Juho Koponen

Documentation of modular game collection

Bachelor's thesis
Information Technology

2019

South-Eastern Finland
University of Applied Sciences

Tekijä Juho Koponen	**Tutkinto** Insinööri (AMK)	**Aika** Toukokuu 2019
Opinnäytetyön nimi Modulaarisen pelikokoelman dokumentointi		35 sivua
Toimeksiantaja Kaakkois-Suomen ammattikorkeakoulu Oy		
Ohjaaja Lehtori Marko Oras		

Tiivistelmä

Tämän opinnäytetyön tarkoitus oli kehittää ja koota yhteenveto modulaarisen pelikokoelman dokumentoinnista, jonka tilaajana on Kaakkois-Suomen ammattikorkeakoulu. Opinnäytetyö käsittelee opetuskäyttöön kehitetyn pelikokoelman dokumentoinnin suunnittelun ja toteutuksen käyttöä varten.

Modulaarinen pelikokoelma toteutettiin Unity-pelimoottorilla ja sen dokumentointia varten käytettiin useita dokumentointityökaluja ja -tapoja. Pelikokoelman vaatimukset muuttuivat kehitysvaiheessa ja muutokset vaikuttivat dokumentaatioon.

Dokumentoinnin vaatimusperusteet ja suunnittelu, sekä näihin kohdistuvat muutokset, käsitellään opinnäytetyön ensimmäisissä osissa. Dokumentoinnin tavoitteena oli kehittää dokumentaatiojärjestelmä, joka yhdessä ohjeiden kanssa mahdollistaisi peliprojektien tuottamisen aloittelevalta pelinkehittäjältä. Opinnäytetyön pääosa keskittyy dokumentaatiotapojen kehitykseen ja käyttöön yhdessä dokumentaatiotyökalujen kanssa. Dokumentaatiojärjestelmä koostuu useammasta osajärjestelmästä muodostaen kattavan kuvauksen modulaarisen pelikokoelman toiminnallisuudesta ja käytöstä.

Opinnäytetyön tuloksena saatiin modulaarisen pelikokoelman dokumentointi ja dokumentoinnin ohjeistus kehitettyä. Toteutettu dokumentaatio toimii opetustyökalun ohjeiden pohjana. Dokumentaation puutteellisuuden ja virheellisyyden kartoittamista ja korjaamista varten vaaditaan jatkossa käyttötestausta.

Asiasanat

dokumentointi, modulaarisuus, pelikehitys

XAMK
South-Eastern Finland
University of Applied Sciences

Author	Degree	Time
Juho Koponen	Bachelor of Engineering	May 2019

Thesis title

Documentation of modular game collection

35 pages

Commissioned by

South-Eastern Finland University of Applied Sciences Ltd.

Supervisor

Marko Oras

Abstract

The purpose of this thesis was to compile and develop the documentation of a modular game collection, a learning tool commissioned by South-Eastern Finland University of Applied Sciences. The design and implementation of the documentation for the game collection is described in this thesis.

The modular game collection was developed with Unity game engine. Various documentation tools and methods were used for the documentation of the game collection. The requirements of the game collection changed during the development and affected the documentation.

This thesis details the requirements and the design of the documentation, including changes made in the process. The objective was to develop a documentation system which, together with instructions, would enable a beginner game developer to construct game projects. The development and utilization of the documentation methods are the focus of this thesis, together with the documentation tools. The documentation system consists of multiple subsystems, forming a comprehensive description of the functionality and usage of the modular game collection.

The system and the related guidelines were developed as a result of this thesis. The implemented system functions as a basis for the instructions of the learning tool. In order to discover and correct the defects and deficiencies in documentation, usability testing of the modular game collection is required.

Keywords

documentation, game development, modularity

CONTENTS

ABBREVATIONS 6
1 INTRODUCTION 7
1.1 Purpose of this thesis 7
1.2 Approach to the subject 8
1.3 Commissioner 9
2 REQUIREMENTS 9
2.1 Modular game collection 9
2.2 Learning tool 10
2.3 Modularity 10
2.4 Instructions for game projects 11
2.5 Documentation 11
2.6 Changes in the documentation 12
3 DESIGN AND PREPARATIONS 13
3.1 Documentation guidelines 14
3.1.1 Code commenting guidelines 14
3.1.2 Error documentation guidelines 15
3.1.3 Graph guidelines 16
3.1.4 Instruction guidelines 16
3.2 Documentation database 16
3.2.1 Selection of the documentation database tools 17
3.2.2 Doxygen 17
4 IMPLEMENTATION 18
4.1 Game design documents 18
4.2 Code documenting 19
4.2.1 Pseudocode 20

4.2.2 Clean code....20
4.3 Unity Editor tooltips....21
4.4 Unity Editor error messages....22
4.5 Doxygen....22
4.6 Documentation for instructions....27
4.7 Graphs....28
4.7.1 Class diagrams....28
4.7.2 Diagrams....29
4.7.3 Game object diagrams....30
5 FUTURE DEVELOPMENT....31
6 CONCLUSIONS....32
REFERENCES....34
LIST OF FIGURES

ABBREVATIONS

Ddoc	Documentation generator for D programming language.
HTML	Hypertext Markup Language, a standardized markup language for creating web pages.
JSDoc	Markup language for JavaScript documentation.
NDoc	Documentation generator.
Pandoc	Document converter.
RDoc	Documentation generator for Ruby programming language.

1 INTRODUCTION

With the growth and progress of the modern software development, the importance of documentation has become apparent. Project-oriented software development requires communication, not only between the developers, but the developer and the software. Communication enables the various tasks to be distributed between multiple developers, and sufficient documentation allows to monitor the advancement of the development. Introducing new developers or maintaining the software will become easier when information is directly readable without the need to interpret it from the code.

Documentation is an important aspect of programming projects of any extent. The purpose of documentation is to communicate the functionality of the program to other developers and users. Documentation is useful for design, development, testing, maintenance and further development.

Documentation can be implemented on a variety of levels: internally within the written code, integrated within the development environments or externally with documents and databases. The intended users and the chosen documentation methods together with other requirements result in a project-dependent documentation.

1.1 Purpose of this thesis

This thesis was commissioned to develop the documentation of a modular game collection. The thesis compiles the utilized documentation tools and methods.

The first part of this thesis describes the requirements of the documentation. The first part is followed by an introduction to the design and preparation of the documentation, describing the process of elaborating the documentation system from subsystems as established by the requirements.

The focus of this thesis is the implementation of the documentation: the development of the documentation methods and guidelines and the utilization of documentation tools. The final, and conclusive part discusses the lessons learned from the documentation of the game collection. The extent of the success of the documentation will also be contemplated.

The development of the modular game collection and the changes made to it during the development are not included in the scope of this thesis. The changes caused the implemented game collection to diverge from its original vision. Briefly summarized, the changes were a result of multiple factors from inadequate design-phase and project management: underestimating the importance of instructions as one of the requirements, misunderstandings in the requirements due to inefficient communication and time-consuming solutions made during the development. This thesis will not describe the changes of the game collection or causes to these changes in further detail. The instructions created for the game projects of the modular game collection will not be included, but the documentation made for the instructions will be described in this thesis.

1.2 Approach to the subject

The development team for the modular game collection consisted of three developers. For the development team, the modular game collection was one of the first large-scale development projects. Previous experience included game projects and game prototypes, none of which compared to a modular, interconnected system. The approach to the modular game collection was to involve the whole development team in constructing the modular base for the game collection and developing the documentation and instructions.

The author of this thesis had minor prior experience of documentation, mainly in code commenting and game design documents. Knowledge of documentation methods and tools was not extensive. Exploration of how to construct a documentation around a system functions as the basis for this thesis.

1.3 Commissioner

GameLab of South-Eastern Finland University of Applied Sciences provides learning tools and environments for game programming students. South-Eastern Finland University of Applied Sciences is a higher educational institution, with campuses in multiple cities. The motto of the university is unlimited learning, hence the GameLab is constantly exploring and experimenting new ways for improving teaching and learning methods.

2 REQUIREMENTS

The thesis commissioned and related instructions were given in September 2018. Discussions were held in order to define the objectives. No software requirement specification or game design document were created as a result of the discussions.

Initial objectives were changed during the development process both by the development team and the commissioner. This thesis will not focus on the changes in the project, and the reasons for the changes will only be briefly mentioned in the conclusive chapter. The effects on the documentation will be addressed at the end of this chapter.

2.1 Modular game collection

The commissioner gave an assignment to create and develop a modular game collection for educational purposes. The purpose of the product was to serve as a learning tool for game development. The users would construct standalone games, but the modular structure would provide possibilities for customization and experimentation.

The modular game collection would contain 10 different games, each varying in genre. The games would be deconstructed to be used as material in game development projects for educational purposes.

The modular game collection was developed using Unity game engine and Microsoft Visual Studio as the program development environment. GitHub hosting service was used for the project version control of the game collection.

2.2 Learning tool

The intended users would be the new students, and the modular game collection would serve as a learning tool for them. Users would reconstruct several specific games from deconstructed game projects. The modular game collection would include elements from the design, development and various other aspects of programming projects. Game reconstruction would directly introduce new students to game design, project management, version control, usage of a game engine, programming, programming terminology and troubleshooting.

2.3 Modularity

Standalone games of the modular game collection were required to consist of modular or shared elements. Modular elements, such as game objects, scripts and data structures, would allow their usage in other games. Users would have been able to customize their reconstructed projects by seamlessly integrating modular elements or even mechanics into their creations. For example, introducing an enemy object or game score system used by other games.

In software development, modularization also means subdividing the program and its development into more easily manageable parts. The benefits of this type of modularization are the maximisation of the individual programmer productivity and the minimisation of the damage caused by the changes in the functional requirements. Dividing the program to subunits and modules, promotes maintainability and reusability; changes to the whole system are redirected into smaller modules. (Robinson 2004, 276-277.)

For the modular game collection, modularity meant efficiency in both development and documentation: modules reduce the amount of repetition and promote the task distribution between the developers. Additionally, the maintenance of the documentation would become facilitated as the changes would affect only a part of the system and, therefore, only a part of the related documentation.

2.4 Instructions for game projects

The modular system and its game projects required instructions for reconstructing the individual games. The importance of the instructions was also emphasised by the intended userbase: new students were assumed to have little or no previous practice or experience in either game development or programming projects. The objective for the instructions was to cover the basics of game development and programming and to explain advanced or complex concepts, such as object pooling. Overall, the aim was to provide instructions for reconstructing the game projects with clearly defined steps and in a clearly defined order, as well as instructions for the utilization of the modular elements.

An introduction to the basics of troubleshooting was also a requirement for the modular game collection project. Instructions were required to contain a controlled troubleshooting scenario as well as support for uncontrolled situations. The intention for the controlled troubleshooting scenario was to introduce problem solving as a part of the programming projects and familiarise the reader with terminology used in the process.

In addition to the intended troubleshooting scenarios, the instructions had to include support for uncontrolled errors. The intended users were not expected to be able to solve the potential problems for example due to missing references. Additionally, error messages created by Unity may contain terminology or concepts unfamiliar to the users.

2.5 Documentation

The documentation is project-specific (Robinson 2004, 284), and the aforementioned requirements for of the modular game collection relied on the documentation. The range of the documentation can vary from structural arrangement and commentary to module description documents, depending on the scope and structure of the project (Robinson 2004, 284-285).

A modular system has to be documented sufficiently to allow the users – teachers and students – to understand the game collection from their respective perspectives. As a learning tool, the modular game collection was required to introduce game programming and game programming concepts to beginner-level users. At the beginning of the project development, instructions were planned to be based on module documentation, referring to it whenever appropriate. These reasons (Figure 1) resulted in the documentation of the game collection becoming a mandatory aspect of the project.

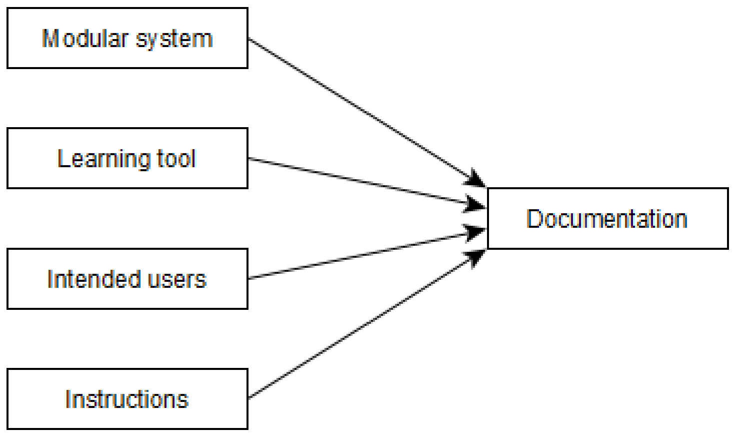


Figure 1. Requirements for the documentation.

2.6 Changes in the documentation

The initial modular system changed during the process into standalone game tutorials, and the documentation of the product was affected by these changes. Changes made to instruction requirements had a substantial effect, as a significant amount of documentation was specifically designed for the instructions.

As a result, documentation methods had to be reconsidered and changed. Some documentation methods were dismissed, and some were introduced to comply with the new requirements. Some documentation methods were not directly affected but their role in the documentation was potentially impacted. This thesis will describe the method-specific changes in the chapters 3 and 4.

3 DESIGN AND PREPARATIONS

As documentation was a mandatory aspect of the thesis, the planning of it began alongside with the design-phase of the modular game collection. Later, the redefining changes made to the modular game collection extended the design-phase of the documentation.

Initially, some major aspects for documentation were identified. The modules of the modular game collection had to be documented, including the usage of the modular elements. The code of the scripts had to be explained for the users having no previous understanding of the used programs. Potential error-scenarios required documentation that would provide support in resolving the problems. Instructing the usage of the game engine was not a requirement for the documentation.

Identifying the key areas of the documentation resulted in four major sections. Firstly, a documentation database for the modular system was required. Secondly, the code had to be documented in order to clarify the functionality of the program. Thirdly, error documentation for initially planned controlled and uncontrolled error-scenarios was required. Finally, visuals to support the documentation in representing the various structures and concepts of the game projects.

In conclusion, the documentation was designed to be not one separate system but multiple integrated subsystems and documentation methods (Figure 2). This design-choice followed the idea of modularization: breaking larger entities into smaller components. Tools for documentation were considered and compared. Designing the documentation also included the consideration of interactions and compatibility between the documentation methods.

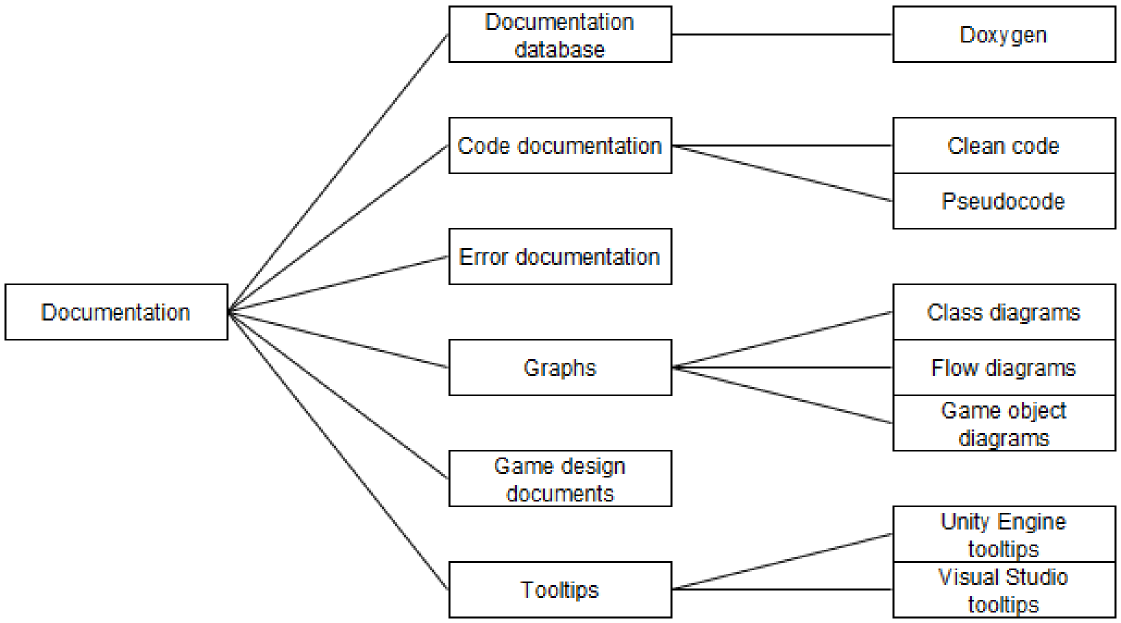


Figure 2. Sections of the documentation and documentation tools and methods.

3.1 Documentation guidelines

Documentation guidelines were designed and established, for controlling the implementation of the modular game collection documentation. Guidelines provided standards in the development with the purpose of promoting consistency in the complex, large scale project. Guidelines were created for the code commenting, error documentation, graphs and instructions.

The primary language used in documentation was chosen to be Finnish, as the primary language of instruction in the degree programme of game programming was Finnish. English terms were used whenever Finnish alternatives were not suitable or available.

3.1.1 Code commenting guidelines

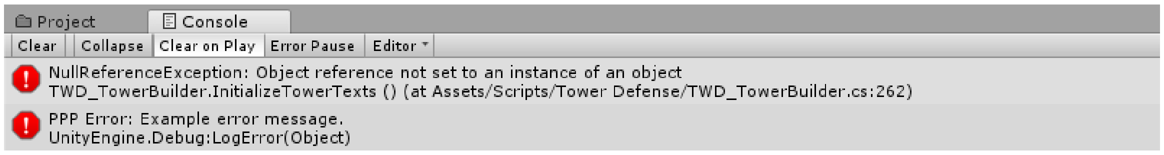
Under normal circumstances, comments are supposed to explain why – not how – a section of the code exists. The explanation of the functionality of the program should not be repeated by comments as the code itself should be sufficiently clear and readable. (Goodliffe 2006, 75–76.) However, in terms of purpose and requirements, the modular game collection was not a typical programming project; the intended users were expected to be unable to read the code and understand how it works. The functionality of the code and the role of the various elements such as classes, variables, methods and data structures, were decided to be explained by pseudocode-like comments within the code.

The purpose of the code commenting guidelines was also to ensure that the organisation of command lines in the code for both Visual Studio and the documentation generation tool remained intact and cause no disturbances.

3.1.2 Error documentation guidelines

Error documentation was initially designed to answer questions what, why and where. Custom-made error messages would contain information on what went wrong and where, and the documentation system for error messages would contain support for troubleshooting. The error messages were planned to contain individual identification and script as well as the method and the line number of how and where in the code the message originated. The documentation of the error messages was planned to contain a list of all the custom error messages, their causes and suggestions for solving the problems.

Custom error messages were planned to appear like the error messages in Unity Editor (Figure 3). In order to be distinguishable from Unity Editor error messages, the custom error messages would contain a prefix at the beginning of the message. The prefix would distinctly set the custom error messages apart from the Unity Editor error messages.



```
NullReferenceException: Object reference not set to an instance of an object
TWD_TowerBuilder.InitializeTowerTexts () (at Assets/Scripts/Tower Defense/TWD_TowerBuilder.cs:262)
PPP Error: Example error message.
UnityEngine.Debug:LogError(Object)
```

Figure 3. An example of custom error message in Unity Editor.

The custom error messages were also planned to appear in the same location as the Unity Editor error messages: console. As a result, all the error messages would be found in the same place. The causes for custom error messages were assumed to commonly be the changes made within the Unity Editor or the incompatibility between the code and the current state of the game project. When double-clicked, the messages in Unity Editor console also offered a quick access to the section of the code where the message was from.

Along with the changes in the modular game collection development, changes were made to the error message documentation. The error documentation system and majority of the custom error messages were discarded. Selected messages were kept for their specific reasons. The further details of these changes will be described in chapter 4.

3.1.3 Graph guidelines

The graphs were planned to follow Unified Markup Language, or UML, whenever applicable. UML is considered a standard amongst the software developers as it enables the representation of structures and their details in various programming languages, including C# (Doran & Casanova 2017, 12). UML diagrams were planned to be utilized in the representation of functional concepts, structures and their relations. UML diagrams would introduce the userbase to the standardized modelling language.

3.1.4 Instruction guidelines

Guidelines were required for the creation of consistent instructions. The instructions for the reconstruction of the game projects had to be manually created and required standards for structure and contents. Instructions were designed to include written guidance and visuals to support and explain the guidance. Navigation within the instructions was also an important aspect because the instructions for a specific game project were planned to refer to the common instructions.

3.2 Documentation database

A wiki-style database for the documentation was required to document the modules of the modular game collection. A database would provide information on various classes and their data members. The database would also explain the usage of the member-functions of the classes and the relationships between the classes.

3.2.1 Selection of the documentation database tools

Criteria for the documentation tool for the modules of the modular game collection included budget, language support, licence, operating system support, programming language support. Of various options, Doxygen was preferred over such alternatives as Ddoc, JSDoc, NDoc, Pandoc and RDoc.

While Doxygen would automatically generate documentation for the modules of the modular game collection, a solution for creating a wiki-site was also initially required. The wiki-site would include instructions, error documentation and documentation generated by Doxygen. Google Sites and GameLab's wiki-base were considered as options for hosting the wiki-site. GitHub repository's wiki-section was also taken into consideration after the utilization of Git version-control. However, changes made to the modular game collection during its development resulted in a requirement for a tutorial-oriented solution.

3.2.2 Doxygen

Doxygen is a documentation generation tool, supporting multiple programming languages. Doxygen can be used to create a documentation browser from the source code. (Doxygen 2019.) It can generate HTML-files from the source code, offering both an overall view to the program as well as detailed views to its modules.

Doxygen automatically recognizes data structures, classes and class members and generates a database from these elements. Users can add comments to the code for Doxygen to incorporate them in the documentation (Doxygen 2019). For example, additional details may be added for specific classes or their data members and the details will be displayed in the documentation. Doxygen does this by reading special commands.

Special commands are lines in the code which start with a character recognized by Doxygen. By utilizing special commands, additional information can be included in the documentation, sections can be excluded from documentation generated by Doxygen or descriptions can be added to code elements which are left undocumented by default.

4 IMPLEMENTATION

Implementing the documentation was started along with the first game project and creation of modular elements. The development of the game projects and the instructions of the modular game collection continued even after the completion of this thesis.

Designing the game collection, the individual games projects, documentation and instructions was carried out by the whole development team. Implementation of the common basis of the game collection, the documentation and the instructions involved all of the developers, while the implementations of the individual game projects were primarily developed as individual works.

Not all the documentation tools and methods were chosen at the start, but during the study, depending on the needs and development stages. Also, not all the planned documentation methods carried over from design-phase to implementation. Utilization of the documentation tools and methods is described in this section of the thesis, including the potential changes made to the specific tools or methods during the development.

4.1 Game design documents

Brief game design documents for the individual game projects of the modular game collection were created. The purpose of these game design documents was to introduce the game project before the intended users would begin its reconstruction. The game design documents described the requirements of the project and detailed the key concepts of the functionality of the game.

Compared to complete large-scale game design documents, the documents of the modular game collection were simplified in order to focus on the essential aspects of the game project. Large-scale game design documents may include the theme, characters, assets, level design, user interface and everything else that is required to be developed for the game. The simplified game design documents included a brief description of the game, its controls and core game mechanics.

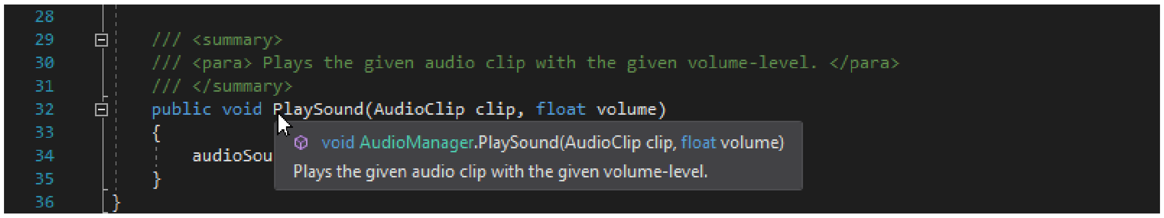
4.2 Code documenting

The implemented code for the modular game collection was required to be of high quality and well documented. The documentation of the code began from the code itself: the code had to be understandable, scripted in an organized manner and described for the intended users.

Algorithms and functions were identified as important aspects of the code which required functional descriptions. Classes – many of which were designed to be reconstructed by the intended users – were planned to be described in the instructions when they were reconstructed. Variables and data structures were designed to be named with simple and clear identifiers, requiring minimal depictions. Code description focused on the methods of the scripts of the modular game collection.

The intent and functionality of the methods were described in code comments. Get- and Set-methods were generally left uncommented unless their functionality was more advanced than to simply control the changes made to the variable. If methods had parameters or returnees, these elements were also given descriptions. With the utilization of Doxygen documentation tool, the method descriptions were carried over to the automated documentation generation.

Visual Studio tooltips were utilized for method descriptions. Summaries were intended to be used to describe the type member of a class (Microsoft Documentation 2015), and public member methods, which were called by other scripts, were summarized. Summaries become visible in the Visual Studio environment when the mouse cursor hovers over the method (Figure 4).



```
/// <summary>
/// <para> Plays the given audio clip with the given volume-level. </para>
/// </summary>
public void PlaySound(AudioClip clip, float volume)
{
    audioSou
}
}
```

```
void AudioManager.PlaySound(AudioClip clip, float volume)
Plays the given audio clip with the given volume-level.
```

Figure 4. Tooltip in Visual Studio.

4.2.1 Pseudocode

Pseudocode is a description of the basic function of the code or an algorithm. The purpose of the pseudocode is to outline the functionality of the program, detached from the programming languages. Pseudocode can be used to design the code before implementing it and may be used in the documentation as it typically describes the purpose of the code.

In the modular game collection, the comments written in the code functioned similarly to the pseudocode, describing the functionality of the code on a basic level. The functionality descriptions were inserted in the code, instead of being introduced before the function. The purpose of this was to present the description of the code functionality in the location of this said functionality. This complied with the notion of close proximity: the comment would be understood in its context and would easily be updated if changes were made to the code (Goodliffe 2006, 74).

4.2.2 Clean code

In the context of this thesis, the term ”clean code” is used to describe code which has been polished to be readable and understandable. Clean structure and clean code are some of the most important documentation methods as the purpose of documentation revolves around understanding the program and its functionality. Obscure code defeats the purpose of documentation.

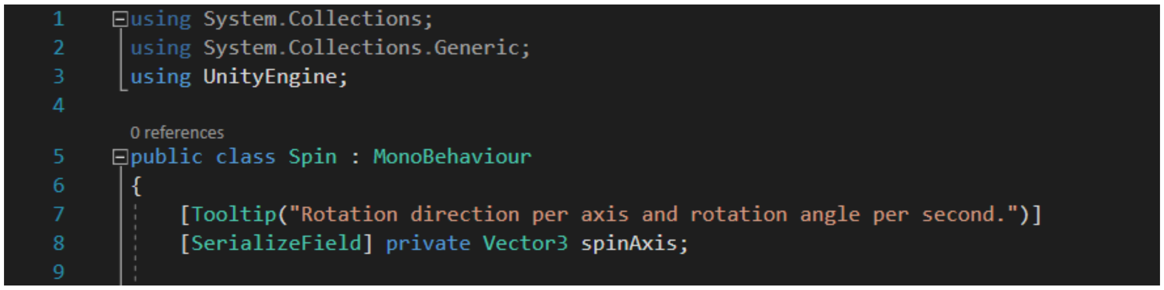
Clean code has several benefits. Readable and understandable code is less prone to errors and misuse. Clean code is also easier to maintain by the author of the code or by other programmers. (Doran & Casanova 2017, 323-324.) Easily readable code is essential for effective teamwork between multiple developers.

From the viewpoint of the modular game collection in this thesis study, clean code was necessary since the intended users were expected to program with the help of instructions. Therefore, classes, variables and methods were precisely named. The structures of the algorithms were planned utilizing pseudocode and reviewed before implementation. The developed code was reviewed several times, and, if necessary, components were renamed and reorganized.

The scripts of the modular game collection were organized according to their use: common scripts were located in a script-folder, and the project-specific scripts had their own folders within the script-folder. In addition, the names of the project-specific scripts included a short prefix of the project.

4.3 Unity Editor tooltips

Variables, that were considered non-self-explanatory were given tooltips, visible in Unity Editor. The tooltip-attribute had to be assigned to the variable within the code (Figure 5). Tooltips are visible in Unity Editor when the mouse cursor hovers over the variable (Figure 6) (Unity Documentation 2019b). The purpose of these tooltips is to explain and clarify the role and the use of the specific variable.

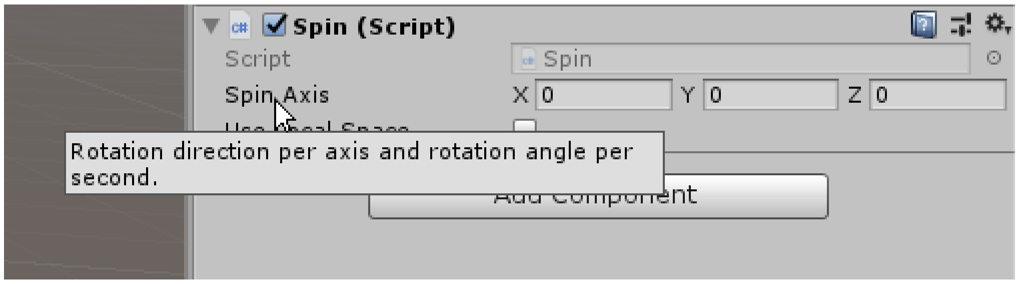


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 references
public class Spin : MonoBehaviour
{
    [Tooltip("Rotation direction per axis and rotation angle per second.")]
    [SerializeField] private Vector3 spinAxis;

```

Figure 5. Implementation of Tooltip-message in code.



Rotation direction per axis and rotation angle per second.

Figure 6. Tooltip in Unity Editor.

4.4 Unity Editor error messages

Due to considerable changes during the development, the initial plans for error message documentation were not implemented; the users were required to write parts of the code by themselves, which no longer guaranteed the reliability of custom error messages.

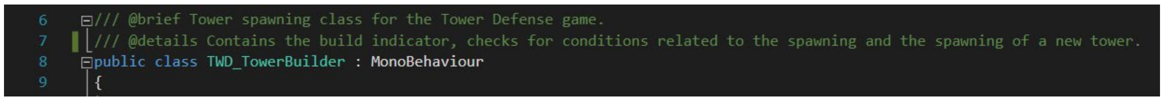
The discarded error messages were left to be processed by Unity Engine. Many of the custom error messages concerned missing references and were originally planned to function as null checks. Before a reference to an object is being used, a variable in the code is inspected to see whether it contains the reference or is missing the reference (Unity Documentation 2019a). Otherwise, Unity Engine creates a Null Reference Exception. Instructions for solving Null Reference Exceptions were planned to be created in the refined error documentation.

Some error messages were chosen to be kept. These error messages concerned game loops and game states. The remaining error messages were located in a state machine, within an Update-function. If the state machine fails to define its state for an unexplained reason, an error message will be created. Without these messages, Unity Engine will not recognize the error, and the user will not be informed. In some of the games, these error messages were located in the premade segments of the code.

4.5 Doxygen

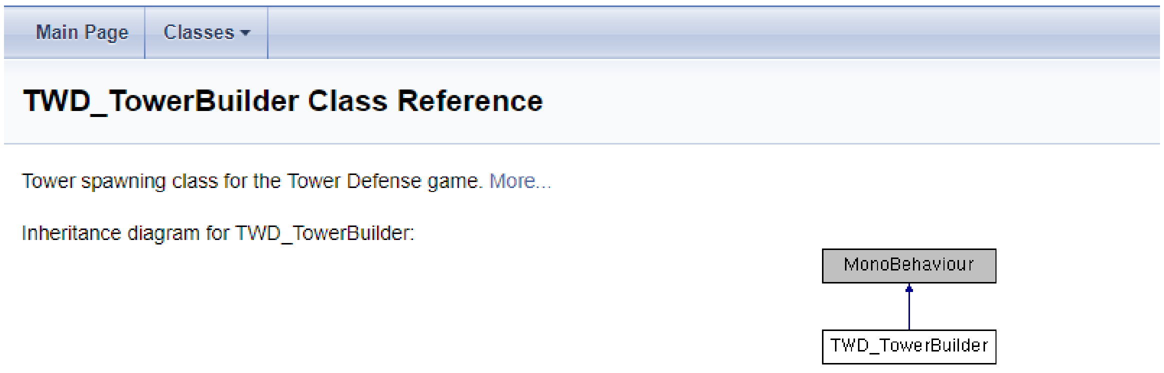
Doxygen generates documentation from defined source files. Doxygen first had to be configured in order to include source files, and a configuration file was set up to generate a documentation database. The configuration file included several options regarding the documentation, such as language, included and excluded files, input and output options, generation and sorting options (Doxygen 2019).

Every class in the game scripts was described for the documentation. Doxygen special commands @brief and @details were used in the code (Figure 7), and the Doxygen documentation generation function conversed the descriptions (Figure 8 & 9). Class inheritance was also represented by a diagram in the generated documentation.



```
/// @brief Tower spawning class for the Tower Defense game.
/// @details Contains the build indicator, checks for conditions related to the spawning and the spawning of a new tower.
public class TWD_TowerBuilder : MonoBehaviour
{
```

Figure 7. Class description using Doxygen special commands @brief and @details in code.

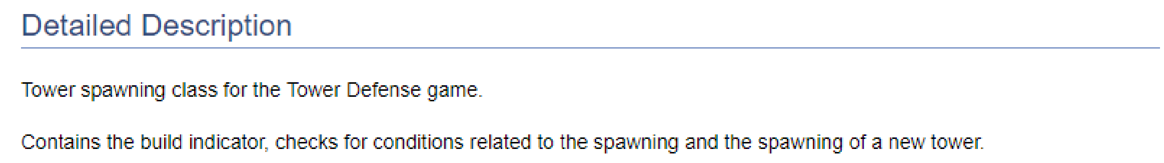


TWD_TowerBuilder Class Reference

Tower spawning class for the Tower Defense game. More...

Inheritance diagram for TWD_TowerBuilder:

Figure 8. Class description generated by Doxygen.



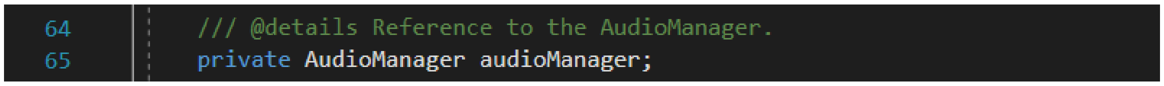
Detailed Description

Tower spawning class for the Tower Defense game.

Contains the build indicator, checks for conditions related to the spawning and the spawning of a new tower.

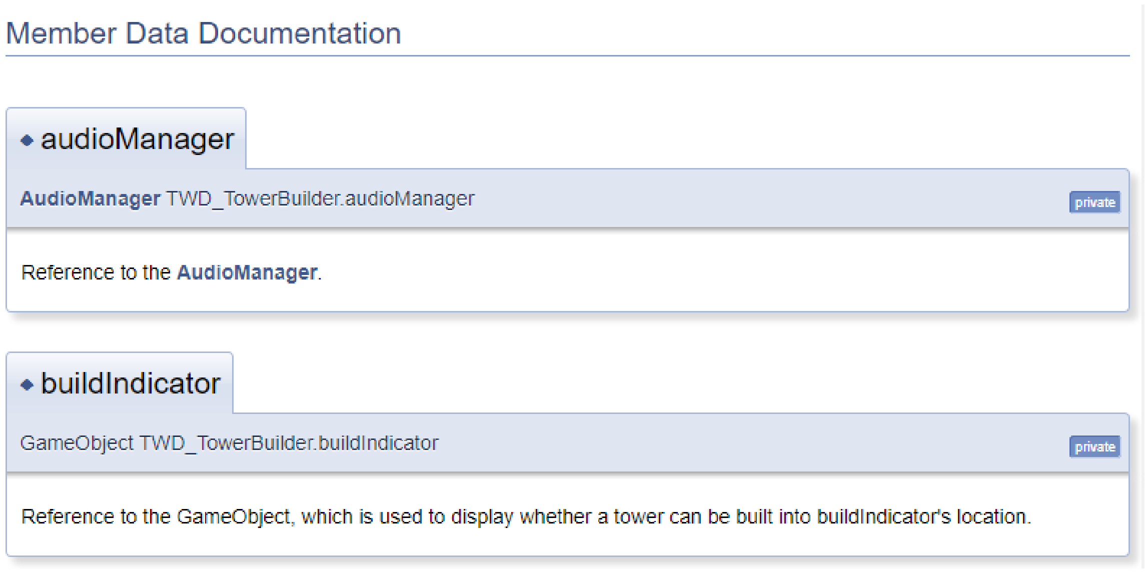
Figure 9. Detailed class description generated by Doxygen.

Variables were described with Doxygen special command @details in the code (Figure 10). When conversed to the Doxygen documentation (Figure 11), the variable information included the type, access modifier and custom description of the variable.



```
/// @details Reference to the AudioManager.
private AudioManager audioManager;
```

Figure 10. Variable description using Doxygen special command @details in code.



Member Data Documentation

◆ audioManager

AudioManager TWD_TowerBuilder.audioManager private

Reference to the AudioManager.

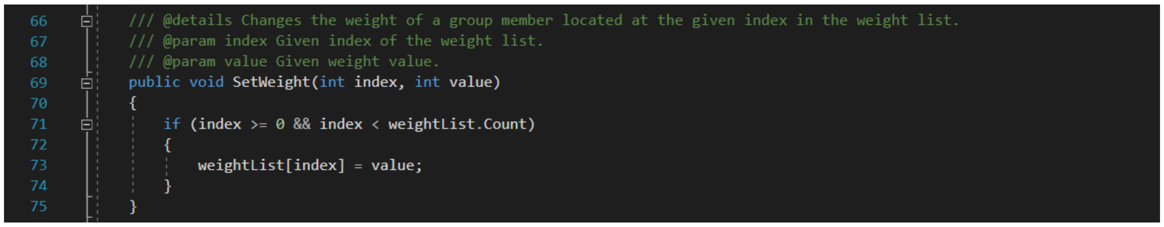
◆ buildIndicator

GameObject TWD_TowerBuilder.buildIndicator private

Reference to the GameObject, which is used to display whether a tower can be built into buildIndicator's location.

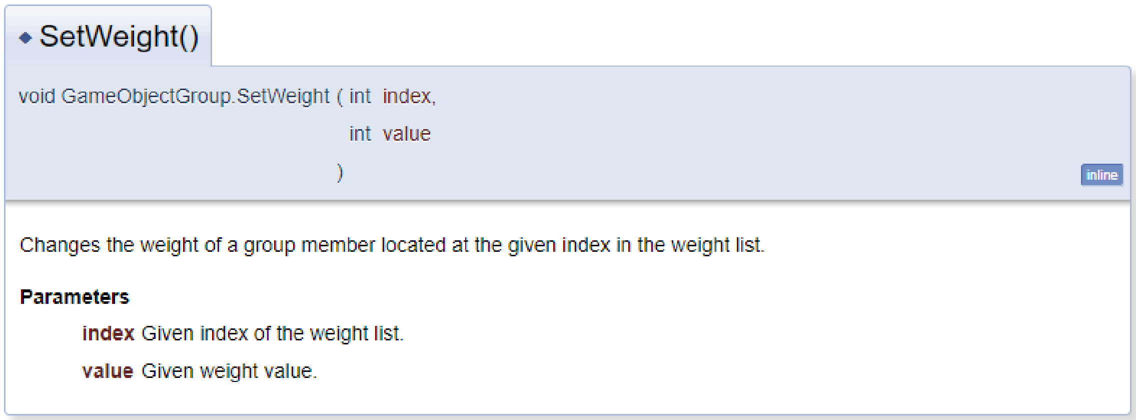
Figure 11. Variable descriptions generated by Doxygen.

Several functions – excluding Get- and Set-functions – were given a description, and also the parameters of these functions were described. In the code, Doxygen special command @details was used for the function descriptions, and @param was used for the parameter descriptions (Figure 12) and conversed into the generated documentation (Figure 13).



```
/// @details Changes the weight of a group member located at the given index in the weight list.
/// @param index Given index of the weight list.
/// @param value Given weight value.
public void SetWeight(int index, int value)
{
    if (index >= 0 && index < weightList.Count)
    {
        weightList[index] = value;
    }
}
```

Figure 12. Doxygen special command @param in code.



◆ SetWeight()

```
void GameObjectGroup.SetWeight ( int  index,
                                 int  value
                               )
```

inline

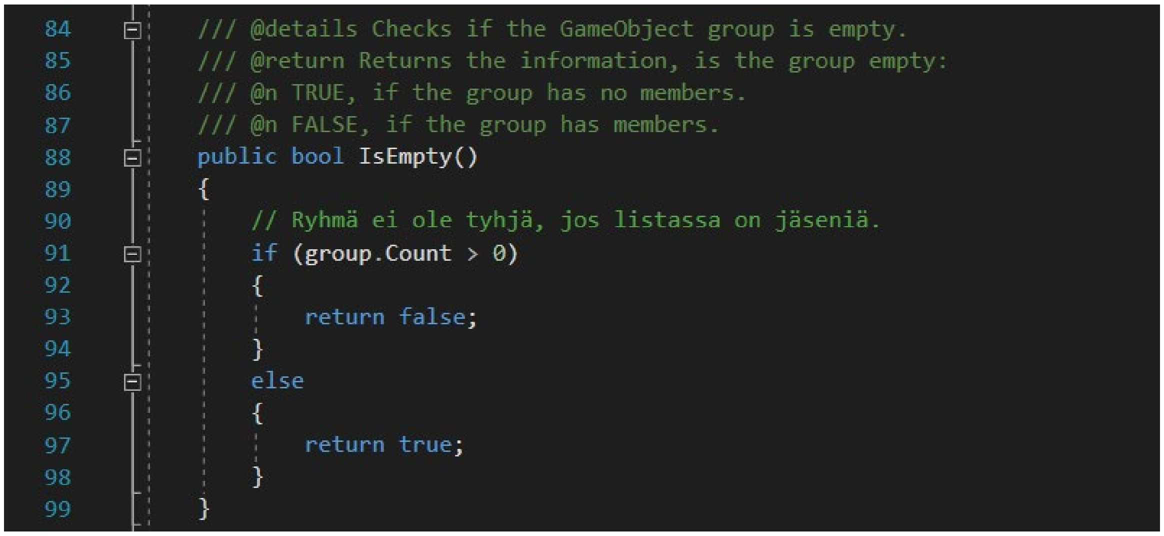
Changes the weight of a group member located at the given index in the weight list.

Parameters

index	Given index of the weight list.
value	Given weight value.

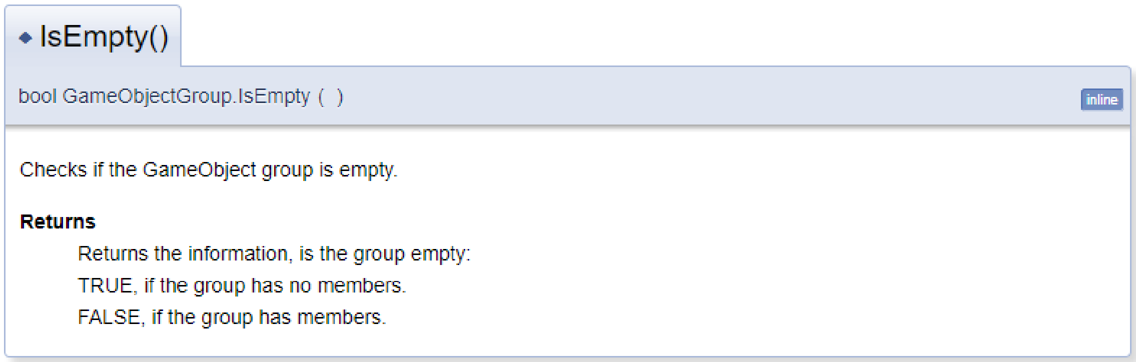
Figure 13. Function description with parameters generated by Doxygen.

Whenever possible, special command @return (Figure 14) was used to include additional information about the functions’ return values in the documentation (Figure 15). If a function returned a Boolean value, additional descriptions were given for all the returnee values.



```
/// @details Checks if the GameObject group is empty.
/// @return Returns the information, is the group empty:
/// @n TRUE, if the group has no members.
/// @n FALSE, if the group has members.
public bool IsEmpty()
{
    // Ryhmä ei ole tyhjä, jos listassa on jäseniä.
    if (group.Count > 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Figure 14. Doxygen special command @return in code.



◆ IsEmpty()

bool GameObjectGroup.IsEmpty ( ) inline

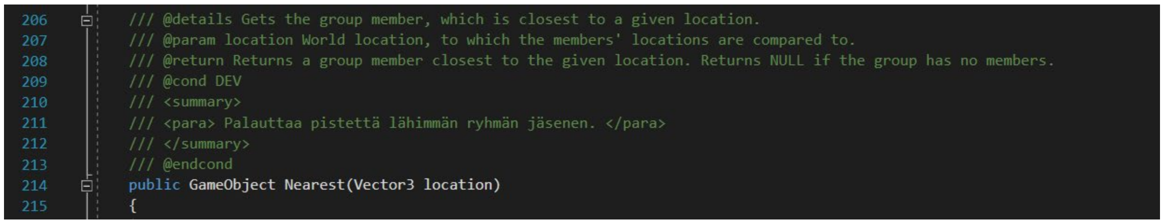
Checks if the GameObject group is empty.

Returns

Returns the information, is the group empty:
TRUE, if the group has no members.
FALSE, if the group has members.

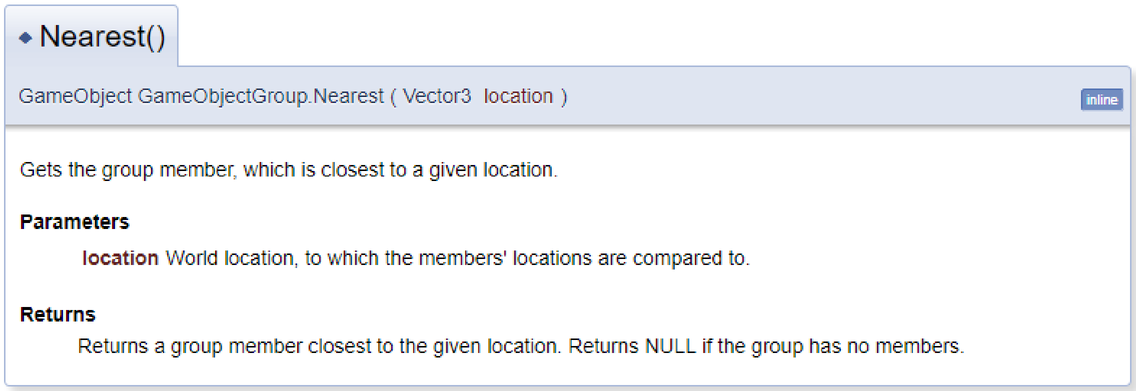
Figure 15. Function description with returnee generated by Doxygen.

Special commands @cond and @endcond (Figure 16) were used to exclude comments not intended to be included for Doxygen documentation from being incorporated by the documentation generation. These special commands prevented the comments used for Visual Studio tooltips from being included in the descriptive section of the documentation (Figure 17).



```
/// @details Gets the group member, which is closest to a given location.
/// @param location World location, to which the members' locations are compared to.
/// @return Returns a group member closest to the given location. Returns NULL if the group has no members.
/// @cond DEV
/// <summary>
/// <para> Palauttaa pistettä lähimmän ryhmän jäsenen. </para>
/// </summary>
/// @endcond
public GameObject Nearest(Vector3 location)
{
```

Figure 16. Doxygen special commands @cond and @endcond in code.



◆ Nearest()

GameObject GameObjectGroup.Nearest (Vector3 location) inline

Gets the group member, which is closest to a given location.

Parameters

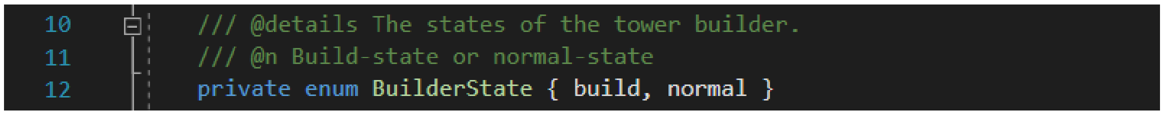
location World location, to which the members' locations are compared to.

Returns

Returns a group member closest to the given location. Returns NULL if the group has no members.

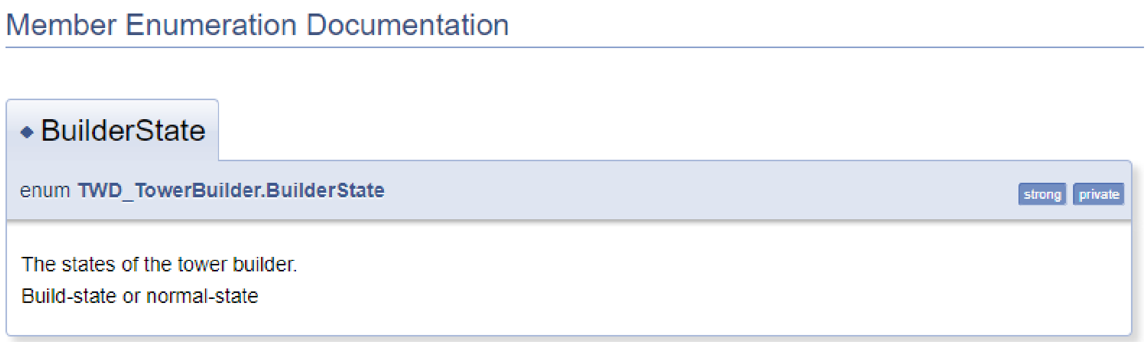
Figure 17. Function description generated by Doxygen.

Every data type such as enumerators in the game scripts were described. Data types were given descriptions using Doxygen special command @details in the code (Figure 18). The possible type values were included in the generated documentation in the description of the data type (Figure 19).



```
/// @details The states of the tower builder.
/// @n Build-state or normal-state
private enum BuilderState { build, normal }
```

Figure 18. Data type description using Doxygen special commands in code.



Member Enumeration Documentation

◆ BuilderState

enum TWD_TowerBuilder.BuilderState

The states of the tower builder.
Build-state or normal-state

Figure 19. Data type description generated by Doxygen.

4.6 Documentation for instructions

The instructions were developed with HTML. The guidelines for the instructions consisted of standard layouts, grammatical guidelines, instructions for text formatting, common terminology and usage and contents of visual aids.

An HTML-base was developed for the instruction pages. The standard structure of the HTML-base consisted of the table of contents, sections and subsections. The table of contents included links to the sections and subsections. These links provided a quick access to any section within the instruction page and enabled the users to return to the previously worked section. The header of an instruction page included a button to return to the top of the page. Sections and subsections consisted of the instructions and visuals.

A common terminology and grammatical style were agreed for the instructions. The used terms related to programming and game development had to be commonly known or clearly explained before they were used. Finnish terms and translations were preferred but if unsatisfactory, English equivalents were used. The common instructions included explanations for various terms and, in addition, introduced the English terms.

Formatting guidelines were created to standardize the use of text highlighting in the instructions. Formatting was used to distinguish and highlight the processed game elements, and game objects and scripts were written in bold typeface, and the names of 3D-models, materials and textures were given in italics.

The content of the images of code stages were agreed to be precise but brief. The images of the code stages focused on the code insertions or adaptations including a sufficient amount of surrounding code and the code lines to communicate their location.

4.7 Graphs

Various graphical figures were created for representation. Graphs were created manually in order to be customizable based on the requirements. Graphical aids can provide significant benefits in information processing because they are able to deliver meaningful and accurate concepts or interpretations while remaining simple and unambiguous (Puri 2006, 325-330). Graphical aids were created to assist the intended user-base in learning concepts and structures. Graphical representations of the structure of the modular game collection, namely class diagrams, were aimed for the developers in order to support maintenance and the potential of further development of the product.

4.7.1 Class diagrams

Class diagrams were created of each of the game projects of the modular game collection. However, it was assumed that the intended users of the modular game collection would be unable to understand the structural representations of the class diagrams.

UML class diagrams represented the data of the classes and the relationships between the classes. A representation of a single class consisted of the name, variables and the functions of the class in three boxes respectively. The variables and the functions had their protection level displayed by plus (+) and minus symbols (-) in front of the class members. The variable types and function return types were displayed after the colon symbols (:). (Doran & Casanova 2017, 13.) The relationships between the classes were briefly described (Figure 20).

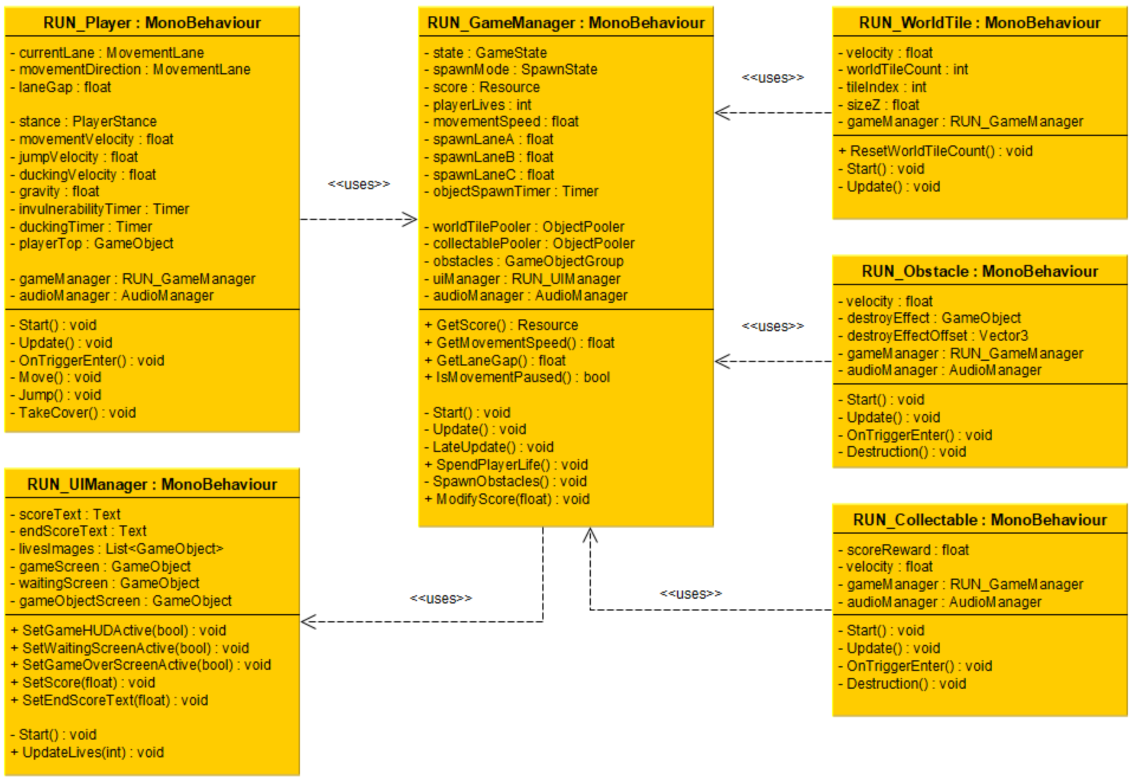

RUN_Player : MonoBehaviour
- currentLane : MovementLane
- movementDirection : MovementLane
- laneGap : float
- stance : PlayerStance
- movementVelocity : float
- jumpVelocity : float
- duckingVelocity : float
- gravity : float
- invulnerabilityTimer : Timer
- duckingTimer : Timer
- playerTop : GameObject
- gameManager : RUN_GameManager
- audioManager : AudioManager
- Start() : void
- Update() : void
- OnTriggerEnter() : void
- Move() : void
- Jump() : void
- TakeCover() : void
<<uses>>
RUN_GameManager : MonoBehaviour
- state : GameState
- spawnMode : SpawnState
- score : Resource
- playerLives : int
- movementSpeed : float
- spawnLaneA : float
- spawnLaneB : float
- spawnLaneC : float
- objectSpawnTimer : Timer
- worldTilePooler : ObjectPooler
- collectablePooler : ObjectPooler
- obstacles : GameObjectGroup
- uiManager : RUN_UIManager
- audioManager : AudioManager
+ GetScore() : Resource
+ GetMovementSpeed() : float
+ GetLaneGap() : float
+ IsMovementPaused() : bool
- Start() : void
- Update() : void
- LateUpdate() : void
+ SpendPlayerLife() : void
- SpawnObstacles() : void
+ ModifyScore(float) : void
<<uses>>
RUN_WorldTile : MonoBehaviour
- velocity : float
- worldTileCount : int
- tileIndex : int
- sizeZ : float
- gameManager : RUN_GameManager
+ ResetWorldTileCount() : void
- Start() : void
- Update() : void
<<uses>>
RUN_Obstacle : MonoBehaviour
- velocity : float
- destroyEffect : GameObject
- destroyEffectOffset : Vector3
- gameManager : RUN_GameManager
- audioManager : AudioManager
- Start() : void
- Update() : void
- OnTriggerEnter() : void
- Destruction() : void
RUN_UIManager : MonoBehaviour
- scoreText : Text
- endScoreText : Text
- livesImages : List<GameObject>
- gameScreen : GameObject
- waitingScreen : GameObject
- gameObjectScreen : GameObject
+ SetGameHUDActive(bool) : void
+ SetWaitingScreenActive(bool) : void
+ SetGameOverScreenActive(bool) : void
+ SetScore(float) : void
+ SetEndScoreText(float) : void
- Start() : void
+ UpdateLives(int) : void
<<uses>>
<<uses>>
RUN_Collectable : MonoBehaviour
- scoreReward : float
- velocity : float
- gameManager : RUN_GameManager
- audioManager : AudioManager
- Start() : void
- Update() : void
- OnTriggerEnter() : void
- Destruction() : void

Figure 20. Class diagram created for modular game collection.

4.7.2 Diagrams

Diagrams were created in order to represent complex concepts or functions. The diagrams were targeted for the intended users to provide visual aid in learning a particular concept or function. At first, flow diagrams were created but they were later discarded due to their complexity and advanced terminology. The original flow diagrams required the understanding of concepts such as index notation and lists but failed to provide a visual representation of the index or list usage.

Custom diagrams were created in order to simplify a concept or function. The custom diagrams were designed to represent and visualize the main steps of the function. Colors were used to distinguish between the different elements in the function, and notes were included to describe the steps.

4.7.3 Game object diagrams

Diagrams were created in order to visually represent the structure and components of game objects. Game object diagrams represented the game objects and their child objects and components created in Unity Editor. In the diagrams, game objects, components and custom scripts were distinctly color coded. Components and scripts were listed below the game object, and the parent game object and its child objects were connected with arrows (Figure 21). If the game object had multiple similar child objects, an additional multiplier in the diagram represented the number of the child objects (Figure 22).

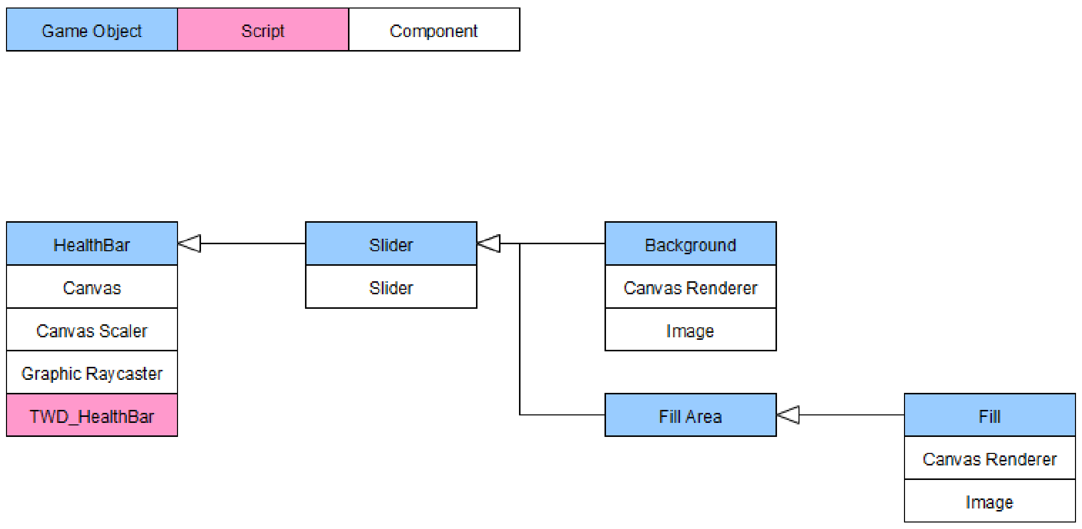


Figure 21. Example of game object diagram. Note that the parent object is the furthest to left.

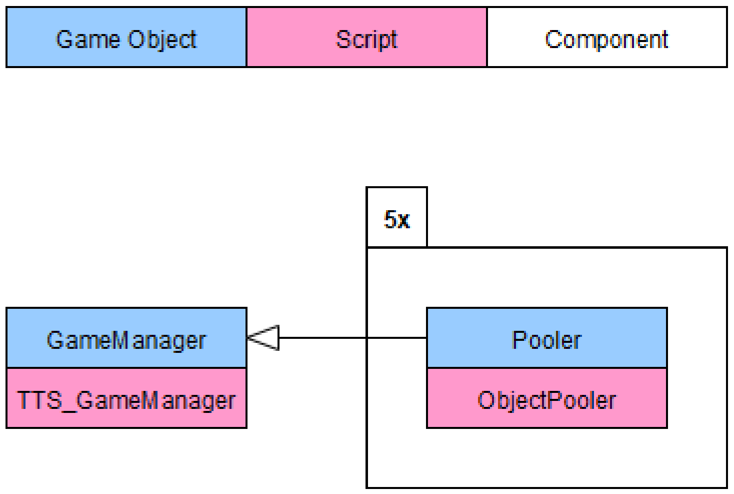


Figure 22. Game object diagram with multiple child objects.

5 FUTURE DEVELOPMENT

As of completing this thesis, the implemented documentation has not been utilized completely in the instructions of the modular game collection. While the remaining game projects and instructions are being developed, some graphs and diagrams remain unutilized. Game design documents have not yet been included in the game project instructions, which would be important, as they detail the project-dependent requirements before the reconstruction process by the intended user.

Documentation that would be unutilized by the modular game collection would be revisited. The possibilities to use this documentation in the instructions, maintenance or further development of the modular game collection would be evaluated. If the unnecessary documentation is estimated to be useful in the future, it may be preserved. Otherwise, the section of documentation may be discarded.

Reviews of the documentation are required upon the completion of the remaining game projects and their instructions. These reviews will be carried out to inspect the consistency of the documentation, and convert the documentation of the game projects up to date with the newest documentation guidelines. The contents and structures of the graphs and diagrams, terminology in the code comments, and overall compliance of the guidelines would be included in the documentation reviews. Potential shortages in the documentation would be implemented and included, making sure, no documentation material would be missing.

Finally, as the last scripts have been reviewed, the final version of the Doxygen documentation will be generated and included in the project package. Guidelines for the documentation would be written in more formal manner and compiled in one file for potential improvements or insertions for the documentation.

6 CONCLUSIONS

In general, documentation requires resources. Depending on the project, the resources used for documentation may include workforce, budget, time or other resources. Documenting the modular game collection mainly required time. Repeating and explaining the code in the comments was necessary and led to additional work required in the development. Changes in the code resulted in a necessity to edit the descriptive comments, and the maintenance of the documentation required a considerable amount of time.

In large scale projects, automated documentation may prove to be helpful as it reduces the time it takes to document the project and helps to keep the documentation more consistent. For the modular game collection, Doxygen documentation tool proved to be a significant aid in generating the documentation. The documentation generated by Doxygen was systematic and consistent. Navigation in the Doxygen documentation was convenient, and it was possible to search the classes of the game scripts. For the author, this thesis has provided a remarkable opportunity in learning and understanding the importance of documentation and its implementation.

The main purpose of the documentation process was to provide a system that could enable a beginner to use the modular game system. The main purpose remained the same throughout the development process, and only the need for different aspects of documentation changed, resulting in changes made in the documentation methods. As a result of this thesis, the documentation of the modular game collection was developed and now it provides a vast range of comprehensive descriptions of the functionality and usage of the used code. The documentation can be considered successful within a specific range: the implemented documentation functions as the basis for the instructions of the modular game collection.

The clarity and understandability of the developed code was not tested, only implemented, reviewed and modified by the developers. Usability testing would be a substantial measure to uncover the potential errors in the documentation and instructions of the modular game collection. Developing the documentation may cause blindness to defects and deficiencies among the developers as they become familiar with the systems and may lack the ability to examine the system from an outsider’s perspective. The usability test could help detect flaws in the documentation which have passed unnoticed by the developers, but testing is not an absolute measure for identifying defects. Testing the program does not guarantee the absence of defects in the code (Kasurinen 2013, 19-20) and this also applies to the documentation. After the usability testing, the documentation of the modular game collection must not be classified as flawless.

The adequacy of the documentation cannot be affirmed without testing. It is difficult to estimate the sufficiency of the documentation, this is even more difficult with the modular game collection since the intended users were assumed to have very limited understanding of game programming and the concepts related to it. In conclusion, usability testing is required to determine the needs for improving the documentation.

REFERENCES

Doran, J. P. & Casanova, M. 2017. Game development patterns and best practices. Ebook. Packt Publishing Ltd. Available at: https://ebookcentral.proquest.com/lib/xamk-ebooks/home.action [Accessed 3 April 2019].

Doxygen. 2019. Doxygen Manual. WWW document. Updated 24 April 2019. Available at: http://www.doxygen.nl/manual/index.html [Accessed 1 May 2019].

Goodliffe, P. 2006. Code craft: The practice of writing excellent code. Ebook. No Starch Press, Incorporated. Available at: https://ebookcentral.proquest.com/lib/xamk-ebooks/home.action [Accessed 25 January 2019].

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Microsoft Documentation. 2015. C# programming guide. WWW document. Updated 20 July 2015. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/ [Accessed 5 May 2019].

Puri, U. 2006. Teaching techniques. Ebook. Pragun Publications. Available at: https://ebookcentral.proquest.com/lib/xamk-ebooks/home.action [Accessed 23 April 2019].

Robinson, J. A. 2004. Software design for engineers and scientists. Ebook. Elsevier Science & Technology. Available at: https://ebookcentral.proquest.com/lib/xamk-ebooks/home.action [Accessed 20 April 2019].

Unity Documentation. 2019a. Unity user manual. WWW document. Updated 15 April 2019. Available at: https://docs.unity3d.com/Manual/index.html [Accessed 1 May 2019].

Unity Documentation. 2019b. Unity scripting API. WWW document. Updated 15 April 2019. Available at: https://docs.unity3d.com/ScriptReference/index.html [Accessed 1 May 2019].

LIST OF FIGURES

Figure 1. Requirements for the documentation.
Figure 2. Sections of the documentation and documentation tools and methods.
Figure 3. An example of custom error message in Unity Editor.
Figure 4. Tooltip in Visual Studio.
Figure 5. Implementation of Tooltip-message in code.
Figure 6. Tooltip in Unity Editor.
Figure 7. Class description using Doxygen special commands @brief and @details in code.
Figure 8. Class description generated by Doxygen.
Figure 9. Detailed class description generated by Doxygen.
Figure 10. Variable description using Doxygen special command @details in code.
Figure 11. Variable descriptions generated by Doxygen.
Figure 12. Doxygen special command @param in code.
Figure 13. Function description with parameters generated by Doxygen.
Figure 14. Doxygen special command @return in code.
Figure 15. Function description with returnee generated by Doxygen.
Figure 16. Doxygen special commands @cond and @endcond in code.
Figure 17. Function description generated by Doxygen.
Figure 18. Data type description using Doxygen special commands in code.
Figure 19. Data type description generated by Doxygen.
Figure 20. Class diagram created for modular game collection.
Figure 21. Example of game object diagram.
Figure 22. Game object diagram with multiple child objects.