Ted’s Entertainment

Ted's Quest
Software Architecture Document

Version <1.1>

Revision History

Date	Version	Description	Author
<29/dez/19>	<1.0>	<initial description>	<Michaela Fleig>
<27/jun/20>	<1.1>	<update information>	<Michaela Fleig>

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

This document applies to the development of the game "Ted's Quest" by Ted's Entertainment. Affected by the SRS and the use cases.

1.3 Definitions, Acronyms, and Abbreviations

TQ Ted's Quest, name of the game

1.4 References

SRS SRS.md, 27/06/20, Ted's Entertainment for Software Engineering Course, https://github.com/GabrielBalzer/tedsquest_documentation/blob/master/srs.md

1.5 Overview

This document describes the architecture of the software application "Ted's Quest".

2. Architectural Representation

As software architecture we planned to use the classical MVC-model. This should make the handling and maintaining of the source code easier and more understandable. By using the Unity Engine, it will be more complicated to implement, since Unity is providing most of the functions. MVC is represented by three parts: model, controller and view. Model collects all data. In case of TQ, it's the players local account, the scores and the local settings. The controller provides the data flow, the logic of the application. Applied to TQ this means the interfering points between the model and the user interface. The user interface is provided by the view. The view renders the data that the controller provides.

3. Architectural Goals and Constraints

Since we are planning on saving the users login data on his local device, the architecture is designed for not publishing any information. If the user is signing up with its Google account, we cannot guarantee the same security, since Google is collecting its own information about its accounts.
The system should be running on computer and Android phones. Which is easy to implement since there is a service from the Unity Engine to build the same application for both devices. This and the used architectural model make the reuse of the source code uncomplicated.
Since we are not copying user data to our own server, it is not necessary at all to use a network connection.

4. Use-Case View

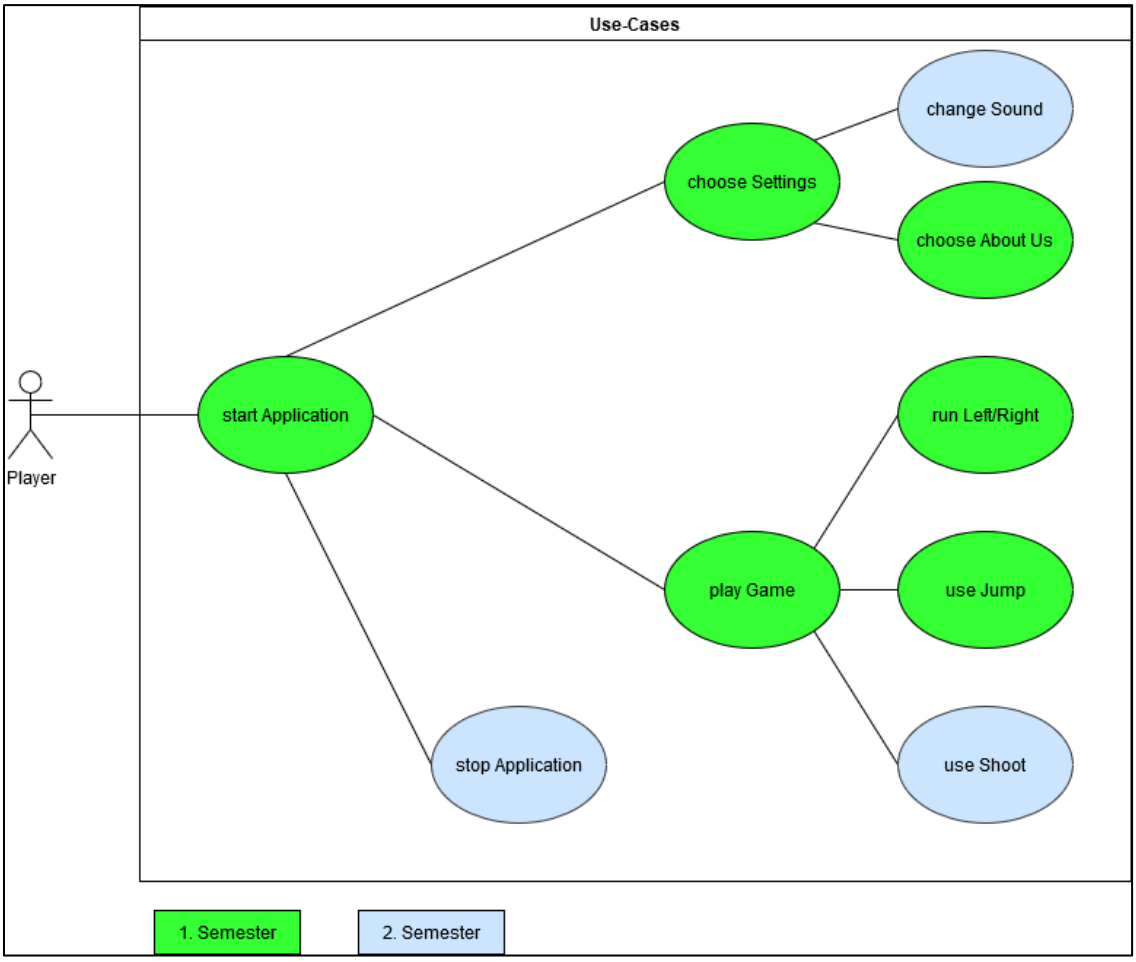

Figure 1: Overall Use-Case Diagram with marked scope.

The figure above shows all use case diagrams with a clearly marked scope for each first and second semester.

4.1 Use-Case Realizations

The following links lead directly to the use cases:

- Use Case – Jump
- Use Case – Main Menu
- Use Case – Move Right
- Use Case – Choose Settings
- Use Case – Shoot
- Use Case - Change Sound
- Use Case – Story Menu

5. Logical View

Significant for the architecture is the separation of the source code into the parts model, view and control. The source code are kept simple and structured.
Therefore, classes for the user interface are structured together. They have a way to communicate with the

control but not with the model. So only control can communicate with all logical views.

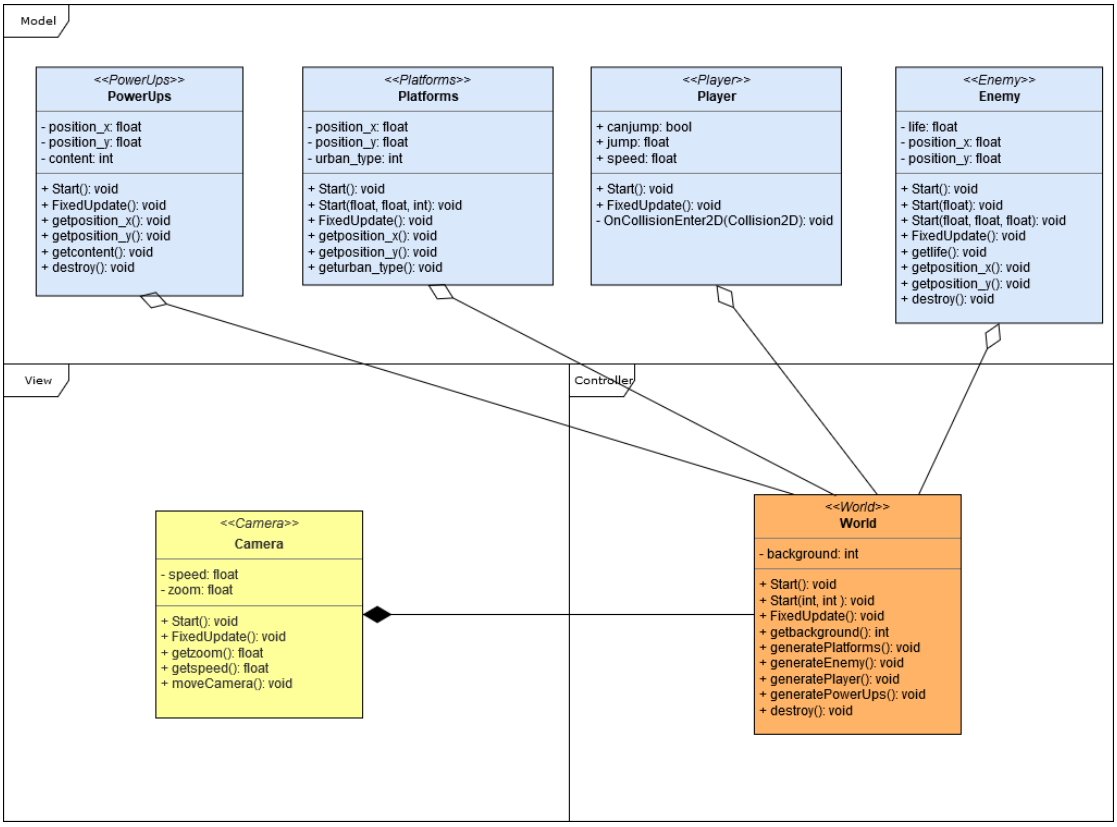

Figure 2: Some classes to demonstrate the MVC-model.

The blue section describes "Model" components, the yellow class describes the "View" and the class "World" is orange and belongs to the part "Controller".
The "Camera" class is the interface of the game to the user. Controlled by the class "World" which itself is controlled by the user input, the data of the class "Player" e.g. to jump or move.

5.1 Overview

Adding design patterns methods to improve code quality and maintainability is limited due to the fact that the Unity Engine automatically generates a great part of the code. Singleton allows to add a design pattern where many classes inherit from a so called "Singleton" class. By using this method, the developer keeps track of the created instances which can be a destroyable object such as a box or a platform but also the player.

5.2 Architecturally Significant Design Packages

Structuring a project is important for further maintaining and developing the code. Therefore the architecture changed slightly.
The figure below shows the significant design by using Singleton classes.

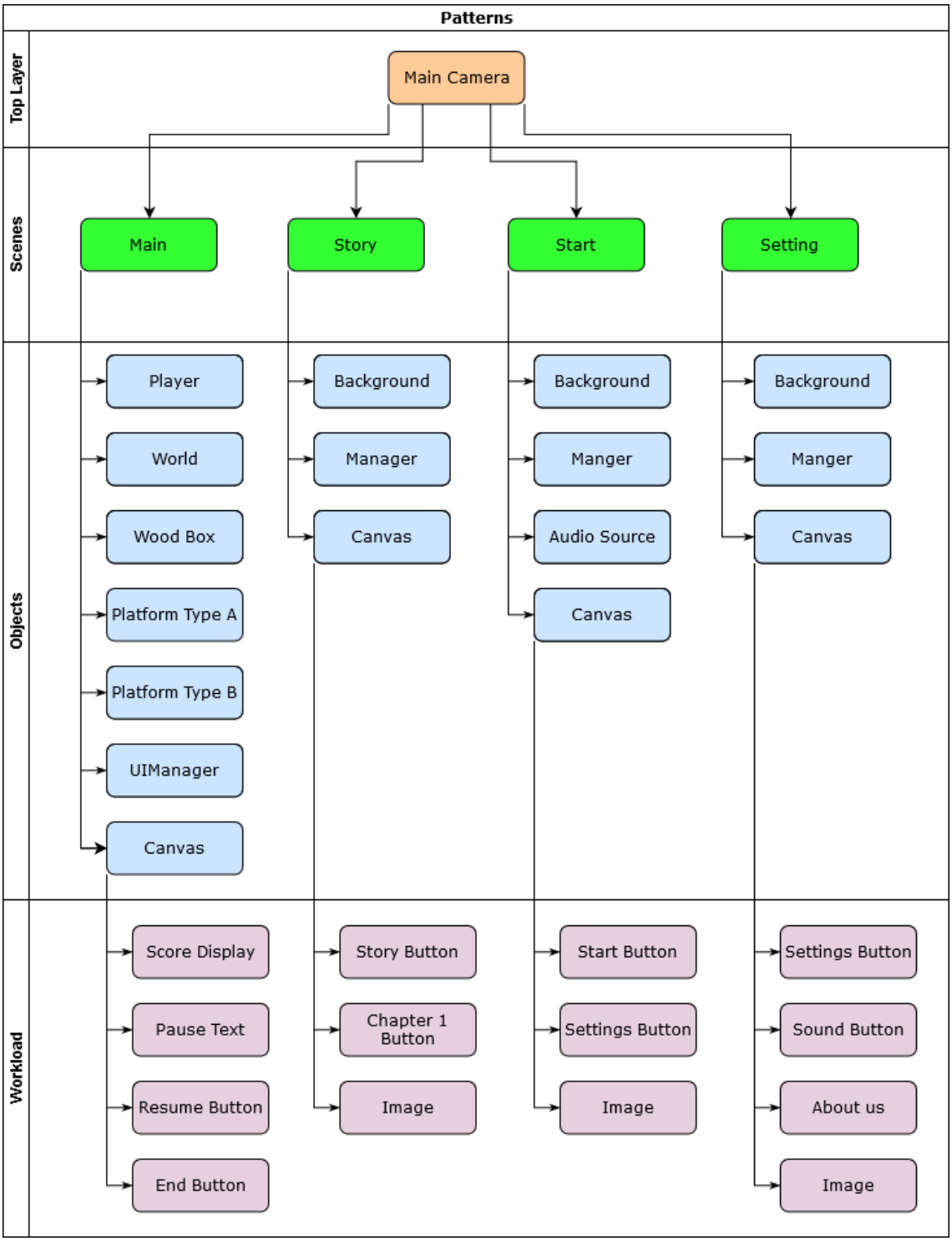

Figure 3: Singleton Design Patterns inherit all modifiable classes.

6. Process View

At this point, there is no real process view possible due to the fact, that there is a too small amount of

classes that the developers can create themselves that could interact.
The user is interfering with the interface-class "World" that passes all valid information to the data storage. This section passes the data back to the interface if necessary and now, the "World" passes this information to the view-part "Camera" for the user to see.

7. Deployment View

There is no delpolyment view for this project possible due to the fact, that there is no hardware connection existing. Either developer nor the user nor the Unity Engine will have access to the hardware. Plus there is no special hardware.

8. Implementation View

Significant for this architecture is the division of the classes into logical groups. Unity makes this harder to implement by taking a lot of orders away from the developer. This implementation will be remarkable for its best effort architecture that relies to the MVC-model.

8.1 Overview

(n/a)

8.2 Layers

(n/a)

9. Data View (optional)

All flowing data from the user input and output is going to and from the Unity Engine. Therefore a figure is not recommended.

10. Size and Performance

The application requires about 60MB free disk space on the computer.

11. Quality

The chosen software architecture contributes to safety from internal data corruption and privacy implications. Portability and reliability are mainly contributed by the Unity Framework, that takes most of the hardware based programming.
By adding design patterns over Singleton to a Unity project and metrics to review the self-written code of the game in Unity, there should be an improved quality for further maintaining of the code. Ted's Entertainment aimes for a full test coverage for the project where the Unity Framework can be excluded since it is not part of the development and provided by Unity.

11.1 Metrics

Metrics can be added by using the web-tool Codacy. Since the Unity Engine only checks if the user defined code is accepted and runs it, there is no other way than manually check for code metrics in the self-written code.
After running Codacy for the first time for the code, the diagram shows as follows 83% issues from the entire self-written code.

Figure 4: Issues in percent after the first execution of Codacy.

After checking some issues, the amount of issues was 42% as shown below.

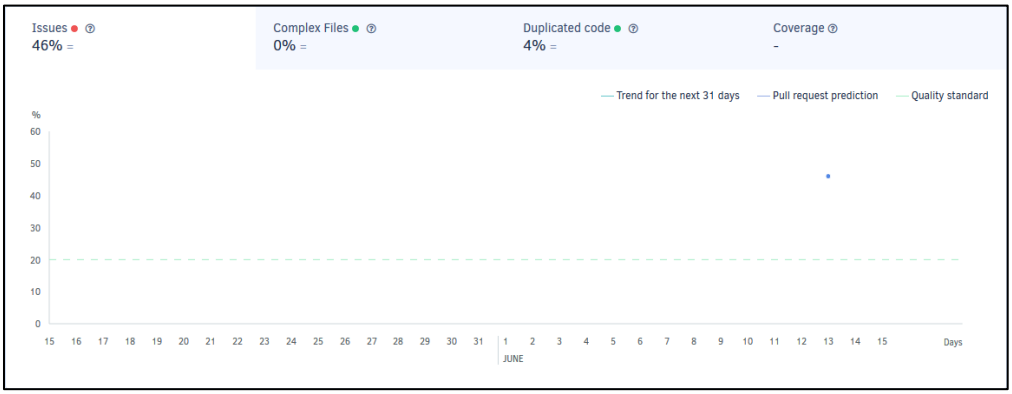

Figure 5: Issues in percent after the second execution of Codacy.

Fixed issues were e.g.:

- Braces Before and After
- Not Supported Before and After

The listing from before and after checking and fixing issues is shown in the pictures below.

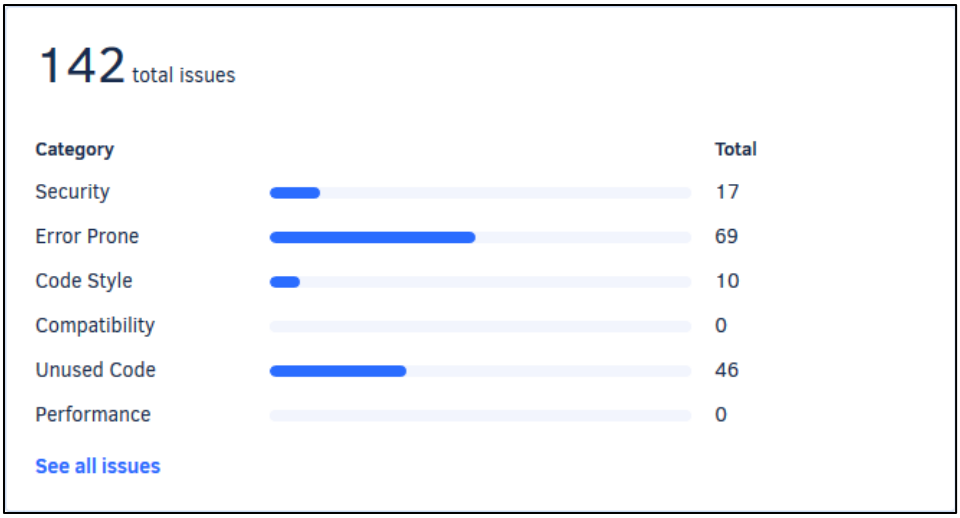

Figure 6: Issues after a first execution of Codacy.

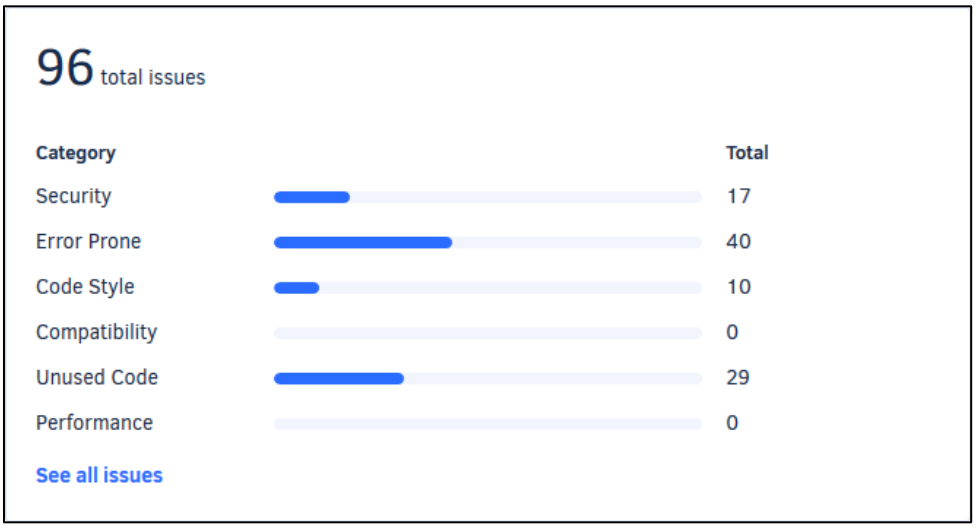

Figure 7: Issues after the second execution of Codacy.

These figures show how many issues can be fixed by looking at few coding styles.