Universal Dialogue System

Honours Stage Project

Jack Ballard

Abstract

Dialogue in video games developed with a separation of skills will be written by a designer, separate from the implementation of that dialogue into the game by a developer. This project develops a tool to allow a designer to write dialogue, attach data and conditions to that dialogue, embed Lua scripting in the text field, and connect dialogue nodes to create branching narratives.

This project also covers the inclusion of files to make the dialogue exported by the designer program work in Unity, which utilises the attributes and conditions written by the designer and executes Lua to interface with global variables and functions in Unity.

This project meets all its stated objectives and ultimately concludes that the tool developed is fit for purpose of writing dialogue to be used in games, but that the tool in its current iteration is simple with room to grow in the way of adding more features and improving the tools quality-of-life.

Contents

1 Introduction

Dialogue is a prevalent part of many games in various degrees, from brief comments from characters too sprawling webs of conversations and player decisions. Keeping track of this dialogue during the production process can be rather difficult for designers without the aid of a purpose created tool.

The goal of this project is to create a Universal Dialogue System, a tool for designing dialogue in games. The system will take the form of a node-based editor that designers can use to create branching dialogue trees and export them for use in other programs.

This system would be independent of any specific game engine or platform, so long as the developer of the end program can interpret the exported file for their dialogue. This project would separate the working environment between the designer, who would write the dialogue, and the developer, who would create the end program the dialogue would be used in.

The aim of this tool is to make designing dialogue simple and intuitive. This will be achieved by visualising branching connections between dialogue nodes and requiring no programming skills of the designer. The designer should be able to associate other data with their dialogue nodes, such as the file name and location of assets, or embedded scripting that can communicate with the end program.

More comprehensive objectives are covered in Section 2.4

2 Literature Review

This section will outline the types of products this project would aim to create and the tools that are currently available to facilitate their creation. These will be used to inform this projects objective, and using these evaluate the technology and techniques that will be employed to create this project.

2.1 Examples of dialogue in games

In The Secret of Monkey Island, there is an activity that players can participate in throughout the game where the player is insulted, and players must select the correct response to any given insult to gain the upper hand, as seen in figure 1. The options available to the player are learned through encountering and memorising insults, so the player does not start knowing many insults or responses, but over the course of these interactions expands on the options the player is able to choose from.

This is a rather simple interaction but demonstrates having a large collection of dialogue options that are revealed based on the game state, as well as player selected dialogue options themselves influencing the game state. In this case winning the insult sword fight or purposely losing to gain more insult responses. These interactions can grow rather complicated in other games where dialogue can branch rapidly from a single starting point, and each decision can have any amount of influence over the rest of the game state.

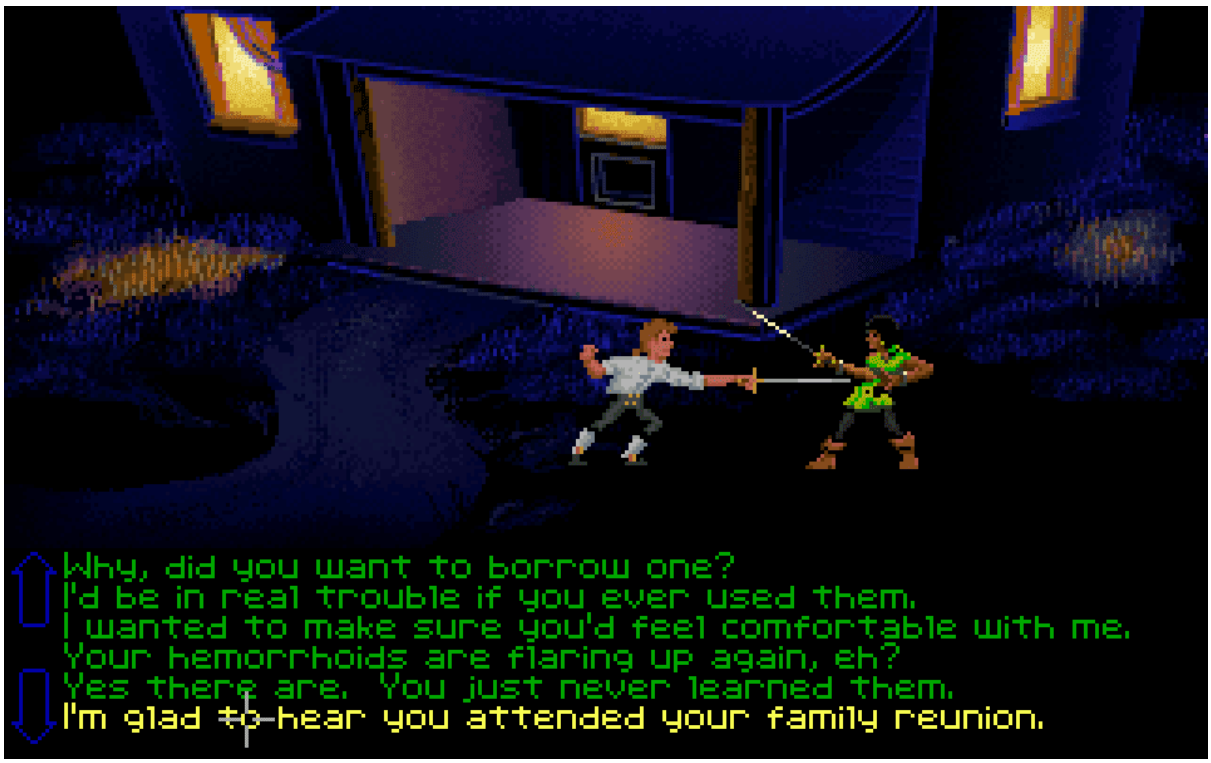


Figure 1 - The Secret of Monkey Island insult sword fight (Lucasfilm Games LLC, 1990)

2.2 Existing Products

This section outlines the kind of existing products that could be created with the Universal Dialogue System in order to determine what features it should include. This will be used to inform this project's objectives later.

2.2.1 Disco Elysium

Disco Elysium is an award-winning role-playing game that boasts to include "A revolutionary dialogue system" (*Disco Elysium*, 2015) It is made in Articy (Disco Elysium., 2014), a narrative design tool covered in section 2.2.3, and includes some features this project should cover. It includes Linear and branching dialogue, which can be presented to the user as selectable options or determined based on other conditions, such as previous user choices or dice rolls.

Some dialogue options have associated checks that determine if an option is showed to a player in the first place, or what happens after a choice has been made. In both cases, the game state informs the options presented, but in the latter case one dialogue option can lead to two different outcomes as based on a dice roll and stats. The outcome of this decision is remembered by the game and can be used to inform further checks, figure 2 shows a conversation which includes a non-repeatable dialogue option and check.

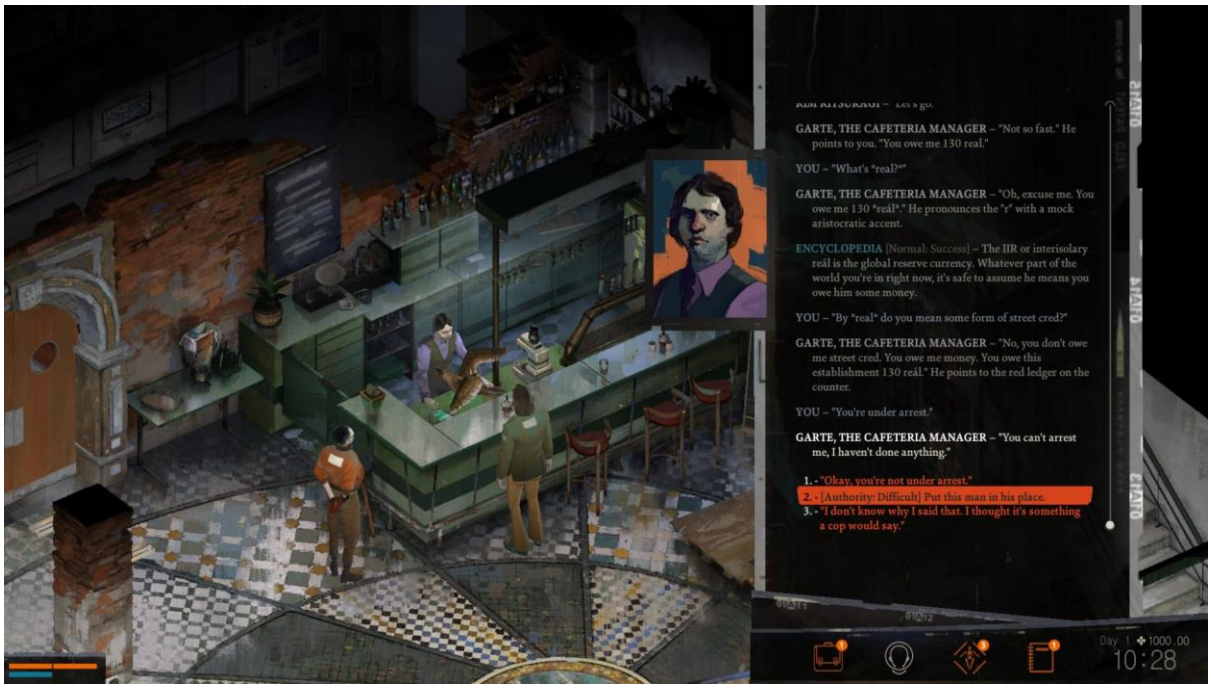


Figure 2 - A conversation in Disco Elysium (ZA/UM, 2019)

For the ease of the user, previously selected options become greyed out but are still selectable, so that the user can easily keep track of what dialogue options they have already seen but have the option to review them anyway. Some dialogue prompts

animations, scene changes, camera moves, music ques, dice rolls, and other situational events. Additionally, the game is fully voice acted and has language options.

Based on the features of Disco Elysium, this project would need to include nodes that can receive input connections from other nodes, store text to be displayed, attributes and events. Attributes, in this context in Disco Elysium, would store who's talking and checks to see if the players stats are high enough to display any given bit of dialogue, whereas events would be single shared event name to be used across multiple nodes.

Within each node would need to be the ability to connect to any other number of other nodes, and have each connection be associated with a condition so that out connections can be presented based on the game state. Disco Elysium does this by only presenting some dialogue options based on previous decisions.

2.2.2 Z.A.T.O. // I Love the World and Everything In It

Z.A.T.O. is a linear visual novel (*Z.A.T.O. // I Love the World and Everything In It on Steam*, 2025) made in Ren'Py, an open source and popular visual novel engine (*The Ren'Py Visual Novel Engine*, 2004). Each character in Z.A.T.O. is represented with a collection of sprites that makes up their expression, which is positioned and animated on screen as appropriate, as shown in figure 3. The player character also has a collection of sprites, but they are displayed in the dialogue box as opposed to the main stage.

Figure 3 - Dialogue in Z.A.T.O. // I Love the World and Everything In It (Ferry // Nopanamaman, 2025)

The text is revealed over time and plays an audio que when text is revealed, which Ren'Py allows to be based on which character is speaking, and the contents of the text

box can be changed to include variables such as the player's name, particular collected items, etc.

As Z.A.T.O. is linear it has no need of branching paths or many of the features this project would seek to include, making a developer work with multiple elements in their nodes that they aren't going to use would be redundant on their part, and it would be impossible for the editor to be tailored to each type of game. Therefore, the base node should be as light and free from clutter as possible, containing only its own ID, text, and the output connection which would have its own ID.

Based on the features of Z.A.T.O. this project should include a global variable system so dialogue can contain custom text and be referenced anywhere in the editor independent of any node. Events could also be removed in leu of placing the agency on the developer to create an event attribute if appropriate for their project.

2.2.3 Undertale

Undertale is an RPG with an "emphasis on humour, character dialogue, and player choice" (*UNDERTALE !?*, 2026). It wasn't created using a dialogue system and was designed to be highly customisable by the developer. For instance, Undertale can change the formatting in its text box to match a character or have the individual characters in the text appear from different positions on screen before flying into position, such formatting can be seen in figure 4.

Figure 4 - Dialogue in Undertale (Toby Fox, 2015)

Some dialogue editors, like Twine, have text formatting options built in (*Story Formats - Twine Cookbook*, 2009)**.** Twines handles it's formatting this way:

"''Hello'',(text-rotate-x:32)+(text-rotate-y:18)[and again], welcome to the (text-colour:red)[Apature Science] computer aided(after:2s)[enrichment centre]."

Twine can customise the text formatting and content simultaneously as it functions as an editor and front end, and this is the case with Undertale. The text written for Undertale was made around these effects and it wouldn't make sense to make the dialogue for Undertale using a dialogue editor, as the formatting of the text is an integral part of the dialogue.

Ultimately, the end program will be able to interpret the text content provided from this solution in any way it likes, so a developer would be able to handle their own formatting and simply write their formatting in the text field if that would be appropriate for their project.

Some other dialogue effects however could benefit from some finer control over the program, such as if the developer could write executable code directly in the editor, so that changes to this code and the triggers for it are not split between the end program and the dialogue editor.

2.2.4 Detroit: Become Human

Detroit: Become Human is a story rich cinematic game with "one of the most intricately branching narratives ever created" (*Detroit: Become Human | Official Site | Quantic Dream*, 2025). It wasn't written in software designed to create branching narratives, instead a branching script was written by David Cage and then the motion capture was filmed (Good, 2017). Without a branching scripting tool, the writing process "quickly becomes, honestly, a nightmare", and inflexible to changes later.

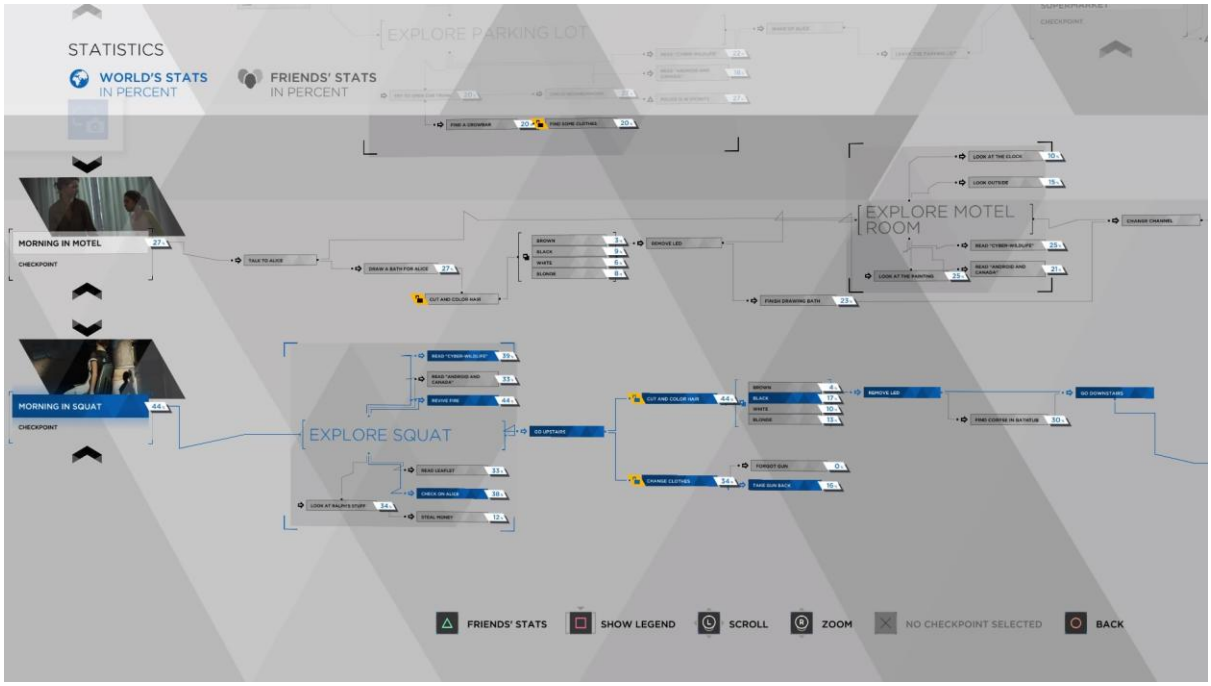


Figure 5 - A choice flowchart in Detroit: Become Human (Quantic Dream, 2018)

A node based external tool would help in the writing process, would allow elements of the game state to be reflected in the script, and could even be what selects camera angles and plays animations, but by working in an external program the designer would be blind to the outcome until it is ran. Detroit is an instance of a game where the dialogue would not be written using an external tool and the designers would work in the game itself. There are, however, some design features of note in how they relate to an external tool.

The choices selected by the player are presented to them as a map after each section, as seen in figure 5, but not every decision affects the outcome of that section, so each choice is not reflected here. Some options are only available based on elements of the game state, such as a character's opinion of the player, and are changed by dialogue options that otherwise have no bearing on the branch the player will select.

The options presented to the player never directly inform them of what the text will specifically say, instead it provides the general direction of that piece of dialogue, such as "aggressive, calm, pressure, etc". This could be an attribute of a node that describes the text it contains, or an attribute of the connection that describes the node it is connecting to. Some of these choices are also time limited, and one of the available options will be designated as the default option for if the timer runs out.

Not every choice that a player makes is a piece of dialogue, some are quick time events, and others are choices made in combat, but both are reflected in the node map presented to the player when it is a branching choice, although that choice does not contain dialogue.

2.2.5 The Henry Stickman Collection

The Henry Stickman Collection is a "Newgrounds choose-your-own-path classic" (*The Henry Stickmin Collection on Steam*, 2020) that focuses on diverting paths. Similar to Detroit: Become Human, it has a map that shows the player the choices they've made, but unlike Detroit it shows every choice and gives the player the option to jump to any node, as in the Henry Stickman Collection "failing is more fun than succeeding" and is focused on showing the player all the choices they could have made.

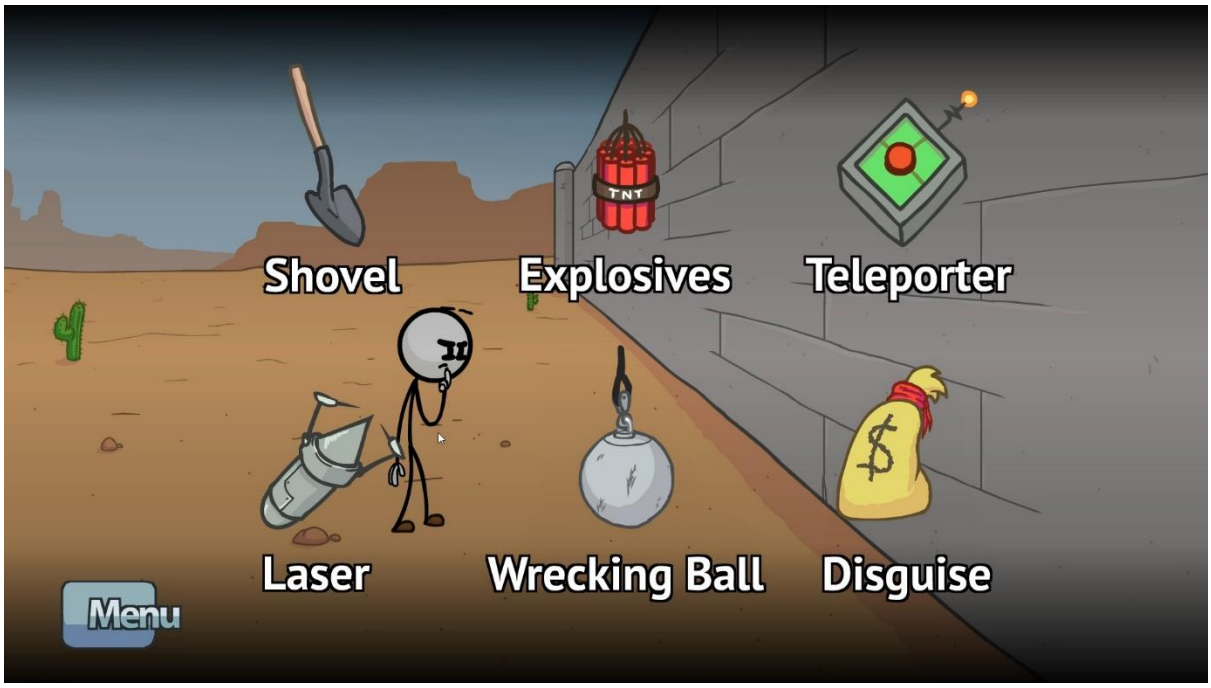


Figure 6 - Choice options in the Henry Stickman Collection (PuffballsUnited, 2020)

The Henry Stickman Collection is of focus in this document because it demonstrates the benefit of attributes per connection, as opposed to each node having its own attribute. In the Henry Stickman Collection, each option presented to the player is accompanied by it's own reveal sound, a custom sprite, a hover animation, and hover sound, as seen in figure 6. Each of these could each be attributed to the selectable option from the root node, instead of as an attribute of each of their respective end nodes. The latter is a valid option, but it may make more sense for a designer for each of these attributes to be associated with the connection.

Some choices in the Henry Stickman Collection are time limited and display a countdown for the player to select an option, and has another option not displayed to the end user for if the timer runs out without an option being selected.

2.2.6 Summary

Based on these existing solutions, the Universal Dialogue System set out by this project should be able to create linear and branching dialogue trees. Each piece of dialogue should include supplemental data to inform the end program of how to handle this text,

and the surrounding context of its delivery, such as with sprites or animations. Each piece of dialogue should have the capability to change based on the current state of the program.

Each decision should have the same capacity for supplemental data and dynamic changeability as each piece of text as set out above. Additionally, these decisions should have in built conditional checks which can be informed by the state of the end program.

Each choice should have the capacity to link to multiple other pieces of dialogue, each with their own conditional checks independent of the conditional checks for that choice. Under this model, the condition for showing a choice is independent of the checks determining where that choice would lead.

Table 1 - Existing products features table

Features	**Disco Elysium**	**Z.A.T.O**	**Undertale**	**Detroit: Become Human**	**The Henry Stickman Collection**
Linear Dialogue	Y	Y	Y	Y	Y
Branching Dialogue	Y	N	Y	Y	Y
Conditional Dialogue	Y	N	Y	Y	Y
Stat-Based Checks	Y	N	Y	Y	N
Variable Text	N	N	Y	N	N
Dialogue Formatting	N	Y	Y	N	N
Voice Acting	Y	N	N	Y	Y
Dialogue Triggers Events	Y	N	N	Y	N
Node-Based Structure	Y	N	N	Y	Y
Connection-Based Attributes	Y	N	N	Y	Y
Time-Limited Choices	N	N	N	Y	Y
Flowchart Choice Map	N	N	N	Y	Y

2.3 Existing Solutions

This section covers existing solutions that would be used to create the products outlined above and will detail the ways these solutions could be used to create them. This will be used to inform this project's objectives later.

2.3.1 Twine

Twine is an open-source text editing tool for telling interactive, nonlinear stories. (*Twine / An open-source tool for telling interactive, nonlinear stories*, 2009) It can be used by downloading a desktop application or in the web app, shown in figure 7. Twine allows users to create branching text-based stories by using a visual GUI, however, it struggles at anything that falls outside strictly text, such as interactions that aren't clicking links

or heavy use of multimedia, which while included in twines feature set, is difficult to use.

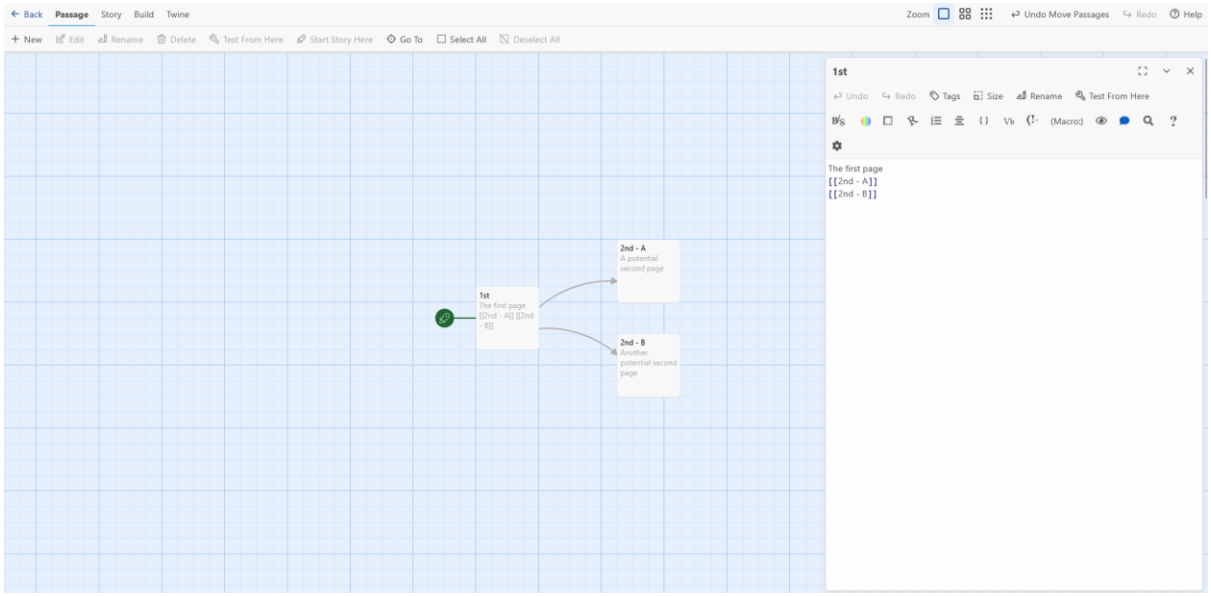


*Figure 7 - Twine web editor (*Twine / An open-source tool for telling interactive, nonlinear stories*, 2009)*

Twine is very focused on text-based narratives and can tailor its tools as such. It comes included with text formatting effects as there is no distinction between the editor and the end program. Twine is designed to not only design dialogue for games but play the games in question. Twine can be published to a HTML file, which allows it to run in a browser and be easily shared. It is also this HTML file that is used to import a version into the editor.

Twine also features a collection of runtime engines called story formats which allow a set of macros and JavaScript functionality (*Story Formats - Twine Cookbook*, 2009; Chapel, 2026). Each one uses a different syntax but uses Harlowe by default. Using these story formats, it's possible to embed the use of variables and logic into the text field of a page using the syntax of the story format.

2.3.2 Inklewriter

Inklewriter is a free tool that allows a user to write branches as they appear, then link them to other nodes later. It is easier to use than Ink, a scripting language for writing interactive narrative for text based and graphical games, but comparatively less powerful. Inklewriter stories can however be converted to Ink (*Inklewriter*, 2019).

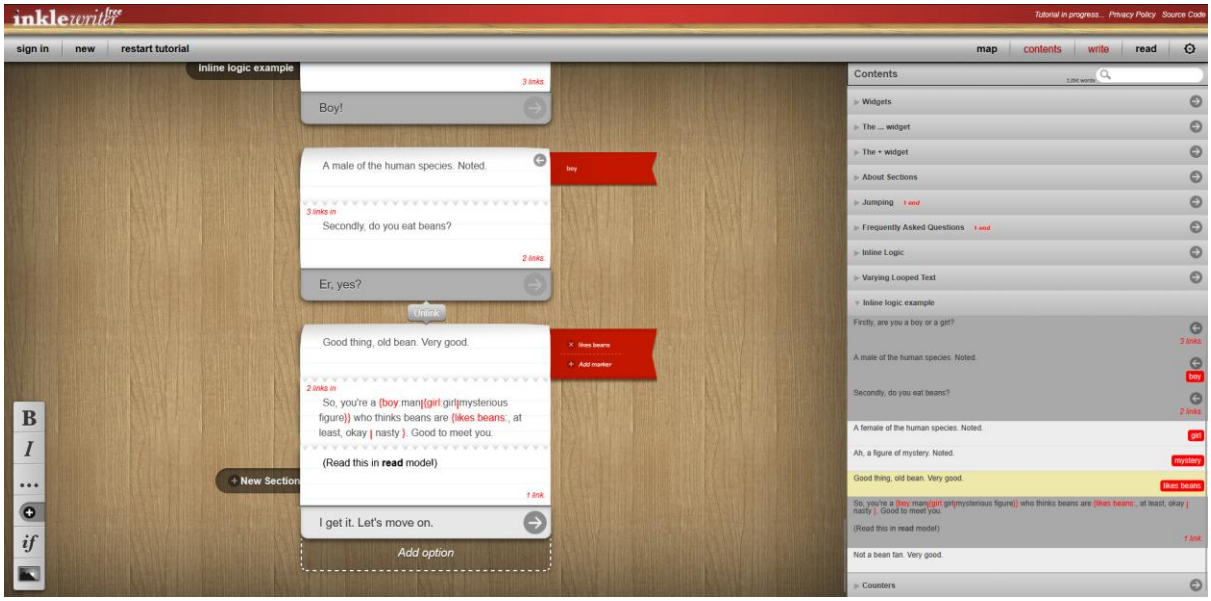


*Figure 8 - Inkwriter dialogue UI showing variables (*Inklewriter*, 2019)*

Inkwriter is split up into three different editing UI's, the main UI allows the user to write additional branches from the currently available node, shown in figure 8. The second is a contents page, which displays the text of each node and its number of outgoing links. Each node is separated by section, and highlighting a node also highlights other nodes that form a path that leads to that node, though critically this only selects one of potentially several viable paths.

The last screen is a map, shown in figure 9, which operates in a similar way to what this project is aiming to achieve. Each node in this section is split up into compartmentalised sections, which is a boon for readability by only showing the section selected, but removes the ability to join connections between sections. Selecting a node in this screen, similar to the contents screen, highlights the nodes that form a valid path to the selected node.

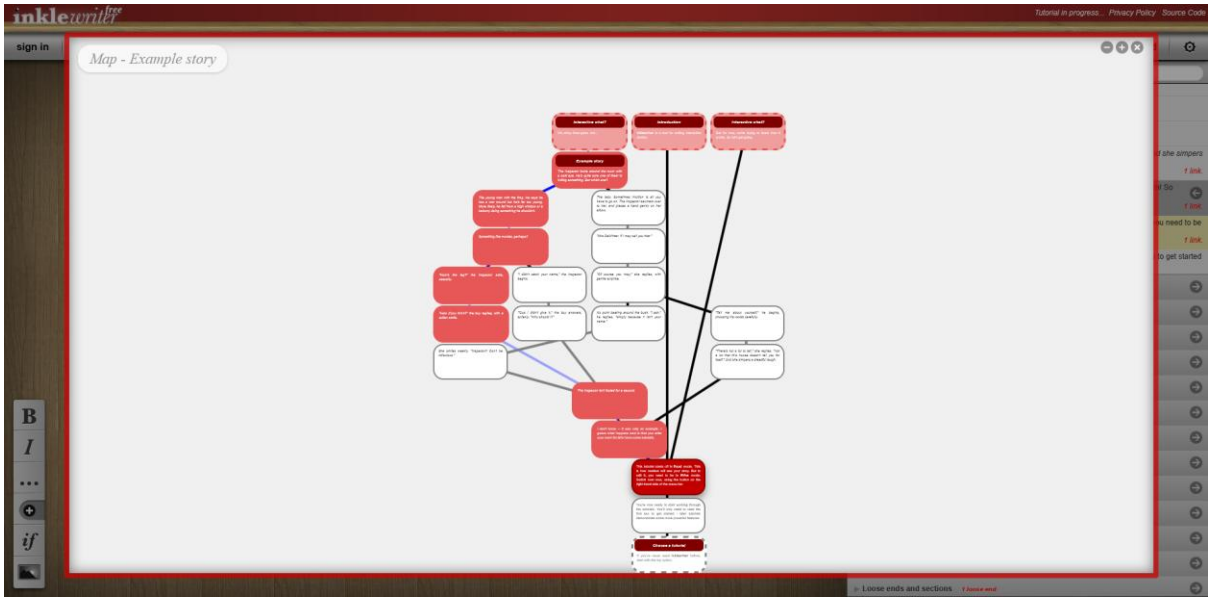


*Figure 9 - Inkwriter branching dialogue map UI (*Inklewriter*, 2019)*

Inklewriter makes use of a marker-based system that allows the program to set the state of markers when the user gets to certain paths and can use these markers as conditions to determine what options to present to the user. There are also counters and loops which can be defined by the user, but only using the logic set out by the program, the user cannot define any of their own logic and this software does not include embedded scripting.

2.3.3 Articy

Articy is a narrative design tool for interactive projects, it is a graphical node editor, as seen in figure 10, designed to work with external programs, such as Unity, Unreal, or with the Articy API, in addition to being exportable. Its nodes have conditions built in, which corresponds to a global list of all variables in a separate menu.

Each text box can also have entities that are based on templates, each with its own local store of attribute data as defined by the user, as shown in figure 11. These Objects inherit from their template, so addition attributes can be added to them. These entities can be used across many different text boxes, and the attributes they contain can be used as textbox connection conditions. (*Features of articy:draft X – Narrative Design Tool for interactive projects*, 2014)

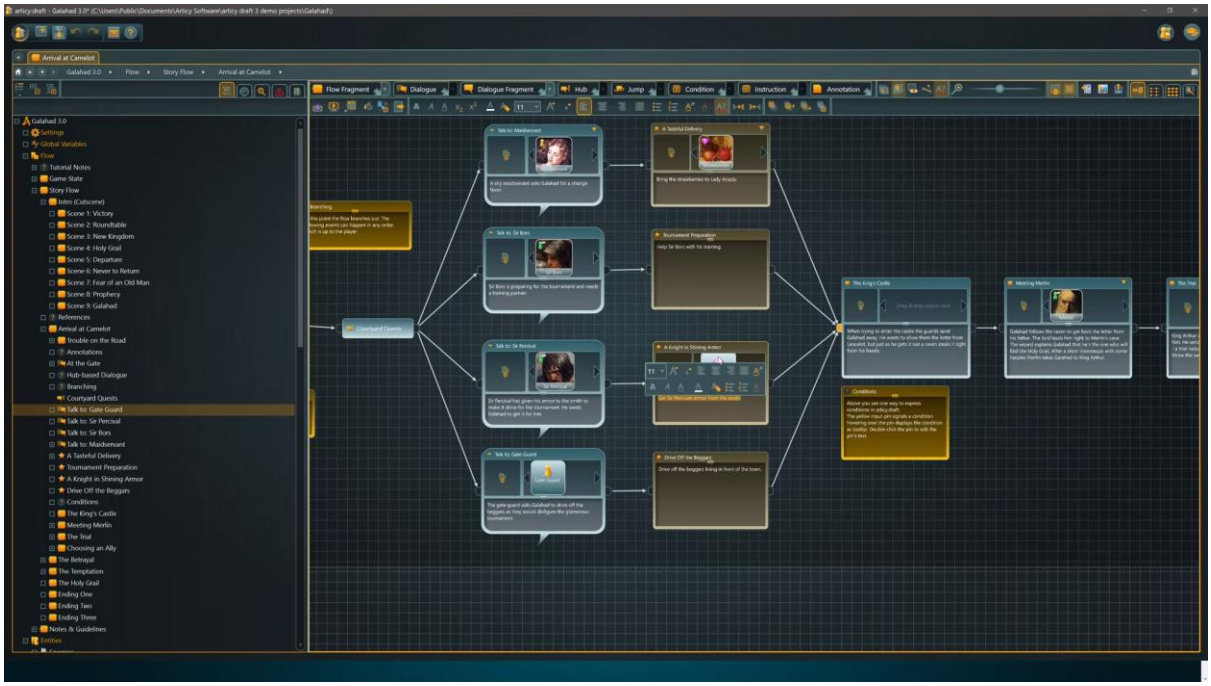


*Figure 10 - Articy branching dialogue UI (*Features of articy:draft X – Narrative Design Tool for interactive projects*, 2014)*

Textboxes can be grouped together in “Nested flows” for organising larger flows. Articy has a database that houses all entities and assets, in addition to the attribute menu. This allows users to view a menu of all listed textbox contents and associated assets. There is also a localisation menu that shows a list of all textbox contents in a reference and translated language, for ease of mass translation.

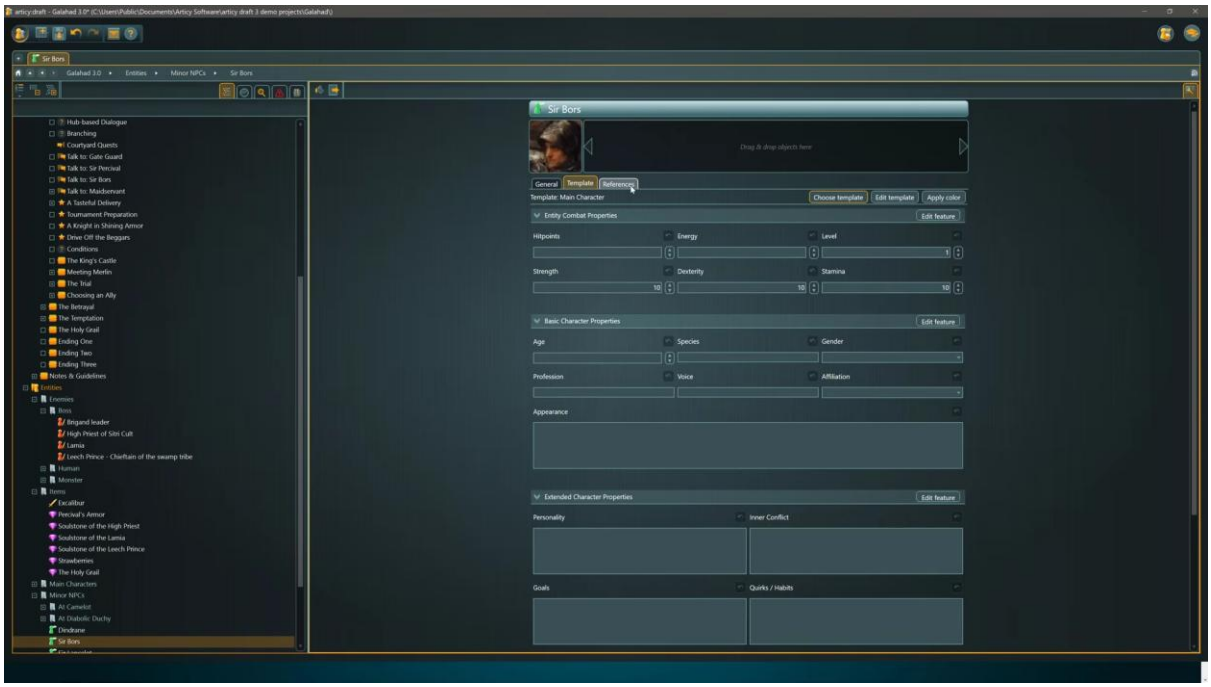


*Figure 11 - Articy template UI (*Features of articy:draft X – Narrative Design Tool for interactive projects*, 2014)*

The left of the screen has also been split up into a file view, and each project can contain an organised structure of different flows, scenes, characters, assets, etc. This is the most advanced covered in this section, and contains the most features, including ones that are not applicable to this project, such as separating textboxes by locations.

2.3.4 Summary

Across these tools, it's clear that ease of use is critically important, Articy is a more feature complete solution that integrates well with other programs, but it's difficulty to use provides a learning curve that the other solutions do not have. By contrast, Inklewriter is an easy-to-use software that designers would be able to easily pick up and use, but it's lack of features mean that it doesn't support more complicated actions or projects.

Twine shows a feature set that is both easy to use with many features built into its logic and formatting, but it is only designed to use with its own integrated environment and is highly tailored for making dialogue within that environment. The tool made by this project should be as system agnostic as possible while still providing features for logic and formatting if applicable.

Table 2 - Existing solutions features table

Feature	**Twine**	**Inklewriter**	**Articy**
Node-Based structure	Y	Y	Y
Visual Graph Editor	Y	Y	Y
Integrated Runtime	Y	Y	N
External Tool Integration	N	N	Y
Branching Dialogue Support	Y	Y	Y
Conditional Logic	Y	Y	Y
Variables	Y	Y	Y
Scripting	Y	N	Y
Multimedia Support	Y	N	Y
Inbuilt Text Formatting	Y	N	N
Localisation Tools	N	N	Y
Asset Management	N	N	Y
Reusable Components	N	N	Y

2.4 Objectives

Based on the features present in the existing options, this project should be a include a node-based graph editor that supports branching dialogue. The nodes should contain some form of conditional logic and variables, as well as embedded scripting. So that this project will work with a pre-existing game editor, there should also be a way to export the dialogue created by the design program and import it into an end program so that it can make use of the data the node contains.

Objective 1 – User Interface
For this project to be successful, the designer should be easily and intuitively able to create branching dialogue for their game, which can then be imported into an end program. To achieve this, the project must include a node-based GUI, with each node containing the literal text to be displayed and connections to other nodes. The developer needs the ability to add, remove, and rearrange these nodes in a window for ease of use.

Objective 2 – Node Attributes
Each node should be able to contain information related to its node, such as the name of associated assets. The information each node requires should be definable by the designer, such that it contains the name of the attribute, and the contents of that attribute.

Objective 3 – Node Connections
Each node should contain the ability to add outgoing and incoming connections, which would connect nodes together in a directed format, such that when one outgoing connection is selected that it corresponds to the connected incoming connection of a different node.

Objective 4 – Connection Components
Each node would have a default outgoing connection slot when one has not been added by the designer, which would not contain any associated data or additional options. Each node should also have the ability to add component connections which would contain supplemental data attached to outgoing connection slots.

Objective 5 – Connection Component Attributes
Each connection component should have the option for corresponding attributes to be defined by the designer, which would act identically to the attributes of the of the node but would be specific to the associated outgoing connection.

Objective 6 – Connection Components Conditions
Each connection component should contain a section for the designer to define conditional logic to be interpreted by the end program, such that the program can determine if this connection component should be displayed to the end user at all. There should be the ability to add multiple individual instances of conditional logic per component connection for the ease of the designer, which can be interpreted by the end program in a way determined by the developer, such as making this component connection available on any of the conditional statements being true.

Objective 7 – Outgoing Connection Components
Each component connection should have a default outgoing connection slot associated with the component connection when no other connections have been added. When additional outgoing connections have been added to the component

connections, each outgoing connection should have an associated section to add conditional logic, so that a designer can state that a component connection could lead to multiple different nodes based on the conditional logic outlined in this section.

Objective 8 – User Interface Features
The designer program should have the ability to move the grid view for ease of use of the designer. The designer program should have the ability to add a start node which will indicate where the dialogue starts. This start node should function identically to a default node, except it will have no connection components, attributes, text field, or accept incoming connections.

Objective 9 – Import and Export
The designer program should also contain the ability to export the dialogue made by the designer, this exported file should contain all information about the nodes, including their presentation in the designer, and all connections. The designer should also contain the ability to import the exported file and have it add all the nodes and connections into the designer such that it is editable and identical to how it appeared when it was exported.

Objective 10 – End Program Importer
The end program should have the ability to interpret the exported file, such that when the end program is run it may read the file and make use of the dialogue, connections, and attributes specified by the designer.

Objective 11 – End Program Stubs
The end program should also specify the variables and functions it wants to make available to the designer, and when run be able to appropriately incorporate the use of those variables and functions as specified by the designer in the text field of the nodes.

Objective 12 – End Program Logic
The end program should also be able to identify which sections of the nodes text field are logic specified by the designer. The program should be able to return and modify variables, run functions and return their values, and run embedded scripts written by the designer, all of which interface with the end program.

2.5 Language

Before any decisions can be made regarding frameworks or structure, the programming language used to develop this project must be decided. Different languages have various levels of complexity, efficiency, control, and support. This foundational decision will influence every step of this project from conceptualisation to the end of production.

2.5.1 C#

C# is a programming language designed to increase programmer productivity, achieved by being simple, expressive, and performant by design (Albahari, 2023). C# is mainly used for application development where hardware access is unnecessary (*C# Tutorial*, 2026) , as it compiles to an intermediate language, and is easy to use due to its clear hierarchy of classes.

C# can be compiled once and work for each platform thanks to the .NET framework (*C++ vs C#*, 2025). The CLR (common language runtime) handles memory using a garbage collector, which decrements references to an object on the heap until that reference reaches 0, at which point the code no longer needs access to that object and the next time the application requires space it deletes the objects that are no longer in use (Richter, 2012).

The garbage collector simultaneously makes C# much easier to use, on account of not being required to manage how objects are allocated in memory directly but comes with the overhead and performance hit of having the garbage collector manage memory. The most prevalent performance issue however is Redundant Data Processing arising from redundant code (Pawlukiewicz & Nedkov, 2024), an issue that can just be overcome with code optimisation.

2.5.2 C++

C++ is a programming language known for its fast speed, being among the fastest of the high-level languages, and low-level memory management using pointers and dynamic memory allocation and deallocation operators (*C++ Programming Language*, 2026). As C++ has no garbage collector, the expectation is placed on the developer to handle their own use of memory, making C++ more difficult to work in.

Due to not having the overhead of a garbage collector and direct access to low-level memory, C++ is very well suited to projects that require speed and performance (*C++ vs C#*, 2025). C++ compiles directly to machine code, so the executable program that is created is for a specific hardware or system (Stroustrup, 2014).

2.5.3 Summary

The advantages to developing a project in C++ are not applicable to this project, so despite the improved performance of C++. the ease of developing C# across a larger group of systems makes it the ideal choice for rapid well supported development.

2.6 File Type

When the designer program exports the data to an external file, it needs to be stored in a serialised way so that it can be imported and read by the end program and the editor itself.

2.6.1 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format (*JSON*, 2008) that is easily human and machine readable. JSON is built on exclusively universal data structurers, almost all modern program languages support them. Due to this and its simplicity, JSON has become very widely used on the web (Bray, 2017), with popular uses ranging from database systems to HTTP API requests and responses (Bourhis et al., 2017).

The two major short comings of JSON are its runtime efficiency and its space efficiency (Viotti & Kinderkhedia, 2022). These shortcomings are directly relevant for data-intensive cloud systems at scale; these shortcomings are negligible in the scope of this project.

2.6.2 BSON

BSON (Binary JSON) is a binary serialisation format designed to be “lightweight, easy to scan, and fast to encode/decode” (Rick-Anderson, 2023). The data is efficiently stored as a binary-encoded version of JSON (*Introduction to BSON and Types*, 2026) that supports fast traversals and supports in-place updates that do not require the entire document to be reserialised (Viotti & Kinderkhedia, 2022).

The advantages gained by BSON over JSON are better suited to handling lots of data quickly, such as network transfers or database storage as opposed to simple human readable data exchange (*Introduction to BSON and Types*, 2026).

2.6.3 MessagePack

MessagePack is a fast and efficient binary serialisation format (*MessagePack: It’s like JSON. but fast and small.*, 2021), that uses minimal extra data for small integers and short strings. The MessagePack serializer for C# outperforms JSON.NET by 9.7 times at serialization and 13.5 times at deserialization (MessagePack-CSharp/MessagePack-CSharp., 2026). MessagePack is very easy to implement for a developer, and due to it’s simplicity has found lots of official and third-party support (Viotti & Kinderkhedia, 2022).

MessagePack is faster than JSON and excels at handling small amounts of data, which is how most of the data would be stored in an external file for serialisation. If this project were to serialise its data to an external file as binary, then MessagePack would be the most suited to this task.

2.6.4 Summary

For this project, JSON would be better suited to storing external files, as although MessagePack is more efficient in terms of size and serialization, these benefits are marginal compared to storing dialogue information in a format that is human readable.

2.7 UI

This project will require a simple and easy to use UI for the designers to use, this will require a framework that handles designing UI objects, drawing the UI, accepting user inputs, and underlying core functionality. This section covers potential UI frameworks.

2.7.1 ImGui.Net

ImGui.Net is a .Net wrapper for ImGui, a popular UI library typically used for prototyping and tool development (Zejdová, 2025), and supports integration with OpenGL, Vulkan, and DirectX. ImGui supports cross-platform desktop applications using Javascript, HTML, and CSS thanks to being developed using the Electron framework (Knoblauch et al., 2017).

ImGui works by outputting vertex buffers that can be rendered in a 3D-pipeline-enabled application (omar, 2026), this makes it well suited for integration into applications that use 3D rendering, such as game engines or full screen applications.

Due to ImGui being an immediate-mode GUI, every frame the UI is being redrawn, which is more intensive but is also harder to use than a retained-mode API (stevewhims, 2022). Designing layouts and adjusting styles requires manual code changes, which can be both time consuming and error prone (Zejdová, 2025).

Additionally, third party extensions include several preexisting node editors (*Useful Extensions · ocornut/imgui Wiki*, 2026), but using one of these extensions undermines this projects aim of creating a node editor, so these extensions are not considered for use as part of ImGui.Net.

2.7.2 Windows Presentation Foundation

WPF (Windows Presentation Foundation) is a UI framework that is designed to run on modern hardware using a vector-based rendering engine (adegeo, 2022). WPF uses XAML (Extensible Application Markup Language) for creating and initialising .NET objects, as this provides a human-authorable way of creating and editing the UI (Sells et al., 2007).

WPF is a retained-mode graphics system, which means that UI elements are represented as a tree of visual objects, and the program only draws or updates the parts that change (Crudu, 2025). Not only is this easier to use as state maintenance, initialisation, and cleanup are all handled by the API automatically (stevewhims, 2022), but the performance is better.

2.7.3 Summary

Of the options presented, WPF is a better solution for this project specifically, as a node-based UI is not graphically intensive and does not benefit from being displayed in an immediate-mode GUI, especially for the needless performance trade off.

2.8 Program Structure

To design this software in a scalable and easily maintainable way, this project should conform to an architectural design pattern that separates the UI, logic, and data.

2.8.1 Model–View–Controller

The MVC (Model-View-Controller) pattern breaks down the architecture of the program into the Model, which holds the data and business logic, the View, which uses UI logic to present the user facing part of the application using the data from model, and the Controller, which handles user input, incoming requests, and handles interacts between the Model and View. (*MVC Framework Introduction*, 2025), shown in figure 12.

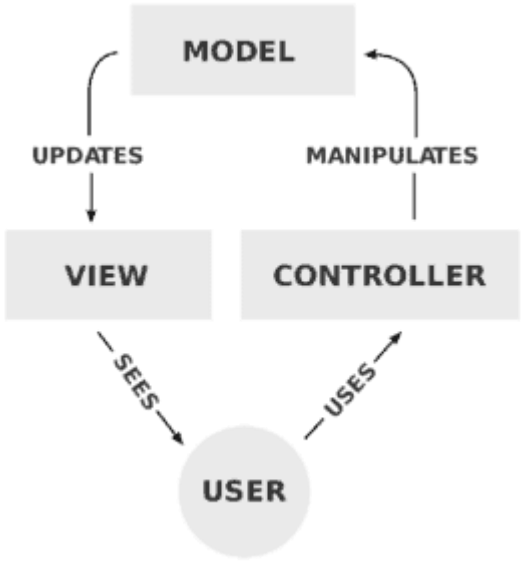


Figure 12 – Model-View-Controller flow of control (baeldung, 2021)

In this pattern, the View can query data from the Model directly as the data processed by the Model is transparent from its perspective (Qureshi & Sabir, 2014). As the component of this pattern are independent of each other, modifying, replacing, or reusing these components are simple (Team, 2023).

Compared to other architectural patterns, MVC can end up with very large controller in big projects (Ibraimi et al., 2025), which can then become a bottleneck.

2.8.2 Model–View–Presenter

The MVP (Model-View-Presenter) pattern is a derivative of the MVC pattern (kexugit, 2010), the Model functions the same way in MVP as it does in MVC, but the View handles user input, notifies the Presenter of changes, and does not use UI logic. The Presenter itself holds the UI logic, it fetches data from the Model and applies UI logic to manage the state of View (*MVP (Model View Presenter) Architecture Pattern in Android with Example*, 2025), shown in figure 13.

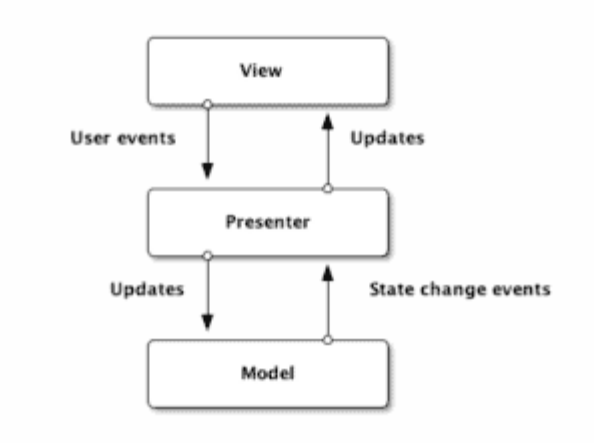


Figure 13 – Model-View-Presenter flow of control (baeldung, 2021)

In this pattern, the View is a passive interface that does not interact with the Model (baeldung, 2021), which allows the Presenter and Model to be decoupled (Qureshi & Sabir, 2014).

MVP allows for great testability but allows for the Presenter to expand to an all-encompassing class if the developer does not follow the single responsibility principle (*MVP (Model View Presenter) Architecture Pattern in Android with Example*, 2025); however, following this principle can result in a large manual wiring effort (Ibraimi et al., 2025).

2.8.3 Model-View-ViewModel

MVVM (Model-View-ViewModel) is a GUI architectural pattern derived from MVC (Fuksa et al., 2025), the Controller is replaced by a ViewModel which serves as a bridge between the View and Model using a data-binding mechanism (Team, 2023). The ModelView holds no reference to View, instead it communicates with the View exclusively through data bindings and makes the data in Model accessible to the View by turning it into a View-friendly format, shown in figure 14.

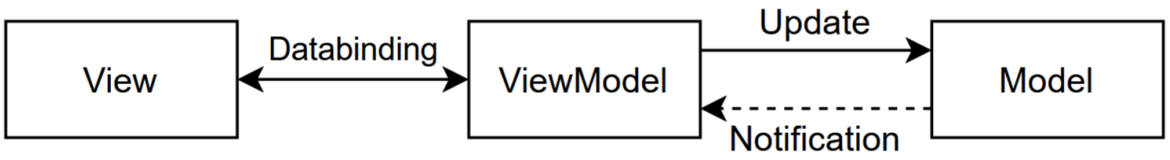


Figure 14 - Model-View-ViewModel architectural pattern (Fuksa et al., 2025)

Two-way data binding allows user interface elements to react as soon as the bound data is changed (Ibraimi et al., 2025), so despite the learning curve this allows developers to easily build reactive UIs. The ease of use and reduced complexity of handling dynamic changes have made MVVM a popular architectural choice.

With large applications, the data bindings can impact performance, but with simple UI's MVVM can be overkill (*Introduction to Model View View Model (MVVM)*, 2023). Additionally, despite offering stronger maintainability and testability, debugging can be difficult on account of complex data bindings.

2.8.4 Summary

For this project, the MVVM pattern makes the most sense, as an application which includes a highly adaptive UI depending on data best fits MVVMs use case. By separating the user interface, business logic, and data handling into distinct components, MVVM also improves maintainability, scalability, and overall code organisation throughout development.

2.9 Embedded Scripting Language

Games that this tool will be developed to assist with are usually made by a multidisciplinary team with various degrees of familiarity with programming, but designers should have some influence over game logic (Renger, 2022).

To this end, designers may wish to include scripts with nodes, so that when the exported file is read by the end program, the program may interpret what has been written as instructions to execute. These instructions need to be written in a language the end program can interpret, so this section will evaluate the available interpreters and the associated languages.

2.9.1 MoonSharp

Lua is a scripting language designed to be simple, portable, fast, and easily embedded into other applications (Ierusalimschy et al., 2007). As an example of it's simplicity, the only kind of data structure that Lua uses is the table, which is an associative array. Additionally, Lua is dynamically typed, variables themselves do not use types and values contain references to structured values.

Moonsharp is a Lua interpreter written in C#, it's designed to be easier to use and to have maximum compatibility with .NET, Mono, Xamarin, and Unity (*MoonSharp*, 2016) . Moonsharp is made up of a collection of functions that allow Lua variables to be edited and read, Lua functions to be called in C#, register C# functions to be called in Lua, and other functionality (Ierusalimschy, 2003).

Lua has been designed to be easily extended, but it has also been designed to be small and fast, notably Lua is more than an order of magnitude smaller than most scripting languages but is independently benchmarked as being one of the fastest interpreted scripting languages (Ierusalimschy et al., 2007).

Lua has become a popular choice for game engines for a couple of reasons, it allows prototypes to be made faster and easier, the language is simpler so can be used by

more than just developers, and the language does not need to be compiled before being executed (Kašuba, 2015). This allows developers to rapidly redeploy and test code.

2.9.2 IronPython

Python is an interpreted and interactive programming language that supports dynamic typing and data types, and has very clear syntax (*General Python FAQ*, 2026). Python can be embedded into existing applications as it is compiled and interpreted at runtime (*Python | Compiled or Interpreted ?*, 2025), which means it has the same ease of testing and redeploying code as Lua. Additionally, python is known for its readability, combined with its popularity outside of embedded systems means that python is easy to understand, maintain, and well supported with a large array of libraries (*Python Tutorial | Learn Python Programming Language*, 2026), and supports multiple platforms such as Windows, Linux, and Mac OS.

IronPython is a .NET compiler for a full implementation of Python written in C# (Foord, 2008), and accessing the .NET framework is incredibly easy as accessing classes from .NET within IronPython does not require type conversions. IronPython is well suited to embedding into web development, such as ASP.NET, and client-side applications, such as WPF.

IronPython is a dynamic language that preforms many tasks at runtime, which allows it to more readily adapt to changing conditions (Mueller, 2010), granting a level of flexibility to embedded systems that static languages could not.

2.9.3 Jint

JavaScript is a dynamically typed interpreted language that supports client and server-side development (*Introduction to JavaScript*, 2026). JavaScript is versatile in its applications and well supported by numerous libraries and frameworks. The language is event driven, so it responds to users' inputs in real time, and an interpreted language, so the language is executed line by line.

Jint is a JavaScript interpreter for the .NET framework that runs JavaScript inside a safe sand-boxed environment (Ros, 2026). Like MoonSharp and IronPython, Jint exposes native .NET objects to embedded JavaScript code. Due to Jint not generating .NET bytecode and doesn't use DLR, small scripts run very fast but is slower than traditional languages for complex tasks.

JavaScript is a complicated language that requires developers to understand core programming concepts, which makes this ill suited to be given to designers (*Introduction to JavaScript*, 2026). Additionally, JavaScript does not require explicit types, which can lead to errors at runtime as type checking is not strictly enforced.

2.9.4 Summary

Lua is the best fit for this project, the language is smaller and faster, simple enough to hand to designers without much programming knowledge, but critically Lua is the industry standard for embedded scripting in the games industry, so will be the most likely to fit with preexisting development pipelines and match current designers' skills.

2.10 Embedded Scripting Patterns

This section outlines the potential options for embedding scripting languages into the program. The designer should be able to create dialogue that integrates with the end system without the developer needing to change it. To that end, this program should make use of embedded scripting so that the exported file would be able to communicate with the end system. There is a collection of patterns that are used to facilitate the communication between the executed Lua and the executing program.

2.10.1 Observer Pattern

The observer pattern, otherwise known as the publish-subscribe pattern, is a design pattern that allows observers to be notified of a change in the state of an object via the subject (*Mastering the Observer Pattern in Lua: Implementing Publish/Subscribe for Event Notification*, 2024). The intent of this pattern is to define a one-to-many relationship that allows all dependant observers to be updated when a change of state is issued, as shown in figure 15.

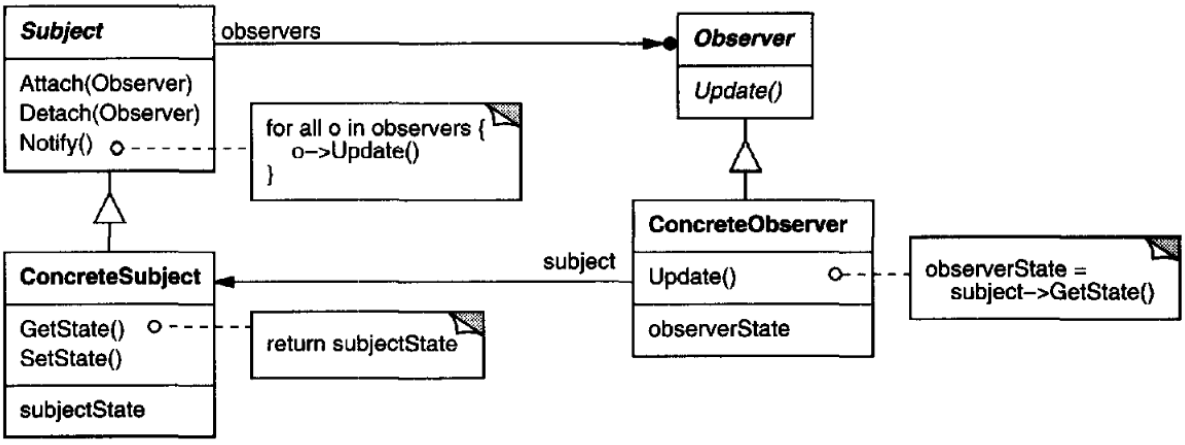


Figure 15 - Observer design pattern UML diagram (Gamma, 1995)

When creating an observer pattern, the objects that change don't know how many dependant objects they need to change, nor do they need to specify which objects those changes would concern. By having objects notify the subject of a change and allowing the subject to publish a notification to all subscribed observers, we allow the objects to not be tightly coupled (Gamma, 1995).

This pattern excels when there are many systems that depend on each other and react to the programs constantly changing state, specifically all objects update the subject, which then issues event notifications that different observers are listening for (*Observer Design Pattern*, 2026). For this project, the only object that would be issuing changes to the rest of the program would be the embedded script interpreted by the end system, and as such an event publishing / subscribing model would be unnecessary compared to other simpler options.

2.10.2 Interpreter Pattern

The interpreter design pattern is used to define the grammar of a language (*Interpreter Design Pattern*, 2026). Each rule for the grammar of a language is represented by classes that define how a specific expression should be interpreted. These expressions are part of a syntax tree that creates more complex expressions (Gamma, 1995). The interpreter evaluates this structure to execute expressions written in the defined language, using the flow defined in figure 16.

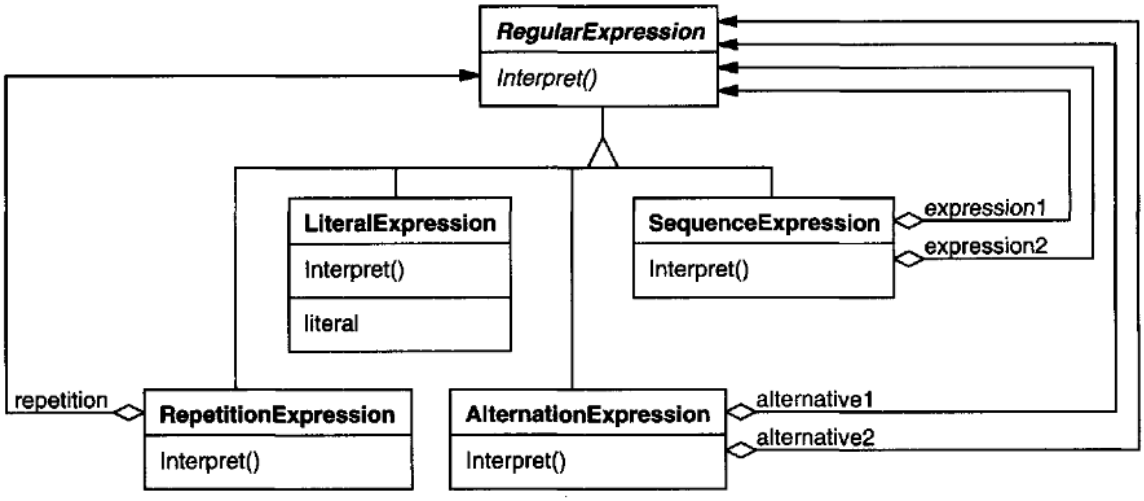


Figure 16 - Interpreter design pattern UML diagram (Gamma, 1995)

This design pattern is relevant to this project as in order to meet the objectives set out above, the designer needs to be able to interact with the end program using embedded scripting. Therefore, the end program must run this code, which has not been compiled, at run time.

This pattern can be used to implement a custom scripting language into this program (*Interpreter Pattern in Lua: Mastering Language Processing*, 2024), but this would be cumbersome and ineffective, as other solutions already exist that designers may be more familiar with. Instead, this program would implement the interpreter that has already been created, and this pattern could be used to evaluate expressions and parse commands from text.

2.10.3 Facade Pattern

The façade design pattern is used to provide a unified interface for a subsystem that hides its internal complexity (*Facade Method Design Pattern*, 2026). This means that a single-entry point to those systems are provided, or sometimes a small collection of simple methods, and all the required subsystem classes are internally called by as the façade delegates requests, shown in figure 17.

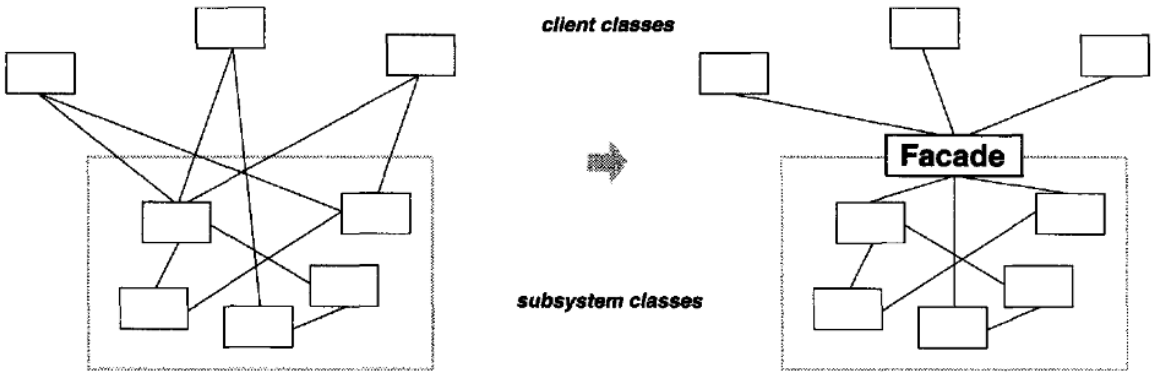


Figure 17 - Facade design pattern UML diagram (Gamma, 1995)

This design is useful as it hides powerful low-level systems behind simple interfaces (Gamma, 1995), such as parsing and code generation. This pattern is also easily maintainable within a larger program, as any of the subsystem classes can be changed and due to the loose coupling between the subsystems and the main program without affecting how the façade is called (*Facade Pattern: Simplifying Complex Subsystems with Design Patterns*, 2024).

This pattern is relevant to the project's embedded scripting as taking the content of the text field and separating out the embedded script, and turning it into code that can be run, among the other functionality the embedded scripting interpreter will need, will be best kept loosely coupled and simply presented to the rest of the program.

2.10.4 Summary

The embedded scripting in the end program should be designed to use the interpreter pattern to handle parts of the script to be run, and the façade pattern to hide this complexity from the rest of the program.

2.11 Data Structure

As each node can have any number of both incoming and outgoing connections, each node is represented by a many to many relationship. Poorly designed databases that store many to many relationships are the largest source of data duplication, desync,

and maintenance pain (*Many to Many Relationships: A Guide to Database Design*, 2026).

When storing and exporting data from the dialogue system, the connections between nodes shouldn't be stored in the nodes themselves, as it would either become possible to have two nodes fall out of sync when referring to another node, or the connections would be formatted such that one node is blind to its own connection.

Additionally, when storing textboxes in JSON, the textbox objects themselves cannot store connections as references to other textboxes as JSON does not support serialised graphs due to circular references (*TypeError: cyclic object value - JavaScript | MDN*, 2025).

This section outlines potential data structures that could be used to store connection information between nodes.

2.11.1 Adjacency Matrix

An adjacency matrix is a method of representing which vertices of a graph are adjacent using a table, with each node being included in both axes (Singh & Sharma, 2012). If a connection is present between two nodes, then the value of those nodes in the graph is 1, otherwise it is 0. In an undirected graph, the graph is symmetrical along the diagonal where nodes represent connections to themselves, but in a directed graph then each side of the graph can represent the direction of the connection (*Adjacency Matrix Representation - GeeksforGeeks*, 2025). For the sake of this project, each side of the graph can represent if a connection between nodes is incoming or outgoing.

For an adjacency matrix, adding, removing, and checking an edge connection can be done in O(1) time simply. This comes at the cost of representing each connection with a value in the table, regardless of whether the nodes are connected or not. This uses $O(V^2)$ memory, so is efficient if the graph is dense, where there are many connections between nodes, but becomes inefficient when the graph is sparse. For this project, it is unlikely that the designer would interconnect the nodes enough to justify this data storage solution.

2.11.2 Edge List

An edge list is a data structure that represents a graph by simply storing connections between two nodes in a list. For this project, which will likely store connections as a sparse graph in almost all use case, they are memory efficient as only the existing edges are stored, so has a memory usage of O(E) (Lee, 2025).

The edge lookup of an edge list is proportional to the number of edges, which slows with larger amounts of edges as the lookup speed is O(E). Large amounts of nodes can also make this storage solution become inefficient as checking the existence of an edge

requires iterating over the entire list. Ultimately, edge lists are the best for projects that require iterating over the edges of a sparse graph.

2.11.3 Adjacency List

An adjacency list is a data structure that stores a list of all nodes, each of which stores a list of nodes that node connects to (*Adjacency List Representation*, 2025). In the case of a directed graph, only one of two connected nodes would need to hold a connection, in the case of this project this would be the outgoing node.

The adjacency list is a compact method of storing connection information as it takes O(V + E) memory, and retrieving the list of connected nodes can be done in O(1) time, however adding and removing connections takes longer, as it takes O(deg(V)) time (Singh & Sharma, 2012). When adding or removing a node, either the data structure either needs to be resized to add the new node, or all references to the node must be removed from all other nodes that connect to it, this is poorly suited to this project as nodes and connections will be frequently changing.

2.11.4 Summary

Out of the potential solutions, an edge list best fits the criteria for this project as a sparce graph that will be frequently changed. As a data structure, the edge list is also best suited to having additional information be added to, such as representing nodes having different outgoing connection points.

3 Risk

This section outlines the risks associated with developing this project, and the steps taken to mitigate these risks. Below is the risk analysis and mitigation table included in the project definition document.

Table 3 - Risk analysis and mitigation table

Risk	Likelihood	Severity	Impact	Mitigation
Program because too unwieldy to properly modify	3	4	12	Good programming practices and proper maintenance.
Failure to meet milestones	2	4	8	Use the inbuilt redundant time to work on other objectives, pre-emptively allocate extra time if missing milestones is expected.
Illness	3	2	6	Build redundant time into the project to afford time not working.
Inaccurate estimation of time in task plan	2	3	6	Build redundant time into the test plan.
Users don't understand how to use program	2	3	6	Provide documentation on how to use this tool.
Failure to integrate with user projects	1	5	5	Provide documentation on how to build into the user's program.
Miss deadlines	1	4	4	Build redundant time into the project so that additional work can be completed alongside
Data loss	1	4	4	Regularly backup to GitHub to minimise the impact of losing data
Performance bottleneck	1	2	2	Debug and rewrite logic for better performance if this occurs.

For the program to not become too unwieldy, "good programming practices" means developing the project using Model-View-ViewModel architecture in practice. Failure to meet milestones and inaccurate estimation of time in task plan was not particularly relevant, as specific milestones and a strict task plan was replaced with a different development methodology that used sporadic sprints for batches of features tracked as

part of that batch. However, all these risks were in service of not missing the deadline, which happened, so this project is working with the extended deadline.

To mitigate the loss of data, regular backups means that progress on this project won't be lost in the event that the device it's being developed on becomes inoperable. A performance bottleneck and failure to meet milestones did not end being an issue and providing documentation as a way to mitigate users not knowing how to use the program is out of scope of this project. Illness ended up being a persistent problem throughout this project, despite steps being taken to avoid and mitigate this problem, to which the solution was simply to continue developing the project while ill.

4 Technical Achievement

This section showcases the software developed for this project and the technical details that went into them. This section is split into the designer program, which creates the dialogue, Unity, the end program of choice for fitting the created dialogue into a game engine, and Lua, how embedded scripting is used across both these applications.

4.1 Designer Program

The following section outlines some of the technical details of the universal dialogue systems design program. This program is intended to be used by only designers to create dialogue for an end program, so it is simple to use whilst comprehensively covering the designers requirements of this system.

4.1.1 Presentation

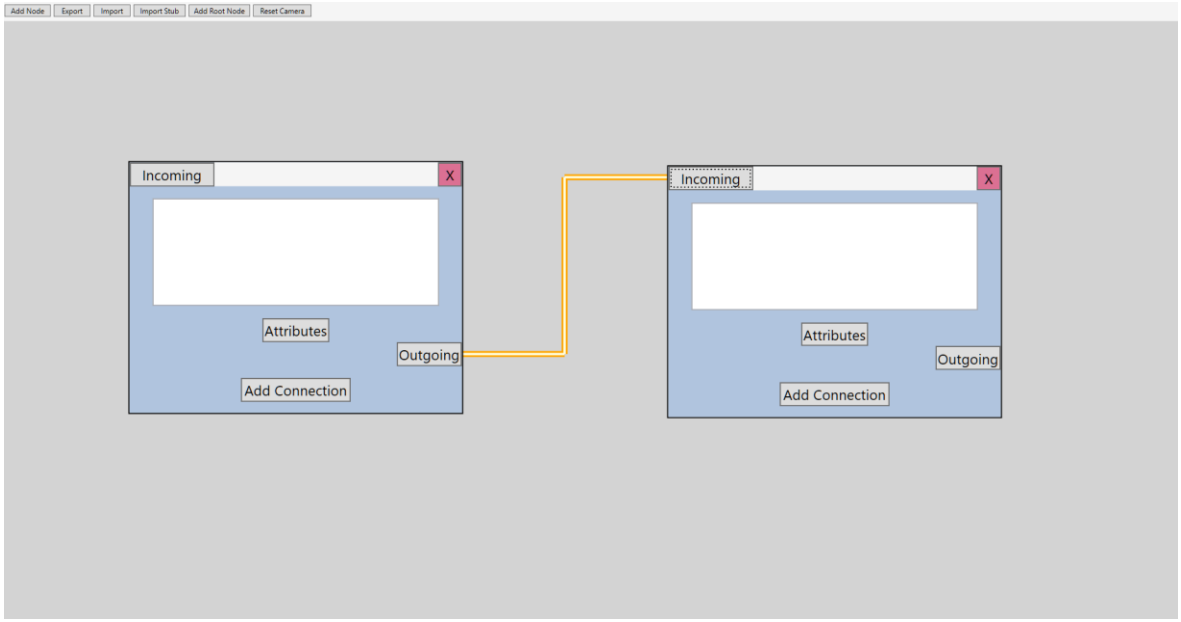


Figure 18 - The design program showing two connected nodes

The designer program is a node-based GUI which has the ability to add, remove, and connect individual nodes that contain the text to be displayed and other fields, as well as add default starting nodes. These nodes can be resized, and the position of the viewport can change and zoom. Figure 18 shows two nodes that have been connected, and the viewport has been zoomed in on these nodes. This meets the specification set in Objective 1 – User Interface and Objective 8 – User Interface Features.

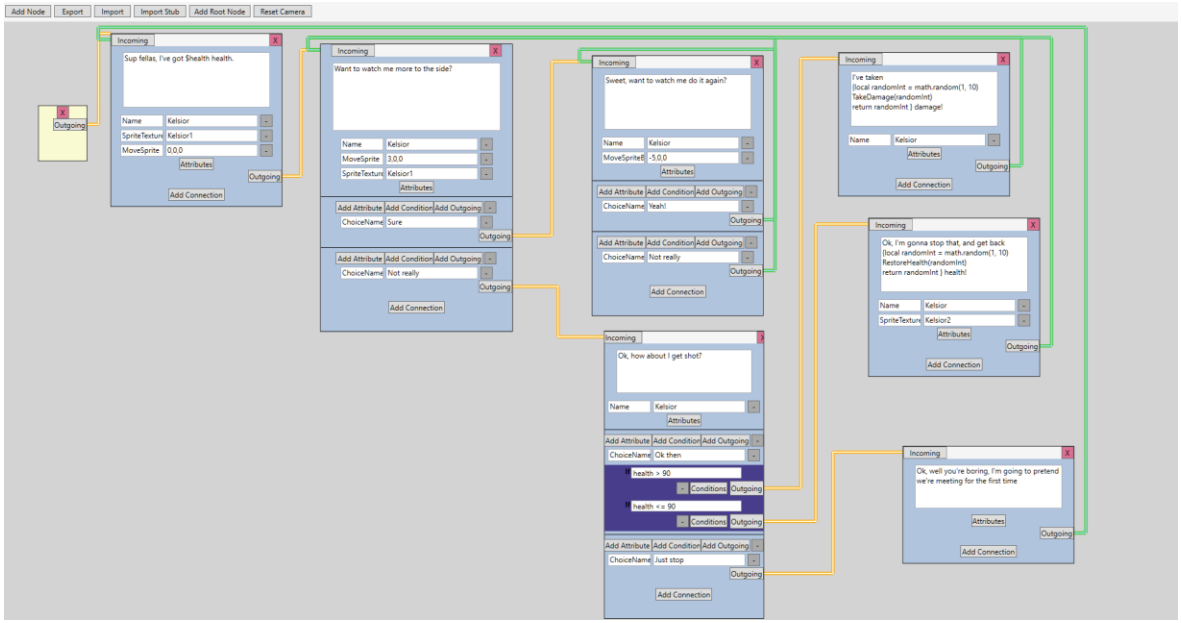


Figure 19 - The design program showing multiple connected nodes

Figure 19 shows a more complicated dialogue tree, which demonstrates that each node can connect to each other node in a directed format, fulfilling Objective 3 – Node Connections, but also showing that the connections bend for ease of readability, and that connections that connect to a point left of them turn green and take a different path in an attempt to not intersect with either connected textbox. This is in service of ease of use, as having connections in the same space of the same colour that overlap with each other makes following the connections harder for the designer to interpret.

4.1.2 Node structure

The structure of the node should account for all the potential configurations a designer may wish to implement, the feature complete design of the node is outlined in the Objectives in section 2.4.

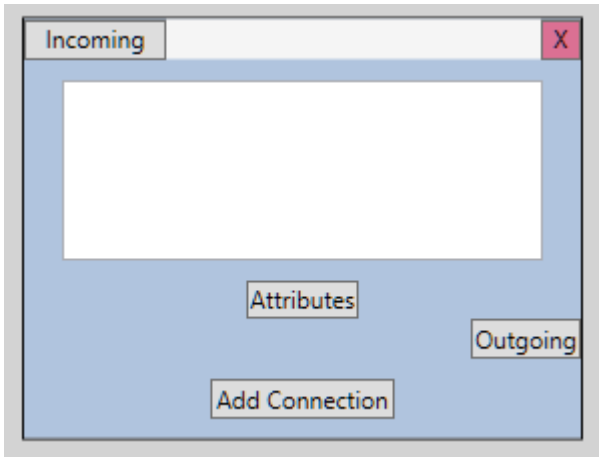


Figure 20 - A defualt dialogue node

The Base Node, as shown in figure 20, has a default outgoing connection that gets removed when a connection component is added and can contain any amount of connection components and attributes.

```
<Button Content="Attributes" HorizontalAlignment="Center" VerticalAlignment="Center"
        Command="{Binding AddAttributeCommand}"
        Style="{StaticResource HideForRootStyle}"/>

<Button Content="Outgoing" Height="20" HorizontalAlignment="Right"
        Command="{Binding AddDefaultConnectionCommand}"
        Style="{StaticResource DefaultOutgoingButtonStyle}"/>

<ItemsControl x:Name="ConnectionComponent"
              ItemsSource="{Binding ConnectionComponents}"
              ItemTemplate="{StaticResource ComponentConnectionView}"
              Style="{StaticResource HideForRootStyle}"/>

<Button Content="Add Connection" Width="92" Height="20" Margin="10"
        Command="{Binding AddConnectionComponentCommand}"
        Style="{StaticResource HideForRootStyle}"/>
```

Figure 21 - XAML code from the dialogue node

This XAML code shown in figure 21 shows that the buttons are bound to commands in the NodeViewModel, such as AddConnectionComponentCommand. When this is clicked, the button labelled outgoing is collapsed and set to not visible, as the DefaultOutgoingButtonStyle which this button is using as it's style contains a data trigger bound to IsDefaultOutgoingVisible, which gets set to false when a connection component is added.

```
<Style x:Key="DefaultOutgoingButtonStyle" TargetType="Button">
    <Setter Property="Visibility" Value="Visible"/>
    <Style.Triggers>
        <DataTrigger Binding="{Binding IsDefaultOutgoingVisible}" Value="False">
            <Setter Property="Visibility" Value="Collapsed"/>
        </DataTrigger>
    </Style.Triggers>
</Style>
```

Figure 22 - XAML code of the default outgoing button style

Additionally, when a connection component is added, the contents of that resource, that being the new buttons and text fields, are added to this node. Their source is bound to that specific connection component, as contained in the NodeViewModel, shown in figure 22. This forfills Objective 4 – Connection Components.

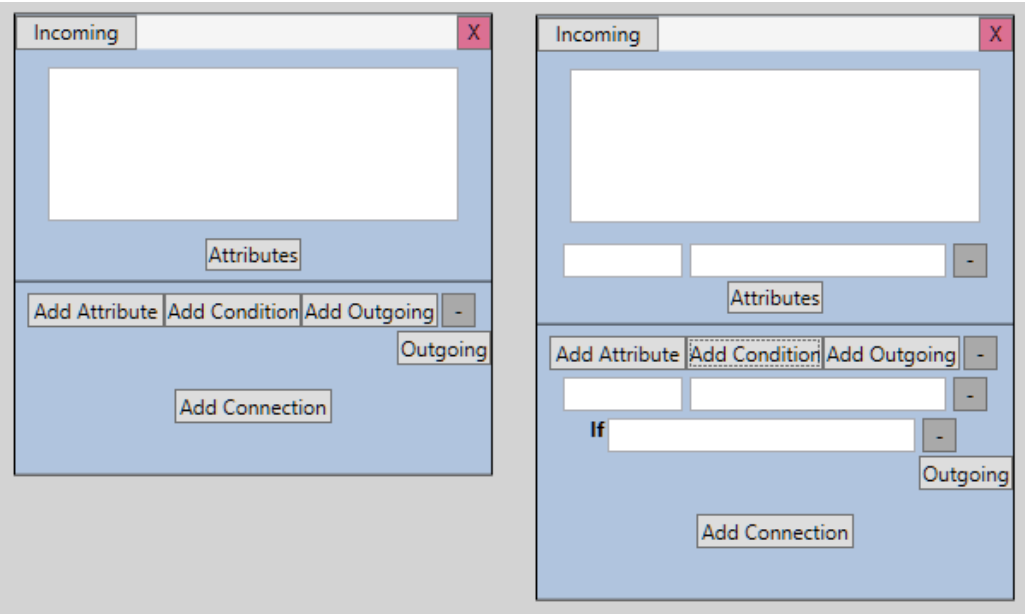


Figure 23 - Dialogue nodes showing attributes

This process of data binding will be covered further in section 3.1.3 Model-View-ViewModel structure and section 3.1.4 Event handling, but it is this mechanism that elements are added to the node, such as attributes, conditions, connection components, and outgoing connections. The node on the right of figure 23 shows an attribute that's been added to the node itself, as per Objective 2 – Node Attributes, and an attribute and condition that has been added to the connection component, meeting Objectives 5 – Connection Component Attributes and Objective 6 – Connection Component Conditions respectively.

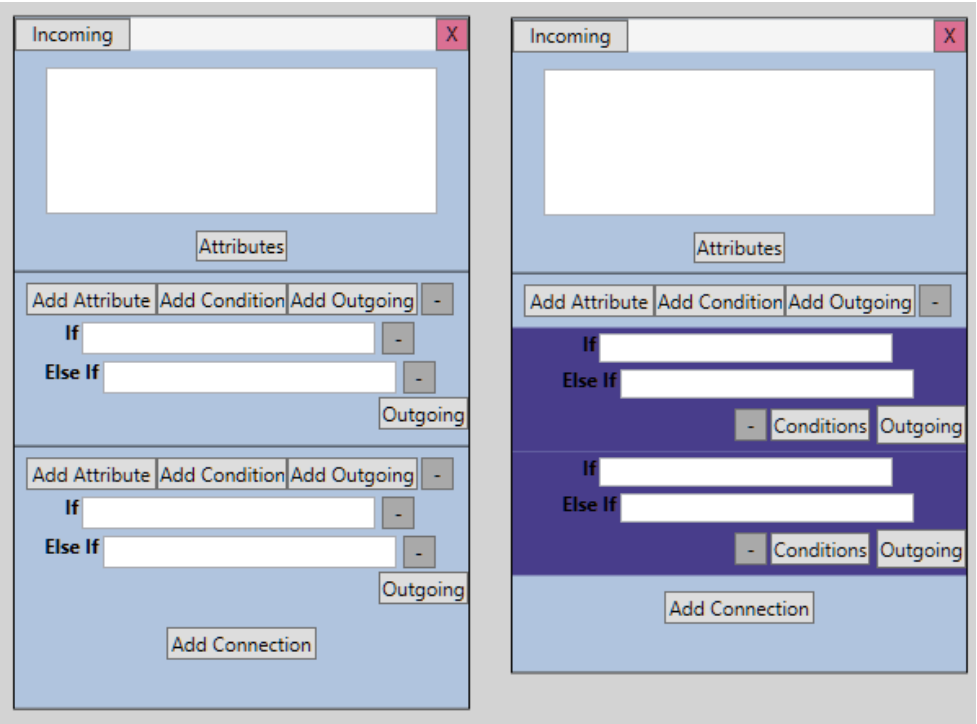


Figure 24 - Dialogue node showing conditions

In the same way that adding a connection component to the node removes the nodes default outgoing connection button and associated connection from the connections list if one had been made, adding an outgoing connection to the connection component removes that components default outgoing button and associated connection, as outlined in Objective 7 – Outgoing Connection Components. Additionally, as per this objective, each outgoing connection component can have conditions be added which will determine which of the connections will be used when this connection component is selected. Of note, despite the seeming similarities between the nodes in figure 24, the conditions for the connection component determine whether that connection will be shown to the user, whereas the conditions for the outgoing connection components determine which connection will be used upon that component being selected.

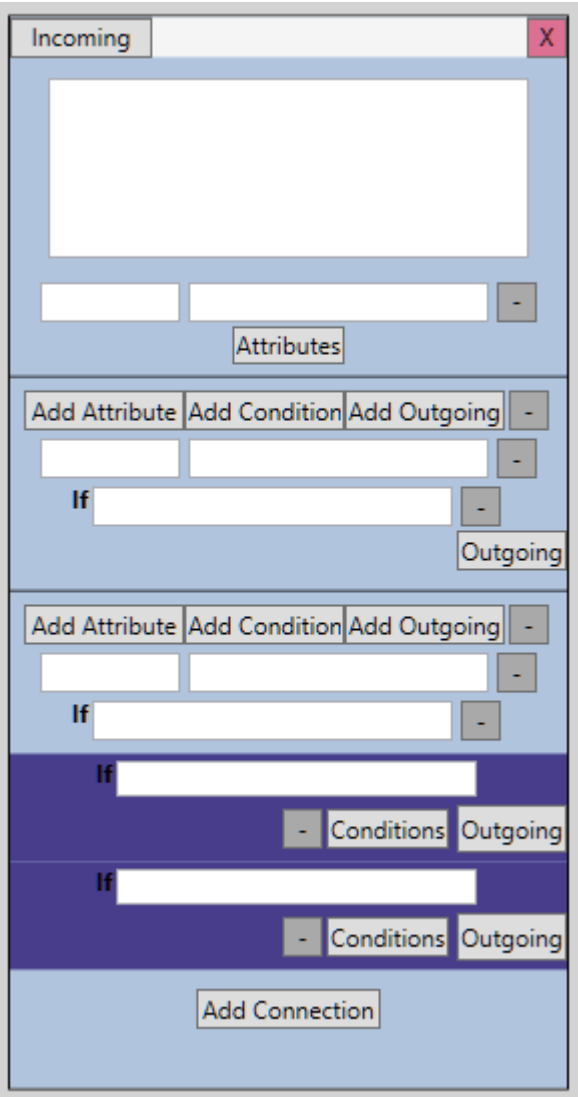


Figure 25 - A dialogue node showing many fields

As shown in figure 25, the nodes can be constructed to string each of these features together, to match whatever the designer needs out of that node.

4.1.3 MVVM structure

The structure of the designer program uses a practical implementation of Model-View-ViewModel architecture to separate concerns between classes, a stricter implementation would have made working with these classes needlessly difficult without the benefit of using this architecture.

The window of this program uses MVVM architecture, but the implementation of this architecture is best shown via the node design. The node is separated into three layers based on the aspects of the node that need to be dynamically added, those layers being Node; which a dynamic amount of may be added to the window, Connection; which in node is called ConnectionComponent to avoid confusion, and OutgoingConnection.

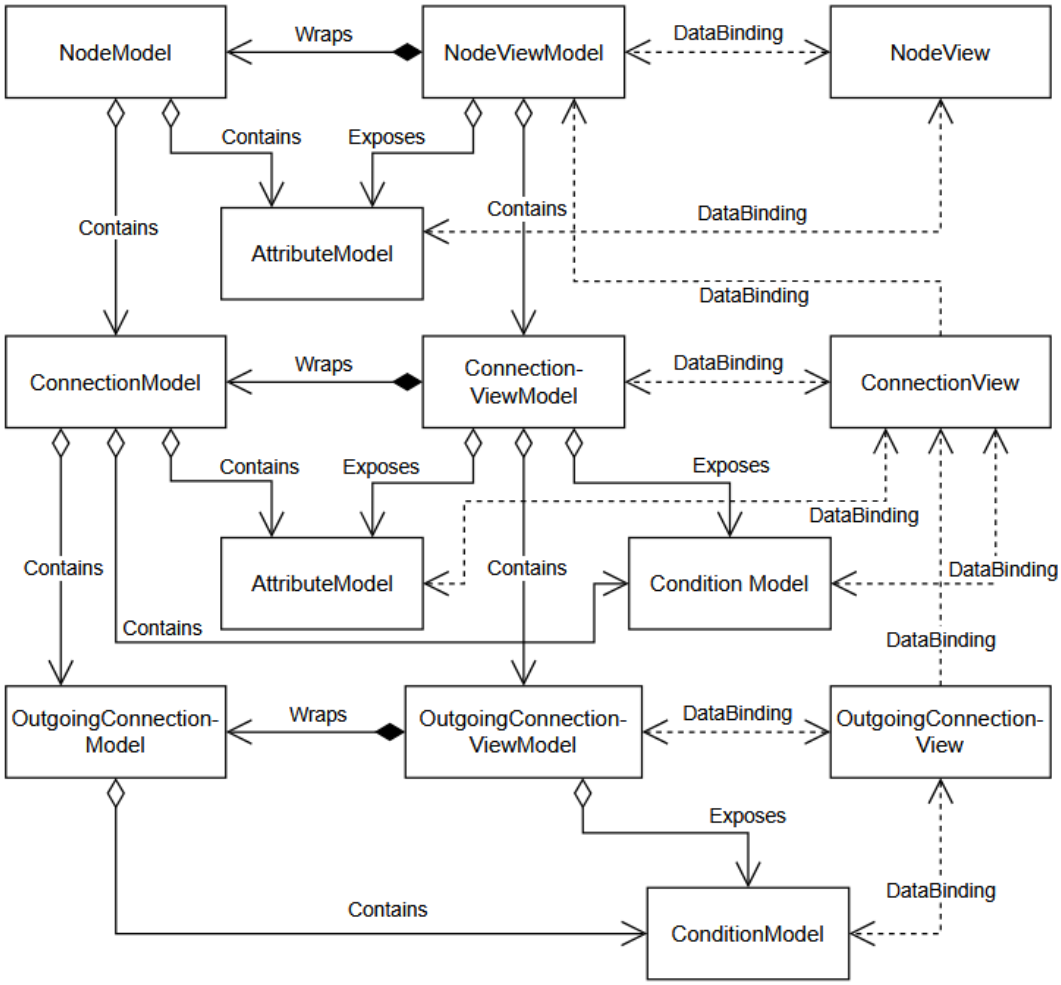


Figure 26 - High level class diagram of a dialogue node

Each of these layers implements MVVM architecture, as shown in figure 26, the ViewModel is what contains the logic of the class, it wraps the Model of the class, which contains data and functions specific to manipulating data, such as import and export logic. The Model and ViewModel both inherit from INotifyPropertyChanged, and are informed when an element is updated so that it is stored properly, more details on this in section 3.1.4 Event handling.

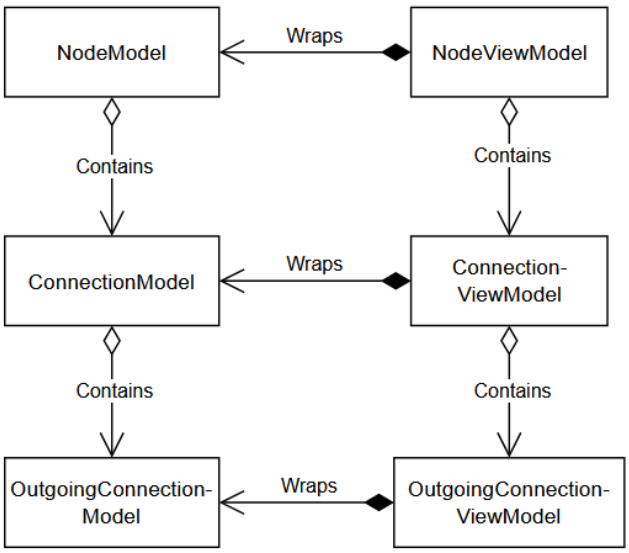


Figure 27 - High level class diagram showing the Models and ViewModels of the dialogue node

The NodeModel contains the ConnectionModel, whereas the NodeViewModel contains the ConnectionViewModel, whilst both ViewModels expose their respective Models, all the while the Model does not know about the ViewModel. This means that the Models don't rely on the ViewModels and that any access to the Model must go through the ViewModel, as shown in figure 27.

The implementation of the View class varies based on if it's a Node, Connection, or OutgoingConnection. All the View classes are written in xaml, but NodeView is an ItemsControl with an associated .cs class, whereas ConnectionView and OutgoingConnectionView are resource dictionaries that are used in NodeView.

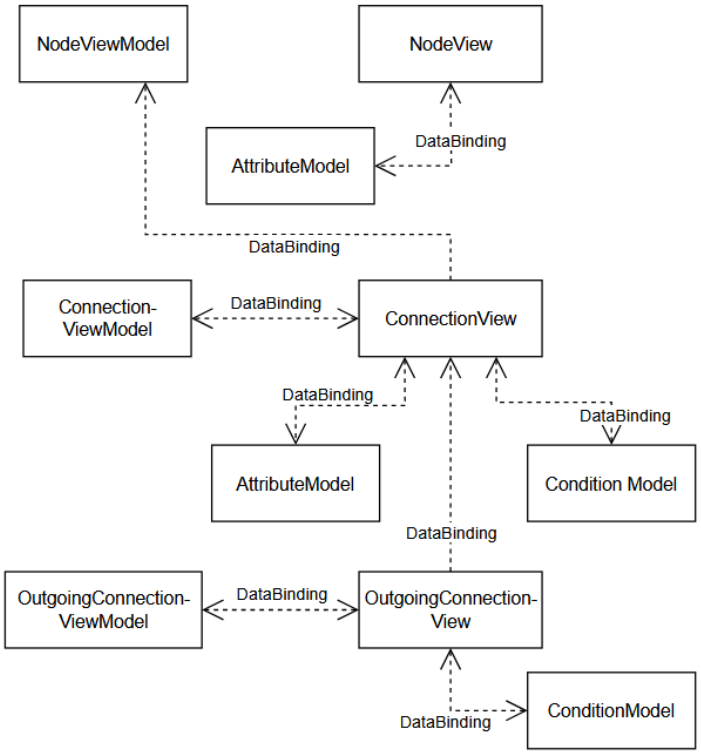


Figure 28 - High level class diagram showing the dialogue nodes Views databindings

Some elements in node view contain a data binding that point to other parts in the program, as shown in figure X. Usually this is a two way databinding that points to that views associated ViewModel, but some elements require communicating with that elements ancestor, such as a ConnectionComponent requiring to communicate with the Node as only the Node has the capacity to delete a ConnectionComponent.

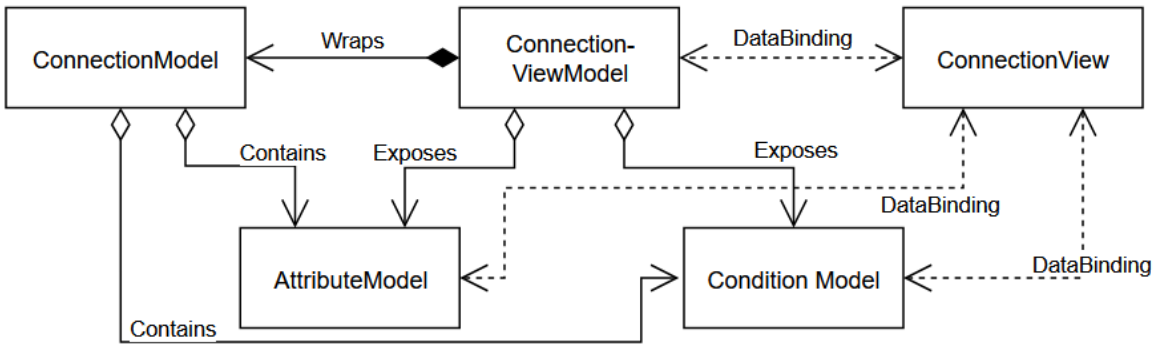


Figure 29 - A high level class diagram showing Connections MVVM architecture

The models also contain attributes and conditions, these are classes that don't have associated Views or ViewModels, as they exist only to store data and specific data logic, such as import and export functions. These classes are stored in Models, exposed via ViewModels, and have data bindings with Views via their respective ViewModels, as shown in figure 29. When exporting a file to json, or importing from json, the data is formatted in this way, in addition to the stored connections, as described in Objective 9 – Import and Export.

4.2 Unity

Unity has been selected as the end program of choice to show how the universal dialogue system can interface with other game engines. The code outlined in this section is specific to unity for this demonstration, much of the logic can be ported to other engines simply, and the code that can't would simply be a matter of reimplementing.

4.2.1 File reading and data structure

In this project the connections and nodes are both their own data types, the node structure is described in section 3.1.3 MVVM structure, connections consist of nodeId, componentId, connectionId, and targetNodeId. When exporting to an external format, these are stored consecutively in json. When including a lua stub in a project, as outlined in section 3.3.2 Lua stubs, the stubs would be added to the json file so that subsequent loading of this file would include that stub.

The unity program has the capacity to import files from json, which happens every time the program is run, as specified in objective 11 – End Program Importer. When this happens the lua stubs are discarded, as are the position and size information about the node, which are only relevant when importing into the design program. When imported, the importer will send the text content to the lua interpreter, then treat that output as the literal text to be displayed, more details in section 3.3.1 Lua interpreter. The text output can also be amended here by attributes, which is covered more in section 3.2.2 Core and demo files.

4.2.2 Core and demo files

In unity, the program is split up into the core and demo files, the core files are the necessary part of every program that uses this universal dialogue system and uses the file exported by the design program. The demo files are the files that are project specific and are not necessary for every project that uses this system to work.

```
public static List<TextBox> textBoxes = new List<TextBox>();
public static List<Connection> connections = new List<Connection>();
public static Dictionary<string, object> variables = new Dictionary<string, object>();
public static Dictionary<string, Delegate> functions = new Dictionary<string, Delegate>();

public static Dictionary<string, Action<NarrativeAttributeContext>> attributes = new Dictionary<string, Action<NarrativeAttributeContext>>();
```

Figure 30 – Global variables in Unity

As part of the core files includes the global variables, shown in figure 30, which contains variables, functions, and attributes. Any demo file can send their variables, functions, and attributes to the globals. The variables can be read and wrote to by lua, covered more in section 3.3.1 Lua Interpreter, the functions can similarly be executed by lua. TextBoxes contain a number of attributes that when each text box is selected is iterated through. Each attribute contains an associated function which is then executed to the parameters set in the designer.

4.3 Lua

This section outlines how Lua is used for embedded scripting between the design program and the end program, and how the dialogue written by the designer can influence and be influenced by the game state, whilst allowing logic to be embedded in the dialogue itself.

4.3.1 Lua interpreter

The designer program allowed for specifying Lua in two ways to allow for embedded scripting, that being prefixing a single variable or function with a dollar value or encasing a larger chunk of Lua in braces. When the program would display the text that contains this logic, it first evaluates the Lua and returns relevant values, that being the contents of the variable, the returned value of a function if that value is not void, or the returned value of a chunk of Lua written by the user.

When evaluating the text for Lua, the text is broken up into sections based on that text meeting the syntax requirements set out above. Based on the type of syntax, the text is then modified to an executable format, such as placing a return value in front of variables and functions and fixing the formatting of functions if they have been written without brackets and when the contents of those brackets are separated into a different string. Chunks of Lua as written in braces by the designer is left as is when executed other than separating it from the other dialogue text, the assumption being that all Lua that gets to this part of the program will be formatted correctly due to the inbuilt error checking, which is covered in section 3.3.2.

Before the scripts are run, the variables and functions from the globals are injected into Lua's global scope. This allows for any variable or function in the program to be used by Lua despite having been written in C#, as described in Objective 12 – End Program Logic.

4.3.2 Lua stubs for error checking

When the program is run, the functions, variables, and attributes added to globals are exported as stubs to a separate json file. These stubs contain information necessary to inform Lua what these variables and functions are, their declaring and returning type, and their parameters. The designer program may import this file, which is then used for the reminder of that session, and will be exported with files that are exported from that moment on.

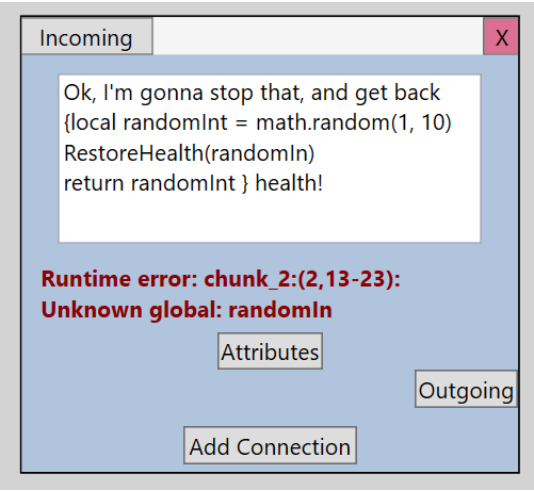


Figure 31 - Dialogue node displaying an error with the embedded script

Critically, this file can then be injected into Lua's global scope in the design program with the globals as defined in the designer. Using these stubs, Lua is able to validate the blocks of logic written by the designer and check if their code will run when imported into Unity, as described in Objective 11 – End Program Stubs. These functions hold no logic or data and will never execute correctly using just the stubs, but Lua will never be run in the designer, only the end program, so stubs are all that is required to validate the code. When the code blocks are edited, if there is an error in the code blocks, then the designer will display the error message under the text field as shown in figure 31.

5 Evaluation

This section evaluates the implementation of this project, how well it met its stated objectives, and the process by which the project was created and managed over time.

5.1 Implementation

This project successfully met all the stated objectives set out in section 2.4, the designer program is able to make comprehensive dialogue trees with conditions and attributes and is easy to use. Unity, as the end program of choice, can use this system for a fully interactive narrative game with separated core and demo specific files.

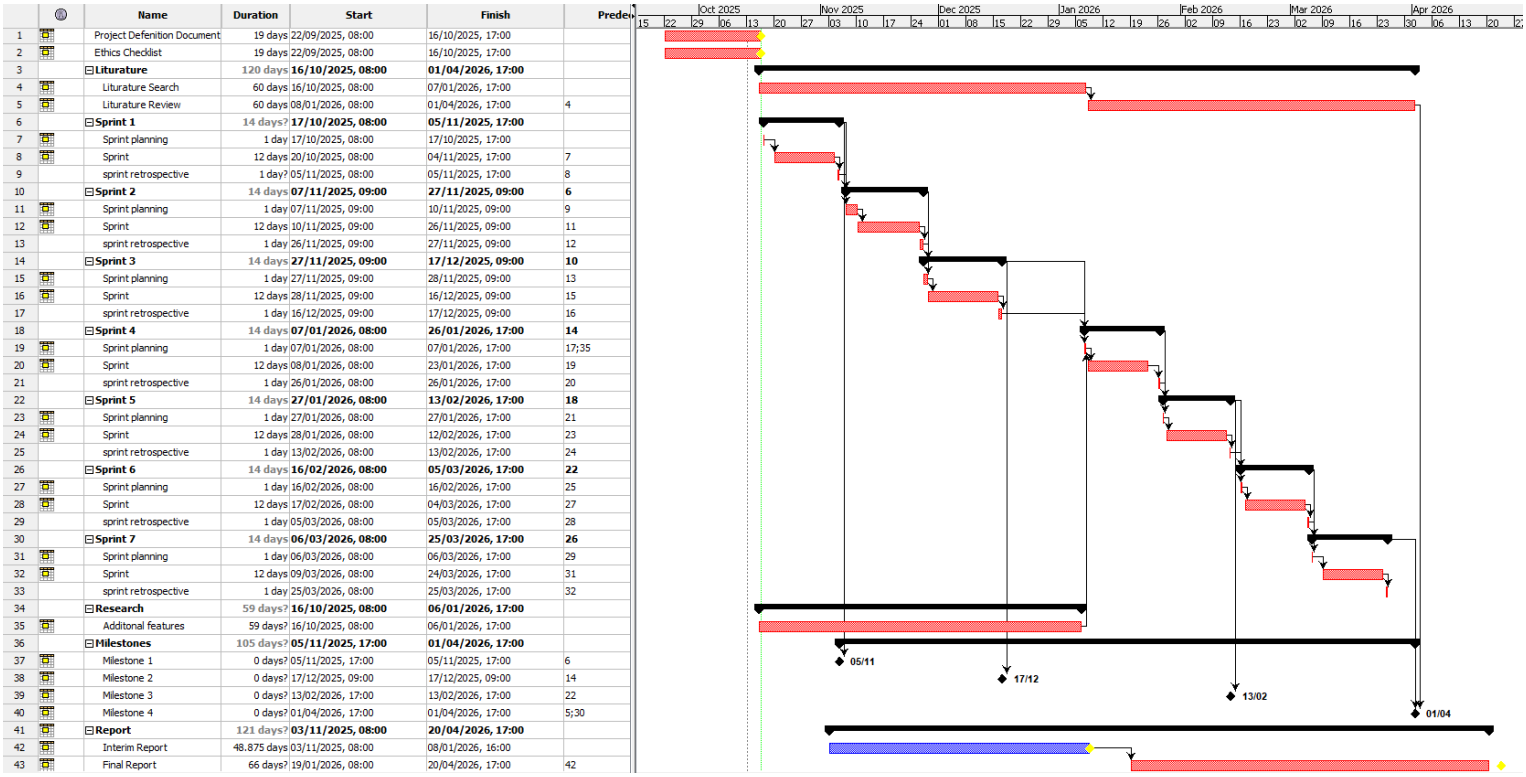
Table 4 - Objective implementation table

ID	Objective	Comments
1	User Interface	The dialogue nodes can be moved and resized, and the UI on the whole looks better than a purely functional piece of software.
2	Node attributes	Everything in the attributes field can technically be done with Lua scripting, but having a concise list of properties makes the software easier to work with. This could be improved by providing confirmation to the designer that their attribute is valid from the Lua stub or providing a dropdown of attribute keys instead of a textbox when a Lua stub has been provided.
3	Node Connections	A handful of small details in how the connections work, that being their bends, the pending connection sticking to the mouse, and the green connections that try to avoid intersecting their own textbox and its incoming connections, all make the UI better to work with than a purely functional design.
4	Connection Components	The connection components, as well as the outgoing connection components, were difficult to implement with MVVM architecture as they are not user controls with an associated .cs class, but rather were resource dictionaries used in the node xaml class and then bound appropriately.
5	Connection Component Attributes	These attributes related to the individual choices, so differ from the attributes in the node, and are stored appropriately to show this. However, as the functions of these two nodes are separate, there is an argument to be made that they should not be counted as the same thing, and that they should be separated in Unity and in the Lua stubs.
6	Connection Component Conditions	These conditions determine whether the choice of this connection component would be shown to the user and is written to work using the same logic as the embedded scripting. So long as what is written in this field returns a Boolean, it is a valid condition.
7	Outgoing Connection Components	These components could be made more intuitive to use for a designer with further work.

8	User Interface Features	This objective only set out to accomplish moving the grid, but implementing this allowed for the grid to be scaled as well.
9	Import and Export	The json file is human readable, and on top of containing the connections and nodes, also contains the Lua stubs associated with those nodes if one is added. When importing and exporting, the file explorer opens to the last location that it was opened to and allows the file to be imported and exported with any name.
10	End Program Importer	When the program is run, the exported file is read and stored to be iterated through, this is done through a unity component that takes parameters that specify the name of the file to be used, as well as other parameters that inform how the dialogue works, such as the text reveal delay, choice buttons, and audio.
11	End Program Stubs	The stubs are exported when the program is run and are attached to the json file when imported into the designer program. The ability to display the error message from Lua in the designer based on the stubs is a notable inclusion for the sake of ease of use.
12	End Program Logic	Unity and the designer program use an interpreter design pattern to construct executable code out of what is placed in the text field.

5.2 Process and Management

The development of this project followed an agile methodology that consisted of sprints and review. When planning this project, the stated objective was to break down work on this project into set sprints over fixed amounts of time as outlined in figure 32 from the project definition document, this is not the process that ended up being followed. Instead, the process of developing this project occurred in brief sprints of feature development, followed by longer periods of bug fixing and work on other projects.



	Name	Duration	Start	Finish	Predec
1	Project Defenition Document	19 days	22/09/2025, 08:00	16/10/2025, 17:00	
2	Ethics Checklist	19 days	22/09/2025, 08:00	16/10/2025, 17:00	
3	**Liturature**	120 days	**16/10/2025, 08:00**	**01/04/2026, 17:00**	
4	Liturature Search	60 days	16/10/2025, 08:00	07/01/2026, 17:00	
5	Liturature Review	60 days	08/01/2026, 08:00	01/04/2026, 17:00	4
6	**Sprint 1**	14 days?	**17/10/2025, 08:00**	**05/11/2025, 17:00**	
7	Sprint planning	1 day	17/10/2025, 08:00	17/10/2025, 17:00	
8	Sprint	12 days	20/10/2025, 08:00	04/11/2025, 17:00	7
9	sprint retrospective	1 day?	05/11/2025, 08:00	05/11/2025, 17:00	8
10	**Sprint 2**	14 days	**07/11/2025, 09:00**	**27/11/2025, 09:00**	**6**
11	Sprint planning	1 day	07/11/2025, 09:00	10/11/2025, 09:00	9
12	Sprint	12 days	10/11/2025, 09:00	26/11/2025, 09:00	11
13	sprint retrospective	1 day	26/11/2025, 09:00	27/11/2025, 09:00	12
14	**Sprint 3**	14 days	**27/11/2025, 09:00**	**17/12/2025, 09:00**	**10**
15	Sprint planning	1 day	27/11/2025, 09:00	28/11/2025, 09:00	13
16	Sprint	12 days	28/11/2025, 09:00	16/12/2025, 09:00	15
17	sprint retrospective	1 day	16/12/2025, 09:00	17/12/2025, 09:00	16
18	**Sprint 4**	14 days	**07/01/2026, 08:00**	**26/01/2026, 17:00**	**14**
19	Sprint planning	1 day	07/01/2026, 08:00	07/01/2026, 17:00	17;35
20	Sprint	12 days	08/01/2026, 08:00	23/01/2026, 17:00	19
21	sprint retrospective	1 day	26/01/2026, 08:00	26/01/2026, 17:00	20
22	**Sprint 5**	14 days	**27/01/2026, 08:00**	**13/02/2026, 17:00**	**18**
23	Sprint planning	1 day	27/01/2026, 08:00	27/01/2026, 17:00	21
24	Sprint	12 days	28/01/2026, 08:00	12/02/2026, 17:00	23
25	sprint retrospective	1 day	13/02/2026, 08:00	13/02/2026, 17:00	24
26	**Sprint 6**	14 days	**16/02/2026, 08:00**	**05/03/2026, 17:00**	**22**
27	Sprint planning	1 day	16/02/2026, 08:00	16/02/2026, 17:00	25
28	Sprint	12 days	17/02/2026, 08:00	04/03/2026, 17:00	27
29	sprint retrospective	1 day	05/03/2026, 08:00	05/03/2026, 17:00	28
30	**Sprint 7**	14 days	**06/03/2026, 08:00**	**25/03/2026, 17:00**	**26**
31	Sprint planning	1 day	06/03/2026, 08:00	06/03/2026, 17:00	29
32	Sprint	12 days	09/03/2026, 08:00	24/03/2026, 17:00	31
33	sprint retrospective	1 day	25/03/2026, 08:00	25/03/2026, 17:00	32
34	**Research**	59 days?	**16/10/2025, 08:00**	**06/01/2026, 17:00**	
35	Additonal features	59 days?	16/10/2025, 08:00	06/01/2026, 17:00	
36	**Milestones**	105 days?	**05/11/2025, 17:00**	**01/04/2026, 17:00**	
37	Milestone 1	0 days?	05/11/2025, 17:00	05/11/2025, 17:00	6
38	Milestone 2	0 days?	17/12/2025, 09:00	17/12/2025, 09:00	14
39	Milestone 3	0 days?	13/02/2026, 17:00	13/02/2026, 17:00	22
40	Milestone 4	0 days?	01/04/2026, 17:00	01/04/2026, 17:00	5;30
41	**Report**	121 days?	**03/11/2025, 08:00**	**20/04/2026, 17:00**	
42	Interim Report	48.875 days	03/11/2025, 08:00	08/01/2026, 16:00	
43	Final Report	66 days?	19/01/2026, 08:00	20/04/2026, 17:00	42

Figure 32 - Gantt chart plan for development

Between these sprints, I had regular meetings with my project supervisor, which allowed me to show what I had developed and seek feedback which I would then act upon. These sprints were irregular and not what I had originally described in my project definition document, but this methodology worked well for this project, as the time it took to develop features and bug fix the program were unpredictable.

This project has also been regularly pushed to github, to serve as not only an external backup, but also a way of viewing previous iterations of the project and viewing overall development.

5.3 Further work

This project would benefit from many quality-of-use features in the designer, as the primary issue is use and readability. This would be done by having collapsable sections where clusters of nodes could be contained and hidden when not being actively used, which would help mitigate the issue of readability due to graph complexity at large scales.

Additionally, a way to dictate default attributes that the designer wishes to include in all nodes, as well as node copy and paste functionality, would help streamline the use of this tool. As each individual file is it's own section of dialogue, many exported files could be used in an end system for different conversations, therefore quick access to recently

edited files, as well as a way to group dialogue files together and rapidly change between working on this dialogue, would streamline the designers use of this tool.

This project would benefit from language translation tools, so that text boxes could contain different text fields for different languages when selected. This could allow a developer to easily translate text they've already written into another language by having immediate access to the original text in context, or allow the dialogue to be written in multiple languages at once if those languages were selected at the start of the project.

The dialogue nodes would benefit from various forms of feedback when being used, such as a way of indicating when an attribute key is valid for the imported Lua stub, or having the text field notify the designer of valid and invalid dollar-prefixed Lua. Similarly, developing this program further would allow for more ways of integrating the program with Unity, such as text formatting.

6 Conclusion

To conclude, this project serves as a fit to purpose tool that a designer could use to design dialogue and dialogue branches for a game, as well as a bridge to allow the files exported by the designer to work in Unity. Additionally, this project was well researched, with all critical decisions being decided in the literature review and informing development.

This project met all of its stated objectives, and as a result this system is feature complete, able to create any dialogue to fit all standard branching methods used in narrative games. If a narrative text-heavy game were to be made in unity, this would be a valid choice for creating the dialogue.

Despite this, the tool is simple and would benefit from the inclusion of many quality-of-life additions, as well as additional features outlined in section 5.3, that while add complexity, would ultimately make dialogue quicker and easier to use at larger scales. The designer program has been written with architecture that would allow adding additional features to be easy to anyone familiar with WPF and given more development time could be included.

With more development time the Unity part of the project could also be developed further. Presently Unity shows a visual novel style game and the core files support this, but the core files could potentially also include a way to easily add dialogue to a preexisting game.

7 References

adegeo (2022) *What is Windows Presentation Foundation - WPF*. https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/ [Accessed 22 Apr 2026].

Adjacency List Representation (2025) *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/adjacency-list-meaning-definition-in-dsa/ [Accessed 21 Apr 2026].

Adjacency Matrix Representation - GeeksforGeeks (2025). https://www.geeksforgeeks.org/dsa/adjacency-matrix/ [Accessed 21 Apr 2026].

Albahari, J. (2023) *C# 12 in a nutshell: the definitive reference*. First edition, O'Reilly Media. (In a nutshell)

baeldung (2021) *Difference Between MVC and MVP Patterns | Baeldung*. https://www.baeldung.com/mvc-vs-mvp-pattern [Accessed 11 Apr 2026].

Bourhis, P., Reutter, J.L., Suárez, F. & Vrgoč, D. (2017) JSON: data model, query languages and schema specification., arXiv

Bray, T. (ed.) (2017) *The JavaScript Object Notation (JSON) Data Interchange Format*, (RFC8259). RFC Editor. RFC8259.

C++ Programming Language (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/cpp/c-plus-plus/ [Accessed 2 Apr 2026].

C++ vs C# (2025) *GeeksforGeeks*. https://www.geeksforgeeks.org/c-sharp/c-vs-c-sharp/ [Accessed 22 Apr 2026].

C# Tutorial (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/c-sharp/csharp-programming-language/ [Accessed 22 Apr 2026].

Chapel (2026) ChapelR/twine-resources.

Crudu, V. (2025) *Understanding the WPF Rendering Process - Key Questions Every Developer Should Ask*. https://moldstud.com/articles/p-understanding-the-wpf-rendering-process-key-questions-every-developer-should-ask [Accessed 20 Apr 2026].

Detroit: Become Human | Official Site | Quantic Dream (2025). https://www.quanticdream.com/en/detroit-become-human [Accessed 17 Feb 2026].

Disco Elysium. (2014) *Articy*. https://www.articy.com/en/showcase/disco-elysium/ [Accessed 12 Feb 2026].

Disco Elysium (2015). https://discowp.zaumstudio.com [Accessed 12 Feb 2026].

Facade Method Design Pattern (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/system-design/facade-design-pattern-introduction/ [Accessed 22 Apr 2026].

Facade Pattern: Simplifying Complex Subsystems with Design Patterns (2024) *Software Patterns Lexicon*. https://softwarepatternslexicon.com/mastering-design-patterns/structural-design-patterns/facade-pattern/ [Accessed 22 Apr 2026].

Features of articy:draft X – Narrative Design Tool for interactive projects (2014) *Articy*. https://www.articy.com/en/articydraft/feature-list/ [Accessed 6 Jan 2026].

Ferry // Nopanamaman (2025) Z.A.T.O. // I Love the World and Everything In It [Video Game]., Ferry // Nopanamaman. https://store.steampowered.com/app/4122860/ZATO__I_Love_the_World_and_Everything_In_It/

Foord, M.J. (2008) *IronPython in action*, Manning

Fuksa, M., Speth, S. & Becker, S. (2025) MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern. In. 163–181.

Gamma, E. (ed.) (1995) *Design patterns: elements of reusable object-oriented software*, Addison-Wesley. (Addison-Wesley professional computing series)

General Python FAQ (2026) *Python Documentation*. https://docs.python.org/3/faq/general.html [Accessed 20 Apr 2026].

Good, O.S. (2017) *Detroit: Become Human's script is 2,000 pages long Polygon.Com*. https://www.polygon.com/e3/2017/6/14/15803834/detroit-become-humans-e3-2017-script-david-cage/ [Accessed 11 Feb 2026].

Ibraimi, A., Memeti, A. & Idrizi, F. (2025) MVC VS MVVM VS MVP: WHICH ARCHITECTURAL PATTERN SUITS MODERN APPLICATIONS BEST?. *JNSM - Journal of Natural Sciences and Mathematics of UT*. 10(19–20), 306–311. https://doi.org/10.62792/ut.jnsm.v10.i19-20.p3196

Ierusalimschy, R. (2003) *Programming in Lua*, Roberto Ierusalimschy

Ierusalimschy, R., De Figueiredo, L.H. & Celes, W. (2007) The evolution of Lua. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, *HOPL-III '07: ACM SIGPLAN History of Programming Languages Conference III*, ACM. https://doi.org/10.1145/1238844.1238846

Inklewriter (2019). https://www.inklestudios.com/inklewriter/ [Accessed 13 Dec 2025].

Interpreter Design Pattern (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/system-design/interpreter-design-pattern/ [Accessed 22 Apr 2026].

Interpreter Pattern in Lua: Mastering Language Processing (2024) *Software Patterns Lexicon*. https://softwarepatternslexicon.com/lua/behavioral-design-patterns-in-lua/interpreter-pattern/ [Accessed 22 Apr 2026].

Introduction to BSON and Types (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/mongodb/what-is-bson/ [Accessed 22 Apr 2026].

Introduction to JavaScript (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/javascript/introduction-to-javascript/ [Accessed 21 Apr 2026].

Introduction to Model View View Model (MVVM) (2023) *GeeksforGeeks*. https://www.geeksforgeeks.org/websites-apps/introduction-to-model-view-view-model-mvvm/ [Accessed 12 Apr 2026].

JSON (2008). https://www.json.org/json-en.html [Accessed 1 Apr 2026].

Kašuba, M. (2015) *Lua game development cookbook: over 70 recipes that will help you master the elements and best practices required to build a modern game engine using Lua*. 1st ed, Packt Publishing. (Quick Answers to Common Problems)

kexugit (2010) *MSDN Magazine: Cutting Edge - Better Web Forms with the MVP Pattern*. https://learn.microsoft.com/en-us/archive/msdn-magazine/2010/september/msdn-magazine-cutting-edge-better-web-forms-with-the-mvp-pattern [Accessed 22 Apr 2026].

Knoblauch, J., Sethuraman, A. & Hey, J. (2017) IMGui—A Desktop GUI Application for Isolation with Migration Analyses. *Molecular Biology and Evolution*. 34(2), 500–504. https://doi.org/10.1093/molbev/msw252

Lee, S. (2025) *Mastering Edge Lists in Graph Algorithms*. https://www.numberanalytics.com/blog/ultimate-guide-to-edge-list-in-graph-algorithms [Accessed 21 Apr 2026].

Lucasfilm Games LLC (1990) The Secret of Monkey Island [Video Game]., Lucasfilm Games LLC. https://archive.org/details/mnkyega [Accessed 1 May 2026]

Many to Many Relationships: A Guide to Database Design (2026). https://www.datacamp.com/blog/many-to-many-relationship [Accessed 21 Apr 2026].

Mastering the Observer Pattern in Lua: Implementing Publish/Subscribe for Event Notification (2024) *Software Patterns Lexicon*. https://softwarepatternslexicon.com/lua/behavioral-design-patterns-in-lua/observer-pattern-publish-subscribe/ [Accessed 3 Mar 2026].

MessagePack: It's like JSON. but fast and small. (2021). https://msgpack.org/ [Accessed 6 Apr 2026].

MessagePack-CSharp/MessagePack-CSharp. (2026), MessagePack-CSharp

MoonSharp (2016). https://www.moonsharp.org/about.html [Accessed 22 Apr 2026].

Mueller, J. (2010) *Professional IronPython*, Wiley Pub. (Wrox professional guides)

MVC Framework Introduction (2025) *GeeksforGeeks*. https://www.geeksforgeeks.org/software-engineering/mvc-framework-introduction/ [Accessed 8 Apr 2026].

MVP (Model View Presenter) Architecture Pattern in Android with Example (2025) *GeeksforGeeks*. https://www.geeksforgeeks.org/android/mvp-model-view-presenter-architecture-pattern-in-android-with-example/ [Accessed 11 Apr 2026].

Observer Design Pattern (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/system-design/observer-pattern-set-1-introduction/ [Accessed 22 Apr 2026].

omar (2026) Ocornut/imgui.

Pawlukiewicz, A. & Nedkov, N. (2024) *Exploring performance issues and patterns in C# by analyzing open-source projects*

PuffballsUnited (2020) The Henry Stickman Collection [Video Game]., Innersloth. https://store.steampowered.com/app/1089980/The_Henry_Stickmin_Collection/

Python | Compiled or Interpreted ? (2025) *GeeksforGeeks*. https://www.geeksforgeeks.org/python/python-compiled-or-interpreted/ [Accessed 20 Apr 2026].

Python Tutorial | Learn Python Programming Language (2026) *GeeksforGeeks*. https://www.geeksforgeeks.org/python/python-programming-language-tutorial/ [Accessed 20 Apr 2026].

Quantic Dream (2018) Detroit: Become Human [Video Game]., Quantic Dream. https://store.steampowered.com/app/1222140/Detroit_Become_Human/

Qureshi, M.R.J. & Sabir, F. (2014) A comparison of model view controller and model view presenter., arXiv

Renger, S. (2022) Investigation into the criteria of embeddability of visual scripting languages within the domain of game development. https://doi.org/10.13140/RG.2.2.11976.39686

Richter, J. (2012) *CLR via C#*. Fourth edition, Microsoft

Rick-Anderson (2023) *BSON Support in ASP.NET Web API 2.1 - ASP.NET 4.x*. https://learn.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/bson-support-in-web-api-21 [Accessed 22 Apr 2026].

Ros, S. (2026) Sebastienros/jint.

Sells, C., Griffiths, I. & Sells, C. (2007) *Programming WPF*. 2nd ed, O'Reilly

Singh, H. & Sharma, R. (2012) Role of Adjacency Matrix & Adjacency List in Graph Theory. *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY*. 3(1), 179–183. https://doi.org/10.24297/ijct.v3i1c.2775

stevewhims (2022) *Retained Mode Versus Immediate Mode - Win32 apps*. https://learn.microsoft.com/en-us/windows/win32/learnwin32/retained-mode-versus-immediate-mode [Accessed 20 Apr 2026].

Story Formats - Twine Cookbook (2009). https://twinery.org/cookbook/terms/terms_storyformats.html [Accessed 19 Feb 2026].

Stroustrup, B. (2014) *The C++ programming language: C++11*. 4. ed, Addison-Wesley

Team, A. (2023) *Architectural Patterns: MVC, MVP, and MVVM Explained | AppMaster*. https://appmaster.io/blog/architectural-patterns-mvc-mvp-and-mvvm [Accessed 8 Apr 2026].

The Henry Stickmin Collection on Steam (2020). https://store.steampowered.com/app/1089980/The_Henry_Stickmin_Collection/ [Accessed 11 Feb 2026].

The Ren'Py Visual Novel Engine (2004). https://www.renpy.org/ [Accessed 12 Feb 2026].

Toby Fox (2015) Undertale [Video Game]., tobyfox. https://store.steampowered.com/app/391540/Undertale/

Twine / An open-source tool for telling interactive, nonlinear stories (2009). https://twinery.org/ [Accessed 11 Dec 2025].

TypeError: cyclic object value - JavaScript | MDN (2025) *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Cyclic_object_value [Accessed 13 Jan 2026].

UNDERTALE !? (2026) *UNDERTALE*. https://UNDERTALE.com [Accessed 13 Feb 2026].

Useful Extensions · ocornut/imgui Wiki (2026). https://github.com/ocornut/imgui/wiki/Useful-Extensions#node-editors [Accessed 20 Apr 2026].

Viotti, J.C. & Kinderkhedia, M. (2022) *A Survey of JSON-compatible Binary Serialization Specifications arXiv.Org*. https://arxiv.org/abs/2201.02089v2 [Accessed 22 Apr 2026].

Z.A.T.O. // I Love the World and Everything In It on Steam (2025). https://store.steampowered.com/app/4122860/ZATO__I_Love_the_World_and_Everything_In_It/ [Accessed 12 Feb 2026].

ZA/UM (2019) Disco Elysium - The Final Cut [Video Game]., ZA/UM. https://store.steampowered.com/app/632470/Disco_Elysium__The_Final_Cut/.

Zejdová, M. (2025) *Interactive GUI Editor for ImGui with Customizable Styles*, Bakalářská práce. Masarykova univerzita, Fakulta informatiky.