Imatge que conté text, captura de pantalla, Font, disseny

Descripció generada automàticament

**Grau en Disseny i Producció de Videojocs**

**Curs acadèmic 24-25**

**Development of an Abstracted**

**Network and Synchronization System**

**for Online Games in Unity**

**TfG**

**Francisco Secchi Tarrés**

**Tutor: Ricard Perea Ros**



**Abstract**

In the abstract, the goal and the most significant aspects of the project have to be shown. It is useful to illustrate the process of development of the project proposal and the results obtained in the project. All these contents should be written in about five-seven lines, in order to fit the three abstracts in only one page. The abstract is very useful because it shows a general idea about the project at a glance. We suggest to write the abstract after the documentation of the project.

**Resum**

En el resum cal indicar l’objectiu i els punts més significatius del treball. És molt útil mostrar el procés de desenvolupament de la proposta de projecte i quins han estat els resultats obtinguts. Tot això cal fer-ho en aproximadament cinc-set línies, de forma que els tres resums càpiguen en una única pàgina. El resum és molt útil perquè mostra una idea general del treball en una lectura ràpida. S’aconsella fer el resum un cop s’hagi redactat tota la documentació.

**Resumen**

En el resumen se debe indicar el objetivo y los puntos más significativos del proyecto. Es muy útil mostrar el proceso de desarrollo y cuáles han sido los resultados obtenidos. Todo esto debe hacerse en aproximadamente cinco-siete líneas, de forma que los tres resúmenes quepan en una única página. El resumen es muy útil porque muestra una idea general del trabajo realizado en una lectura rápida. Se aconseja hacer el resumen una vez se haya redactado toda la documentación.

Contents

[List of figures 6](#_Toc191920604)

[List of tables 6](#_Toc191920605)

[1 Introduction 1](#_Toc191920606)

[2 Theoretical framework 1](#_Toc191920607)

[2.1 General concepts 1](#_Toc191920608)

[2.1.1 Object-Oriented Programming 1](#_Toc191920609)

[2.1.2 Composition 1](#_Toc191920610)

[2.2 Design Patterns 1](#_Toc191920611)

[2.2.1 Singleton 1](#_Toc191920612)

[2.2.2 Command 1](#_Toc191920613)

[2.2.3 State 1](#_Toc191920614)

[2.2.4 Observer 1](#_Toc191920615)

[2.2.5 Factory 1](#_Toc191920616)

[2.2.6 Prototype 1](#_Toc191920617)

[2.2.7 Strategy 2](#_Toc191920618)

[2.2.8 Mediator 2](#_Toc191920619)

[2.3 Unity Overview 2](#_Toc191920620)

[2.3.1 Scripting 2](#_Toc191920621)

[2.3.2 Input System? 2](#_Toc191920622)

[2.3.3 Built-in Patterns 2](#_Toc191920623)

[2.3.4 Packages and libraries 2](#_Toc191920624)

[2.4 Networking 2](#_Toc191920625)

[2.4.1 Concepts 2](#_Toc191920626)

[2.4.2 Models 2](#_Toc191920627)

[2.4.3 Synchronization techniques 2](#_Toc191920628)

[2.4.4 Networking Libraries 2](#_Toc191920629)

[3 Objectives 2](#_Toc191920630)

[3.1 Principal Objectives 2](#_Toc191920631)

[3.2 Secondary Objectives 2](#_Toc191920632)

[4 Methodological design and timeline 3](#_Toc191920633)

[4.1 Methodology 3](#_Toc191920634)

[4.1.1 MoSCoW 3](#_Toc191920635)

[4.1.2 Version Control 3](#_Toc191920636)

[4.2 Planning 3](#_Toc191920637)

[4.2.1 Definition of required features 3](#_Toc191920638)

[4.2.2 List of features using MoSCoW 3](#_Toc191920639)

[5 Development 3](#_Toc191920640)

[5.1 Project Setup 3](#_Toc191920641)

[5.2 Core Modules 3](#_Toc191920642)

[5.2.1 Networking Manager 3](#_Toc191920643)

[5.2.2 State Management 3](#_Toc191920644)

[5.2.3 Rollback System 4](#_Toc191920645)

[5.2.4 Input Buffering/Prediction? 4](#_Toc191920646)

[5.2.5 Synchronization 4](#_Toc191920647)

[5.2.6 Serialization 4](#_Toc191920648)

[5.2.7 Utilities and Debugging 4](#_Toc191920649)

[5.3 Gameplay Framework 5](#_Toc191920650)

[5.3.1 Game State 5](#_Toc191920651)

[5.3.2 Object State 5](#_Toc191920652)

[5.3.3 Object Controller 5](#_Toc191920653)

[5.3.4 HUD 5](#_Toc191920654)

[5.4 UML Design 5](#_Toc191920655)

[5.5 Implementation 5](#_Toc191920656)

[6 Conclusions 6](#_Toc191920657)

[7 Bibliografia/Referències 6](#_Toc191920658)

# List of figures

Fig. 3.1. Interlineat i espaiat del text, en el processador de text MS-Word10

Fig. 3.2. Exemple de figura, on cal remarcar les unitats en els eixos.13

# List of tables

Taula 3.1. Resum del tipus de lletra a emprar en la memòria.11

# Introduction

Unity is one of the bigger, most used, game engines

# Theoretical framework

* 1. General concepts
     1. Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming environment that organizes software around data, or objects, rather than functions and logic. Objects encapsulate both state (attributes) and behaviour (methods), which promotes modularity, reusability, and maintainability in complex systems (Meyer, 1997; Weisfeld, 2019).

Software structured in this manner uses classes to represent real-world entities, making the code more intuitive for users and developers. Each object acts as a self-contained module, facilitating clearer architecture and easier debugging, testing, and scaling of software applications (Weisfeld, 2019).

OOP is built on four foundational principles:

* **Encapsulation**: This restricts direct access to certain components of an object. Privacy is controlled through access modifiers (e.g., public, private), ensuring that internal implementations are hidden while exposing only what is necessary via defined interfaces (Meyer, 1997; Weisfeld, 2019).
* **Abstraction**: Abstraction allows programmers to manage complexity by exposing only the essential features of an object and hiding the implementation details. This principle simplifies interactions between objects and supports high-level problem-solving (Meyer, 1997).
* **Inheritance**: Inheritance provides logical hierarchy. A subclass inherits attributes and methods from its superclass, allowing developers to extend and specialize behaviour efficiently while reusing code (Meyer, 1997; Weisfeld, 2019).
* **Polymorphism**: allows methods to take on multiple forms while keeping the same name, through method overriding. It lets objects be treated as instances of their parent class while still exhibiting distinct behaviour, enhancing flexibility and scalability (Weisfeld, 2019).

OOP is especially powerful in component-based systems, where objects are composed of modular components that define distinct behaviours. This model encourages composition over inheritance, further increasing the adaptability of software architecture (Weisfeld, 2019).

* + 1. C# Language

C# is a statically typed, object-oriented programming language developed by Microsoft as part of the .NET platform. It offers robust features for building scalable and maintainable applications, including support for reflection and attribute annotation.

Reflection in C#

Reflection is a powerful feature in C# that allows programs to inspect and interact with their own structure, metadata, and code at runtime. Using reflection, developers can access information about assemblies, modules, and types, and can dynamically create instances, invoke methods, or access fields and properties. This capability is particularly useful for scenarios such as plugin architectures, serialization, and testing frameworks.

Attributes and Custom Attributes

Attributes in C# provide a way to associate declarative information, or metadata, with code elements such as assemblies, types, methods, and properties. These attributes can be queried at runtime using reflection, enabling programs to modify behavior based on the presence or values of attributes.

Developers can define custom attributes by creating classes that inherit from the “System.Attribute” base class. Custom attributes allow for the addition of user-defined metadata to code elements, which can then be retrieved and utilized at runtime.

* + 1. Composition
    2. Basic network (client, server, etc)
    3. UML
  1. Design Patterns
     1. Singleton
     2. Command

Guardar i enviar els inputs com a Commands per executar a tots els clients o per fer rollback?

* + 1. State
    2. Observer
    3. Strategy

Per custmització del framework (diferents estrategies de sincronització etc)

* 1. Unity Overview
     1. Scripting

In Unity, scripts are typically written in C# and are used to create custom behaviors for GameObjects. These scripts are attached to GameObjects as components and can control various aspects of the game's functionality, such as responding to user input, managing game state, and handling physics interactions (Unity Technologies, n.d.-a).

Unity's scripting environment supports both object-oriented and data-oriented programming paradigms. Developers can use traditional object-oriented design by creating classes that inherit from Unity's built-in types, such as MonoBehaviour, or leverage Unity's Data-Oriented Technology Stack (DOTS) for high-performance, multithreaded code (Unity Technologies, n.d.-a).

The scripting workflow in Unity involves creating C# scripts within the Unity Editor, editing them using an integrated development environment (IDE) like Visual Studio, and attaching them to GameObjects to define their behavior. Unity's scripting system provides a robust and flexible framework for implementing game logic and interactive features (Holik Studios, n.d.; Unity Technologies, n.d.-a).

* + 1. Composition
    2. Packages and libraries

A Unity package follows a convention structure of files and folders

A Unity Package has only one required file that lets Unity and users know what the package is and how it fits into the project. That is the manifest, a required file so Unity can identify and recognize the project as a valid package to be included using Unity’s Package Manager as a local or external Git package.

* + 1. Test Runner

The Unity Test Runner is an integrated tool within the Unity Editor that facilitates the creation and execution of automated tests for game development projects. It supports both Edit Mode and Play Mode tests, enabling developers to validate code functionality within the editor and during gameplay.

* **Edit Mode Tests**: These tests run within the Unity Editor without entering Play Mode. They are ideal for testing pure logic and editor scripts. Edit Mode tests are executed in a single frame, and Unity APIs that depend on runtime behaviors are not available (Unity Technologies, n.d.-a).
* **Play Mode Tests**: These tests run in Play Mode, allowing access to runtime features such as physics, animation, and scene management. Play Mode tests can be written as coroutines using the [UnityTest] attribute, enabling frame-by-frame execution and interaction with behaviors (Unity Technologies, n.d.-b).

Integration with NUnit

Unity Test Runner is built upon the NUnit framework, a widely-used unit testing library for .NET languages. This integration allows developers to utilize familiar testing constructs such as [Test], [SetUp] and [TearDown] attributes, as well as assertions like “Assert.AreEqual” and “Assert.IsTrue” (Unity Technologies, n.d.-a).

Creating and Running Tests

To create a test, developers can add a new C# script with test methods marked by NUnit attributes.

Tests can be executed via the Test Runner window (Window > General > Test Runner), where developers can run all tests, selected tests, or rerun failed tests. The Test Runner provides detailed feedback on test outcomes, including execution time and error messages (Unity Technologies, n.d.-a).

* 1. Networking
     1. Concepts

Transport Protocols

Dedicated & Listen Server

Ownership

Replication

* + 1. Models

Authoritative and Non-Authoritative

Peer-to-Peer

Hybrid Peer-to-Peer

Others

* + 1. Synchronization techniques

Rollback

State Replication

Event-based

Delay-based

* + 1. Networking Libraries

Enet-CSharp vs LiteNetLib for UDP

Telepathy vs System.Net.Sockets for TCP

# Objectives

* 1. Principal Objectives
* Develop a standardized networking system for game developers in Unity. This framework does not try to be a solution for all multiplayer games and genres, so it won’t implement all networking models. It focuses on a Host-Client solution with a State Replication system and Rollback capabilities for synchronization.
* Code a well-structured, de-coupled, and fully scalable piece of software using the best programming techniques and patterns. Letting developers implement their own network and synchronization strategies to suit their needs.
* Create simple scenes to test and visually show all the different functionalities of the framework.
  1. Secondary Objectives
* Write a clear and useful framework documentation.
* Test the framework by adapting a single player game, implementing a multiplayer mode.

# Methodological design and timeline

* 1. Methodology
     1. MoSCoW

The MoSCoW method is a structured prioritization technique employed in project management, business analysis, and software development to determine the relative importance of requirements. The acronym stands for:

* **Must have**: Critical features essential for the project functionality.
* **Should have**: Important but not vital requirements; their absence may be inconvenient but not catastrophic.
* **Could have**: Desirable requirements that can enhance the project’s value but are not necessary.
* **Won’t have**: Features classified as least critical, to be excluded from the current scope.

Developed by Dai Clegg in 1994 for rapid application development, the MoSCoW method was later integrated into the Dynamic Systems Development Method (DSDM) framework to facilitate timeboxing and iterative delivery (Clegg, 1994). It encourages stakeholder collaboration to achieve consensus on priorities, ensuring that the most critical requirements are addressed first (Agile Business Consortium, n.d.).

This is a method aids in managing expectations and ensuring the most critical requirements are addressed first, especially in time-constrained projects.

* + 1. Waterfall

Waterfall is a linear and sequential software development methodology that divides the project into fixed stages: requirements, design, implementation, testing, deployment, and maintenance. Each phase must be completed before the next can begin, emphasizing comprehensive documentation and up-front planning. This model is most effective when requirements are stable and well-understood from the outset, though it is often critiqued for its inflexibility in adapting to change (Royce, 1970).

The decision to use this methodology comes from the dependency between modules. Although each module has its independence and is de-coupled in a certain way with the rest, the layer structure of the framework requires a module to be finished (atleast its Must features) to begin the development of the next one. It is possible to start implementing the Network layer, but without a Transport module fully functional it won’t really be covered and tested. Combined with robust MoSCoW defined features, it is the most suited development methodology.

* + 1. Version Control

Git is a distributed version control system that enables users to track changes in files and coordinate work on those files among multiple people. Each user maintains a complete copy of the repository, including its full history, which facilitates collaboration, supports non-linear workflows, and enhances data redundancy. Git is widely adopted in both software development and data science for managing code, data, and documentation, thereby supporting reproducible research practices. (Vuorre, M., & Curley, J. P., 2018)

Git is the main and only version control of the project. It has been selected because it has fast upload and download file states, it’s very easy to work with and well-known. A branch will be created for each module, enhancing the workflow of the project, reducing redundancy and improving revisions.

* 1. Planning
     1. Definition of required modules

This section abstracts the first idea of the project’s structure, constructed by six different modules. Each responsible for implementing a part of the framework, they will be independent at a certain level from each other while having a clear and modular dependency between them.

Transport Abstraction

This module will be the responsible for all network-related operations. It either sets up the local machine as a server or as a client. It will also manage LAN broadcast and discovery for local net area game sessions.

It aims to be completely de-coupled, meaning that it defines an Interface that can be implemented and injected in the framework as it just handles the low level networking. This is very useful for letting users implement their own transport protocols and logic.

Serialization

Another completely de-coupled module, aims to set up the serialization strategy. An implemented methodology will be already set up for the framework, but they can switch to their own Serializer by implementing a simple two-method interface.

This module will be responsible for serializing and de-serializing messages sent through the network.

Message Handler

A centralized messaging system that handles the distribution of messages across the framework. Although the name might be misleading, it will only handle who executes received messages in the framework, based on message type. It will not be sending data.

This module will have no connection with the Transport and Serializer layers, but it will be tightly coupled with the Network and Synchronization layer. Will be open for users to use, at their own care, to define messages and message handlers.

Network Management

The network layer will act as a gate to the Transport layer for the whole framework. It is also acting as a window façade for the users to set up hosts or clients.

Here is where users will define the Transport and Serializer strategies; start and connect clients and hosts; track, spawn and destroy networked objects; get transport information and set up other configuration properties.

Game Synchronization

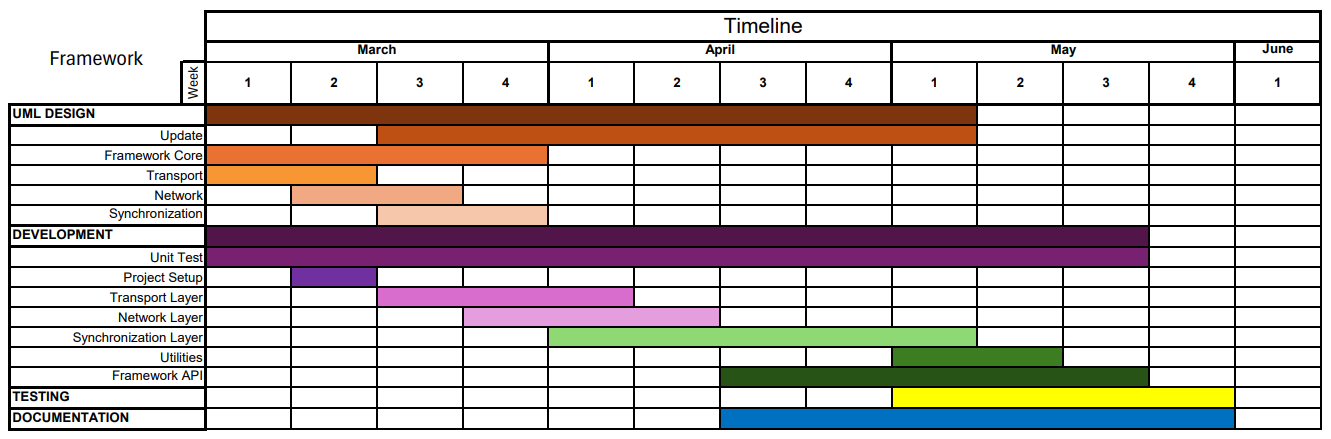
Here lays the core of the framework usability. A system for state changes detection on networked objects, synchronizing them between all the players. It’ll be responsible for defining networked objects and its owner, sending and receiving game states and Remote Procedure Calls to manage the synchronization of the game.

Debugging and utilities

A set of functions and windows that will help the user to follow the execution flow of the framework. This module will try to give easy access to information about the server, the clients and their connection to developers in runtime. As well as the current objects and messages sent and received.

* + 1. List of features using MoSCoW

|  |  |  |  |
| --- | --- | --- | --- |
| **Priority** | **Module** | **Functionality** | **Description** |
|  | Transport | Setup Transport | Defines the network transport layer and sets up the default protocol. |
|  | Transport | Start | Starts a server/host |
|  | Transport | Connect | Initiates a connection to a server/host |
|  | Transport | Disconnect | Disconnects from the server/host |
|  | Transport | Send | Sends already serialized data to the specified client. |
|  | Transport | Receive | Receives data from the network |
|  | Network | Start Server | Starts a server as host |
|  | Network | Start Client | Starts a client |
|  | Network | Join Host | Connects a client to the host. |
|  | Network | Client Disconnection | Manages client disconnection. |
|  | Network | Host Disconnection | Manages host disconnection. |
|  | Network | Listen | Listens for incoming client connections. |
|  | Network | Send To All | Sends a message to all clients. |
|  | Network | Send To Client | Sends a message to a specified client. |
|  | Network | Send To Host | Sends a message to the host. |
|  | Network | Disconnect | Disconnects the client. |
|  | Sync | Object Identity | Assigns object identity via a unique ID |
|  | Sync | Object Ownership | Assigns an object to a client via a client ID |
|  | Sync | OnStart Network | Called when the object is first spawned in the network. |
|  | Sync | OnDestroy Network | Called when the object is removed from the network. |
|  | Sync | Synchronize variable | Synchronizes a variable across the network |
|  | Sync | Owner Transfership | Handles ownership changes. |
|  | Sync | Call RPC | Sends an RPC to a specific player |
|  | Rollback | Set a State | Switches between GameStates |
|  | Rollback | Clone State | Clones a State to send to all clients |
|  | Rollback | Save State | Saves a State of a specific frame for rollback |
|  | Rollback | Load State | Restores to a past State during rollback |
|  | Rollback | Handle desync | Detects and resolves desynchronizations |
|  | Rollback | Store Input | Saves Input history for rollback |
|  | Rollback | Rollback | Restores the game to a specific frame |
|  | Rollback | Reapply Inputs | Runs stored inputs after rollback |
|  | Rollback | Validate Rollback | Checks if the state is consistent on all clients |
|  | Sync | Network Transform | Synchronize Unity Transform |
|  | Sync | Network Rigidbody | Synchronizes physics of an object |
|  | Message | Message Manager | Handles message execution. |
|  | Serializer | Serialize Data | Convert data for network transportation |
|  | Serializer | Deserialize Data | Converts network data back into objects. |
|  | Transport | Custom Protocols | Allow developers to implement custom transport protocols |
|  | Network | Migrate Host | Transfers host role to another client if the host disconnects. |
|  | Network | Handle Reconnection | Allow disconnected clients to rejoin a session. |
|  | Sync | Custom GameStates | Enables developers to define custom game states. |
|  | Network | Create Lobby | Matchmaking lobby |
|  | Network | Join Lobby | Join an existing lobby |
|  | Sync | Scene Load | Synchronizes scene load. |
|  | Sync | Compress Data | Serialize only changed data, not full states |
|  | Sync | Adaptive Interpolation | Adjust smoothing based on latency |
|  | Sync | Latency Compensation | Adjust all clients for laggy players |
|  | Network | Peer-to-Peer solution | Habilitate Peer-to-Peer |
|  | Network | NAT Punching | Enables connection outside local area nets. |
|  | Sync | Input Delay Buffer | Align frame execution across all players |
|  | Debug | Logging utilities | Log rollback, frame count, GameState, etc. |
|  | Debug | Network Stats | Provide network statistics and log issues |
|  | Network | Dedicated Server Model | Only host-client model |
|  | Network | Cloud matchmaking system | Matchmaking is handled outside locally lobbies |
|  | Network | Managed Game Hosting Services | Cloud-based hosting |
|  | Sync | Built-in Chat voice | Developers can integrate it using third-party solutions |

* 1. Timeline

# Development

* 1. Project Setup

Folders and Assembly Definitions

Following Unity’s package structure, the solution has three main folders with an assembly definition setting code boundaries within the package. We first split the project by Runtime, Editor and Tests logic.

* **Runtime/**: here lays all the framework’s code, the whole functionality is structured inside this directory and is the core of the package. Every module here is defined inside SimpleNet’s namespace.
* **Editor/**: this module holds all the code executed outside Unity’s runtime environment. Here’s where debugging Unity windows and menus are defined. It is completely dependent of Runtime’s folder and it is organized inside its own namespace named “SimpleNet.Editor”.
* **Tests/**: this assembly implements all Unit Tests inside SimpleNet.Tests namespace, importing all tested Runtime’s code. This will be referencing Runtime’s and Unity’s TestRunner assembly and must have a module for each Runtime module, since it aims to test cover all the framework’s code.
* **ThirdParty/**: this is a special folder where all external libraries used by SimpleNet are located. Frameworks such as LiteNetLib for UDP transportation and MessagePack for serialization are installed in this folder and used by both, Runtime and Tests modules. Assemblies reference the “dll” files of the external libraries.

Markdown and required files

* **package.json**: SimpleNet’s manifest will start as version 0.1.0 under the name of “com.franysan.simplenet”. Sets a minimum version of Unity 2023.1 and its only dependency will be Unity’s test framework.
* **Markdowns**: the README file contains a short explanation of the package followed with usability points. LICENSE states a standard MIT license and copyright. CHANGELOG will have a list of features implemented by the end of this project, since this will be counted as the first and release commit and CHANGELOG will be updated from then and onwards with the package’s changes and new features.
  1. Core Modules
     1. Networking Manager

Hosts (servers) will receive and accept connections depending on the maximum number of clients permitted, defined by the user of the framework. It will receive and send data from and to clients, either specific or broadcast. Triggers an event whenever it has received data or successfully connected/disconnected a client.

Clients in the other hand can connect to a valid IP address and Port

- transporte de paquetes según protocolo

- conexión P2P híbrida.

- P2P y pasar a host-client por condición

- selección de host, si falla pasar a uno nuevo

* + 1. State Management

Crear y guardar GameStates. Proveer de clases abstractas/interficies para guardar estados de jugadores y objetos.

* + 1. Rollback System

Guardar GameStates, rollback si se desincroniza, resimulación después del rollback.

* + 1. Input Buffering/Prediction?

Cuando se pierde un paquete(frame) con inputs, predecir o replicar el ultimo input.

Guardar un buffer con los ultimos inputs?

Cuando llega el paquete 🡪 rollback

* + 1. Synchronization

Sistema de estrategias de sincronizacion.

* + 1. Serialization

Encargado de data converter para transporte.

* + 1. Utilities and Debugging

Debug varios: problemas de conexión, rollbacks...Hacer logs de frames o mostrar estadisticas de net?

* 1. UML Design
  2. Implementation

# Conclusions

# Bibliografia/Referències

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON*, 26(8), 1–9.

Vuorre, M., & Curley, J. P. (2018). Curating research assets: A tutorial on the Git version control system. *Advances in Methods and Practices in Psychological Science*, 1(2), 219–236. <https://doi.org/10.1177/2515245918754826>

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice Hall.

Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5th ed.). Addison-Wesley Professional.

Unity Technologies. (n.d.-a). Introduction to scripting. Unity Manual. <https://docs.unity3d.com/Manual/intro-to-scripting.html>

Unity Technologies. (n.d.-b). Scripting. Unity Manual. <https://docs.unity3d.com/Manual/scripting.html>

Holik Studios. (n.d.). Introduction to Scripting in Unity (C#). <https://holikstudios.com/gamedev/unity-engine/scripting/>

Unity Technologies. (n.d.-a). Unity Test Runner. Unity Manual. Retrieved from <https://docs.unity3d.com/Manual/testing-editortestsrunner.html>

Unity Technologies. (n.d.-b). Writing and executing tests in Unity Test Runner. <https://docs.unity3d.com/Manual/PlaymodeTestFramework.html>