-6.

[28] W.-L. Wang, M.-H. Tang, and M.-H. Chen, "Software architecture analysis—a case study," in Proc. Twenty-Third Annual International Computer Software and Applications Conference (COMPSAC '99), Phoenix, AZ, USA, Oct. 27-29, 1999, pp. 265-270. doi:10.1109/CMPSAC.1999.812714.

[29] "Software Architecture Definition," Software Engineering Institute, Carnegie Mellon University. [Online]. Available: https://www.sei.cmu.edu/our-work/software-architecture/. [Accessed: May 9, 2024].

[30] Bethke, Erik. *Game development and production*. Wordware Publishing, Inc., 2003.

[31] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. Northrop, "A Basis for Analyzing Software Architecture Analysis Methods," Software Quality Journal, vol. 13, no. 4, pp. 329-355, 2005. doi:10.1007/s11219-005-4250-1.

[32] N. Altoyan and D. E. Perry, "Towards a well-formed software architecture analysis," in Proc. 11th European Conf. on Software Architecture (ECSA '17), Canterbury, United Kingdom, Sep. 11-15, 2017, pp. 173-179. doi: 10.1145/3129790.3129813.

[33] "Candy Crush Saga," Google Play Store. [Online]. Available: https://play.google.com/store/apps/details?id=com.king.candycrushsaga [Accessed: May 9, 2024].

[34] J. Juul, A Casual Revolution: Reinventing Video Games and Their Players. Cambridge, MA, USA, pp. 45-52: MIT Press, 2009. ISBN 978-0-262-01337-6.

[35] J. Juul, "Swap Adjacent Gems to Make Sets of Three: A History of Matching Tile Games," Artifact, vol. 1, no. 4, pp. 205-216, Dec. 2007. doi: 10.1080/17493460601173366.

[36] C. Fabricatore, "Gameplay and Game Mechanics Design: A Key to Quality in Videogames," 2007. doi: 10.13140/RG.2.1.1125.4167.

[37] L. S. Ferro, "The Game Element and Mechanic (GEM) framework: A structural approach for implementing game elements and mechanics into game experiences," Entertainment Computing, vol. 36, 2021, Art. no. 100375, ISSN 1875-9521, doi:10.1016/j.entcom.2020.100375.

[38] International Organization for Standardization, "ISO/IEC 25000:2014 Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - Guide to SQuaRE." [Online]. Available: https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html [Accessed: May 9, 2024]

[39] "Flip This House," Ten Square Games - archived game. [Online]. Available: https://flip-this-house.en.uptodown.com/android [Accessed: May 9, 2024]

[40] J. Gustafsson, "Single Case Studies vs. Multiple Case Studies: A Comparative Study," 2017.

[41] "VContainer - DI for Unity," hadashiA. [Online]. Available: https://vcontainer.hadashikick.jp/ [Accessed: May 9, 2024]

[42] R. Colby and R. S. Colby, "Game design documentation: four perspectives from independent game studios," Commun. Des. Q. Rev., vol. 7, no. 3, pp. 5-15, Sep. 2019, doi:10.1145/3321388.3321389.

[43] "UniTask," Cysharp UniTask, async/await integration to Unity. [Online]. Available: https://cysharp.github.io/UniTask/ [Accessed: May 9, 2024]

[44] Shvets, A. (2018). Dive into Design Patterns [Online] Available: https://refactoring.guru/design-patterns/book [Accessed: May 9, 2024]

[45] R. Nystrom, Game Programming Patterns, pp. 43-61, 156-168. Genever Benning, 2014. ISBN: 978-0-9905829-0-8.

[46] T. Yamakami, "An Exploratory Analysis of KPI in Mobile Social Battle Games," in Human Centric Technology and Service in Smart Space, J. Park, Q. Jin, M. Sang-soo Yeo, and B. Hu, Eds., vol. 182, Lecture Notes in Electrical Engineering. Dordrecht: Springer, 2012. doi: 10.1007/978-94-007-5086-9_16

[47] "Unity Case Study: Vampire Survivors," Simon Norton, Medium. [Online]. Available: https://medium.com/@simon.nordon/unity-case-study-vampire-survivors-806eed11bebb [Accessed: May 9, 2024].

[48] T. Paiva, A. Damasceno, E. Figueiredo et al., "On the evaluation of code smells and detection tools," J. Softw. Eng. Res. Dev., vol. 5, art. no. 7, 2017.doi: 10.1186/s40411-017-0041-1.

[49] H. Kleinwaks, A. Batchelor, and T. H. Bradley, "Technical debt in systems engineering—A systematic literature review," First published, Apr. 10, 2023. doi: 10.1002/sys.21681

[50] "Unity Case Study: Vampire Survivors," Simon Norton, Medium. [Online]. Available: https://invogames.com/blog/how-much-does-it-cost-to-make-a-video-game/ [Accessed: 09 May 2024]

[51] I. Marcis, Software Engineering. Rutgers University, New Brunswick, New Jersey, pp. 217-245: Rutgers University, 2012.

[52] JetBrains, "Rider," JetBrains s.r.o., [Software]. Available: https://www.jetbrains.com/rider/. [Accessed: May 9, 2024].

[53] SonarSource, "SonarQube," SonarSource SA, [Software]. Available: https://www.sonarqube.org/. [Accessed: May 9, 2024].

[54] M. Hitz and B. Montazeri, "Chidamber and Kemerer metrics suite: A measurement theory perspective," IEEE Transactions on Software Engineering, vol. 22, no. 4, pp. 267-271, May 1996. doi:10.1109/32.491650.

[55] NDepend, "NDepend - 14-days trial," ZEN PROGRAM, [Software]. Available: https://www.ndepend.com/. [Accessed: May 9, 2024].