

Game Systems Assignment 1

22.03.2022

**─**

Ryan Bassil (Lunar)

# 

# Changelog

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Changes** |
| 1.0.0 | 22/03/2022 | Initial Setup |
| 2.0.0 | 22/03/2022 | Filled in information on Introduction and Research topics |
| 2.0.1 | 25/03/2022 | Info from class added to document in System Architecture |
| 2.1.0 | 26/03/2022 | Filled in information about SOLID principles |
| 3.1.0 | 27/03/2022 | Completed programming methodologies and everything in Task 2 except for 5 UML diagrams and the pseudocode. |
| 3.2.0 | 28/03/2022 | Changed system architecture diagram to activity diagram, completed everything in Task 2 except sequence and communication diagrams, and pseudocode. |
| 3.3.0 | 29/03/2022 | Completed pseudocode, added assignment to GitHub, completed System Architecture diagram after renaming previous one to ActivityDiagram. |

# Contents

[Changelog 2](#_Toc98447406)

[Contents 3](#_Toc98447407)

[Introduction 5](#_Toc98447408)

[Rationale 5](#_Toc98447409)

[Background 5](#_Toc98447410)

[Terminology 5](#_Toc98447411)

[Non-Goals 5](#_Toc98447412)

[Proposed Design 6](#_Toc98447413)

[Software and Hardware Requirements 6](#_Toc98447414)

[Research 6](#_Toc98447415)

[Procedural Programming 11](#_Toc98447416)

[Object-Oriented Programming 12](#_Toc98447417)

[Functional Programming 12](#_Toc98447418)

[Logical Programming. 12](#_Toc98447419)

[System Architecture 14](#_Toc98447420)

[Architecture Diagram 14](#_Toc98447421)

[Data types 14](#_Toc98447422)

[Data Model 14](#_Toc98447423)

[Interface/API Definitions 15](#_Toc98447424)

[Risks 16](#_Toc98447425)

[Alternatives 17](#_Toc98447426)

[Pseudocode 17](#_Toc98447427)

[System Pseudocode 17](#_Toc98447428)

[UML Diagrams 21](#_Toc98447429)

[UML Class Diagrams 21](#_Toc98447430)

[UML Activity Diagrams 22](#_Toc98447431)

[UML Sequence Diagram 22](#_Toc98447432)

[UML Communication Diagrams 24](#_Toc98447433)

[UML State Diagrams 25](#_Toc98447434)

[Sign Off 25](#_Toc98447435)

[Testing 26](#_Toc98447436)

[Errors and Buggs 26](#_Toc98447437)

[Evaluation 26](#_Toc98447438)

[Reflection 26](#_Toc98447439)

# 

# Introduction

## Rationale

In this Technical Design Document I propose to demonstrate knowledge of object-oriented design in writing scripts for user interfaces, and iterate on broader OOP concepts and methodologies.

There are many concepts and technical terms relating and included in OOP and design and most researched and covered in this document will be used to produce a Test Project written in C#, a common object-oriented programming language.

## Background

The object-oriented programming language, C#, has had many different versions in the past forming its basis roughly off C. Different versions slightly affects the ways in which some code concepts are written. In this project I will be using version 9.0.

Like C#, Unity has had many iterations in the past and continues to update to this day, but for this project I will be using Unity version 2020.3.27f1

In this project I will be designing and producing a working project within Unity from scratch, starting with this document.

## Terminology

**C#** - An object-oriented programming language

**Unity** – a game engine used in the production of games and software in the IT industry

**TDD** – a Technical Design Document (this document) is a document that aids in the critical analysis of a problem that is presented to a development team and the proposed solution, while also communicating priority and effort of tasks, and impact with various stakeholders (in this case, Odin Evans).

**SDLC** – A Software Development Life Cycle is essentially a method or process followed by the development team (or company) for a software project.

**Keyboard** – clickedy clackedy typing instrument

**Mouse** – something you’ll use until its filthy and then replace it with an RGB one without realising it’s not wireless

**OS** – Operating System of a computer or electronic device

**Processor** – essentially a computer’s brain for calculating and input/outputs of the software running on the computer

## Extra-Goals

* Record how many total attempts have been made to complete the game
* Record the number of successful attempts made and list them in a “hall of victory” screen
* Add timer and display the time of successful runs in the “hall of victory”
* Get a job

## Proposed Design

Given that this project is a solo-developer project and is relatively small, the proposed process to follow for its creation will be the Agile development methodology.

Following written pseudocode to draw up UML diagrams on the flow of the game, I will construct simple UI elements within a scene in Unity, and then test the results. Following successful result, I will continue by implementing the gameplay itself in incremental steps, again according to pseudocode as a guide, and testing the results upon completion.

Check ins, or meetings, with the Lead Programmer will be regular (after every milestone).

## Software and Hardware Requirements

**OS**: Windows 7 or Greater

**Processor**: 2 Ghz Intel Pentium 4 or equivalent

**Memory**: 1 GB RAM

**Graphics**: Integrated Graphics Chip (minimum Nvidia GTX 700 series or AMD Radeon R7 200 series)

**DirectX**: Version 9.0

**Storage**: 50 MB available space

# Research

Software Development Platforms

Software Development Platforms for games is fundamental software that allows the development of game software or applications to be built with it, hence it is a metaphorical *platform* for a game or application.

Some SD platforms for games include, but are not limited to:

1. **Unity Game Engine** – Unity was first released in 2005 and its main development focus has been tools to build 3D games, though through the years 2D development has been increasingly supported, and Virtual Reality games have also popularly used Unity as its platform (such as *Beat Saber*). It’s runtime coding language and scripting API are different, being C++ and C# respectively.
2. **Unreal Game Engine** – Unreal was first released in 1998 and shortly from its release was a major driving factor of massively multiplayer online games (*MMOG*), and in mid-2000’s focused more development on support for mobile games. Later, in 2014, Unreal developed a very intuitive and unique tool within its game engine called *Blueprints*, which allowed people to develop games with very minimal coding.
3. **Godot Game Engine** – A game engine that is entirely free and *open-source* with increased support for the Linux OS comparatively, is a game engine released initially in 2014 and continues to be developed. Its focus is both 3D and 2D game development, though encourages use of its own scripting language called *GDScript*, which is akin to JavaScript or Python, though uniquely developers can use multiple languages to write games in this engine, such as C, C#, C++, GDScript, and VisualScript which is handled by its GDNative technology. It is one of few SDP’s that allow the use of multiple languages in one project or game.

Software Development Methodologies

A Software Development Life Cycle, or Software Development Methodology, is essentially a method or process followed by the development team (or company) for a software project.

[www.tutorialspoint.com](http://www.tutorialspoint.com), a common help source for software developers around the globe, says this about SDLCs:  
“*[A] Software Development Life Cycle* *is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.*”

**Examples:**

1. **Waterfall Model** (Traditional methodology)

A model of development that endeavours to follow the “get it right the first time” philosophy, whereby the developer gathers as many details and requirements about the system needing to be developed as possible, and then follows through with developing the software until it meets all those requirements before being delivered back to the client.

1. **Agile Model**

This methodology highly focuses on customer involvement in order to bring a solution to the problem needing to be solved through rapid and multiple smaller iterations of a new model in order to reassure appropriate adaptation and satisfaction of customer requirements.

1. **Test Driven Development**

A methodology that focuses on building and structuring the code around testing the code written. For every function or item of code written there is a test that is written after it. This methodology is similar to the Agile development but adds tests much more frequently through the cycle of the project and focuses on robustness of the code.

1. **DevOps Model**

A methodology that is much larger than most other methodologies as its goal is to bring multiple ‘departments’ of a project into harmonisation and collaboration (where other methodologies have them separate and limited communication between them) to provide more rapid and cleaner codebase and software management. These departments are developers (who write the code) and the operators who deploy and manage the software.

DevOps has a lot of emphasis on testing and delivery automation to streamline the development/production process of the project.

1. **Prototype Model**

This methodology works around getting basic requirements and interface in a state that can be shown to a stakeholder(s). This prototype might not be designed internally as it will be later, but it provides a means to discuss the current results and projected progress of the project, and details and more intricate topics will be finalised at a later time.

This allows for designers to get ample feedback on what the client requires before embarking on developing the software in the method the designers have agreed.

The software is built, tested, and then reworked when needed until an acceptable prototype is achieved. Regular meetings are essential to keep the project on time and avoid costly delays in the prototyping approach. Missing functionality can be identified, which helps to reduce the risk of failure as Prototyping is also considered as a risk reduction activity in SDLC.

1. **RAD Model**

This methodology is essentially a planning-minimalist methodology that focuses on breaking the project into modules and completing working prototypes as fast as possible, allowing feedback received from clients in a much shorter timeframe, and then repeating the process until the clients are satisfied.

It is an iterative process that begins with gathering requirements from clients through workshops or focus groups, and during the development process the prototypes are reused, allowing for rapid delivery to the stakeholder(s).

1. **V-Shape Model (Verification and Validation Model)**

The V-Model methodology is an intense process that requires clear requirements, clear scope, and minimal to no iteration. Every phase of development requires a series of rigorous testing phases and only once these tests are passed for that phase can the next phase begin.

The V-model’s strengths lie in the robustness and clarity, and weaknesses lie in flexibility and time consumption.

Like the Waterfall model on which it is based, it is good for simple projects with a clearly defined scope and minimal changes to requirements.

Strict linearity means that it is not ideal for large projects or anything that requires a lot of iteration.

1. **Iterative Model**

This methodology focuses on gathering only part of the system requirements, then designs and develops that part of the system right to implementation and is reviewed when the next part of the system requirements or steps are gathered/provided. The existing component(s) developed are then modified to account for the new requirements as per the next section of the system. This is a sort of evolution of the software through the project.

This methodology relies on having clear requirements for each step, and sometimes knowledge of the full system required; though multiple modules or components can be developed in parallel.

It is not suited for smaller projects due to its evolving and modularly iterative nature where components are developed in parallel and improved upon. It is ideal for projects where time is crucial and new technology is being introduced to a project.

1. **Big Bang Model**

This methodology is a highly flexible, high-risk model that follows no formal process of development. The beginning of this process starts with the stakeholder(s) providing resources/money to the development team and describing possibly very vague requirements and letting the development team interpret those requirements as they wish in hopes that it will align with the stakeholder’s interests.

Since there is no formal process, this model exercises plentiful creativity and minimal planning, but doesn’t support high complexity, large or ongoing projects as it leaves too much allowance for devastating risks.

1. **Spiral Model**

This methodology is a combination of sorts of the waterfall, prototype, and agile methodologies. The beginning of the process starts similarly to waterfall whereas many requirements of the overall system are gathered, then follows the proof of concept stage at which some feedback is received by the stakeholder(s), and then the risk management phase before leading to the design stage of the general architecture of the software. Following this construction and development begins to produce different builds which start at 1 and new builds are only created once the previous has been reviewed by the client and more design and modification of that build has been taken into the design of the following build.  
This spiral can be continuous if resources or stakeholder(s) allow, continually building and reiterating on the previous build until the stakeholder(s) is happy.

Overall, those methodologies that provide iteration as a baseline for the model are always superior over others due to the everchanging nature (and rapidly developing) IT industry and technology. Game development needs to be flexible. Out of the 10, this leaves us with *Agile, Iterative, Prototype, RAD, Spiral, DevOps, and Test-Driven* models.

Next thing that is good for game development is clear scope and planning, and from these 7 models, *Agile, Prototype, Test-Driven, DevOps, Spiral, and Iterative* models are suited for planning purposes.

Following that, time is usually crucial in most game development situations, however, in relation to this TDD and over-arching project in which I am a sole developer, we can conclude that I would require a model suited to a small project. This leaves models *Agile, Prototype,* and *Test-Driven*, since Spiral, Iterative, and DevOps are rather large and complex methodologies and for the most part require a lot more time provided in the project scope.

The next requirement for a small project game project would be determined on either project scope or requirements; *Agile* provides more robust and thought-through development compared to *Prototype*, but *Prototype* is faster and needs fewer requirements in early stages of development. Meanwhile, *Test-Driven* development is the most robust and automated but requires the most time and knowledge compared to the others.

For this instance, I would recommend the *Agile* model due mostly to the amount of time provided by a brief glance at the scope of the project.

Integrated Development Environment

It is a software application that provides a platform and tools for software developers to build applications and software. Visual Studio is the IDE being used for this project and is common in game dev.; JetBrains Rider, Monodevelop (by Unity before being discontinued), Eclipse, and PyCharm are all IDE examples.

S.O.L.I.D Design Principles

/ What does S.O.L.I.D stand for in reference to Object Oriented Design Principles? Explain all 5 principles within S.O.L.I.D.

<https://www.gamedeveloper.com/programming/solid-principles-for-game-developers>

**S – Single Responsibility Principle**

*“There should never be more than one reason for a class to change.”*

Separate your classes by their responsibilities/functions. Don’t let one class hold multiple responsibilities that cause coupling between scripts or classes and therefore bugs and breaks if you change either one or all of them.

**O – Open Closed Principle**

*“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”*

Every class, module, or function should be able to have their behaviours changed without their original source code to be changed/altered in order for these new behaviours to be incorporated.

**L – Liskov Substitution Principle**

*“Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.”*

Basically any classes’ behaviour must be evident in their children classes, and anything using/calling the parent class of that child must be able to use its attributes and functions polymorphically through its children (derived) classes without caring of the type of derived class. Every derived class of the parent must follow the designed function of the parent class. I.E: **Food** > Pizza. Pizza follows the design of the parent because it is indeed a food and will have its basic attributes if anything wants to call the food, because Pizza has an *ingredients* attribute for instance. In other words, Pizza can substitute for Food any time if needed.

However, here: **HealthyFood** > Pizza, the design of the parent is not being met by the child (derived) class because while HealthyFood and Pizza may both have *ingredients*, Pizza from a behaviour perspective is not a healthy food by any means. So, Pizza class would not be able to substitute for HealthyFood class except for very few circumstances, and thus violates the LSP law.

**I – Interface Segregation Principle**

*“Clients should not be forced to depend upon interfaces that they do not use.”*

When a class implements an interface it should not have to implement functions of that interface that it is not going to use. Using this principle, the interface in question would be separated into two or more interfaces (depending on the uses of the functions the original contained) and the class would implement the one in which it would make use of all that containing interface’s functions.

**D – Dependency Inversion Principle**

*“High level modules should not depend upon low level modules. Both should depend upon abstractions.”* *“Abstractions should not depend upon details. Details should depend upon abstractions.”*

Having classes, methods, or containing objects similar, that instantiate or directly manipulate other classes, methods, etc. causes tight coupling between these modules. I.E: if one changes or breaks, everything breaks because it depends on all of them working together. It also makes it super difficult to test one specific thing if it implements or uses things from other modules, the test would become too complicated even for the simplest module.

Instead, you have something like an interface to do the implementation of any of the required modules, and that decouples the modules from each other and instead all of them are responsible for themselves. The module can change however it wants without causing severe ramifications in every other part of the project that relies on one instantiation of module. The interface handles those instantiations. Testing on one single module becomes very easy since it no longer relies on instantiation or implementation of other modules, as long as the interface is maintained.

Programming Methodologies

### Procedural Programming

A programming paradigm, often referred to as POP (Procedural Oriented Programming), is where the program is structured focused on a ‘top down’ approach, executed from start to finish. A task in the program is broken into variables, data structures, and subroutines, as opposed to OOP where a program is broken into ‘objects’ that contain their own data and methods and relate to each other.

For example, a person going to use an ATM is broken down into POP and OOP (respectively):

* *Withdraw, Deposit, Transfer*
* *Customer, Money, Bank*

### Object-Oriented Programming

The OOP paradigm is structured around the concept of ‘objects’ and ‘classes’, where an object is an instantiation of a class, and a class is a blueprint of sorts that contains attributes and functions of what an object might have, but not specific definitions.

The function of OOP is a way to group programming tasks, data structures, methods, etc together in a way that relates to the real world. IE: A *Person* class might describe methods like *Breathe, Eat, Walk*, and any object or class inherited from this class (or “blueprint”) will have those methods.

OOP is a non-linear programming paradigm in which code can execute in multiple “directions” of the program depending on how the objects relate to each other and external inputs. Not too dissimilar from Event-Driven Programming.

### Functional Programming

The Functional programming paradigm is one that focuses on its ability for methods to remain independent of external values and variables, in that a single method takes parameters, operates its code with those parameter values, and returns values based on its operation without ever changing the source of those parameter variables’ values. In this regard, any method (or function) could be taken out and placed in theoretically any program and operate as normal, so long as the parameters are utilised correctly.

### Logical Programming.

The Logical programming paradigm is a way of writing a computer program that enables the computer to make logical decisions based on previously executed or existing code for it to move forward in its program. It’s the difference between the programmer telling the computer that A = C and the computer using an algorithm that the programmer wrote to prove to itself that A = C. It uses logic to make this conclusion.

Open-Source Development tools

Something that is “open source” means that any person or entity in the public domain can use, modify, or distribute the material. In programming, it refers to the software that is developed, used, modified, and distributed.

Some examples for game development programs that are open source are:

* Godot Game Engine
* Visual Studio Code
* Gitlab

# System Architecture

## Architecture Diagram

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Diagrams/SystemArchitectureDiagram>

Diagram, schematic

Description automatically generated

## Data types

**Int** – a 4-byte integer data type that stores whole numbers between a limited negative and positive range.

**String** – a collection of Char data types, often used to store sentences or multiple characters in one variable.

**Char** – a single character value data type that relates to ASCII values.

**Bool** – a boolean data type, true or false.

## Data Model

Integers will be used to:

* store a player’s number of incorrect guesses
* the size of an array of characters for the word
* store and track the player’s total number of successful games
* store and track the number of total times, across all games, the player has guessed any character in the alphabet

Strings will be used to:

* store the word required for the game, pulled from an array
* store an array of strings containing all the words used within the game
* store static phrases for messages displayed to the player at certain event times (when they fail a run, or choosing an invalid character)

Characters will be used to:

* store a character in a parsed string, in particular the word in question for each run
* store an array of all the parsed characters in a string to allow access to them mathematically, using integers
* store user input
* compare user input to the array of characters that make up the word
* displayed on the letter places when guessed correctly by the user

Booleans will be used to:

* set whether the player has guessed a letter correctly or not

Methods will manipulate this data to form the rest of the structure of the game.

## Interface/API Definitions

**Components**:

* **Image** (used to place and manipulate image files within the Unity editor and extend functionality to other components that make use of the Image component)

<https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UIVisualComponents.html>

* **Text Element** (used to display, define, and manipulate text within the Unity engine/editor)

<https://docs.unity3d.com/ScriptReference/UIElements.TextElement.html>

* **Button** (used to run events on a click of the game object this component is attached to)

<https://docs.unity3d.com/2018.3/Documentation/ScriptReference/UI.Button.html>

* **Mesh Filter & Mesh Renderer** (used to render the GameObject the component is attached to and display it onscreen.)

<https://docs.unity3d.com/Manual/class-MeshRenderer.html>

* **Script**

<https://docs.unity3d.com/ScriptReference/ScriptableObject.html>

* **GameObject**: base class used to describe and define all entities in a Unity Scene)

<https://docs.unity3d.com/ScriptReference/GameObject.html>

**Libraries**: (need links and descriptions of what the library allows you to do)

* **UnityEngine**: connect to Unity, attach scripts to objects, reference Unity components, access unity’s classes and methods

<https://docs.unity3d.com/ScriptReference/index.html>

<https://docs.unity3d.com/Manual/ScriptingImportantClasses.html>

<https://answers.unity.com/questions/524862/why-does-one-need-to-use-unityengine.html>

* **UnityEngine**.**UI**: to access the UI Canvas Components and manipulate gameplay user interfaces in Unity

<https://docs.unity3d.com/Manual/com.unity.ugui.html>

* **System**: to access parsing and conversion of data types (in relation to this project), but according to Microsoft: *“Contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.”*

<https://docs.microsoft.com/en-us/dotnet/api/system?view=net-6.0>

* **System.Collections**: In relation to the project, allows us access to arrays, Microsoft: *“Contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.”*

<https://docs.microsoft.com/en-us/dotnet/api/system.collections?view=net-6.0>

* **System.Collections.Generic**: in relation to this project it allows access to stronger typed (more generic) collections and methods for those collections (ie: List<T>, which would be much more useful for Sort()-ing through multiple specific game object types, versus an ArrayList). Microsoft: *“…allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.”*

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-6.0>

## Risks

* Time to completion based on the scope of the project.
* The resulting graphical integrity of the game.
* Properly incorporating the SOLID principles, especially DIP.
* Incorporating Interfaces is new

Research to be done:

* Parsing strings
* Displaying a UI GameObject at a set position from another UI GameObject
* Hangman historical characteristics
* Lists in C#
* Creating a save file (extra)
* Creating and displaying a timer (extra)

## Alternatives

* Using Godot game engine: passed up due to unfamiliarity with the development platform and tools.
* Using the SceneManager library: it would add another layer of complexity to the code and cause unnecessary stress and time bloat to the development of the game.
* Not setting the letters of the word in relation to the underline UI elements: while it would avoid unnecessary code complications, it may create visual bugs with screen scaling and resizing if not moved in relation to the elements.

# Pseudocode

## System Pseudocode

For convenience, find the source script on my GitHub:

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Hangman/Assets/Scripts/Pseudocode.cs>

//Load Main Menu object

//MAIN MENU

//Every button has OnClick event referencing the GameManager script

/\*New Game button

\*Creates a new save file (non-goal)

\*Deactivate the Main Menu

\*Activates the Game UI object (the object the main game takes place through) \*/

/\*Load Game button (non-goal)

\*Checks if there is more than 1 save file that exists

\*If save files > 1

\*Activate prompt UI to choose a save file

\*Text box for user to enter a number relating to the save file

\*Check if valid number

\*If true, load save file and activate Game UI object

\*Else, activate error prompt UI and ask for different save number

\*Back button

\*Deactivate prompt UI for save files

\*Else if save files !null

\*Back button

\*Deactivate prompt UI for save files

\*Load first existing save file in directory

\*Else (if there are no save files)

\*Back button

\*Deactivate prompt UI for save files

\*Activate prompt UI to tell player there are no saves \*/

/\*Exit Game button

\*Update current save file if there is one

\*Check if the game is run in Unity Editor

\*If true: exit play mode

\*Else: quits the application to the Desktop \*/

//PAUSE MENU

//Stop time

/\*Resume button

\*Deactivate Pause Menu

\*Resume time \*/

/\*Main Menu button

\*Deactivate Game UI object (the game)

\*Activate Main Menu object

\*Resume time \*/

/\*Exit Game button

\*Check if the game is run in Unity Editor

\*If true, exit play mode

\*Else, quits the application to the Desktop \*/

//END MENU

/\*Retry / Replay button

\*Update current save file

\*Set total incorrect guesses to 0

\*Activates the Game UI object \*/

/\*Main Menu button

\*Deactivate Game UI object (the game)

\*Activate Main Menu object \*/

/\*Exit Game button

\*Update current save file

\*Check if the game is run in Unity Editor

\*If true: exit play mode

\*Else: quits the application to the Desktop \*/

//GAME

//Declares and assigns List<string> of words

//Sorts word list by length of word

/\*Check difficulty variable

\*Choose a word in the list at random with a certain length, dependant on variable, and store it in a variable

\*Convert chosen word to char array \*/

//Get word length, generate, and display letter placements to the player equal to the length of the word

//Display total incorrect guesses on Game UI

//While user has incorrect guesses left:

//Display text box for user input

//Text box has Enter arrow symbol faintly on right side of bar for visual direction of action

//User inputs character(s)

//Check if Enter key

/\*If enter key is true:

\*Check if input string is null

\*Is null is true:

\*Activate prompt UI to display error that the input cannot be empty

\*Wait seconds

\*Deactivate prompt \*/

//Check if input is longer than 1 character

/\*Input longer than 1 char true:

\*Compare input from input text box to final word

\*If not the same:

\* increment total incorrect guesses count

\*If the same:

\*Deactivate Game

\*Activate End Menu object

\*Activate prompt UI to tell player they won \*/

/\*Input is 1 char

\*Convert input from input text box to char

\*Loop through the char array variable storing the previously converted word

\*Compare input to each item in the char array

\*Every time input = current item in array, store array index in another array

\*Check if other index array is null (meaning no identical letters found in word)

\*If null:

\*Increment total wrong guesses variable and UI

\*Compare total wrong guesses to max guesses

\*If equal:

\*Update Hangman sprite to reflect game loss

\*Wait seconds

\*Deactivate Game

\*Activate End Menu object

\*Activate prompt UI to tell player they lost

\*Else: continue game

\*If other index array !null:

\*Update every letter placement UI using the indices stored in the array with the player's input

\*Loop through and check if every letter placement UI text is !null

\*If true:

\*Deactivate Game

\*Activate End Menu

\*Activate prompt UI to tell player they won

\*If false:

\*Continue game \*/

//Check if Esc key

/\*If Esc key is true:

\*Stop time

\*Activate Pause Menu\*/

//Otherwise, if not Enter or Esc key, convert input string to character array

/\*Loop through and check if any character is invalid character (not alphabetical or Space)

\*If true:

\*Activate prompt UI to display error that characters must be of [valid character] type

\*Wait seconds

\*Deactivate prompt

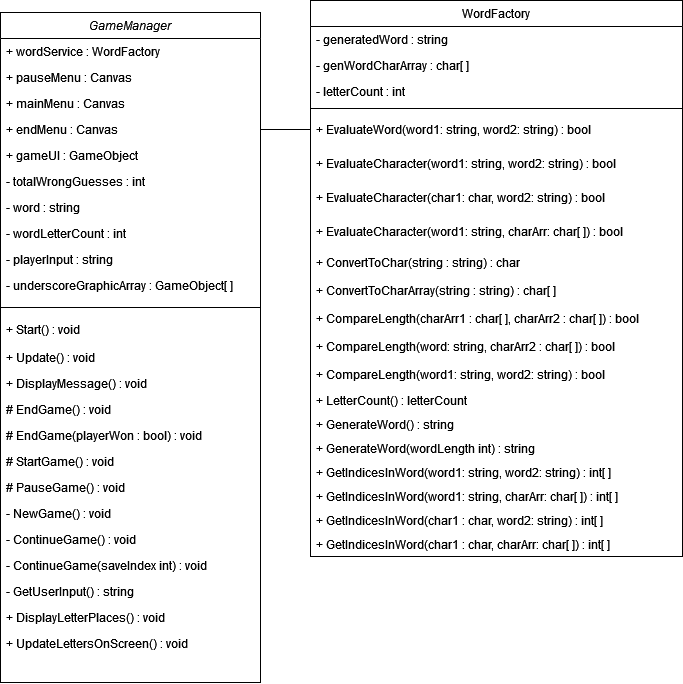
\*If false:

\*Add character to player input text box \*/

# UML Diagrams

## UML Class Diagrams

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Diagrams/InitialClassDiagram>



## UML Activity Diagrams

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Diagrams/ActivityDiagram>

## Diagram Description automatically generated

## UML Sequence Diagram

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Diagrams/SequenceDiagram>

A picture containing diagram

Description automatically generated

## 

## UML Communication Diagrams

<https://github.com/LunarBlacksmith/CertIV-GamesDevelopment_2022/blob/main/GameSystems/Assignment-1/Diagrams/CommunicationDiagram>

Diagram

Description automatically generated

## 

## UML State Diagrams

<https://github.com/LunarBlacksmith/Diagrams/blob/main/CertIV-GamesDevelopment/StateDiagram.drawio>

Diagram

Description automatically generated

# Sign Off

Name

James-Rae

Role

Lead Programmer

Signature

Jaymie-Rae Sargent

Date

05/04/2022

# Testing

## Errors and Bugs

Outline the test classes used. Add rows to table as required.

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Name** | **Description of Error** | **Screenshots of testing** | **Solution** |
| [Name of Class or Object that the error is connected to] | [Description of the error/error message] | [Add and resize relevant screen shots] | [Explain solution/fix to error] |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Evaluation

## Reflection

/Provide a self-reflection on your performance. /