Graduation Report

Codeglue Cross-Platform Library

Saxion University of Applied Sciences
Creative Media & Game Technologies (Game Engineering)

Student
Name: Igli Milaqi
Student Number: 445119
Contact: milaqiigli@gmail.com

Saxion Coach:
Name: Paul Bonsma
Contact: p.s.bonsma@saxion.nl
School: Saxion University of Applied Sciences

Company Supervisor:
Name: Rens Van Mierlo
Contact: rens@codeglue.com

Date of submission: 15-06-2021

1 Contents

2 Introduction

Cross-platform development has always been a challenge in the game development industry since each of them performs the same tasks in different ways. Platform vendors provide SDKs that expect data in a certain way, and they are different for each platform. For this reason, different implementations are required even for the same behaviour, as may be unlocking and handling achievements. For example, handling achievements on PS4 is very different from handling achievements on Xbox.

Codeglue hereinafter referred to as “The Company” deals mainly with porting video games in a variety of platforms. For this reason, the company wants to solve the issue of having to handle platform specific dependencies throughout their projects.

The main task for this graduation period is creating a cross-platform library that can be connected to a game project, and it should automatically handle platform-specific API implementation.

2.1 The Company

The Company is a game development and porting studio, founded in 2000 by Maurice Sibrandi and Peter de Jong. The company is located in Rotterdam, Netherlands and currently has 20 employees.

On their website, the Company refers to itself as “technology specialists” meaning the latter can collaborate with publishers and other game developers, and they are always tinkering with the latest innovations in gaming such as VR, AR and blockchain technology. The Company has ported popular games on different platforms, including “Terraria”, “Human Fall Flat” and “The Escapists 2”.

3 Reason for the assignment

As mentioned above, The Company deals with porting games to a variety of platforms. Each platform has its unique SDK given by the vendor and requires different setup and interfacing, which causes the Company to write the same systems or implementations throughout different projects.

The code is mostly the same over all the projects. Therefore, the Company wants to create a library that they can add to a project that will contain many of the basic API calls. This enables the development of the game code separately from the platform dependencies. If needed, new platforms can be added to the library.

The main reasons why the Company has given this assignment are to:

- reduce the rewriting of the same functionality over and over again;
- create a unified codebase for the platform logic that can be maintained and tested in isolation;
- split the game codebase from the platform-specific implementation so that different tasks can be performed in parallel without interfering with each other;
- automize the importing process.

4 Objectives of the client

The main objective of the client is to find a way to handle platform-specific API calls that are needed throughout different Unity projects.
In order to reach their objective, the company wants to develop a library that handles these calls and that can be deployed into different Unity projects without interfering with the game codebase.
The library is intended to be used by the developers within the company and it should be easy to maintain and extend in the future.

Handling every single platform out there is outside the scope of this graduation project. For this reason, this library will focus on handling the two main console platforms (PlayStation, Xbox). The library should however be extendable so that in the future the developers within the company can add other platforms.

The library is going to be deployed in different projects that need to handle the platform API, and it should provide a unified interface that the game layer can use.
There are many platform behaviours and thus the scope would be too big to handle everything, thus this library will handle only the following generic modules:

- Storage Module. This module will be responsible for storing the data on the device. Makes sure everything is converted to the right formats. Only the storage of the active account can be reached;

- Achievement Module. This module will take care of everything related to achievements, such as providing the list of achievements, unlocking achievements, checking if an achievement has been unlocked already, etc;

- User Module. This module will take care of user-related actions caused by the user or the systems, like user signing in/out, controllers disconnecting;

- Authentication Module. This module will check if the player has access to certain privileges. For instance, an Xbox gold membership is needed to be able to use multiplayer functionality on the Xbox One platform;

- Friend Module. This module will deal with everything related to friends and contacts within the platform;

- Text Module. Each platform needs virtual keyboards. Everything Text related will be handled by the text module including the profanity filter.

5 Problem Definition

The client's problem is to create an architecture/library that can fulfil the following goals:

- Ease of extension;
- Platform separation;
- Library packaging and deployment;
- Module decoupling.

For the "Platform separation" and "Module decoupling" the "Instability & Abstraction Analysis" will help determine if the new architecture is meeting the specified goals and if it has improved in relations to the old architecture.

For the "Ease of extension" goal, multiple modules will be implemented for at least 2 console platforms (Ps4 and Xbox one).

The library is going to be used by the company and it is not meant to be distributed publicly. After the modules are implemented feedback from the company's coach will be gathered to determine if these goals are met.

Regarding packaging and deploying the library, different ways of packaging a codebase in C# will be analysed. The library will also be deployed in an already existing game within the company to evaluate the deployment process.

The top priority for the Company is to have an independent codebase that handles platform-specific functionality and adapts to different SDK versions.

6 Main and Sub Questions

6.1 Main Question

How to create an independent cross platform library for the company that handles console-specific SDKs, in order that the company can reuse and extend the codebase throughout different projects?

To answer the main question, the following sub-questions will be analysed:

1. What is the most suited architecture for the cross-platform library?
2. How will the library be packaged and deployed throughout different projects?
3. How to test the addition of new platforms to the library?

6.2 What is the most suited architecture for the cross-platform library?

6.2.1 Theory

Before looking into already existing architectures that are suited for cross-platform development, theory on what makes a good architecture and how to measure its effectiveness is gathered and analysed.
The book "Antipatterns" by William J. Brown et al (1998), introduces an antipattern called "Architecture by implication". In contrast to design patterns that show best practices in a certain field, antipatterns showcase a common problem and how to avoid it.
This antipattern explains the drawbacks of implementing an architecture without planning it first. Some of the negative effects this antipattern creates are shown below:

• Lack of architecture planning and specification; insufficient definition of architecture for software, hardware, communications, persistence, security, and systems management.

• Hidden risks caused by scale, domain knowledge, technology, and complexity, all of which emerge as the project progresses.

This Antipattern is characterized by not having architecture specification for a given codebase. The developers responsible for the codebase have prior experience with the system architecture and therefore assume there is no need for documentation and high-level specifications.
To avoid the antipattern and define a clear architecture planning, the Antipatterns book suggests using the "Goal-Question Architecture" (GQA).

Goal-Question Architecture (GQA)

GQA is the opposite of the "Architecture by implication" antipattern mentioned above. It states that planning is an essential part of the systems development and that planning itself is an iterative process, meaning that an initial architecture definition should have good strategies but as the development goes on it should be refined to suit the goals of the architecture.

To define a "Goal-Question Architecture" the following steps must be taken:

1. Define the architecture goals. What must this architecture achieve? Which stakeholders, real and imaginary, must be satisfied with the design and implementation? What is the vision for the system?

2. Define the questions. What are the specific questions that must be addressed to satisfy? Define the stakeholder issues and prioritize the questions to support the view selection.

3. Select the views. Each view will represent a blueprint of the system architecture. In GAQ a "view" is a lightweight specification that defines the system based on the perspective of the stakeholders.

Architecture Goals:

- Handling platform-specific functionality for the required consoles
- The gameplay code is independent of the platform code.
- Easy to test.
- Handling multiple or be able to switch SDK versions.

The Questions:

- What is the most fitting architecture pattern for the cross-platform library?
- How to unify the current existing code as one independent library that can be deployed on multiple projects within the company?
- How to allow the extension of new modules and support for new platforms in the future?

The Views:

In this case, the stakeholders are the developers at the company who are going to use the library. From the perspective of the stakeholders the library should handle:

- Achievements
- Storage (Saving & Loading) data
- Users
- Authentication
- Friends
- Text

Screaming Architecture Pattern

The screaming architecture is an architectural pattern mentioned in the book "Clean Architecture" by Robert C. Martin. This pattern states that architecture should tell readers about the system, not about the frameworks used in the system.
Just by looking at the high-level interfaces, it should be clear what your overall architecture is, what is the purpose of your architecture or as the name suggests what does your architecture scream.

In the library's architecture, all the views that were chosen above should handle behaviour specific to their module. For example, the "Achievements" module should handle achievement-related behaviours; no more and no less.

Instability & Abstraction Analysis

The book "Clean Architecture" by Robert C. Martin shows a method to analyse the instability and abstraction of a given architecture.
Instability (I) measures how easy or hard it is for a component/module to change. To measure the instability three variables are introduced:

• Fan-in: Incoming dependencies. This metric identifies the number of classes outside this component that depend on classes within the component.
• Fan-out: Outgoing dependencies. This metric identifies the number of classes inside this component that depends on classes outside the component.
• I: Instability: I = Fan-out / (Fan-in + Fan-out). This metric has the range [0, 1]. I = 0

indicates a maximally stable component. I = 1 indicates a maximally unstable component.

Column1	Achievement Module	Storage Module	User Module	Friend Module	Text Module	Platform Module	Network Module	
Fin	0	1	5	0	0	0	1	
Fout	2	1	1	1	1	2	1	
I = Fout / (Fin + Fout)	1	0,5	0,16	1	1	1	0,5	
Column1	Achievement Module	Storage Module	User Module	Friend Module	Text Module	Platform Module	Network Module	
NA	1	1	1	2	1	1	1	
NC	4	6	4	8	4	3	1	
A = NA / NC	0,25	0,16	0,25	0,25	0,25	0,33	1	
Column1	Achievement Module	Storage Module	User Module	Friend Module	Text Module	Platform Module	Network Module	
D = |A + I -1|	0,25	0,34	0,59	0,25	0,25	0,33	0,5	Average = 0,358

Figure 1. Instability and abstraction data gathered for the existing codebase.

The first table in Figure 1 above shows the calculated "I" value for all the modules implemented for the old codebase.

To measure the abstraction of a module, three new concepts must be calculated:

- Nc: The total number of classes in the module (including the abstract classes).
- Na: The number of abstract classes and interfaces in the component.
- A: Abstractness. A = Na ÷ Nc

The second table in Figure 1 shows the calculated "A" for each module. If "A" is equal to 0 it implies the module has no abstract classes at all and if "A" is equal to 1 it means, there are no concrete classes in the module.
The graph below shows the relationship between Abstraction (A) and Instability (I) for a given module. In the perfect scenario, we will find the components that are maximally stable and abstract at the upper left at (0, 1).
The components that are maximally unstable and concrete are at the lower right at (1, 0).

Since not all modules fall under these two positions "The Main Sequence" defines a reasonable position for a module to be. To calculate how close a module is relative to the main sequence another variable is introduced:

D: Distance. D = |A+I−1|. The range of this metric is [0, 1]. A value of 0 indicates that the component is directly on the Main Sequence. A value of 1 indicates that the component is as far away as possible from the Main Sequence.

The last table in Figure 1 shows the calculated D for each module. The average of all the modules is around 0,358.

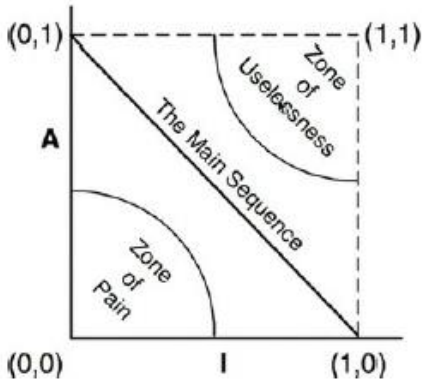


Figure 2. Main sequence diagram. Taken from the book "Clean Architecture" by Robert C. Martin, 2018.

If the distance "D" is somewhere in the zone of pain it means that the codebase has a lot of concrete classes with many dependencies. Contrary to the foregoing, if the distance is on the zone of uselessness, the codebase has only abstract classes and it contains no implementations.

Layered Architecture

The layered architecture, also known as the n-tier architecture, splits the overall library into multiple layers. Each layer contains components that change or update the client's request which goes from the top layer to the bottom layer, and it does not skip any layers unless the layer is defined as an open layer (all layers are closed by default).

The most common layers are:

- Presentation Layer => Handles all the user interfacing as well as serves as an entry point to the architecture.
- Application Layer => This layer serves as an abstraction layer between business layer logic and presentation layer logic. It can also provide a place to put certain coordination logic that does not fit in the business or presentation layer.
- Business Layer => All models and logic that is specific to the business problem you are trying to solve go in this layer.
- The last two layers (Persistence & database) are database specific layers and are not specific to the library's architecture.

A diagram showcasing this architecture is shown in Figure 3 below:

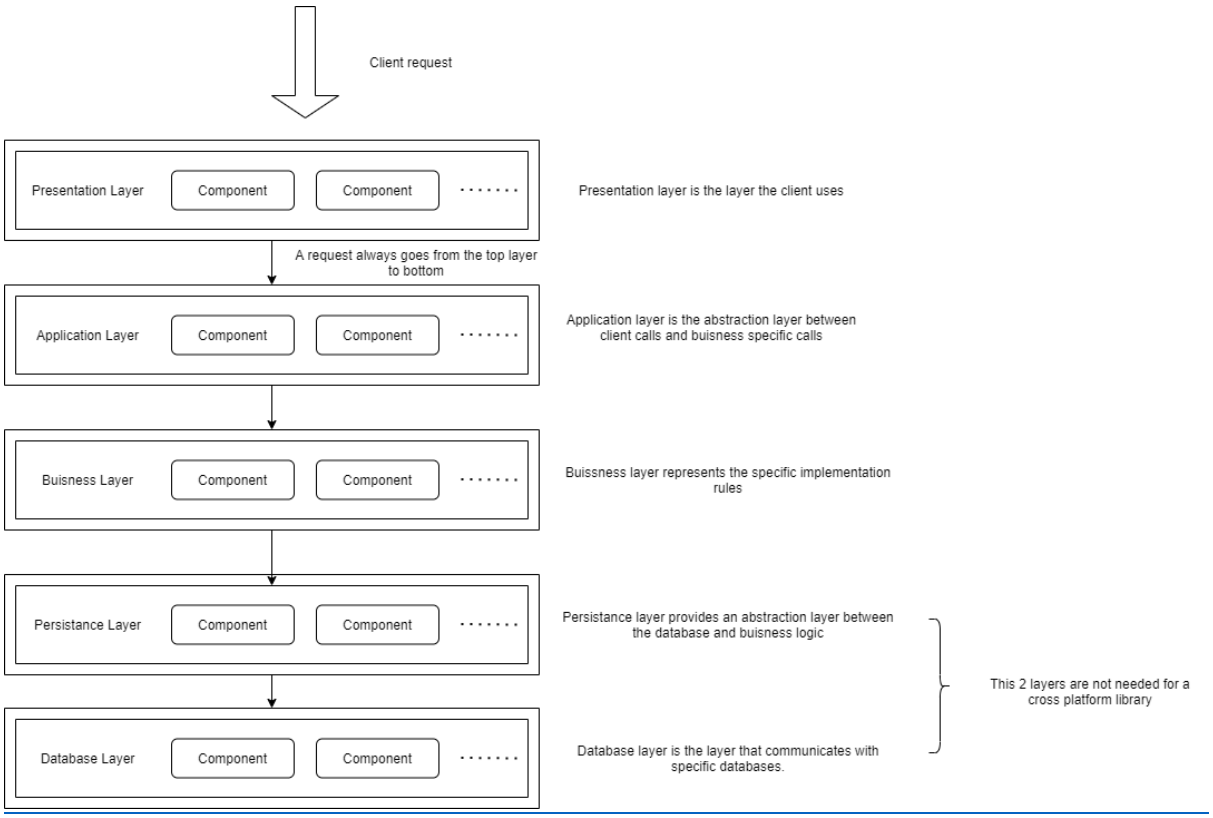


Figure 3. Diagram showcasing the layered architecture design.

The Clean Architecture

Another architecture that is very similar to the layered architecture is the "Clean Architecture." As shown in the picture below the clean architecture design is also providing three main layers: the "Business/Enterprise layer", "Application layer" and "Presentation/Interface layer".

In the book "Clean Architecture" by Robert C. Martin "The further you go in the circle, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies. The overriding rule that makes this architecture work is The Dependency Rule: "Source code dependencies must point only inward, toward higher-level policies." This means that classes represented in the inner circle cannot know anything about the outer rings of the circle.

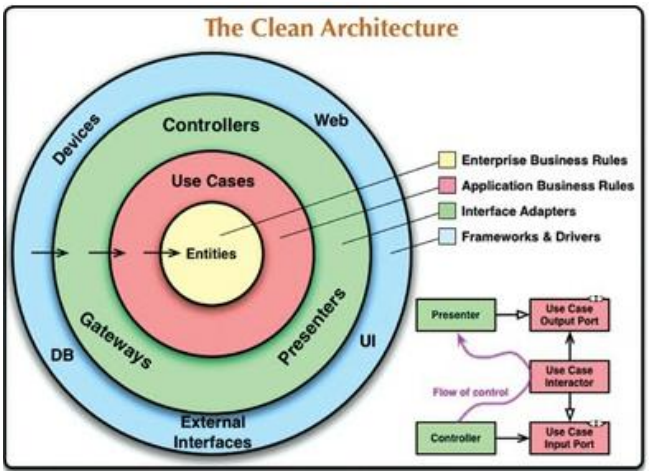


Figure 4. Clean Architecture diagram. Taken from the book "Clean Architecture" by Robert C. Martin, 2018.

Event-Based Architecture

The Event-Based architecture uses a different approach to handle the decoupling of the classes and behaviours. Instead of using an interface approach, this architecture uses event objects that are thrown by the user when a specific behaviour happens. All the interested systems can then listen to these events and execute their functionality. This makes sure systems are decoupled from each other and they do not have to know who is listening to the event or what they are doing when the event is raised. From the user's perspective, all they need to do is throw an event and let the internal systems handle the event. This way the user does not have to change or extend the internal systems.

In the book "Software Architecture Patterns" by Mark Richards there are shown two main topologies for this architecture.
The first one is the "Mediator Topology", which consists of 5 main concepts (Event, Event Queue, Event Mediator, Event Channel and Event Processor). The user uses an Event Queue to raise an Event, Event Mediator decides which channel should be notified for the event. The event Processor executes the behaviour required for an event by listening to the specified event channel. The unique concept of this topology is the Event Mediator which serves as a layer of orchestration for the architecture. The Mediator decides which channels should be notified by the event, there can be another type of event used by the Mediator that notifies the channels when a user event is thrown. An example of this architecture is shown in Figure 5 below.

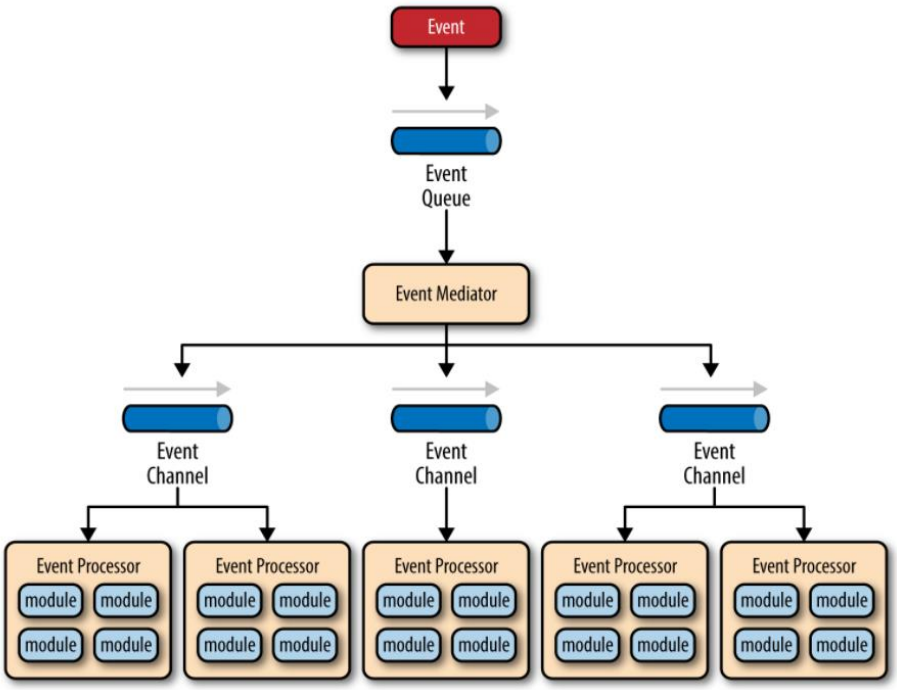


Figure 5. Event Queue architecture (Mediator Topology). Taken from the book "Software Architecture Patterns" by Mark Richards, 2015.

The second topology shown for this architecture is the "Broker Topology". The only difference between the Broker Topology and the Mediator one is that there is no Event Mediator concept, thus the event does not go through another processing layer, instead, it goes directly to the Event Channel that is listening for that event. An example of this architecture is shown in Figure 6 below.

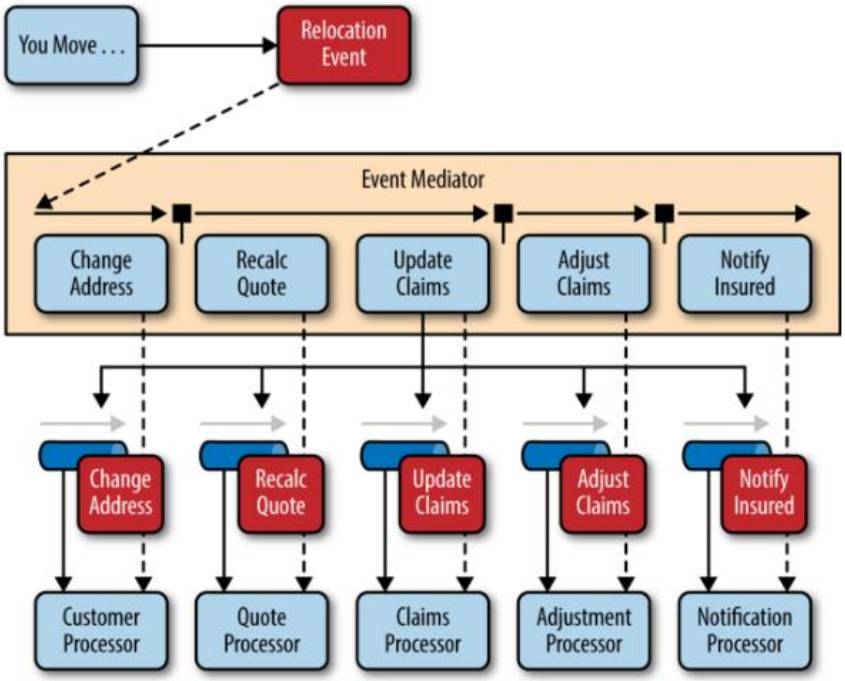


Figure 6. Event Queue architecture (Broker Topology). Taken from the book "Software Architecture Patterns" by Mark Richards, 2015.

Another architecture researched was the ""Microkernel Architecture". This architecture is meant for large scale codebases. More information about this architecture can be found on Appendix A.

As for theory material for cross-platform SDK management websites and existing cross-platform libraries will be analysed.

"Designing Cross-Platform Mobile Applications with Xamarin" (Heywood, 2018) is a blog post that shows concrete examples on how to handle cross-platform development for IOS and Android devices using Xamarin. Although mobile development is different from console development, the architecture and code structure are very similar.

Another reason why there is no console theory gathered is that all consoles are closed environments meaning there are no public online documentation or examples.
As for concrete examples, Unity AR Foundation was analysed. Unity AR Foundation (AR Foundation, 2020) is a cross-platform AR plugin created by Unity Technologies. AR Foundation was chosen because it tries to handle two different native APIs (Apple's AR Kit and Google's AR core).
The architecture uses a "Providers" (AR Foundation Manual, "Subsystems" section, 2020) to wrap native functionality and then expose them to the higher-level layers. In case one API does not support e certain feature (e.g., object occlusion is supported by AR Kit but not by AR core) the library will expose the functions however it will not do anything if called by the platform which does not support the functionality. These architectural choices provided an example of how to wrap/handle multiple different APIs in one library architecture.

6.2.2 Design

The proposed architecture design is a mixture of Layered Architecture and EventBased Architecture.

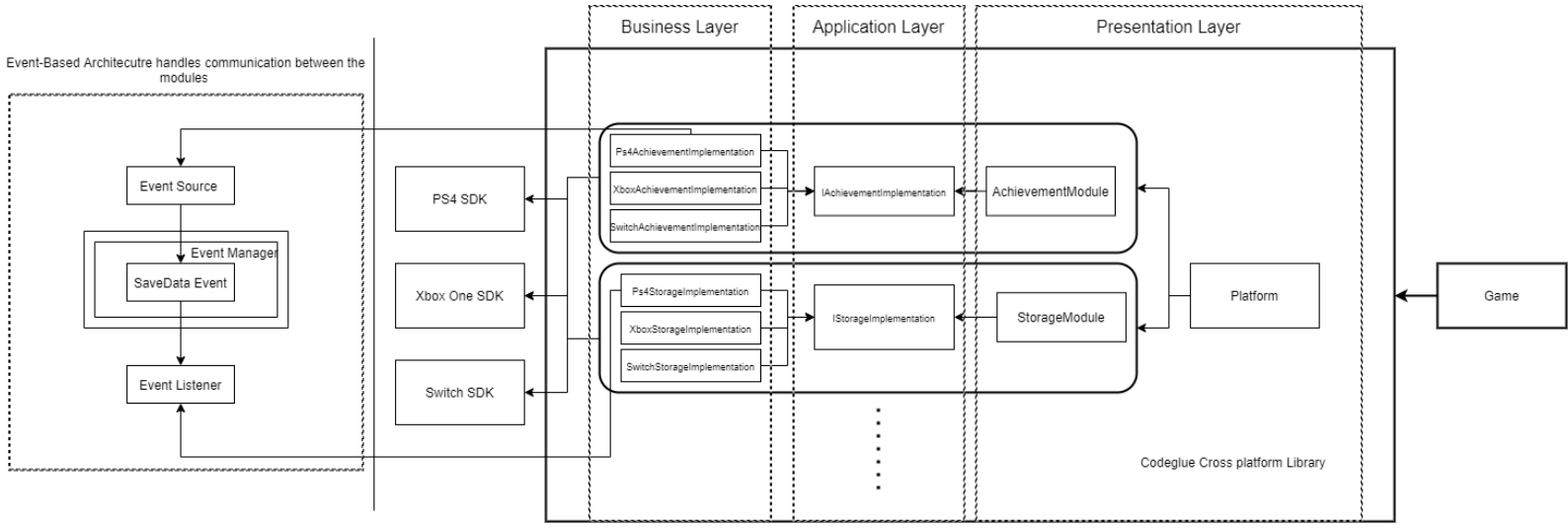


Figure 7. Diagram showing the library's modules in a layered architecture design.

The "Platform" class provides an entry point to the presentation layer. It acts as the Factory Pattern (Gamma et al., 1994) meaning it creates the platform modules and allows access to them. It will also initialize each module and tell the game layer when a module is ready.
"The Modules" define the core of the presentation layer while their "Platform Implementation" Interfaces act as an application layer by just defining the higher level between platform and implementations. Each module automatically creates and stores the correct type of the "Platform Implementation" interface based on the target platform.
Last each concrete platform implementation class represents the business layer. The platform-specific code and SDK usage goes on this layer. This layer is also completely hidden from the presentation layer.
The "Platform Implementation" classes are similar to "Providers" of "AR Foundation" mentioned in the previous "Theory" section. They wrap platform specific implementation and expose it to higher level classes.

This module based layered architecture is mentioned in the web article "Creating truly modular code with no dependencies" (Gadziniwski, 2018).

In addition to the Layered Architecture, the Event-Based Architecture is responsible to handle the communication between the modules, thus decoupling the modules from each other and going close to the desired **main sequence** line mentioned in the "Instability and Abstraction Analysis" section.

6.2.3 Implementation

In the final architecture implementation, both the Layered Architecture and the Event-Based architecture are implemented following the design shown in the previous section.
The new architecture handles the following modules:

- Initialization Module
  handles the creation and initialization of the core SDK classes for the specified platforms. The initialization module and the user module are the core of the platform, and they are also the only two modules needed to get an empty project to run;

- User Module
  handles and updates the main user (the user that started the application) and notifies other modules (using the event architecture) when this user has been updated.
  It can also request to sign in a user as well as set the presence of a user;

- Achievement Module
  handles unlocking, incrementing, and getting a list of achievements;

- Authentication Module
  allows checking if a user can go online or has purchased an online subscription or if the user has certain privileges or not;

- Friend Module
  This module only provides a way to get the list of friends for a specific user;

- Storage Module
  The storage module allows saving, loading and file checking for the main user. Unlike the other modules, its function does not request a user to be specified however, it automatically maintains the main user and adjusts accordingly when that user changes;

- Device Module:
  allows the users to check if a controller is connected or not;

- Text Module
  handles platform's virtual keyboard input.

The modules communicate with each other using the Event Manager, which handles the raising and subscribing of events. An example of using the event architecture is shown in Appendix C.

The platform scripts handle the creation and initialization of the modules. Some of the modules are initialized asynchronously to follow the guidelines of the platform SDKs. This means that the platform script needs to keep track of when a module is ready and give information about the status of the module to the game layer. A more detailed look into the code of the initialization function can be found in Appendix D.

Another bigger change done to the architecture was the addition of the "Platform User" class. This class wraps the SDK specific user data and serves as a handle from the gameplay side. In

the previous architecture, users were not passed into platform functions. Instead, the platform assumed they were for the main user only. Figure 8 below shows an example of a function call from the previous architecture:

```
achievementManager.Unlock(Achievement.SomeAchievement);
```

Figure 8. Achievement unlock functionality from the previous codebase.

To unlock an achievement previously only the achievement had to be passed and the manager would unlock that achievement for the main user. This is how the previous architecture worked. To make the new architecture adaptable to the addition of support for more users the "Platform User" class was defined and is requested to be passed on the function. Figure 9 below shows how the new Unlock function looks like:

```
achievementModule.Unlock(userModule.MainUser, achieve: Achievement.AnotherAchievement);
```

Figure 9. Achievement unlock functionality from the new architecture.

This change also made sure the other modules do not have any dependencies with the whole user module but rather just the user data that is required.

In the new architecture, it is required to pass a user. The user module provides a field to get the main user if needed. This main user is automatically updated by the user module for each platform.

6.2.4 Evaluation

The architecture goals were the following:

- Platform separation:

  In the new architecture, the implementation is completely hidden from the game/application layer. This is because the "Layered Architecture" design introduces a new layer in between the game layer and the implementation layer thus hiding these two from each other completely. The game layer can access a specific module and then the module chooses which implementation to use based on the platform the application/game is running on.

- Ease of Extension:

  In order to test the extension aspect of the architecture at first only two modules were implemented (Initialization and User module). These two modules are also the only modules needed to get a build running on a platform. After the implementation of the above-mentioned modules, five new modules were implemented. This process provided an idea of how well the architecture supports the addition of new modules. The steps for adding a new module are as follows:

  - Creating the module class and define the functions the game layer can access.
  - Creating the platform implementation interface for the module. This is what is required to be supported for each platform.
  - Creating the correct implementation in the module for each platform. (e.g., if we are on ps4 create "PS4Implementation" etc.)
  - The last step is creating the module in the platform script so that the game can access it.

  The addition happens in complete isolation from other modules and if the information is required from other modules, e.g., the user data is needed, then the new implementation can subscribe to the "user changed" events.
  Another aspect of the extension is the ability to handle multiple platforms, for this aspect the PS4 and Xbox one platforms were supported. At first, the PS4 modules and implementations were created and then the Xbox one implementations were added. If a new platform needs to be added to the library, the only things that need to be created are the new implementation classes for that platform.

  The modular architecture design allows easy extension of new modules or adding support for a new platform.

- Module decoupling

  In order to handle the module decoupling, the Event-Based architecture is implemented. To evaluate this architecture the instability and abstraction analysis is performed to find out how the new architecture compares to the old one.

Figure 10 below shows the analysis performed after the new architecture was implemented. As it is shown therein, the distance to the main sequence has decreased from 0,358(Old Architecture) to 0,273(New architecture).

Column1	Achievement Module	Storage Module	User Module	Friend Module	Initialization Module
Fin	0	0	2	0	0
Fout	1	1	1	2	2
I = Fout / (Fin + Fout)	1	1	0,333333333	1	1

Column1	Achievement Module	Storage Module	User Module	Friend Module	Initialization Module
NA	1	2	2	2	1
NC	5	7	7	8	4
A = NA / NC	0,2	0,285714286	0,285714286	0,25	0,25

Column1	Achievement Module	Storage Module	User Module	Friend Module	Initialization Module	Average
D = |A + I -1|	0,2	0,285714286	0,380952381	0,25	0,25	0,273333333

Figure 10. Instability and abstraction analysis performed for the new architecture.

The main reason the new architecture is closer to the main sequence line is the introduction of Event-based architecture which removed most of the dependencies between the modules. Some outgoing dependencies can still be observed from the user module; however, these are not dependencies to the user module itself rather the new “Platform User” class which wraps around the SDK user data.

To go even close to the main sequence one thing that could be done is to introduce more abstract classes. However, this would make the overall codebase more complex and harder to extend since the addition of new platforms would require implementing all the new abstract classes.

6.3 How will the library be packaged and deployed throughout different projects?

6.3.1 Theory

The company's library needs to support Unity projects and thus it needs to be written in C#. To package the library independently two main ways were analysed and compared:

C# native packaging (Assemblies/DLL) & NuGet packaging

Chapter 7 "Understanding and packaging .Net Types" of the book "C# 8.0 and Net Core 3.0" by Mark J. Price, introduces C# assemblies. The assemblies can be built as an executable or as a DLL. If built as a DLL they cannot be executed however they can be included in other projects.
Assemblies in .NET simply define the scope of the binaries — i.e., DLLs. One assembly is one DLL and is defined by the existence of an assembly definition file ".asmdef". The assembly definition is used by the compiler to assemble the DLL but also to understand and follow the linkages between assemblies. Thus, without the correct ".asmdef" file the package scripts will not be accessible (Harwood, 2021).
Assemblies can reference other assemblies, but they cannot have circular dependencies (two assemblies both referencing each other).
The DLLs are dynamically linked at runtime, and they are loaded when required by the application.
In native C# multiple DLLs can be packaged and deployed as NuGet packages. To create one, it is required to fill in the Package info window. Figure 11 below shows an example of the package window.

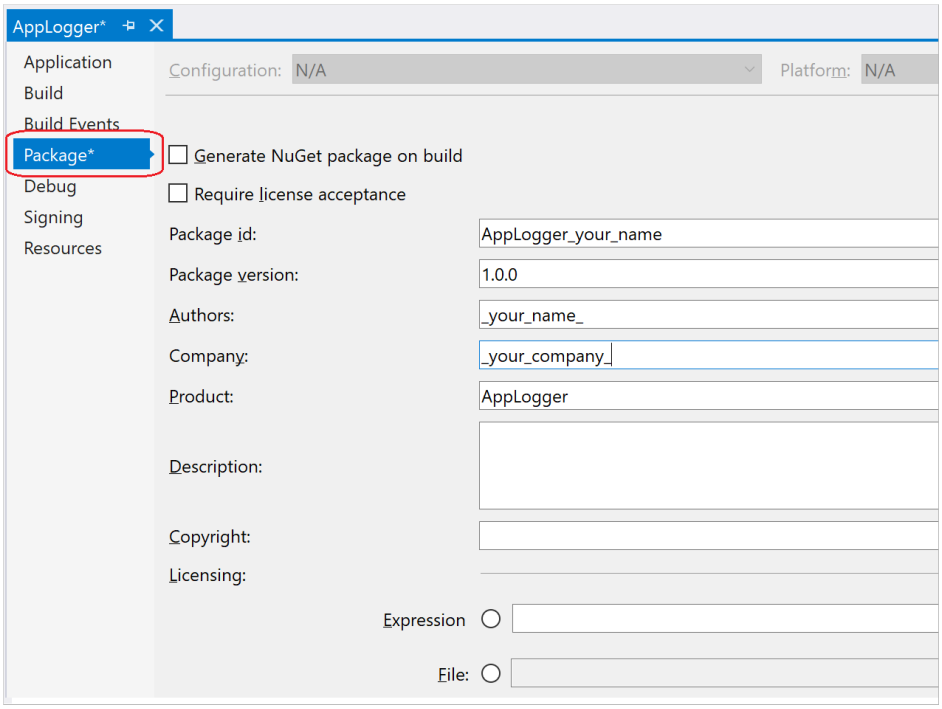


Figure 11. Visual studio packaging settings.

After the info is filled in, the "Pack" command can be run to generate the ". nupkg" file as shown in Figure 12. This process can also be set to execute when the project is built.

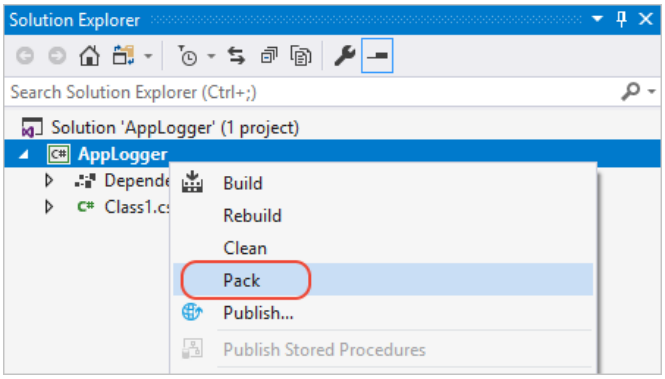


Figure 12. Pack command inside Visual Studio.

After the ".nupkg" file is created it can be shared and linked with other projects. DLLs can be included inside Unity by dragging them in the assets window; however, Unity cannot include NuGet Packages by default.
In order to include a NuGet package in Unity "NuGetForUnity" created by Patrick McCarthy (2008) has to be used. This free asset provides all the NuGet functionality within Unity Editor.

Strengths	Weakness	Opportunities	Threats
• Native C# library, making it independent from unity. • Completely hidden from the Unity project. • Easy to import (Just include the DLL file).	• Needs to be built again if changes are made. • Cannot access Unity features by default.	• Can be used outside Unity Engine.	• If in the future unity features are required to be used the whole architecture needs to change.

Table 1. SWOT Analysis for C# native packaging method.

Unity Package

The Unity Package Manager is the official package management system for Unity which performs the following:

- Allows Unity to distribute new features and update existing features quickly and easily.
- Provides a platform for users to discover and share reusable components.
- Promotes Unity as an extendable and open platform.
- The package's folder layout must follow Unity's folder structure as shown below:

In order to be deployed and included in other projects, the package must follow Unity's folder layout convention as shown in Figure 13.

Package layout

This is the package layout convention followed by official Unity packages:

```
<root>
  ├── package.json
  ├── README.md
  ├── CHANGELOG.md
  ├── LICENSE.md
  ├── Editor
  │   ├── Unity.[YourPackageName].Editor.asmdef
  │   └── EditorExample.cs
  ├── Runtime
  │   ├── Unity.[YourPackageName].asmdef
  │   └── RuntimeExample.cs
  ├── Tests
  │   ├── Editor
  │   │   ├── Unity.[YourPackageName].Editor.Tests.asmdef
  │   │   └── EditorExampleTest.cs
  │   └── Runtime
  │       ├── Unity.[YourPackageName].Tests.asmdef
  │       └── RuntimeExampleTest.cs
  └── Documentation~
      └── [YourPackageName].md
```

Figure 13. Unity package required folder layout design. (Unity Technologies, n.d.)

In the root directory, a “package.json” file is required to be created and filled correctly. This file is similar to the NuGet package. Unity package manager uses this file to import the entire package and its dependencies. An example of this file is shown in Figure 14.

Package manifest example

```
{
  "name": "com.unity.example",
  "version": "1.2.3",
  "displayName": "Package Example",
  "description": "This is an example package",
  "unity": "2019.1",
  "unityRelease": "0b5",
  "dependencies": {
    "com.unity.some-package": "1.0.0",
    "com.unity.other-package": "2.0.0"
 },
 "keywords": [
    "keyword1",
    "keyword2",
    "keyword3"
  ],
  "author": {
    "name": "Unity",
    "email": "unity@example.com",
    "url": "https://www.unity3d.com"
  }
}
```

Figure 14. Example manifest file for a unity package (Unity Technologies, n.d.)

Unity packages work similarly to C# native NuGet Packages by using “Assembly definition (.asmdef)” files to group scripts in separate modules. These .asmdef files can be created in the asset window. Figure 15 below shows an example of an assembly definition file inside Unity.

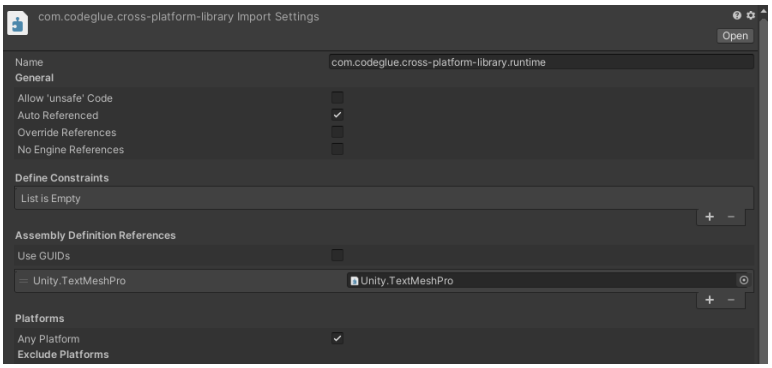


Figure 15. Assembly definition file for the cross-platform library.

Other definition files can be linked by dragging them in the “Assembly Definition References” or if the “Use GUIDs” option is ticked by specifying the GUID if the definition file needed to be linked.

Each Unity package contains three special folders recognized by the package manager. There are Runtime, Editor and Test folders. The Runtime folder contains scripts that are going to be included when the application is running. Editor folder scripts are included only when running the application inside the Unity Editor and they are not included when the project is built. The test folder contains all the automated test cases scripts.
Each of these folders (Runtime, Editor and Test folders) is required to contain an Assembly definition file inside them.

Strengths	Weakness	Opportunities	Threats
• Integrated with unity. • Can access unity’s functionality. • The code source is within the project. • Can be imported via GitHub or a .zip file. • Uses unity’s assembly’s definition files which can be edited to completely hide certain scripts.	• Not as decoupled as a DLL • Cannot be used outside a Unity project.	• Can use unity’s editor features that can make the importing process easier for the user.	• If the project needs to be used outside Unity Engine, it will not work.

Table 2. SWOT Analysis for Unity’s packaging method.

6.3.2 Design

After analysing both of the packaging methods, the package method that fitted the library the most was the Unity packaging method. This is mainly due to the fact that the company wants the library to be used within Unity and to be able to use Unity specific functionality like Coroutines/Debugging etc. Another reason for choosing this packaging method is that it works by default with unity in contrast NuGet packages require extra asset packs made by third-party companies in order to fully work inside unity.
An important design aspect when creating a package is the File layout, not only it has to follow some of the Unity's requirement mentioned above but it also needs intuitive for the users of the library.

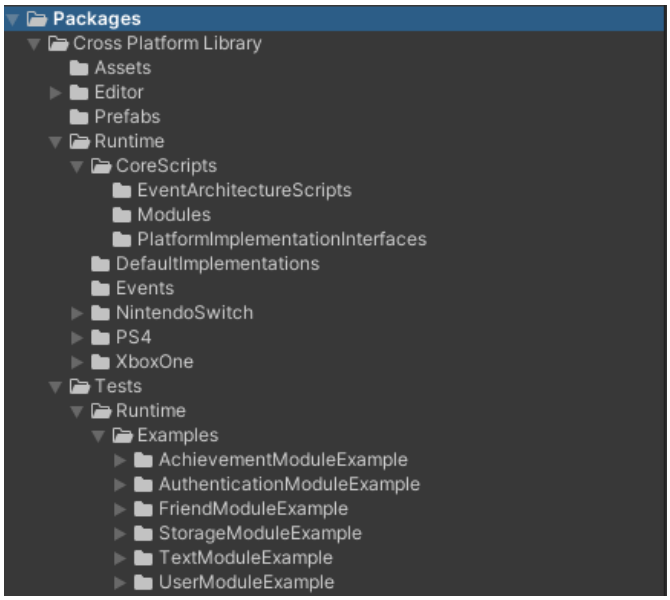


Figure 16. Library's folder structure

Once imported into a project the package can be found under the "Packages" folder which Unity automatically generates. In the root of the package the "package.json" file is found and all the three main folder (Editor, Runtime, Tests). Each of these folders contains different Assembly definition files.
The Editor folder contains the setting window script and some platform-specific editor scripts. (Some of the SDKs have custom window editor functionality as well).
The Runtime Folder contains "Core Scripts" which include the base scripts for the architecture such as module/interface definitions and event queue architecture scripts.
Each Platform has a unique folder where all the DLL/SDKs and specific implementation scripts for that platform are stored.

The Tests folder contains Example tests scenes for each module. These scenes can be opened to test the functionality of that specific module. Testing is explained more in detail in the next sub-question.

Another feature designed for the package is having a settings window where settings for each platform can be assigned. If the user imports the library the window should pop up and if the

user fills in all the required settings the library should be able to run on the supported platform without doing extra coding.

6.3.3 Implementation

The first step in implementing the package was following Unity's package guidelines mentioned above. First, the package.json file was created and placed in the root directory of the package. The JSON file is shown in the 17.

```
{
  "name": "com.codeglue.cross-platform-library",
  "version": "0.0.1",
  "displayName": "Cross Platform Library",
  "description": "Library that maintains console api.",
  "unity": "2019.4",
  "unityRelease": "21f1",
  "references": [
    "Unity.TextMeshPro"
  ],
  "keywords": [
    "platform",
    "library",
    "console"
  ],
  "author": {
    "name": "Codeglue",
    "email": "rens@codeglue.com"
  },
  "type": "library",
  "dependencies": {
    "com.unity.textmeshpro": "2.1.1"
  },
  "hideInEditor": false
}
```

Figure 17.Library's package.json file.

The next step was the creation of the folder structure including the assemblies for the main folders(Editor, Runtime, Test).
The editor assembly definition only references the runtime assembly to get access to the core platform scripts. The settings window then just saves a setting struct to disk and packages it up with the application when built. The runtime assembly definition references only "TextMeshPro" assembly. This is used for a debug popup when certain variables needed to be inspected easily and displayed at runtime.
As for the Tests assembly definition, it only references the runtime definition files to get access to the library's core scripts.

When the library is imported, a custom editor window becomes available for the user. This window contains all the settings required for the platform library. The settings window will save the filled fields in a settings file and bundle that with the build application. Since the platform assembly does not know about the editor assembly all the platform does is loading the settings file at runtime and providing access to the settings fields to the other modules.

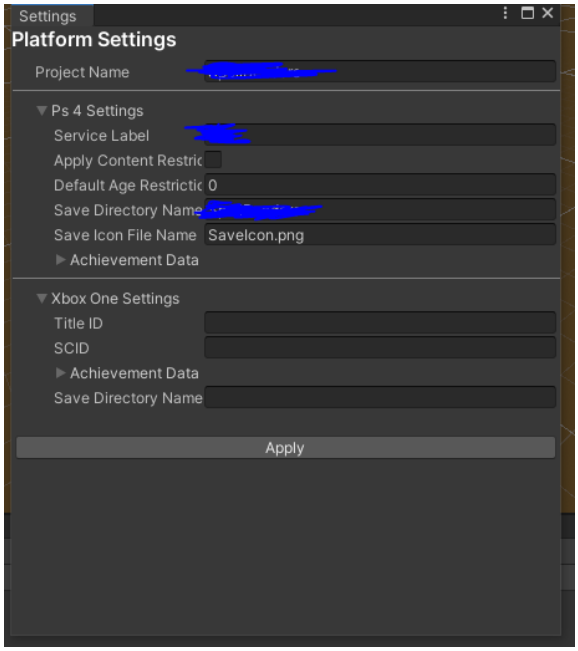


Figure 18.The custom window for platform-specific settings. Hitting apply saves the settings on disk.

6.3.4 Evaluation

In order to evaluate the chosen packaging solution, the library was deployed in one of the existing projects within the company, called "Spellbenders". Spellbenders is a team-based online action game that is currently still in development and is not ported for consoles yet, which made the latter a good test caser for how the library handles the deployment in a project that needs to be ported.

To get a new project to build on a specific console the following steps need to be taken:

- Importing the library unity package. This can be done either via locating the folder of the package in the local device or via a GIT repo (as long as the package manifest file is in the root folder of the repo).
- Filling the custom settings window for the required console.
- Adding the Platform Mono Behaviour in the initial scene of the project. (This behaviour does not get destroyed between scenes thus adding it to the first thing is enough).
- Adjusting specific console settings found in the "Player Settings" of the unity's "Build Window".

After the above-mentioned steps were taken Spellbenders was successfully built for the PS4 platform, and the only thing left was linking the game code with the library.
The library package was then shared within the company to get feedback from the programmers.
A questionnaire was created with questions regarding the architecture, packaging, and deployment of the library. The answers to this questionnaire can be found in Appendix B. As shown in the first questionnaire by Rens, the project took 2 weeks to fully set up and run on PS4 and Xbox One.

6.4 How to test the addition of new platforms to the library?

6.4.1 Theory

Software testing is a very important part of the architecture. It not only prevents bugs and ensures the current codebase is running as intended, but it also makes sure that new additions to the architecture are verified properly.

The C# language has unit testing already build in the language. A unit test is an automated test that verifies small pieces of code (also known as units); it does it quickly and in an isolated manner. (V. Khorikov, 2020).

Creating a unit test method in C# is trivial, the below conditions must be fulfilled:

- It's decorated with the [TestMethod] attribute. (There are more optional attributes for more specific occasions, for more information refer to Microsoft documentation.);
- It returns void;
- It cannot have parameters.

In order to use unit testing inside the Unity engine, "Unity Test Framework"(UTF) must be used. UTF uses a Unity integration of the "NUnit" library, which is an open-source unit testing library for ".Net" languages. (Unity Documentation,2020).
The unit test can be run via the "Test runner" in the editor while being in either "Play" or "Edit" mode. However, they cannot be run once the application is built.
Similar to the native C# unit testing the only thing needed to show and run a test function is to use an attribute. In UTF this attribute is "[UnityTest]". In Unity, the tests also need to be in a separate assembly from the rest of the project. Once the assembly is linked and a test script is created it will appear on the Test runner as shown in Figure 19.

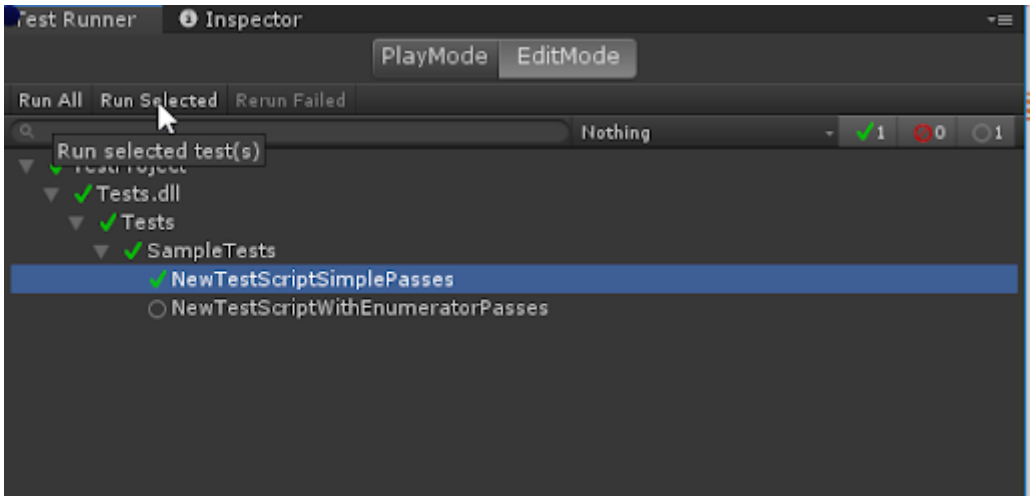


Figure 19. Test cases running in editor mode inside Unity.

If a test passes the assertion, it will be marked with a green check and if it fails it will be marked with a red stop sign. The Gray circle indicates all the test that have not been run.

The library's platform SDKs cannot run in the editor thus a solution where the tests could be performed at runtime for the built application needed to be found.

6.4.2 Design

As mentioned in the previous section C# unit test do not run when the application is built. Since software testing is an important aspect of a codebase, custom tests were designed for the library. It is important however that the created test follow the principles of a unit test, which were:

- Verify small pieces of code (also known as units).
- Perform the tests quickly.
- Perform the test in an isolated manner.

The units that we want to test are the modules of the platform. For each module, a test scene needs to be created where all the functionality of that module is tested in isolation. Since the tests are custom made for the platform, Unity UI features can be used to display information as well as results for the test.

Another design requirement set for the test is that that can be used at runtime for any unity project which contains the library. This means as long as the platform is in a scene any module test can be dragged into a scene and run quickly without having to code extra behaviour.

The test scripts should be in their own assembly definition separate from the platform and game classes. The only dependency the tests need is the platform assembly, as the latter provides the test codebase with the platform-specific API.

6.4.3 Implementation

As shown in the packaging and deployment sub-question, the tests are stored inside a different folder and are part of a different assembly definition. The tests assembly has access to the platform assembly however the platform or the game classes do not have access to the test classes.

Before implementing the test scenes for each module, the platform boot scene was implemented. This scene serves as the hub to all the available test scenes and also waits till the platform fully initializes before allowing the users to load one of the test scenes.

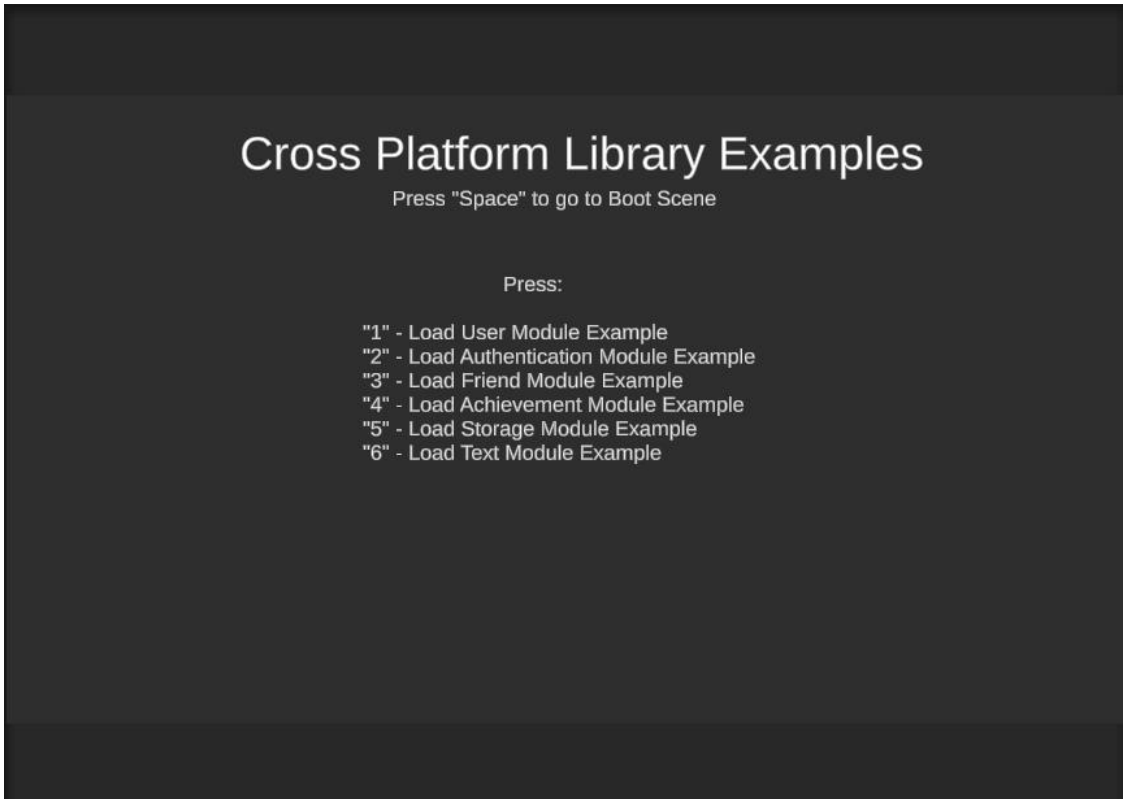


Figure 20. Boot example scene which initializes the platform and can load all the other example scenes.

In this section, only the achievement module example scene is shown. The rest of the examples are shown and explained in Appendix E.

The picture below shows the achievement module example scene. In this scene, all the supported functionalities of the achievement module are tested. The first field displays the current user's username. The programmer can select an achievement from the dropdown menu and increment or unlock it for the user. On the other hand, "Achievement Data" shows all the registered achievements on the platform's backend as well as their current state.

Figure 21. Achievement module test scene.

All the UI elements and canvas is stored as a prefab. This prefab can then be dragged in any other scene if required to show information or test achievement module's behaviours.
The prefabs contain all the behaviours required to update the UI and it will throw an error if a platform script is not in the scene.

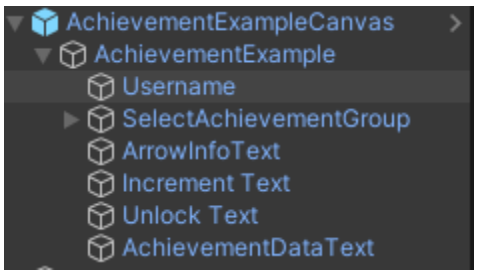


Figure 22. Prefab containing the test logic and UI for the achievement module.

The user can press the "Space" key in any scene to go back to the "Boot" scene. All the test scenes are created in such a way where the only input needed is the keyboard input. This is due to the fact that some platforms only simulate virtual keyboards and not the mouse.

6.4.4 Evaluation

As mentioned in the previous sections, software testing is an important aspect when developing a complex architecture. For this reason, even though unit test were not supported for the library, it was decided that the custom tests should still be implemented following the principles of unit testing.

To evaluate the custom testing method, a testing scene was implemented for each of the supporting modules and tested for both PS4 and Xbox One. Afterwards, feedback from the company's tech lead (Rens) was gathered. This can be seen in his answer to the question "What are some of the improvements the new architecture introduces?" and in the "Overall thoughts" provided under Appendix B, Questionnaire 1.
Rens highlights testing as one of the key improvements to the architecture.

7 Conclusion

In conclusion for the main question, "How to create an independent cross-platform library for Codeglue that handles console platform-specific SDKs, in order that the company can reuse and extend the codebase throughout different projects?", a Unity library was created to handle platform-specific SDKs.

The library's architecture is a mixed architecture between the modular version of Layered Architecture and the Event-Based Architecture that allow the extension of the library by adding new modules with minimal dependencies.

The library is packaged as a unity package that can be deployed throughout multiple unity projects while being an independent codebase.
Currently, the library supports PS4 and Xbox one platform; however other platforms can be added to the library as long as they support the library's implementation interfaces.

For software testing, custom example scenes are implemented which test the functionality of each module in isolation. The test prefabs can be deployed in different scenes and still execute the designed test cases.

8 Discussion

Creating and designing the right architecture is not a trivial task. There is not one architecture to solve all of the problems, meaning certain architectures are good at serving some problems and each of them has its own drawbacks. Another issue is the fact that it is hard to measure how good the architecture is, without having some unit of measurement.

The main focus of this report was analysing different architectures, comparing them with each other and finding a way to measure how the architecture is performing. These measurements were performed for the old codebase and the new proposed architecture.

All the methods were analysed and researched, and concrete evaluation was done to determine if the research was successful or not. Based on this fact, the research done for the library is valid.

9 References

1. Codeglue Website, About us section. Available at: https://codeglue.com/about/. [Accessed 16 March 2021].

2. Brown, W., Malveau, R., McCormick III, H. and Mowbray, T., 1998. Antipatterns. John Wiley & Sons Inc.

3. Martin, R., 2018. Clean architecture. Prentice Hall.

4. Richards, M., 2015. Software architecture patterns. O'Reilly Media, Inc.

5. Gadziniwski, K., 2018. Creating Truly Modular Code with No Dependencies. [online] Total Engineering Blog. Available at: https://www.toptal.com/software/creating-modular-code-with-no-dependencies [Accessed 16 March 2021].

6. Heywood, J., 2018. Designing Cross-Platform Mobile Applications with Xamarin. [online] Capgemini Engineering. Available at: https://capgemini.github.io/.net/designing-mobile-cross-platform-applications-with-xamarin/ [Accessed 16 March 2021].

7. Docs.unity3d. 2020. About AR Foundation | AR Foundation. Available at: https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.1/manual/index.html [Accessed 16 March 2021].

8. Easymobile.sglibgames.com. 2017. Easy Mobile. [online] Available at: https://www.easymobile.sglibgames.com/ [Accessed 16 March 2021].

9. Gamma, E., Vlissides, J., Helm, R. and Johnson, R. (a.k.a "The Gang of Four"), 1994. *Design patterns*.

10. Unity Technologies. Unity - Manual: Creating custom packages. Retrieved 10 May 2021, from https://docs.unity3d.com/Manual/CustomPackages.html?utm_source=YouTube&utm_medium=social&utm_campaign=evangelism_global_generalpromo_2020-09-02_packman-package-manager-docs.

11. Harwood, P. (2021, 3 January). *Simple Guide to Unity Package*. Medium. https://medium.com/runic-software/simple-guide-to-unity-package-management-4aea43d1baf7.

12. Patrick McCarthy (2018). "NuGetForUnity" asset. GitHub page: https://github.com/GlitchEnzo/NuGetForUnity

10 Appendices

10.1 Appendix A

Microkernel Architecture

The microkernel architecture consists of 2 main concepts, The core system, and Plugins. The core system is only the thing needed for the architecture to work and set up and all the extra functionality goes into the plugin modules. The plugin modules are completely independent of each other. They should be treated as self-contained components or libraries. The core system is responsible for providing access to the plugins via some sort of plugin registry however, the architecture pattern does not define how the plugin registry is implemented or how the plugins interface with the core system, the only important thing is that a “contract” exists between the core system and plugins and that the plugins are completely independent of each other.

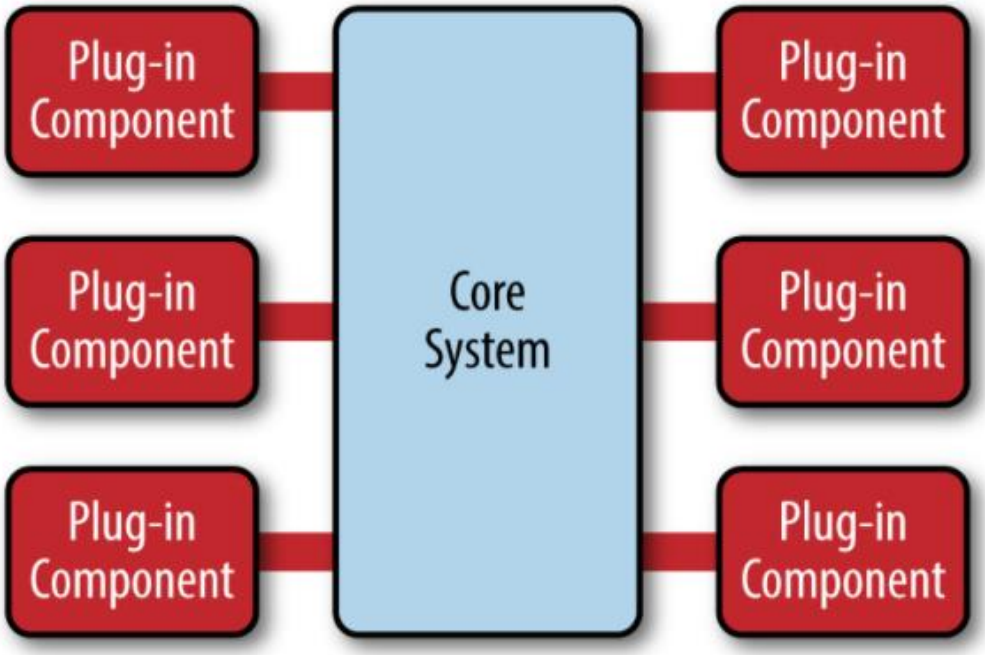


Figure 7. Microkernel architecture diagram. Taken from the book “Software Architecture Patterns” by Mark Richards 2015.

10.2 Appendix B

Questionnaire 1

The questionnaire below was filled by Rens Van Mierlo, Tech Lead at Codeglue and the student's coach. Rens followed the development of the library from the beginning and actively gave feedback and approval for the new architecture implementation.

Codeglue Library Questionnaire

* Required

Is the chosen architecture an improvement over the previous architecture?

There was no reusable architecture before. The the extent it was possible we would copy over classes we used in other projects and "re-create" our platform code and elaborate on it. Now we have a good architecture that provides a stable base for the platform API's.

What are some of the improvements that the new architecture introduces?

Separation of API and Game code. Many of the basic console interactions are taken care off. With the platform settings window we have a separation of code and project data, now we don't have to go into the code base to link the new project to the right backend. The Library also provides a testing application to test new API's without the noise of game code.

How well does the library support the addition of new modules and platforms?

Very well. This has been one of the main focus points for the library. All decisions were made to keep the dependencies among the modules low. This also allows us to remove modules that we do not need for a project. e.g. a network module for a single player game.

How well did the library handle the deployment on an existing project (SpellBenders)?

Very good. The game was deployed and running in 2 weeks time on multiple platforms. The main time consuming tasks were game related code that need to be adjusted / elaborated instead of the library code.

Will The Company consider using/extending the library?

Yes

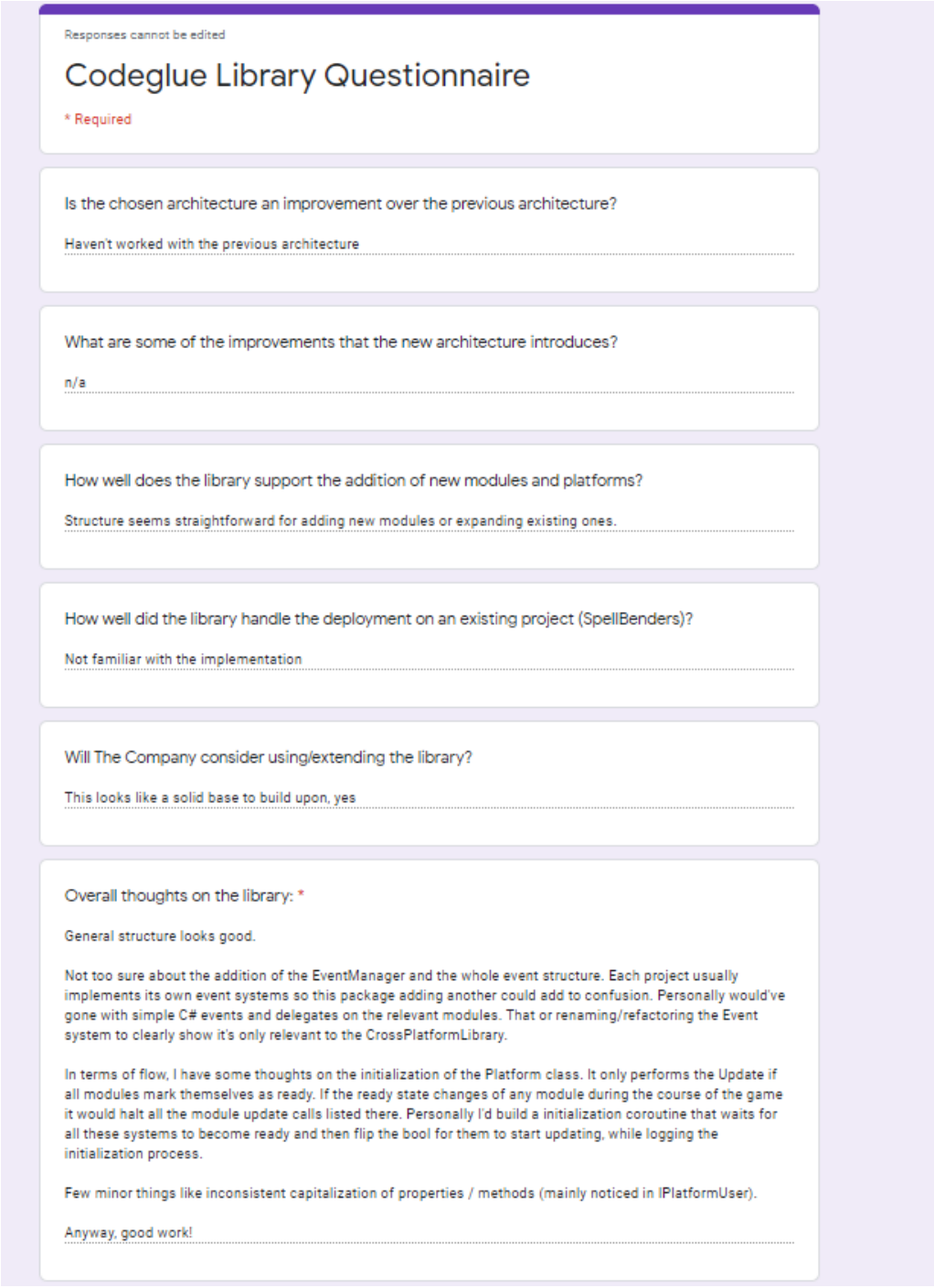
Overall thoughts on the library: *

Very happy with the result. The library provides a stable base that we can extent further on for future projects and API's. There is a good separation of API code, Game code, Project data and test applications. The addition of the test application is a big plus to the library and any future console updates that Codeglue will need to work with.

Submitted 6/10/21, 2:41 PM

Questionnaire 2

The questionnaire below was filled by Joeri van der Velden, a game programmer at Codeglue. Joeri had not used the previous codebase of the library thus he only gave feedback on the new implemented architecture.



Responses cannot be edited

Codeglue Library Questionnaire

* Required

Is the chosen architecture an improvement over the previous architecture?

Haven't worked with the previous architecture

What are some of the improvements that the new architecture introduces?

n/a

How well does the library support the addition of new modules and platforms?

Structure seems straightforward for adding new modules or expanding existing ones.

How well did the library handle the deployment on an existing project (SpellBenders)?

Not familiar with the implementation

Will The Company consider using/extending the library?

This looks like a solid base to build upon, yes

Overall thoughts on the library: *

General structure looks good.

Not too sure about the addition of the EventManager and the whole event structure. Each project usually implements its own event systems so this package adding another could add to confusion. Personally would've gone with simple C# events and delegates on the relevant modules. That or renaming/refactoring the Event system to clearly show it's only relevant to the CrossPlatformLibrary.

In terms of flow, I have some thoughts on the initialization of the Platform class. It only performs the Update if all modules mark themselves as ready. If the ready state changes of any module during the course of the game it would halt all the module update calls listed there. Personally I'd build a initialization coroutine that waits for all these systems to become ready and then flip the bool for them to start updating, while logging the initialization process.

Few minor things like inconsistent capitalization of properties / methods (mainly noticed in IPlatformUser).

Anyway, good work!

10.3 Appendix C

Event Architecture Example

The picture below showcases how the event manager is accessed and raises an event of type “UserStateChangedEvent”. This class inherits from the interface “IEvent”.

```
EventManager.Instance.RaiseEvent(e: new UserStateChangedEvent(user, StateChanged.SignInState,
    LogInState.LoggedIn, SignInState.SignedIn));
```

Then another class can use the event manager to subscribe to an event as shown below.

```
EventManager.Instance.Subscribe<UserStateChangedEvent>(HandleUserStateChanged);
```

Last, the function “HandleUserStateChanged” needs to be implemented. An example of an implementation is shown below.

```
private void HandleUserStateChanged(UserStateChangedEvent e)
{
    if (!e.platformUser.IsMainUser) return;

    IPlatformUser user = e.platformUser;

    if (user == null)
    {
        Debug.LogWarning(message: "UpdateAuthenticationInfo: Main User is Null returning...");
        return;
    }

    userUIBehaviour.UpdateUsernameText(user.GetUserName());

    if (e.stateChanged == StateChanged.SignInState)
    {
        userUIBehaviour.UpdateIsSignedInText(e.signInState);
    }
    else
    {
        userUIBehaviour.UpdateIsLoggedInText(e.logInState);
        if (e.logInState == LogInState.LoggedOut)
        {
            userUIBehaviour.UpdateControllerStatusText(currentlyConnected: false);
        }
    }
}
```

10.4 Appendix D

Platform's Initialize Function

Since the platform needs to inform the game of the status of a module as well as allow the game to continue running while initializing, a coroutine function is used. The first step is loading the settings file. This file is saved by the custom settings editor window, where platform settings are specified. Then all the platforms are registered and initialized. Afterwards, the initial users are established letting the other modules know about the initial state of the users. The next step is waiting for the modules to be ready and debug the status of a module. The last thing the function does is call the "OnFinishedCallback" parameter if one was passed to the function.

```
public IEnumerator Initialize(Action OnFinishedCallback = null)
{
    Debug.Log( message: "Initializing the platform...");

    //Loads the settings file
    LoadSettings();

    //Register The modules
    //Order matters, they will be initialized in the order they are registered!
    platformInitializationModule = RegisterPlatformModule<PlatformInitializationModule>();
    userModule = RegisterPlatformModule<UserModule>();
    storageModule =  RegisterPlatformModule<StorageModule>();
    achievementModule = RegisterPlatformModule<AchievementModule>();
    authenticationModule =  RegisterPlatformModule<AuthenticationModule>();
    friendModule =  RegisterPlatformModule<FriendModule>();
    deviceModule = RegisterPlatformModule<DeviceModule>();
    textModule = RegisterPlatformModule<TextModule>();

    //create and initialize the implementation for each module
    //user and platformInit are considered the core modules.
    CreateAllRegisteredImplementation();

    //Raises user events to inform the other modules for the state of the initial users
    userModule.EstablishInitialUsers();

    //Give some info on what the platform is loading/still waiting for...
    foreach (IModule module in moduleRegistry.Values)
    {
        Debug.Log( message: "Waiting for "+ module.ToString() + " to initialize...");
        yield return new WaitUntil(()=>module.isReady);
        Debug.Log( message: module.ToString() + " is Ready!");
    }

    Debug.Log( message: "ALL Modules Ready! Platform initialization complete!");

    //platform is ready, now call the "OnFinishedCallback" (if one is specified)
    OnFinishedCallback?.Invoke();

    yield return null;
}
```

10.5 Appendix E

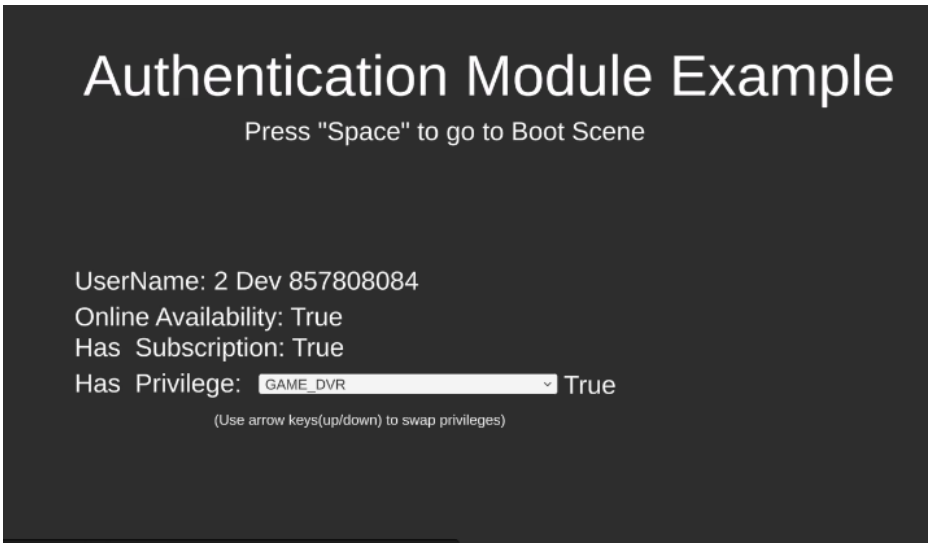
Test Scenes

- User Module

The picture below shows the User Module Test scene. It displays the information about the main user, and it will automatically update that information if the main user changes.

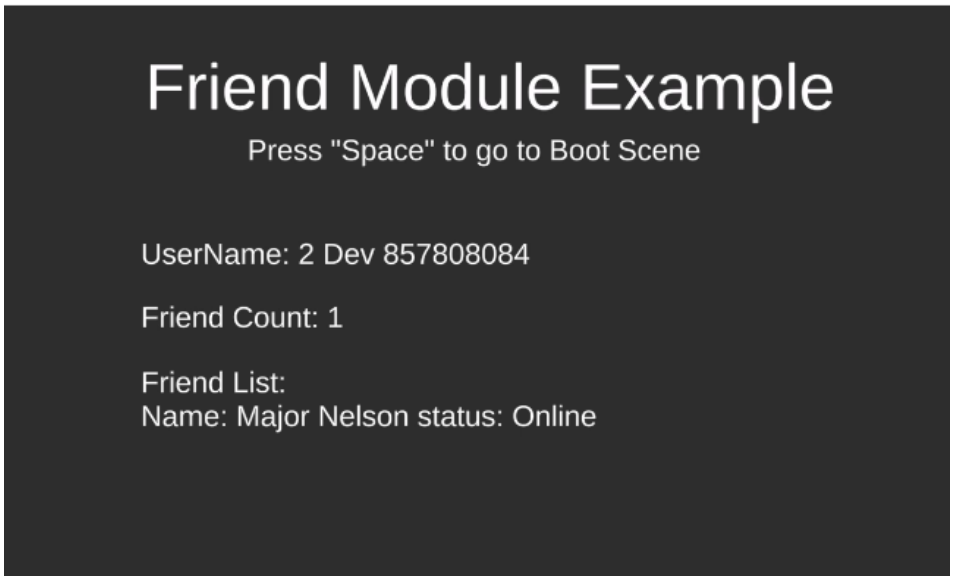
- Authentication Module

  The authentication Scene shows if a user can go online or has an online subscription for the platform. The programmer can also check if the user has certain privileges by using the dropdown UI.



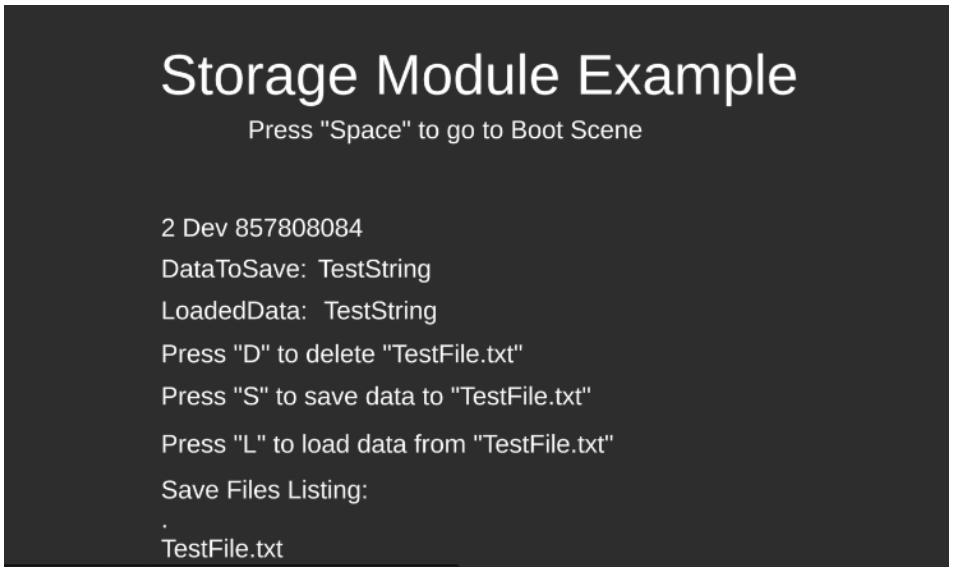
- Friend Module

  The friend example scene displays the friend count and the friend list for the main user.



- Storage Module

  The storage module scene showcases the struct that is going to be saved and loaded. This file can also be deleted. All the files saved on the cloud storage of the user are displayed in the “Save Files Listing”.



- Text Module

  The text module allows the programmer to open the virtual keyboard in case the platform supports one. After typing on the virtual keyboard and pressing the apply key the string is displayed in the “Keyboard Output” text.