# Procedural Side-Quest Generation Using Graphs in Unity3D

Antonis Danas

January 2021

Electrical And Computer Engineering,
Technical University Of Crete

Thesis Committee
Associate Professor Katerina Mania, Thesis Supervisor
Associate Professor Georgios Yannakakis
Associate Professor Georgios Chalkiadakis

# Acknowledgements

For this thesis, I would like to especially thank Prof. Katerina Mania for providing me the opportunity to work on this idea. I would also like to thank Georgios Yannakakis and Georgios Chalkiadakis for accepting to be on my committee and approved the idea of this side-quest generation tool.

I would also like to thank my colleagues in the MUSIC lab, for their assistance during the implementation of this project. Special thanks to Daskalogrigorakis Grigorios and Shalva Kirakosian for their support. Lastly, I would like to thank my family and friends for their emotional support throughout the my studies and the writing of this thesis.

# Abstract

The field of Procedural Narrative Generation is becoming increasingly more popular amongst the gaming industry for the past two decades. From the AI Director of Left 4 Dead, where the game decides when and where to place enemies, ammo or weapons, depending on the player's situation and skill, to The Elder Scrolls: Skyrim's Radiant AI which -among other things- dynamically generates quests and all the entities necessary for that quest to complete, while also setting that quest on new areas for the player to explore. Although such systems generally use simple methods to generate a sequence of events or quests, like a string where every character represents a specific event, there have also been proposed more complex systems using other methods that can help the designers add more depth in their game.

The purpose of this thesis is to showcase a practical implementation of a procedural side-quest generation system using graphs instead of strings, in the Unity3D game engine. Such a system dynamically generates narrative, and more specifically plot, in the form of side quests. For the implementation of this system, the 2019.4 version of the Unity3D game engine was used as well as the .NET Standard 2.0 framework for C#. By implementing one such system on a widely used game engine, it provides a proof of concept on how such a system would work in a game.

The system contains a game world represented by a graph, the Interactables which represent the actual game instances of the game world, a generator which searches that graph for patterns in order to generate a side-quest by taking into account relationships between the player and the Interactables and an event broker who is handling events between those three components.

# Contents

# 1 Introduction

The topic of interactive storytelling and interactive narrative, has long been of interest in to AI research. In the last 40 - 50 years, there have been multiple papers that were published, describing new ways of creating narratives and even performing sentiment and story analysis on novels [19]. But the need to be able to quantify a story in to distinct parts, has been existing for almost a century.

Even in the early 20th century, there has been an attempt to decompose a story to a set of actions - or functions as Propp described them [13], the thought being that one could take such a story and break it into distinct parts. Propp in his paper took a set of Russian folklore tales and tried to extract a number of rules that could potentially describe a general structure all those said tales. Specifically, he described that every Russian folktale, started with one out of a set of specifically described "functions".

Having all those papers on how to decompose genres of literature, like for example creating a set of "functions" for Eskimo Folktales by Colby [4] or the genre of old French epic [11], eventually caught the attention of AI research. Researchers started thinking those sets of functions as grammars and by using a set of one them, they could create a new story computationally. As Propp was the first one to describe such a grammar, there was also a paper that uses his morphology of the Russian folktale, to generate narrative, by Gervás [5].

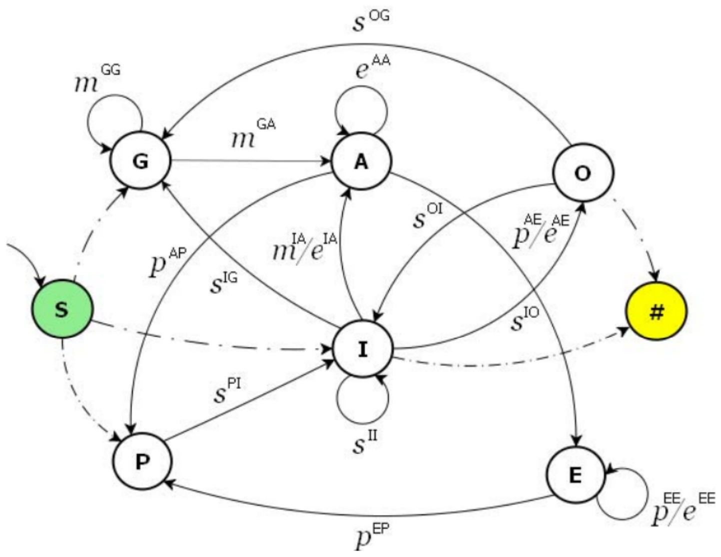


Figure 1: A grammar finite state-machine that represents a plot generation approach by Bui et al. [3]

Following that same trend, a lot of papers have been published regarding a research field called Procedural Narrative Generation. This field, as it will also be described below, is one that focuses on creating - and also analyzing[17] - narratives, using different kinds of approaches, with the latest being with the use of Machine Learning [18], that aim to tailor the experience according to the individual user/reader/player.

## 1.1 Purpose

The focus of this thesis, is to create a world-driven procedural content generation system, using a game engine called Unity3D. The aforementioned system will be able to procedurally generate side quests for a player in an Role Playing Game, depending on relationships between Non-Playable Characters (NPCs) themselves and between the NPCs and the player.

More specifically, the game world is represented by a graph where each node represents a world entity which is called an Interactable, and each edge represents a relationship between those Interactables. In order to generate a quest, the game world is being traversed to find specific patterns of Interactables and their relationships, which are called Rules, that could trigger such a quest. Once a pattern is found, its respective Rule is applied and a new side quest is generated in a form of another graph, which contains all necessary information i.e who the target of the side quest is or which Interactable will invoke said side quest.

Additionally, when the player is taking on a side quest, they should be aware that their actions have consequences in the game world and the relationships between Interactables can change. The system is designed in a way though, that their actions can have consequences regardless of the side quest. While they can affect the frequency and type of what is generated, it is not the quest itself that influences the game world, but solely the actions of the player.

This system resembles the one used in Skyrim, which is a Role Playing Game developed by Bethesda. In their game, Bethesda created a side-quest generation system that produces content for the player on the fly, depending on their actions.

Although, the biggest influence on this thesis has been the ReGEN system proposed by Kybartas and Verbrugge [9]. In their system, Kybartas and Verbrugge used a method of graph re-writing, to generate stories according to relationships existing between nodes on a World Graph. While this approach will be better described on subsequent sections, it is worth mentioning that this method allows the branching of stories, leading to different outcomes and repercussions for the entities residing in that World Graph.

## 1.2 Thesis Structure

In this section of the thesis, an overall structure of the document will be provided, giving a brief description of each section.

In Section 2, an introduction to Procedural Narrative Generation is provided.

Since PNG is a section of a field that is called Procedural Content Generation, we also provide a brief introduction to that as well. We discuss more details on what is considered as narrative, what methods have been used in research to generate different parts of narrative and also give a description on what Skyrim's Radiant Quest System is and what the ReGEN system is.

In Section 3, a description on our approach on generating side-quests in Unity3D is given. Specifically, we describe why we used graphs instead of regular expressions - which is commonly used in plot generation, as we will discuss in the next section. We will also talk more on why and how we used Unity3D as our main game engine.

In Section 4, a description on how the system is used is given in the context of a demo created for the thesis's purposes. We give a description on how a Designer would use the tools created for this project, as well as how a developer could potentially expand it.

In Section 5, a more detailed description is provided on how the whole system is structured. Specifically, a detailed insight on the implementation of each individual component of the project, is shown. Additionally, a detailed explanation is given on how the player interacts with the world around him, how the Quest Scheduler schedules the generation of a quest, but also manages all the active ones, and give a small demo on the system in use.

Finally, in Section 6 an overall conclusion on the project is given. Thoughts on future work are also mentioned in the last part of the conclusion. Specifically, there are details of ways on how the project could be improved in the future in subsequent iterations.

# 2 Procedural Narrative Generation

In the this section, more details on the field of Procedural Narrative Generation are going to be provided. But before we can talk about PNG, we have to talk about what Procedural Content Generation is. So firstly, there is a brief introduction on what PCG is and following that, there are more information provided on what PNG is and what are Skyrim's Radiant Quest system and the ReGEN proposal.

## 2.1 What is Procedural Content Generation (PCG)

The field of Procedural Content Generation is a vast one. PCG refers to generating game content automatically, through algorithmic means [16]. The term game content, though, does not necessarily only mean the graphics of the game. By content we mean all aspects of the game, from graphics - as mentioned - to music, to NPCs' behavior and to even the generation of a story.

One recent example of PCG that promised to provide an almost infinite amount of content for its players, was the game *No Man's Sky* by Hello Games in 2016. That game is an exploration - survival game, that focused on creating an seemingly infinite amount of content for the player to explore. Through the game's procedural generation system, it was possible to create different planets with their own flora, fauna and even different alien species.

Figure 2: An example of the content that, the No Man's Sky procedural generation system, can produce.

No Man's Sky, though being only one example of PCG, can show one great benefit on using these types of algorithms, and that is space. By being able to produce massive amounts of content computationally, game developers can now reduce the memory consumption of their game, by creating content on the fly based on only a few parameters, whether that content may be graphics, game entities or even whole stories - like in this thesis.

## 2.2 What is Procedural Narrative Generation (PNG)

So as mentioned in the previous section, Procedural Content Generation is the process of generating new content computationally, in a game. By using that definition, we can also define Procedural Narrative Generation as the process of creating narrative using algorithms.

### 2.2.1 Brief History

Before we delve more in to what narrative is and how it is used and generated, we will have a brief introduction to the history of this research field. As mentioned in the previous chapters, the first paper to be published and showed the potential of being able to procedurally generate a story, was the one written by Vladimir Propp in 1928 called *Morpholoy of the Folktale.* Ever since his book was translated in 1958 and again in 1968 [13], there has been an increasing amount of interest in the structural analysis of different story genres, like the one by Colby [4].

Eventually, studies appeared that were starting to use those proposed grammars to generate narrative. One such approach was the one by Grasbon and Braun [6], where they used Propp's functions to generate a variety of stories, just from a small database of pre-written sections. One other interesting approach, was the system called BRUTUS by Bringsjord and Ferrucci [1], which could successfully create complex stories but was only geared towards betrayal narratives.

While more and more approaches were using grammars to create narrative, there were other efforts that used AI techniques to generate context. One such example, was the work of Porteous et al. [12] where they used a planning-based system to direct the main story according to the player's actions. Specifically, they took the plot of Shakespear's Merchant of Venice and introduced a character's Point of View, which changed depending on the players actions, which consequently changed their available actions for the next part of the story.

Since we are in an age where machine learning, a lot of research has been made that utilizes the power of ML to generate or analyze narrative. Regarding the generation of narrative, one such example would we the work of Wang et al. [18], where they used Deep Reinforcement Learning, in conjunction with a set of adaptable events, to generate a form of personalized narrative depending on the interaction between the player and the system.

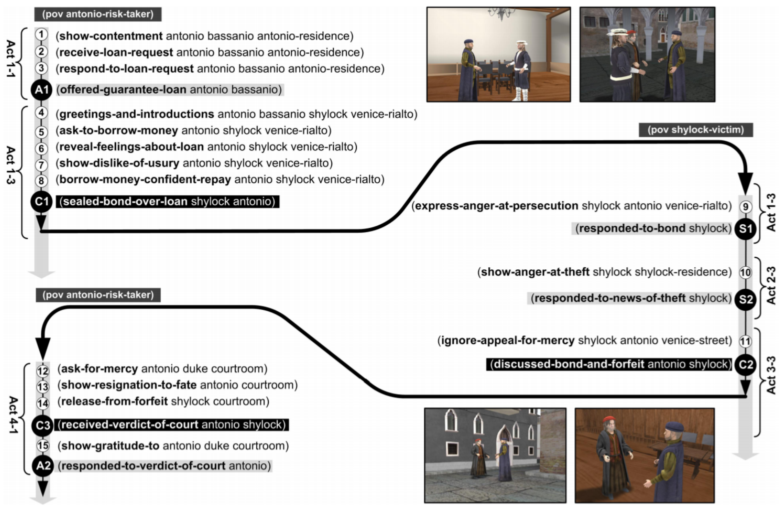

(pov antonio-risk-taker)
Act 1-1
1 (show-contentment antonio bassanio antonio-residence)
2 (receive-loan-request antonio bassanio antonio-residence)
3 (respond-to-loan-request antonio bassanio antonio-residence)
A1 (offered-guarantee-loan antonio bassanio)
Act 1-3
4 (greetings-and-introductions antonio bassanio shylock venice-rialto)
5 (ask-to-borrow-money antonio shylock venice-rialto)
6 (reveal-feelings-about-loan antonio shylock venice-rialto)
7 (show-dislike-of-usury antonio shylock venice-rialto)
8 (borrow-money-confident-repay antonio shylock venice-rialto)
C1 (sealed-bond-over-loan shylock antonio)
(pov shylock-victim)
Act 1-3
(express-anger-at-persecution shylock antonio venice-rialto) 9
(responded-to-bond shylock) S1
Act 2-3
(show-anger-at-theft shylock shylock-residence) 10
(responded-to-news-of-theft shylock) S2
Act 3-3
(ignore-appeal-for-mercy shylock antonio venice-street) 11
(discussed-bond-and-forfeit antonio shylock) C2
(pov antonio-risk-taker)
Act 4-1
12 (ask-for-mercy antonio duke courtroom)
13 (show-resignation-to-fate antonio courtroom)
14 (release-from-forfeit shylock courtroom)
C3 (received-verdict-of-court antonio shylock)
15 (show-gratitude-to antonio duke courtroom)
A2 (responded-to-verdict-of-court antonio)

Figure 3: An example from the Porteous et al. [12] publication, where the PoV changes from (pov antonio-risk-taker) to (pov shylockvictim) and back to (pov antonio-risk-taker).

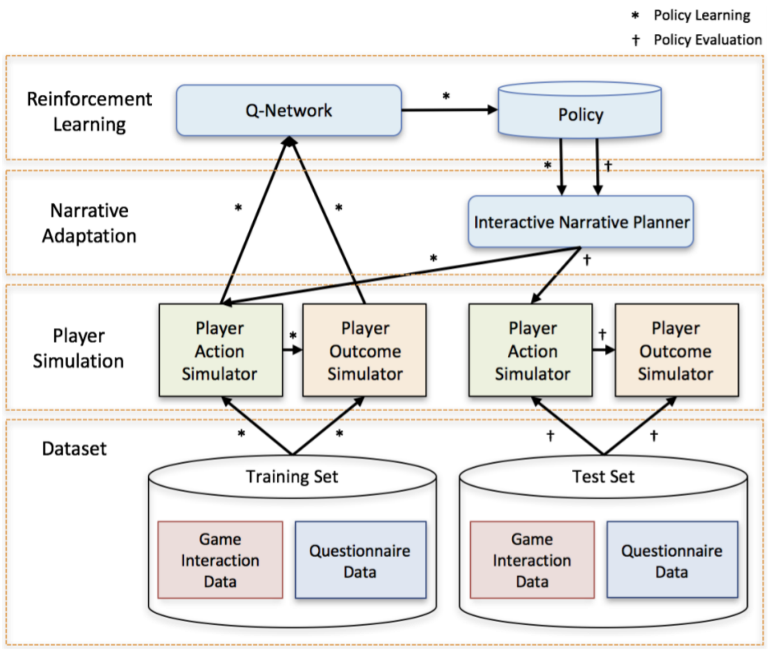


Figure 4: The Reinforcement Learning framework that was used in Wang et al. [18].

### 2.2.2 What is Narrative

Up until now, we have talked about narrative without really giving an explanation to what it really means. In their survey, Kybartas and Bidarra [8] define narrative as the union of what they call a **story**, which is then also split into two sub-parts **plot** and **space**, and finally **discourse**.

Starting with what a **story** is, it is defined as the conjunction of what we call **plot** and **space**. As a plot we define a sequence of story events, i.e. someone was

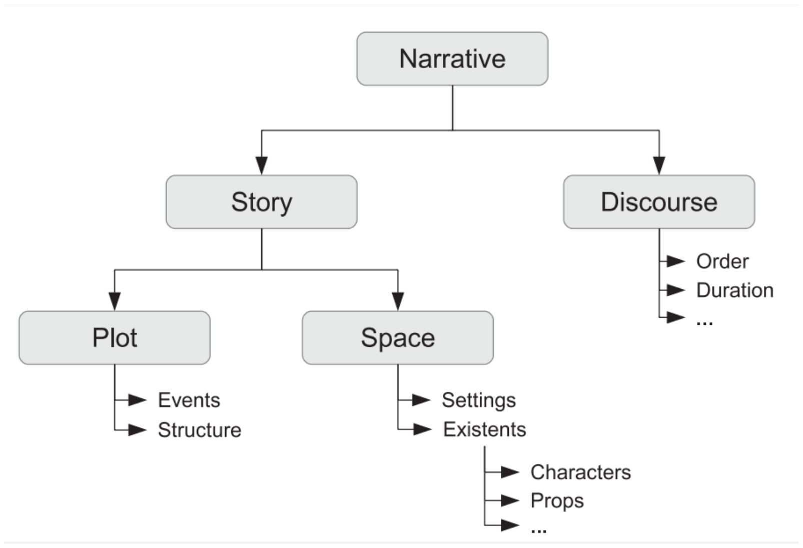


Figure 5: The narrative structure as proposed by Kybartas and Bidarra [8].

murdered, but in a chronological order. So a plot is essentially what happens in a story from start to finish, in that order. Space is then defined as the collection of entities that exist in a story. By entities we mean all the objects that exist in there, all the locations that are available to the story and most importantly, all the characters that exist. Combining then those two elements of the narrative, we make a story.

But a good narrative is not just a chronological view of all the events that happened with a collection of characters, items and places. Most notably, in movies, especially in crime ones, we see a lot of flashbacks showing past events, hidden characters that reveal themselves at the right time and all those elements that make the narrative more ”enjoyable” to the receiver. That part is defined as **discourse**. Discourse, is the way the story is presented to the audience. The way the plot is transformed from a chronological series of events to a more compelling form, and also the way the space is presented to the audience.

### 2.2.3 Fields of Research in PNG

Having defined what narrative and its sub-parts are, in their survey Kybartas and Bidarra, present the fields of research according to the structure defined in the previous section. They show a matrix that separates all those fields depending on their level of automation and what they try and generate, may it be plot, space or both etc. Their matrix is show in Figure 6.

If we take into account the sections defined in Figure 6, we can then differentiate between four distinct fields of research, Manual Authoring, Space Generation, Plot Generation and finally Story Generation. As Manual Authoring does not contain any automation in the plot or space generation, as the name suggests, only a brief description of it will be given here. A more detailed description of the rest of those

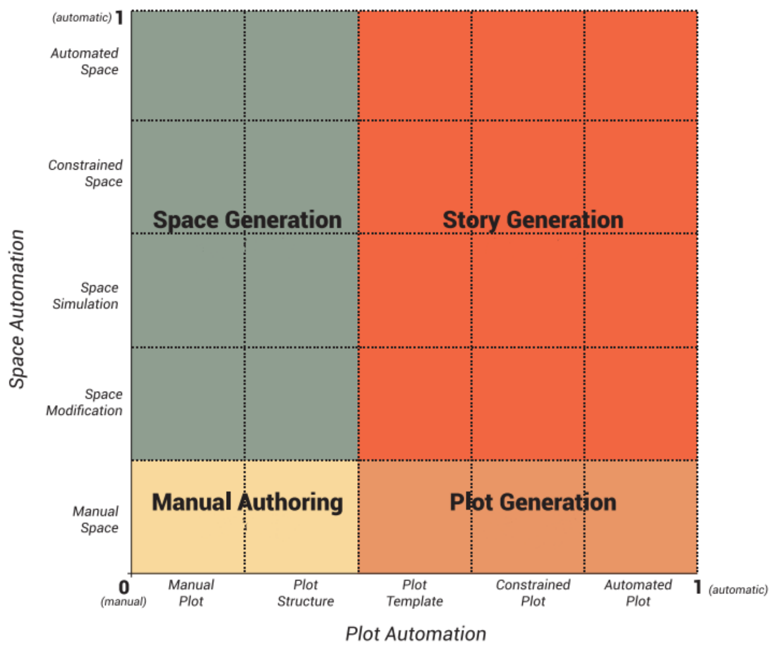


Figure 6: Different fields of story generation, meaning both plot and space generation, in a range from manual to fully automated.

fields, will be provided in subsequent sections.

Manual Authoring tools, are basically tools that help their users generate a story, mostly in a visual manner. They do not use any automation techniques, regarding the plot or space generation, and they just use the information provided by the user and help them create a form of narrative in a user-friendly way.

One such example of manual story authoring is in what is called Narrative Exposure Therapy which uses tools that help patients recreate their memories as narrative, as part of their therapy.

## 2.3 Space Generation

Space Generation refers to the process of creating the world entities of a story, which are, as mentioned in previous sections, all the characters, locations, objects etc. that are used in that story. It is important to point out that, by space generations, we do not mean the process of creating the actual graphics of those entities, rather than their representation in the world in which the story is taking place.

In the space generation process, a PNG system would take as input a predetermined story, and generate a virtual world that the player can play that story on. One such notable example is a system called Game Forge by Hartsook et al.. Game Forge uses plot points, which are high-level representations of events that happen in a specified time period and location. Then in conjunction with designer and player preferences, it uses a Genetic Algorithm to generate a believable game world, that matches those preferences.

## 2.4 Plot Generation

Plot Generation refers to the process of creating a sequence of events in a chronological manner. Plot generators use as input a pre-authored space and/or a fabula that is given to them by the user. Fabula, is defined as the more abstract layer of events that happen within a story. Then we can also define a plot as a subset of the fabula.

One such system that takes a fabula as input and generates a story, is the one proposed by Bui et al.. Their system takes the fabula of the famous fairy tale Little Red Riding Hood, and uses a regular grammar to produce several plots, which then finally manipulates through a genetic algorithm to generate a final version of that plot. One interesting fact about that system is that, since it is using the fabula of the Little Red Riding Hood, which are all the events that occur in that fairy tale, they were able to produce that same story from a different perspective, like the one of the wolf.

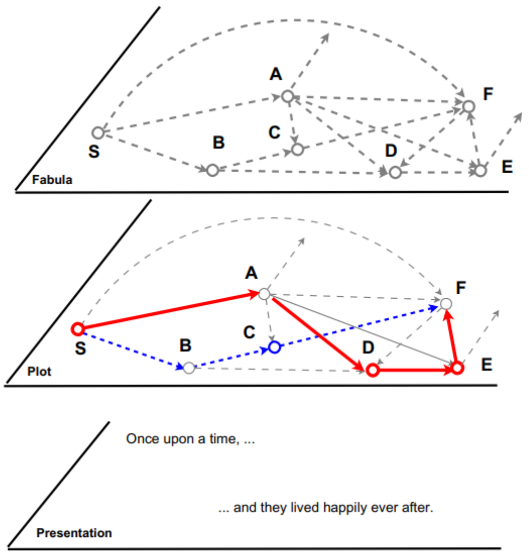


Figure 7: Plot generated by fabula which is then represented to the audience, by Bui et al. [3].

## 2.5 Story Generation

Story Generation is the automated process of creating both Space and Plot, creating a final output of a narrative. As stated by Kybartas and Bidarra [8] in their survey, approaches on generating a story will either try and create a plot based on a given space, which they will also try to modify to fit the plot's needs, or generate space elements that could be used to generate a plot. In other words, in the former approach the plot modifies the space elements, and in the latter, the space elements influence the generation of plot.

One example of the first approach, is the emergent *Virtual Storyteller* system by Swartjes et al. [15]. In their system they use a technique called *late commitment*, in which character agents do not generate plot on a predetermined space, but rather commit elements in the process. As they describe in their paper, it's like actors trying to improvise a play, where they do not work on an agreed upon story world, they rather figure it out on the way. If those character agents, set a goal for themselves - which is determined by what they describe as in character (IC) decisions based on a set of rules - the system will try and generate the world elements necessary for the agents to complete those goals.

On the second approach, in their paper Li and Riedl [10] present a plan-based system that is capable of creating what they call gadgets, to fill the gaps in an existing world space. At first, the system plans a plot that consists of actions. Each action then, has temporal and causal links between them. The plot is then generated by taking into account an initial state of the world and a desired goal. Each action then has certain preconditions undefined, so that it could potentially generate the necessary objects - gadgets - for the agents to reach their goal.

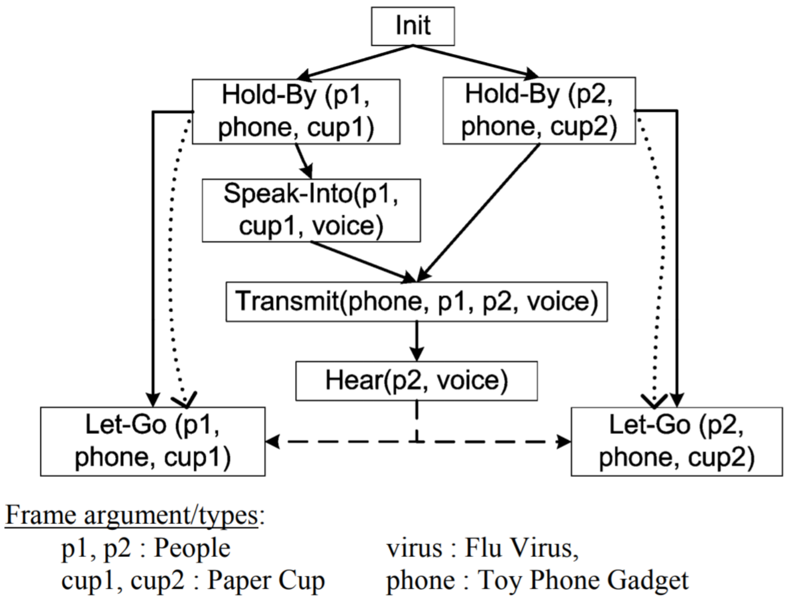


Figure 8: Usage frame of a toy phone, as shown in their gadget generation framework of Li and Riedl [10].

## 2.6 Skyrim's Radiant Quests

The Elder Scrolls V: Skyrim, is an action role playing game created by Bethesda Game Studios in 2011, where the player, called the Dragonborn, is set to defeat Alduin the World-Eater, a dragon who is prophesied to destroy the world. This is an open world game in which the player, other than the main quest, can complete side-quests. Those quests can be **Radiant Quests**, which are procedurally generated through its Radiant AI system.

Figure 9: Gameplay screenshot of The Elder Scrolls V: Skyrim by Bethesda Game Studios.

The Radiant AI system, as described by The Elder Scroll Wiki page, is the dynamic reaction to the player's actions by both characters and the game world in The Elder Scrolls V: Skyrim. Basically what this means is that, depending on the player's actions there is a dynamic reaction from the world around him. For example, if the player commits a crime a against one NPC, they might send assassins as retaliation.

Skyrim's Radiant AI can also produce procedurally generated quests, called **Radiant Quests**. The concept here is that, given a set of pre-authored plots, the Radiant system can pick one and generate its targets on the spot. For example, if the system selects a quest where the player has to free a prisoner, the game will then procedurally generate that character and use that as a target for the quest. One equally important aspect of the generation process, is the fact that the system tries to take into account the locations that the player has not yet explored, and guide those quest into taking place in those places. That process can be categorized as space generation, as it already has a predetermined plot template, so it generates all the characters, objects and then places the target location to a previously unexplored location, all at runtime.

## 2.7 ReGEN system by Kybartas and Verbrugge

Following the concept of Radiant Quests, Kybartas and Verbrugge [9] implemented a similar system, which they called REwriting Graphs for Enhanced Narratives, or ReGEN. In their paper, they describe a system that given a predetermined set of plot rules and a graph that represents the world the player resides in, like all the NPCs, locations etc., and all their relationships, and generates a side-quest that takes into account all those parameters. Also, upon completion, the quest can have an impact on the world, meaning it can dynamically change relationships between world entities.

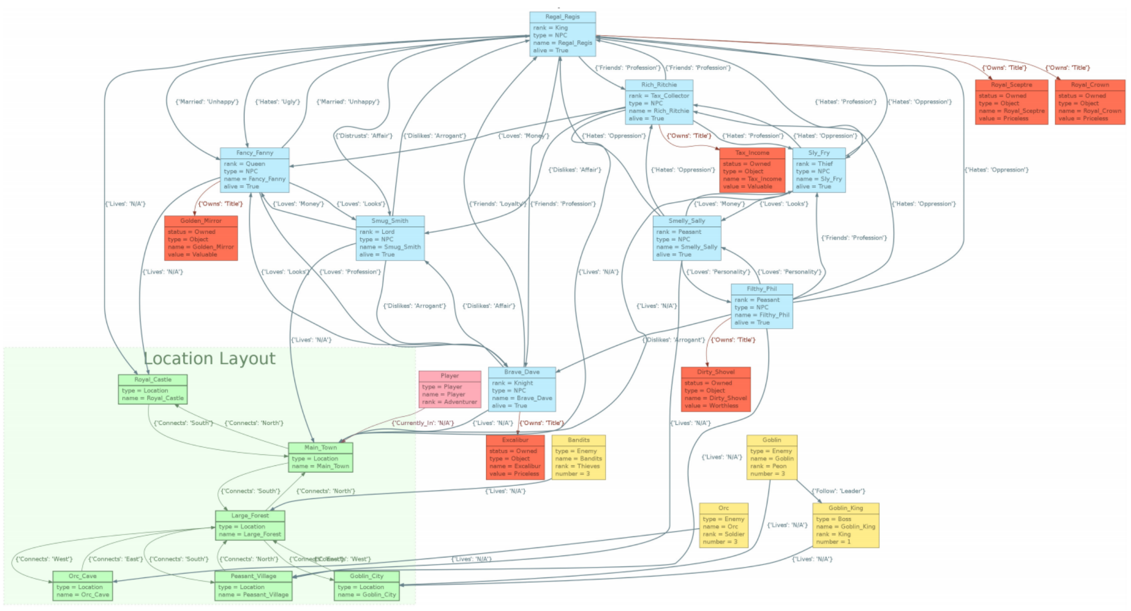


Figure 10: An example of a world graph used in ReGEN.

As mentioned above, the ReGEN system uses a set of rules in order to generate the side-quests. Those rules are basically patterns that are searched in the world graph. That pattern is called an Initial Rewrite Rule (IRR). When it finds a path that has said IRR, then it produces a plot graph, which is a graph that has all the events that will occur in the side-quest, and fills that graph with the entities that it found during the IRR search.

When ReGEN produces a plot graph, it tries to make it more "interesting" by trying to apply another set of rules, called Secondary Rewrite Rules (SRR). Contrary to IRR, SRR is taking into account both world conditions, meaning the relationships between world entities, and narrative condition, meaning what was generated as plot. What those sets of rules are trying to do is, to introduce some sort of variance in the resulting quest, by adding branches. Each branch then can have a different impact on the world depending on the player's actions. The SRR then re-writes the story graph and generates a new one, hence the graph re-writing property that is mentioned in the system's name. An example of this overall procedure is shown in Figure 11.

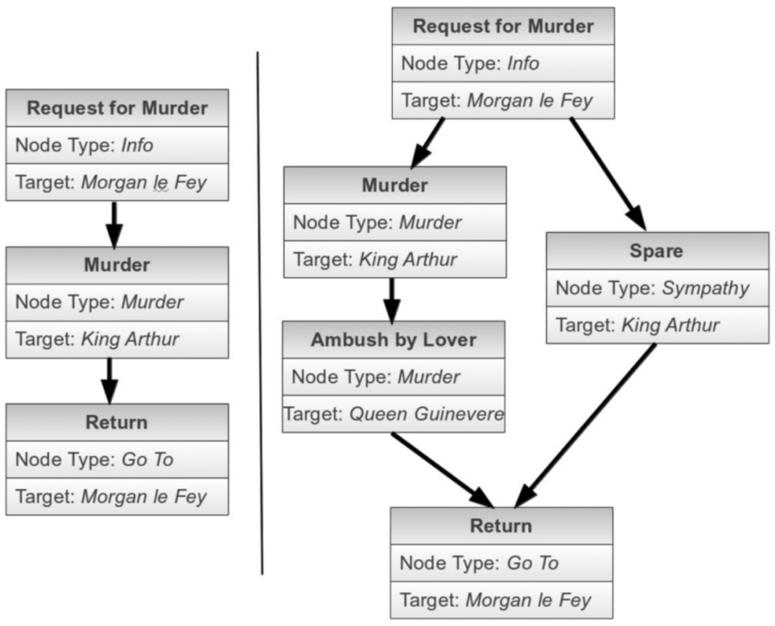


Figure 11: An example of story graph re-writing. On the left we see the plot generated by the system after the use of IRR. On the right we see the final output after implementing an SRR.

# 3 Procedural Side-Quest Generation in Unity3D

Having made a brief introduction to the field of Procedural Narrative Generation and having also discussed more on the research that has been done in said field, this section is dedicated to presenting the key concepts of this thesis.

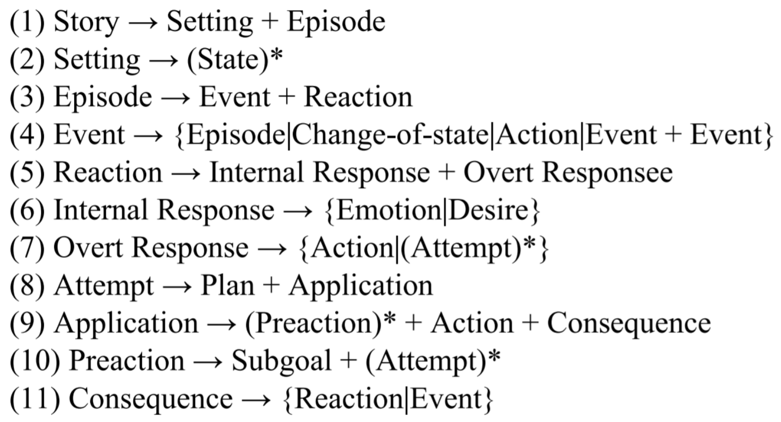
Specifically, we shall provide a more detailed analysis on the graphs and graph searching that were used in the implementation of this thesis and compare their use to the more usual approach of strings. It will also be discussed why the use of this method could potentially provide a better approach on the dynamic change in the relationships of the entities that reside a space world.

Finally, we shall discuss more about the Unity3D platform that was used in the implementation of the thesis. A brief description of the Unity's RPG template, will also be provided, as it was utilized in order to create a demo scene for the demonstration of the proposed system.

## 3.1 Using Graphs instead or Strings

Most early work in Procedural Narrative Generation, has been using a string representation of plot events. Since Propp in his work used a form of grammar to represent events that occur in a story, other research have generated and in turn used such techniques to generate a story in a form of a string.

Specifically, taking as an example Figure 12 we can see a set of rules for a grammar. If we use that set of rules we can then produce a string where each of its symbols represent an event in the plot of a narrative. This procedure is fast and the plot's representation is quite simple, easily understandable and it does not require any significant amount of storage, so it is a good candidate to be used in a game as they need data that can be accessed fast.



(1) Story → Setting + Episode
(2) Setting → (State)*
(3) Episode → Event + Reaction
(4) Event → {Episode|Change-of-state|Action|Event + Event}
(5) Reaction → Internal Response + Overt Responsee
(6) Internal Response → {Emotion|Desire}
(7) Overt Response → {Action|(Attempt)*}
(8) Attempt → Plan + Application
(9) Application → (Preaction)* + Action + Consequence
(10) Preaction → Subgoal + (Attempt)*
(11) Consequence → {Reaction|Event}

Figure 12: Rules of Rumelhart's story grammar in his paper [14].

While a string has all those advantages described above, it also has some drawbacks. The biggest one of them is the level of detail they can provide to the system that utilizes them. As it is previously described, each symbol in a string represents an event. But that symbol does not contain any other information like for example the entities that participate in said event. If for example, we have an event where one has to steal an object from a character at a specific location, then there is no way of passing that information through a string. That will require a different type of system to import those details later.

Having considered that disadvantage, among other ones as well, in their paper on the ReGEN system, Kybartas and Verbrugge propose the use of graphs in order to represent a sequence of events in as much detail as we'd like. This thesis, which is heavily influenced by their work, also uses graphs to represent the sequence of events that happen in a generated side-quest.

The use of graphs has many advantages. Since each node in a story graph, which is the side-quest generated, can contain any amount of information we want, then we can include all the targets that are impacted by that side-quest. In addition, we can take into advantage the same flexibility, while using an IRR while searching the world graph for potential quests. Those patterns can have any amount of information, making them significantly flexible in the level of detail each IRR can have.

On the other hand, while the use of graphs has many advantages, it also comes with some drawbacks. Fist of all, in order to search a world graph and then generate an output graph, there has to be a complex system that can handle all those types of operations. In addition, the world graph and the rule graphs are also much more memory consuming in comparison with the string use case. Last but not least, searching a big world graph can be resource and time consuming for the system, which is not optimal when used in games.

## 3.2 On Dynamically Changing NPCs' relationships

Both Radiant AI and ReGEN systems introduce some kind of dynamic change between entities that reside in their respective worlds. In Radiant AI, depending on the player's actions NPCs around them may act differently than usual.

For example, when a player has high stats in his pickpocketing skills, guards will be more cautious when the player is present. Also, when the player slays a dragon, then the nearby residents will start recognizing him.

While Radiant AI is a very sophisticated system where the player is actively interacting with the world around him, when it generates a new side-quest it does not take into account in-game relationships. It generates the content needed for the side-quest to be playable, on the spot. Also, upon completion of the side-quest, there is no real impact in the world.

The ReGEN system on the other hand, relies on in-game relationships between all entities. In order to generate a new quest, the system must firstly search for patterns in those relationships, and produce a respective side-quest. In addition, upon side-quest's completion, there is an impact in the relationships between the entities of the game. For example, if in one side-quest, the player has to kill an NPC, then all NPCs that previously like that NPC will then start hating the player.

In this thesis, the proposed system is trying to combine both the ReGEN system and the Radiant AI. More specifically, it uses a world graph to represent the relationships between entities in a pre-authored world, and creates side-quests from a set of rules which essentially describe a relationship pattern to be searched in that world. Upon completion of the quest, the relationships in that world change as the rules dictate. One key difference from the ReGEN system, is that those relationships can change from the actions of the player **regardless** of whether he is in a quest or not. For example, if the player kills an NPC then other NPCs will start hating the player - and consequently give him much less quests in the future - even if that player was not currently on a quest.

## 3.3 Using Unity3D

Unity3D is a cross-platform game development engine created by Unity Technologies and released in June 2005. Over the years Unity has increased in reputation, being amongst the top used game engines alongside with Unreal Engine and GameMaker. It is mostly used by Indie developers, but it also has some major games to showcase its potential, like the popular game Hearthstone created by Blizzard and Ori and the Blind Forest, published by Microsoft.

Unity supports both three-dimensional and two-dimensional game development in a variety of platforms like Android, Windows, iOS etc. It also supports virtual reality and augmented reality which is being used more and more in serious games and industrial applications.

Figure 13: Gameplay screenshot of the game Ori and the Blind Forrest.

### 3.3.1 Use of platform

For this thesis the Unity3D engine has been used to demonstrate the use of the proposed system. Unity has a simple user interface, a lot of training material on streaming platforms like YouTube and in their own Unity Learn website.

Additionally, Unity's latest stable scripting runtime supports two different profiles: .NET Standard 2.0 and .NET 4.x. That means it supports the latest updates on the C# programming language and also incorporates Unity's own scripting API.

While Unity also provides an asset store with tools that help developers quickly prototype their games, with some tools even requiring no programming, such tools have not been used in this thesis, as there would be less control of the underlying system. Although, it should be mentioned that some assets were acquired in order to create a basic game user interface.

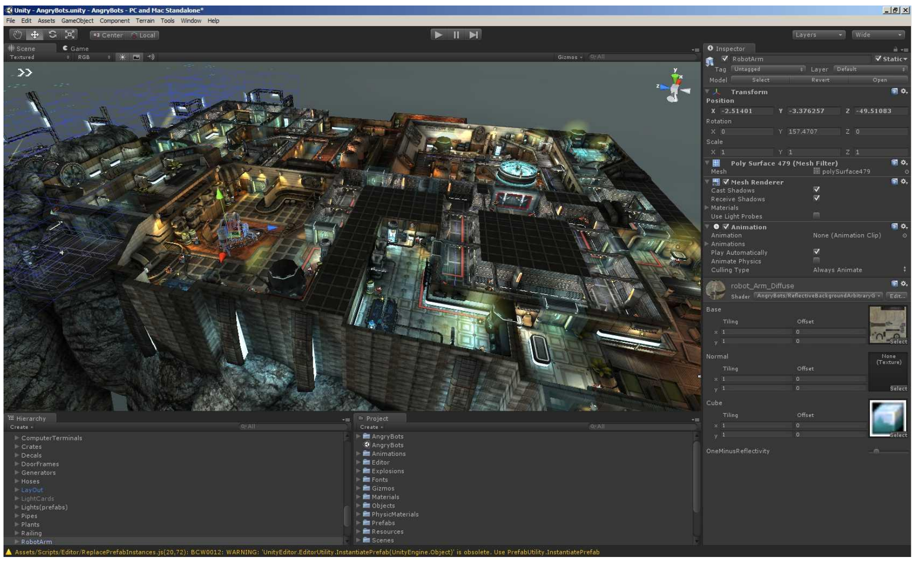


Figure 14: Screenshot of Unity's User Interface.

### 3.3.2 Use of Unity's RPG template

As stated, Unity provides a variety of useful tools for developers and designers to quickly prototype their ideas and later build on them. Due to its increasing population the last few years unity has also imported a number or pre-made projects that, in addition with their online tutorials, can teach new developers of its workflow, and more experienced developers its more advanced and newer features.

One of those templates is the Unity Creator Kit: RPG. This template is designed to familiarize the developers with the Unity Editor and its newer features, like the new Tilemap tool, in a code-free way.

For the purposes of this thesis, the aforementioned template has been used for convenience. The assets of the template as well as the animations and the movement scripts were kept as is. The parts of the templates that were changed to fit what was implemented were, the built in quest system which was bypassed to use the new quest system and elements of the UI, which were interconnected with the old quest system.

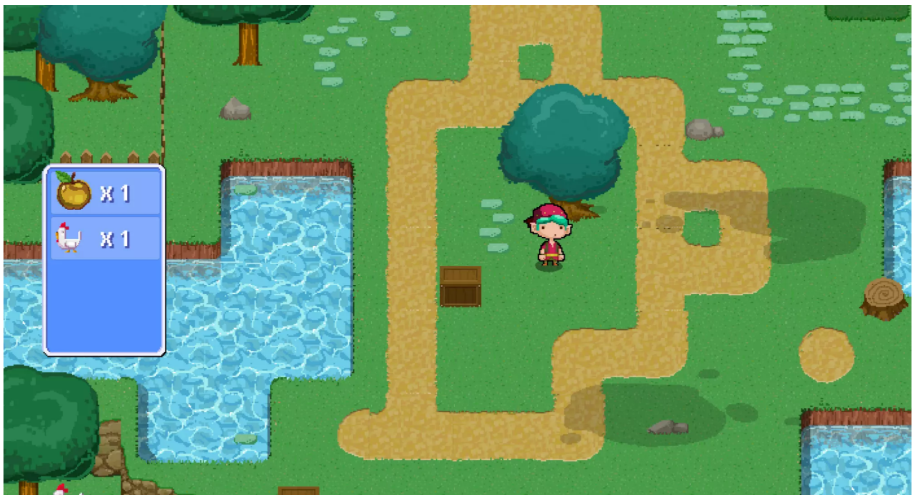


Figure 15: Game view of the template Creator Kit: RPG.

# 4 Use Cases

Having discussed about topics surrounding Procedural Narrative Generation and on how this thesis was influenced by other PNG systems like the ReGEN system and Radiant AI, this section is dedicated to giving a more detailed explanation on how the proposed system could be used by potential designers and developers.

Before we move on to the next subsections, a more solid explanation of the goal of the thesis should be provided. The purpose of this project is to create a procedural side-quest generation system, that takes into account the current space condition and a set of rules that can traverse that space, and decide which relationships to use to create that side-quest.

Specifically, given a set of rules, which for this implementation, are predetermined, the system will first try and decide the best suited rule to use and start traversing a world graph that is created by a designer, to find paths that satisfy said rule. Then provided that at least one of the starting entities of the path does not already have a quest, depending on the relationship between the player and the selected entity, the system may or may not provide a side-quest.

It is important to point out a few things first. First of all, the rules mentioned above are already programmed in the system, but there is an available workflow set to expand those rules by a developer, which will be discussed in detail in later sections.

Secondly, the best suited rule, which is decided in the first steps of the side-quest generation process, is determined by a basic cost function, which will also be described in later sections in more detail.

## 4.1 Designer's Use Cases

In the first subsection, we discuss the use case from a designers perspective. The scope of this implementation was to provide the ability to a designer to import their game assets and functionality, create a world graph that contains details on the entities that reside in their world as well as the relationships between them.

### 4.1.1 Creating a World Graph

Provided that the designer has already imported their game and this system into Unity, their first step would be to create a World Graph. As mentioned, for the purposes of this thesis and for convenience, the RPG prototype of Unity was used.

In order to help designers create the World Graph, a custom World Graph Editor was created to be used as an interactive tool. This tool was created using the UnityEditor library of Unity, which provides an API to help developers create custom tools, to assist the in the production of their project.

If the system has been imported correctly, a new menu item should appear that gives access to the World Graph Editor as shown in Figure 16. When pressed a new window should appear like Figure 17, which is the editor. The left side of the interface is called the World Graph view, which provides a visual representation of the world graph and the right side is the Attribute view, which provides details on the attributes of a selected node of the graph.

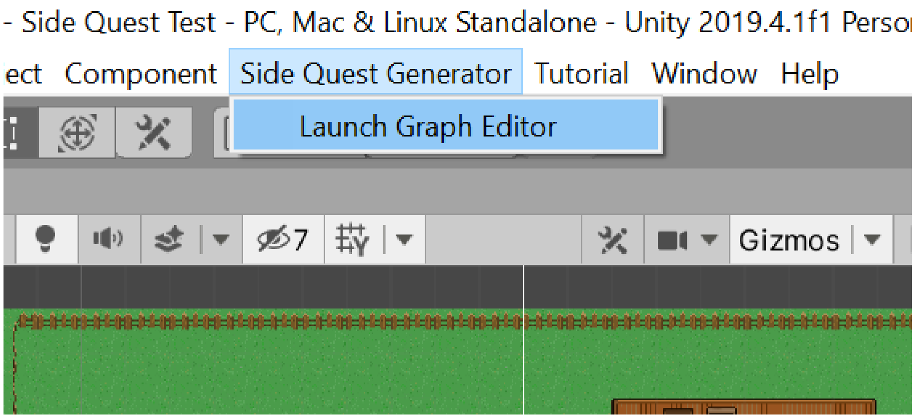

- Side Quest Test - PC, Mac & Linux Standalone - Unity 2019.4.1f1 Perso
ect Component Side Quest Generator Tutorial Window Help
Launch Graph Editor
Gizmos

Figure 16: Menu item of the custom World Graph Editor tool.

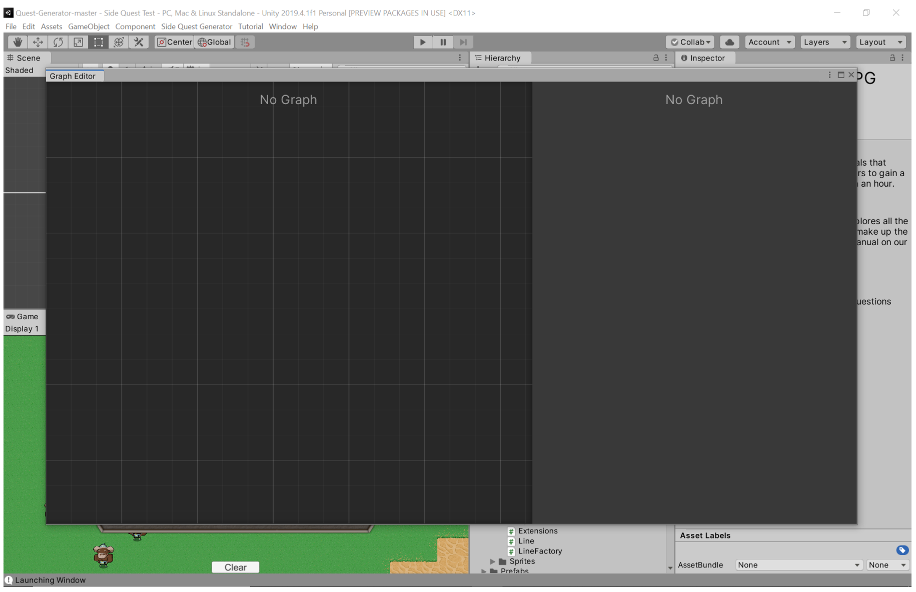
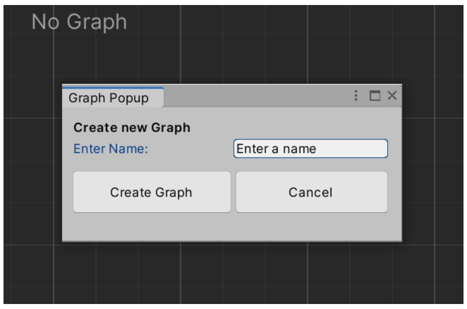


Figure 17: The World Graph Editor Tool interface.

To create a new world graph, the editor has to right-click on the World Graph view, and select **Create Graph**. If the designer already has world graph they can select **Load Graph** to load it. A pop up will then appear, that asks for the world graph's name. The name is required and will show an error if not set.

Having set the world graph's name the designer can now start populating the graph. To start adding nodes, the designer presses right-click on the world graph


No Graph

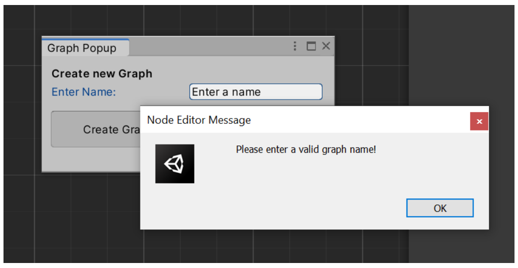
Graph Popup

Create new Graph

Enter Name: Enter a name

Create Graph | Cancel

Figure 18: Pop up prompting user to give a name to the world graph.


Graph Popup

Create new Graph

Enter Name: Enter a name

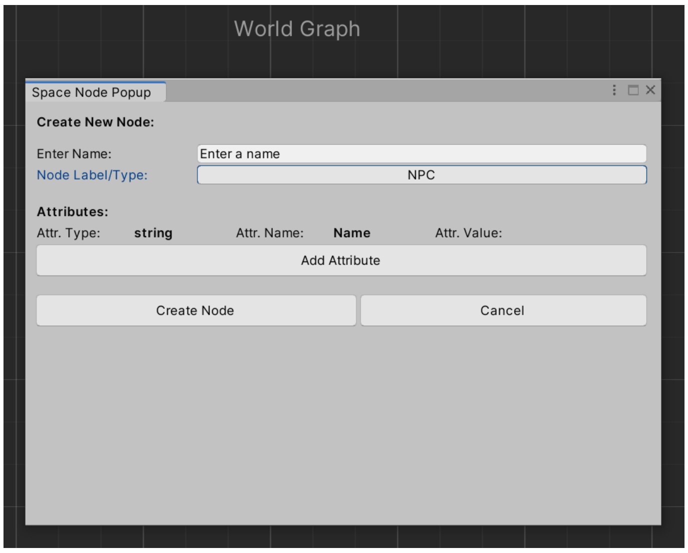
Node Editor Message

Please enter a valid graph name!

OK

Figure 19: Error message when a name is not valid.

view and then **Add Node**. A window will then appear, to help the designer instantiate the node, as shown in Figure 20.


World Graph

Space Node Popup

Create New Node:

Enter Name: Enter a name

Node Label/Type: NPC

Attributes:

Attr. Type: string Attr. Name: Name Attr. Value:

Add Attribute

Create Node | Cancel

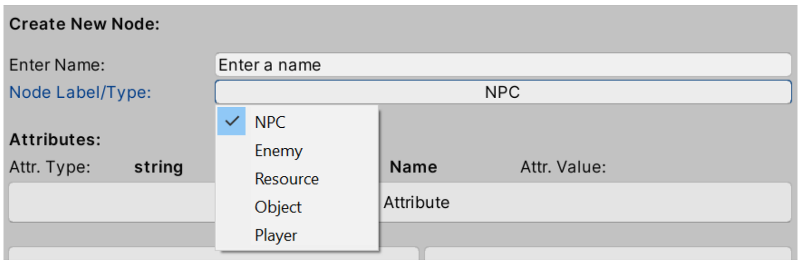
Figure 20: Node creation pop up.

On that new window, the designer can set up a new node in the world graph giving it a name first, which is also its first attribute, and then defining what kind of node it is and its attributes if it needs any.

The attribute defining what kind of node this is, is the **Node Label/Type**. The node Label - as it will be called from now on - has five (5) different options, **NPC, Enemy, Resource, Object** or **Player**. The label is rather important to be set up correctly, as it is used by the system when it traverses the world graph when trying to apply a rule. For example, when it uses the **Resource Gathering** rule it searches for nodes labeled as Resource.

Next step in setting the node details, is giving that node its attributes. Giving nodes attributes is important because a rule can search for specific attributes in nodes. For example, when we have a **Kill** rule, we search for all nodes that have an **Alive** boolean attribute set to **True**.

The first attribute, as mentioned, is the node name which is set by typing it in the *Enter Name* text input. If the designer chooses to give more attributes to the node, they can press the *Add Attribute* button. When the button is pressed two things happen, a new attribute input appears and a *Remove Attribute* button appears. The Remove Attribute button only appears when there are more than one attributes in a node, and when pressed it removes the last attribute. An attribute can be one of three (3) types, **String, Int** or **Boolean**. When a type is selected, the designer can then give the name of the attribute and its value, depending on the type. When the node is ready, the designer presses *Create Node* and a new node is then created. We can then select that node in the world graph view and see its attributes in the attribute view as shown in Figure 23. If the designer desires, they can edit the attributes in the attribute view. Finally, the designer can delete the node after being created by pressing right-click on it and selecting **Delete Node**.

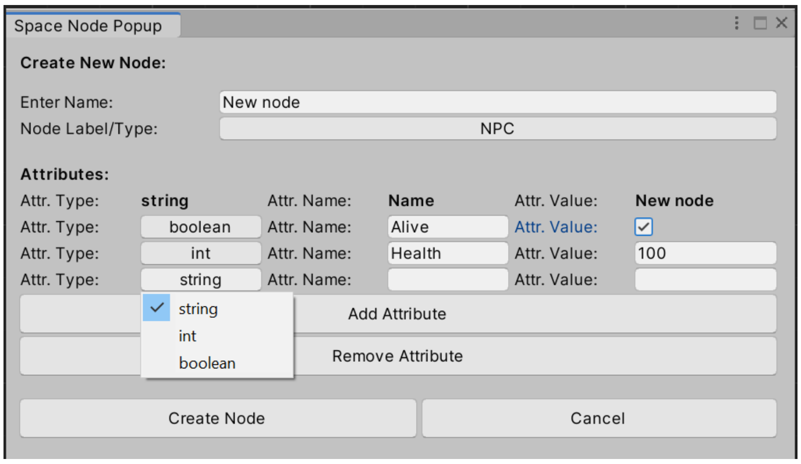

**Create New Node:**

Enter Name: Enter a name
Node Label/Type: NPC

✓ NPC
Enemy
Resource
Object
Player

**Attributes:**
Attr. Type: **string** **Name** Attr. Value:
Attribute

Figure 21: Dropdown with all available label types of a node.


Space Node Popup

**Create New Node:**

Enter Name: New node
Node Label/Type: NPC

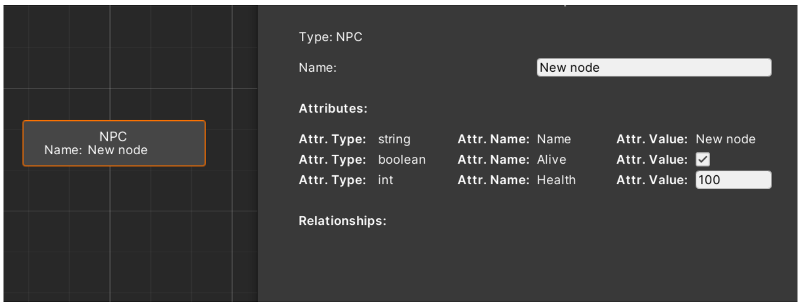
**Attributes:**

| | | | | | |
|---|---|---|---|---|---|
| Attr. Type: | **string** | Attr. Name: | **Name** | Attr. Value: | **New node** |
| Attr. Type: | boolean | Attr. Name: | Alive | Attr. Value: | ✓ |
| Attr. Type: | int | Attr. Name: | Health | Attr. Value: | 100 |
| Attr. Type: | string | Attr. Name: | | Attr. Value: | |

✓ string
int
boolean

Add Attribute
Remove Attribute
Create Node
Cancel

Figure 22: Dropdown with all available attribute types. It can also be seen that, the first attribute is the node name.


NPC
Name: New node

Type: NPC
Name: New node

**Attributes:**

| | | | | | |
|---|---|---|---|---|---|
| **Attr. Type:** | string | **Attr. Name:** | Name | **Attr. Value:** | New node |
| **Attr. Type:** | boolean | **Attr. Name:** | Alive | **Attr. Value:** | ✓ |
| **Attr. Type:** | int | **Attr. Name:** | Health | **Attr. Value:** | 100 |

**Relationships:**

Figure 23: Newly created node with its attributes seen in the attribute view.

The last thing the designer has to do to finalize the world graph, is to define the relationships between two nodes. To do that, first they need to create at more nodes

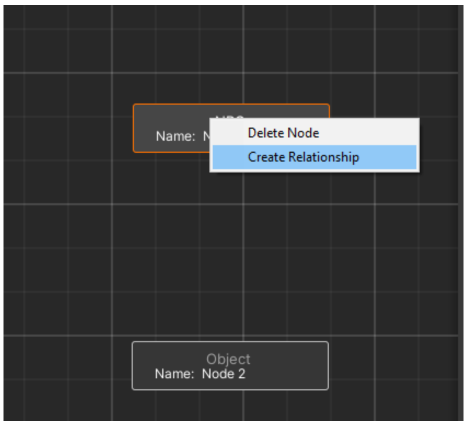


Figure 24: Right-clicking on a node.

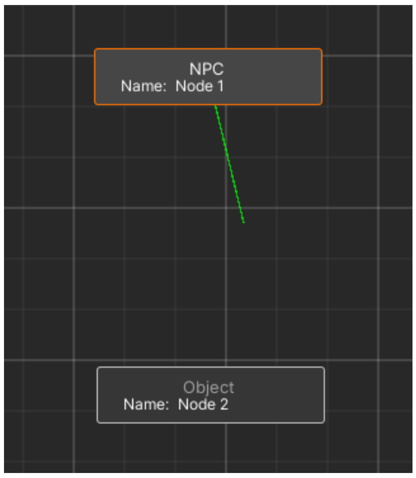
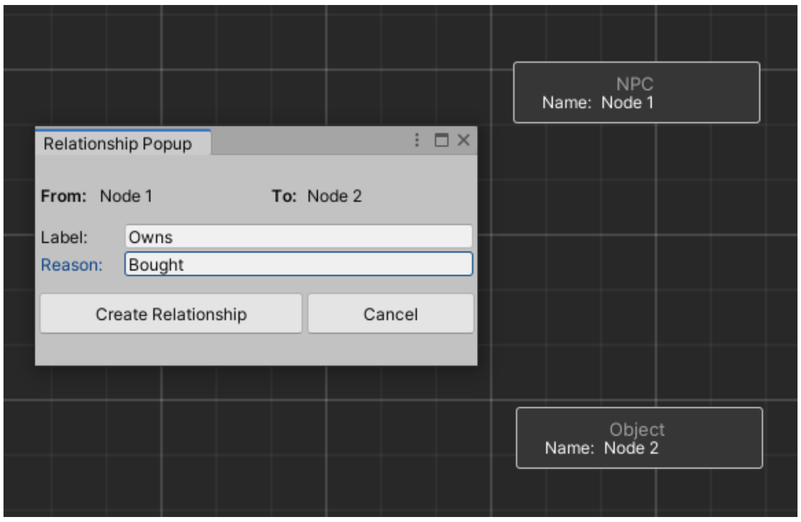


Figure 25: Green line depicting a potential new relationship.

using the same procedure as described above, then press right-click on the **starting node** and select **Create Relationship** (Figure 24). Then a green line will appear starting from the starting node to the mouse cursor (Figure 25). Select the ending node pressing **right-click** again and then a new relationship pop up will appear.



**From:** Node 1 **To:** Node 2

Label: Owns

Reason: Bought

Figure 26: New relationship pop up.

In that relationship pop up, the relationship direction is shown in a manner of **From:** *(starting node)* to **To:** *(ending node)*. The designer can then set the relationship's **Label** and **Reason**. Both Label and Reason are used in the system while searching for potential paths and they define the type of the relationship. The Label is required while adding a Reason is not. It is worth noting that the designer can add **multiple** relationships between two nodes. The designer then presses the **Create Relationship** button to create the new relationship between the nodes.

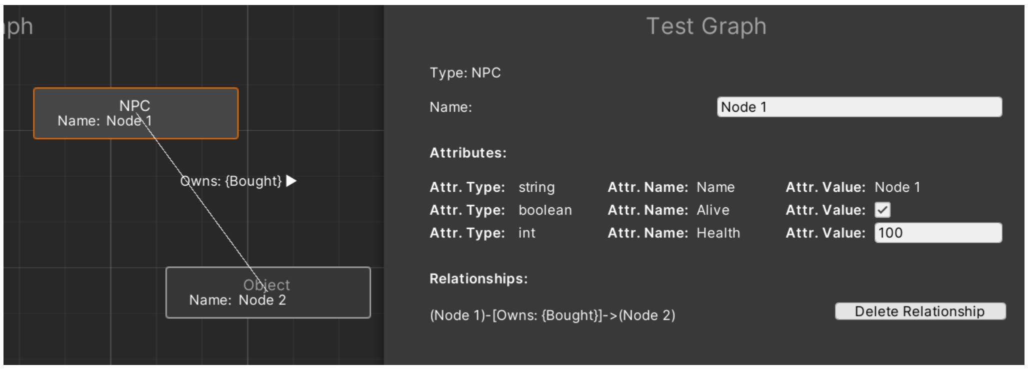
When a relationship is created you can see a white line connecting the two nodes with a text in the middle of that line, showing the Label and Reason (if any), in the form of **Label: {Reason}**. There is also an arrow in either side of that text, pointing to the direction of the relationship taking into account the x-axis difference between the two nodes.

The relationship, also, appears in the Attribute view when selecting a node. It is shown in a form of:

(*starting node*)-[*Relationship Label*: {*Relationship Reason*}]->(*ending node*)

which shows both the node names and the relationship details. The designer can also delete the relationship by pressing the **Delete Relationship** button in the Attribute view.

One last important note to make, is that all changes done to the graph like creating or deleting nodes or relationships, are saved in real time so there is no need for a Save button. The graph is saved in Assets/Database folder.



Test Graph

Type: NPC

Name: Node 1

Attributes:

| | | |
|---|---|---|
| Attr. Type: string | Attr. Name: Name | Attr. Value: Node 1 |
| Attr. Type: boolean | Attr. Name: Alive | Attr. Value: ✓ |
| Attr. Type: int | Attr. Name: Health | Attr. Value: 100 |

Relationships:

(Node 1)-[Owns: {Bought}]->(Node 2)

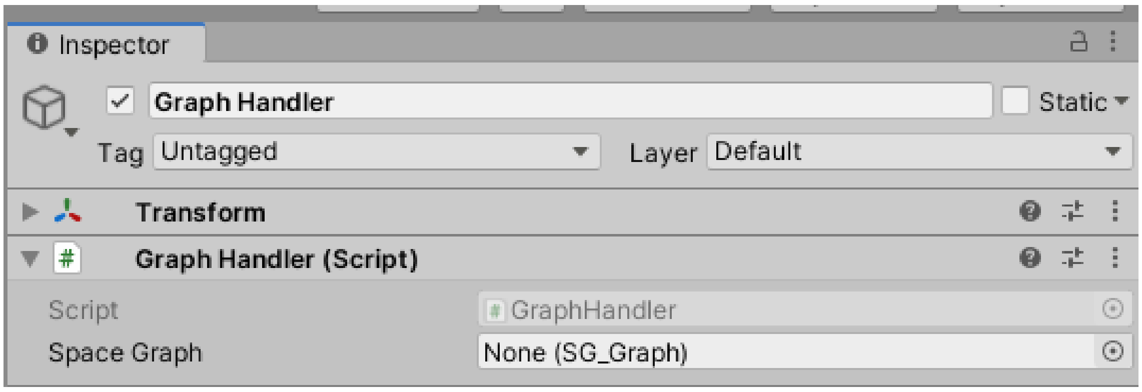
Figure 27: Relationship between two nodes as shown in the World Graph view and the Attribute view.

### 4.1.2 Connecting World Graph to Game

Having created the desired World graph using the Graph Editor tool shown in the previous section, the designer must now connect that graph to their game. The process of doing that will be described below.

Firstly, as a good practice, it is recommended a new persistent scene to be created, to put all system Game Objects. Te first essential Game Object to be set is the **Graph Handler** which will take the graph the designer created, and initialize it during runtime. So first, create an empty Game Object, rename it to Graph Handler and drag and drop the Graph Handler script. This script has one input, which takes the world graph instance that was created. The only thing left to do is drag that instance to the input.

The next step would be to connect the game entities to the ones in the graph. Provided that the system would be imported to a game in progress, which means that all Game Objects like NPCs, resources etc. are already set, then the next steps are as follows.


Inspector

Graph Handler   Static

Tag Untagged   Layer Default

Transform

Graph Handler (Script)

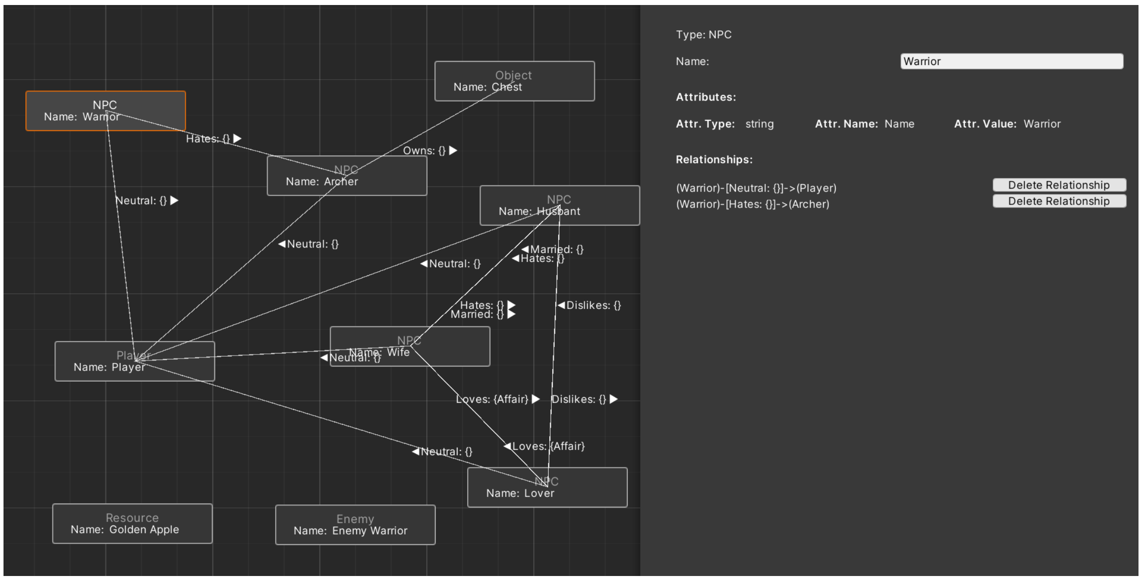
Script   GraphHandler

Space Graph   None (SG_Graph)

Figure 28: The Graph Handler Game Object with the script attached to it.

Depending on the type of the entity the designer has to put in its Game Object one of four options, **InteractableCharacter, InteractableResource, InteractableObject or InteractableEnemy**. All those classes extend the **Interactable** class so they share the same core. It is also important to set the Interactable script to the correct type, as it will not be able to connect to the world graph.

For this example, a world graph is already created as shown in Figure 29. It will now be demonstrated how the designer would create an InteractableCharacter, by using the the Warrior node of the graph. It should also be noted that it is the same procedure for all other types of Interactables.



Type: NPC

Name: Warrior

**Attributes:**

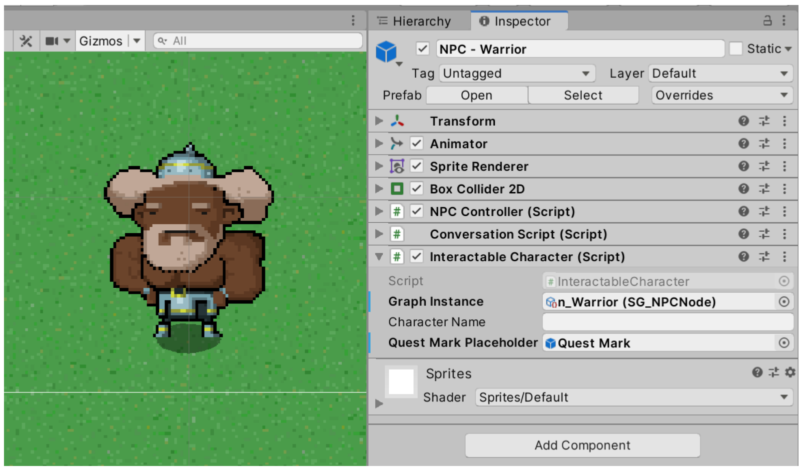
**Attr. Type:** string   **Attr. Name:** Name   **Attr. Value:** Warrior

**Relationships:**

(Warrior)-[Neutral: {}]->(Player)   Delete Relationship
(Warrior)-[Hates: {}]->(Archer)   Delete Relationship

Figure 29: The world graph created for the demo of this thesis.

First, we select the Game Object we want to set as the Warrior, and drag and drop an InteractableCharacter script. Next, go to the Assets/Database folder, find the world graph, expand it, and find the appropriate node, which has a n_ prefix to indicate it is a node, and the node Name. For this example we would need to search for n_Warrior. Drag and drop that instance to the **Graph Instance** input of the InteractableCharacter script. The **Character Name** will be populated on runtime by the script. Lastly, the **Quest Mark Placeholder** requires a game object that represents the Quest Mark that shows on top of the Interactable. This is also required for the script to work and it will throw an error if not populated. The final outcome should look like Figure 30.



NPC - Warrior
Tag Untagged
Layer Default
Prefab Open Select Overrides
Transform
Animator
Sprite Renderer
Box Collider 2D
NPC Controller (Script)
Conversation Script (Script)
Interactable Character (Script)
Script InteractableCharacter
Graph Instance n_Warrior (SG_NPCNode)
Character Name
Quest Mark Placeholder Quest Mark
Sprites
Shader Sprites/Default

Figure 30: The InteractableCharacter script correctly populated for the Warrior NPC of this demo's world graph. The Quest Mark is not shown as it is disabled on instantiation and only enabled when there is a Quest on the character.

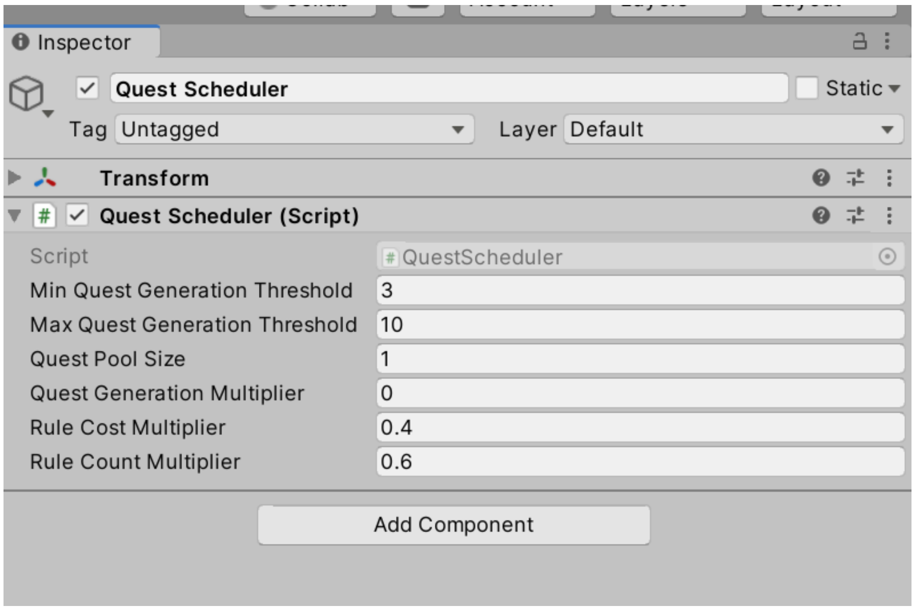
In the same way as we have set up the Warrior instance of the graph, the designer can set up the rest of the graph nodes. The system during runtime will gather all those instances and recreate that graph, with which the game will interact.

After connecting all Interactables with the Game Objects, the designer will have to introduce the system as well. To do that, create an empty Game Object in the persistent scene and add the Quest Scheduler script. The Quest Scheduler has six (6) input variables that are required to be set up. The inputs are given a brief description below.

- **Min Quest Generation Threshold:** The minimum amount, in seconds, that the system has to wait when creating a new side quest.
- **Max Quest Generation Threshold:** The maximum amount, in seconds, that the system has to wait when creating a new side quest.
- **Quest Pool Size:** The amount of side quests that can exist at the same time.
- **Quest Generation Multiplier:** A multiplier that increases the overall chance of producing a quest.
- **Rule Cost Multiplier:** A multiplier that defines the importance of the cost of the rule to be selected, when selecting the best fitting rule.
- **Rule Count Multiplier:** A multiplier that defines the importance of the count of the amount of times the rule was selected, when selecting the best fitting rule.

The last three inputs will be described in more detail in the Implementation section, as they are used in the calculation of getting the best fitting rule.

The last thing to be done for the system to be operational, is to set up the interactions between between the player and the Interactables through the game. Specifically, when the player interacts with an Interactable Character that has a quest available, then the player gets assigned that quest. For this step to be completed, the assistance of a developer is required as there has to be some coding in order to connect all the necessary events. As such, it will be discussed in the nest subsection.



Quest Scheduler
Static
Tag Untagged
Layer Default
Quest Scheduler (Script)
Script QuestScheduler
Min Quest Generation Threshold 3
Max Quest Generation Threshold 10
Quest Pool Size 1
Quest Generation Multiplier 0
Rule Cost Multiplier 0.4
Rule Count Multiplier 0.6

Figure 31: Quest Scheduler instance of demo.

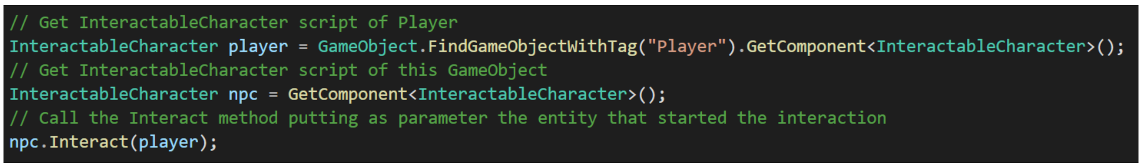
## 4.2 Developer's Use Cases

In this second subsection of the use cases, the process of completing the system connection to the game will be given in more detail. Furthermore, ways that a developer might expand the system by adding more rules, quest events and events, will also be provided.

### 4.2.1 Binding the UI

As mentioned in the previous subsection, showing the designer's perspective on how to import the system to a game, the part of connecting the said system with the actual game can only be achieved with the help of a developer. It will now be demonstrated how the the game can trigger certain events of the Interactables and in extend, the system itself.

The first thing to be done is setting up the interaction between the player game object and the Interactables. All of the Interactables have an **Interact** method that sends out an event through the **EntityEventBroker** class - which will be described in more detail in the Implementation section - that passes the invoker and the target as parameters. The overall structure of the system uses a **Publish - Subscribe pattern**, in which the Interactables send event messages through the EntityEventBroker to whichever class needs to listen to them. For this demo, the Interact event is used more to inform UI elements that the player has interacted with an InteractableCharacter so that a UI element will pop up showing information about a quest, if there is one available.



```
// Get InteractableCharacter script of Player
InteractableCharacter player = GameObject.FindGameObjectWithTag("Player").GetComponent<InteractableCharacter>();
// Get InteractableCharacter script of this GameObject
InteractableCharacter npc = GetComponent<InteractableCharacter>();
// Call the Interact method putting as parameter the entity that started the interaction
npc.Interact(player);
```

Figure 32: Use of Interact method in current demo. The player is the Invoker as he is the one interacting with the InteractableCharacter.

Like the Interact method, there are several other that trigger events which are used by both the system and other event listeners that can subscribe to them. These methods will be described below. Firstly, when an InteractableCharacter is killed, the **CharacterKilled** method is called, which takes an Interactable as a parameter, to specify which entity killed that InteractableCharacter. This method then invokes the EntityDeath method from the EntityEventBroker, which in turn invokes the OnEntityDeath event. Following that Publisher - Subscriber pattern, all objects that are subscribed to that event will be called. The world graph is subscribed to that event, so that it can change all relationships between the killed entity and all other entities that are connected to it. Also, the QuestScheduler class is subscribed to it, so that it can progress a quest, if the character was part of that quest. It should be pointed out that the way the entity is killed, from the game play aspect, must be set by the developer. For example, in this demo, for simplicity, we can just click on a character in order to kill them.

For an object to be picked up, there is a PickUpObject method in the EntityEventBroker, which is called from the Interact method that the InteractableObject has. Whenever an object is picked up, which again is specified by the developer on how it is picked up, the Interact method must be called, which in turn calls the PickUpObject method, which triggers the OnObjectPickUpSuccess event. Like the OnEntityDeath event, this one also has the world graph and QuestScheduler subscribed for the same reasons.

Same as the two methods above, for killing InteractableEnemies there is the KillEnemy method on the script, that invokes the OnEnemyKilled event. Also, for gathering resources the Interact method is called, which invokes the OnResourcePickUp event. As for the previous methods, the way the enemies are killed and the resources are picked up, from the gameplay's perspective, is set by the developer.

Since this is an event based system, the developer can freely create gameplay elements, such as UI or sound effects etc. and subscribe them to the respective events. That way the developer, can connect every element of their game to the system, which was the purpose of the overall structure of the system.

### 4.2.2 Creating new Events

The EntityEventBroker has a set of necessary events that the system uses, as described in the previous section. If the developer needs to expand the that set, we shall now describe the way to do it.

First step in creating a custom event, is to actually declare it in the EntityEventBroker class. There are two steps in that process. First, declare the event type like shown below:

**public static event Action**<ParamType1, ParamType2,...> OnEventName**;**

As a convention, at the start of the event name the **On** prefix is used, to differentiate between a regular method and an event. What this declaration means, is that we have created an event that any method can subscribe on, as long as, that method has the same arguments like specified in the Action<ParamType1, ParamType2,...>. The event will then be used by the Subscriber part of the pattern.

Next step is creating the method, that the Publishers can call to trigger that event. This is a regular method which is registered in the EntityEventBroker. That method can have its own functionality, like for example deciding on whether the event should be called or not. Calling the event is done like this:

**OnEventName?.Invoke(param1, param2,...);**

As stated above, the arguments must be the same as the event. This method is then called by the Publisher side of the pattern, to inform other entities that an event has happened. It should also be stated that, both the event and the method are static so they can be called without creating an instance of the EntityEventBroker.

Now that the event is in place, the developer must create the Publisher side of the pattern, which basically means that depending on how the developer wants to call that event, they need to implement a method that calls the one created in the EntityEventBroker, like shown in Figure 34. It is important not to call the event directly, as the method could potentially have a way to decide whether to call the event or not.

Having created the Publisher side, the developer must now subscribe all the necessary classes that need to listen to the event. This is also implemented like shown in Figure 35. After finishing this step, the developer has now implemented a new event, and expanded the capabilities of the EntityEventBroker.

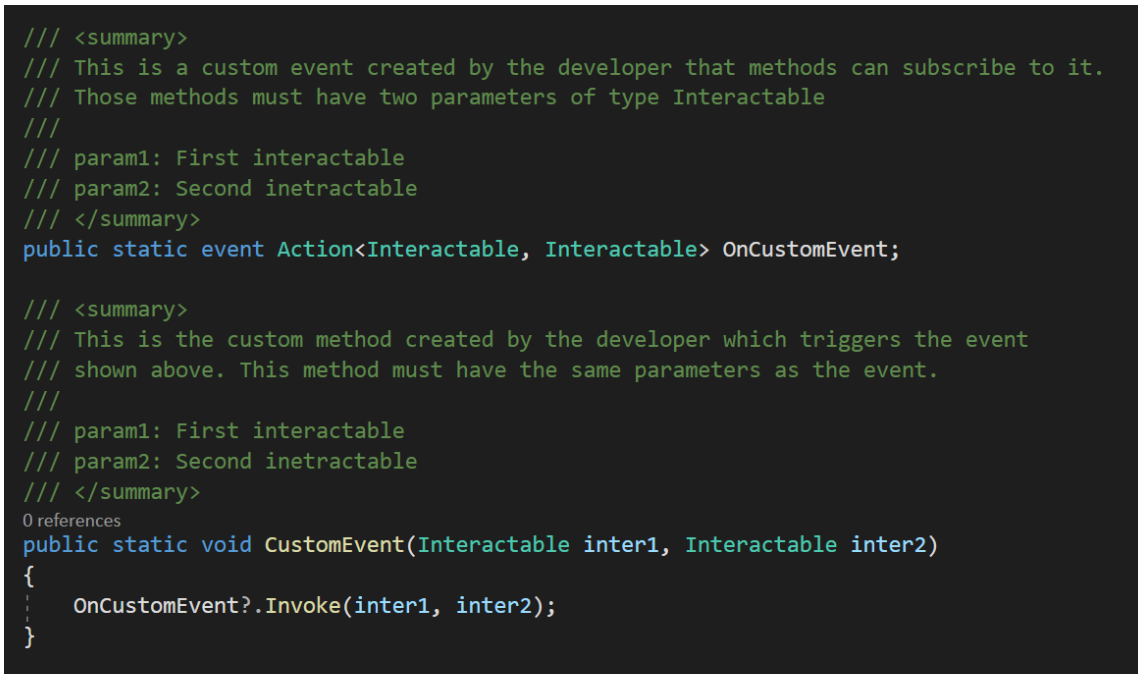

```
/// <summary>
/// This is a custom event created by the developer that methods can subscribe to it.
/// Those methods must have two parameters of type Interactable
///
/// param1: First interactable
/// param2: Second inetractable
/// </summary>
public static event Action<Interactable, Interactable> OnCustomEvent;

/// <summary>
/// This is the custom method created by the developer which triggers the event
/// shown above. This method must have the same parameters as the event.
///
/// param1: First interactable
/// param2: Second inetractable
/// </summary>
0 references
public static void CustomEvent(Interactable inter1, Interactable inter2)
{
    OnCustomEvent?.Invoke(inter1, inter2);
}
```

Figure 33: The necessary additions to the EntityEventBroker class, to create a new event.

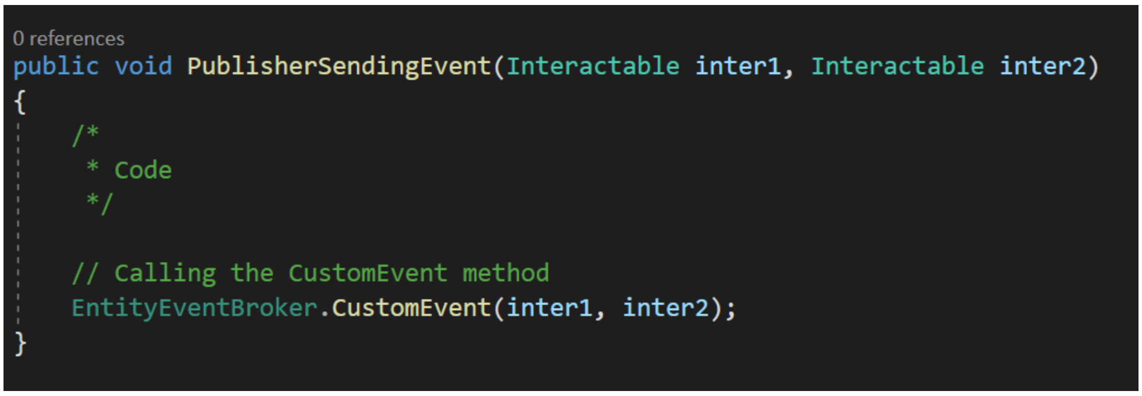

```
0 references
public void PublisherSendingEvent(Interactable inter1, Interactable inter2)
{
    /*
     * Code
     */

    // Calling the CustomEvent method
    EntityEventBroker.CustomEvent(inter1, inter2);
}
```

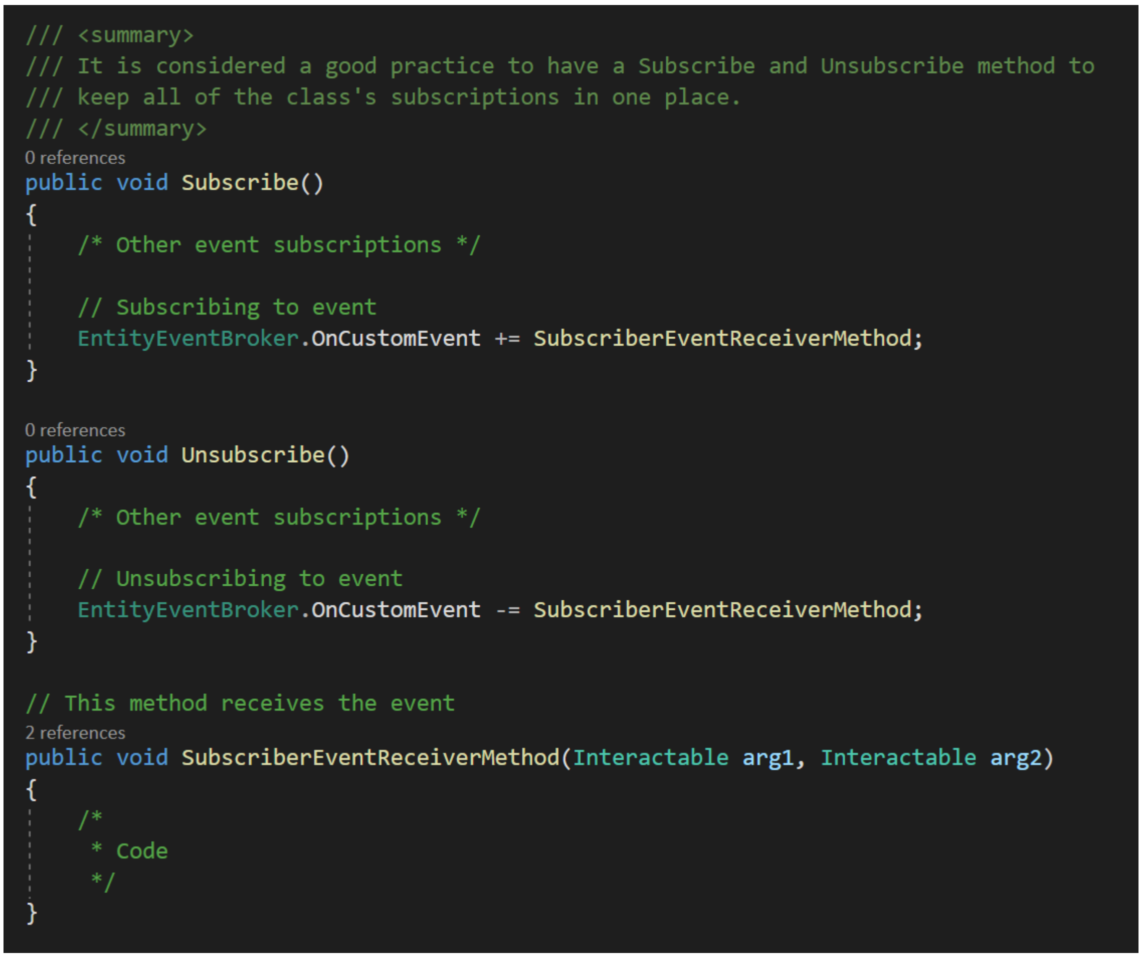
Figure 34: The code necessary for the Publisher to publish an event via the EntityEventBroker.

### 4.2.3 Creating new Rules

As described before, the system functions by using a set of pre existing rules, to search on the world graph and then generate a new side quest. If the developer finds this set limiting, the system provides the capability of expanding those rules and letting them creating their own, in a way which will be described below.

The new rule must expand the Rule abstract class which has three abstract methods, which the new rule must implement, like shown in Figure 36. Those three methods are the ones used by the system to search the world graph and then generate a new Quest if that rule is selected.

The first method used by the QuestGenerator - which as the name suggests, is



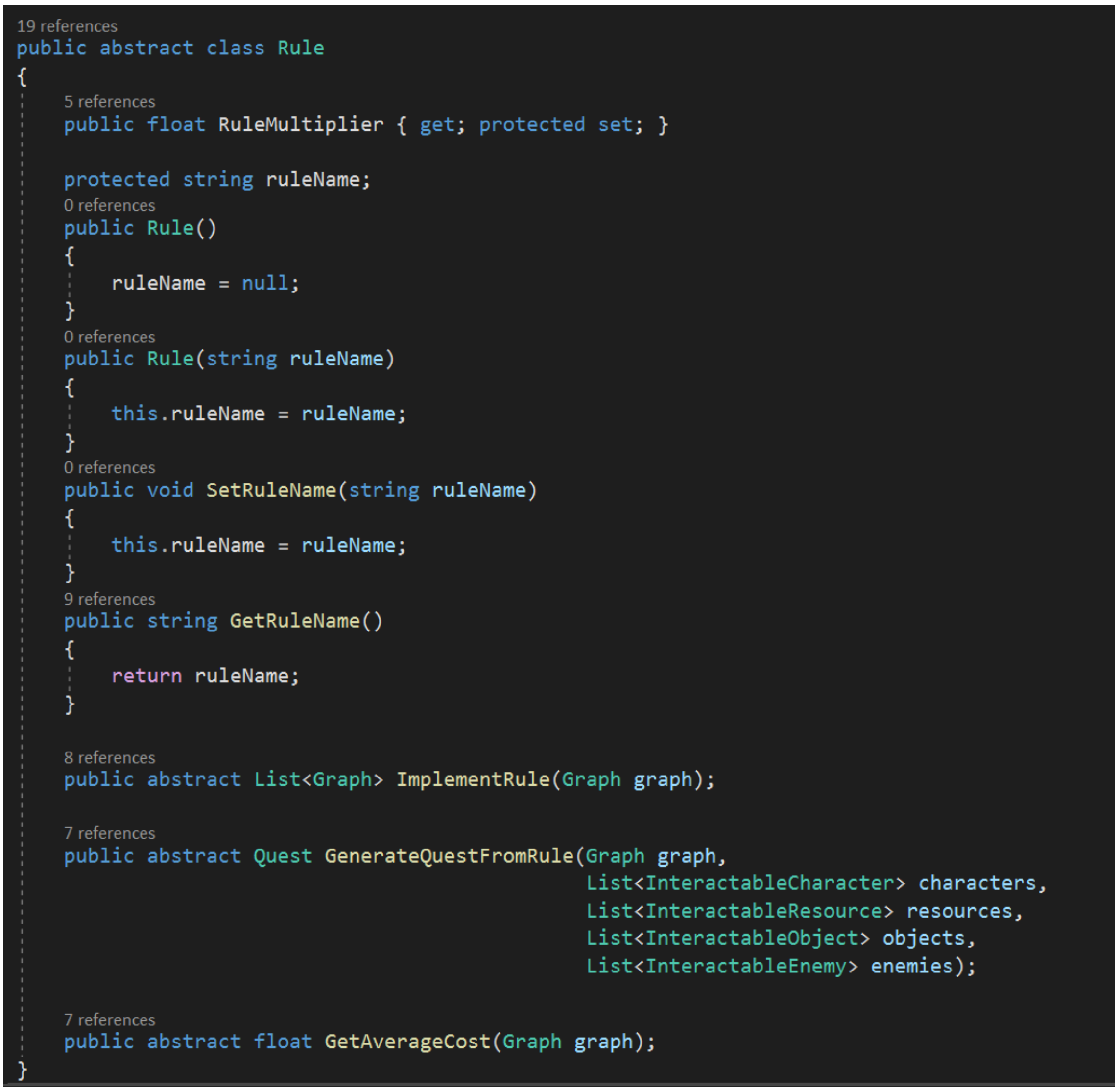
```
/// <summary>
/// It is considered a good practice to have a Subscribe and Unsubscribe method to
/// keep all of the class's subscriptions in one place.
/// </summary>
0 references
public void Subscribe()
{
    /* Other event subscriptions */

    // Subscribing to event
    EntityEventBroker.OnCustomEvent += SubscriberEventReceiverMethod;
}

0 references
public void Unsubscribe()
{
    /* Other event subscriptions */

    // Unsubscribing to event
    EntityEventBroker.OnCustomEvent -= SubscriberEventReceiverMethod;
}

// This method receives the event
2 references
public void SubscriberEventReceiverMethod(Interactable arg1, Interactable arg2)
{
    /*
     * Code
     */
}
```

Figure 35: The code necessary for the Subscriber to subscribe to and receive an event via the EntityEventBroker.

responsible for generating the quests - is the **ImplementRule**. This method, takes the world graph as input, searches for a pattern that's specified by the developer, and returns a list of graphs with all the available graphs that match said pattern. The Graph class has a set of customly developed API that allows the developer to parse the world graph any way they like.

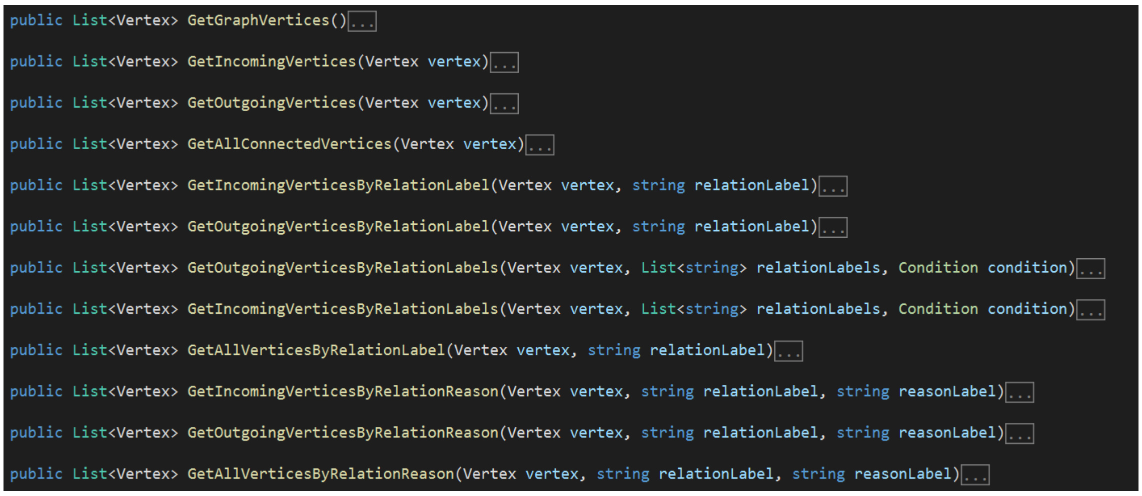
The second important method, is the **GenerateQuestFromRule**. This method will generate the side quest of the respective rule. This method takes as input a graph instance that represents a world graph path, that will be used to create said side quest, and all of Interactables. It then matches the the graph instance nodes to the Interatables, creates all the necessary **Quest Events** and outputs the final side quest.

Quest Events will be described in more detail in the next section, but two of them the most important ones and should be used when a developer is implementing a custom rule. Those Quest Events are the **InvokeQuestEvent** and the **Complete-QuestEvent**. The InvokeQuestEvent is the first event that should happen on every side quest. This QE takes as input a target InteractableCharacter, which is the entity that will give the player the quest and hence shall invoke the quest, and



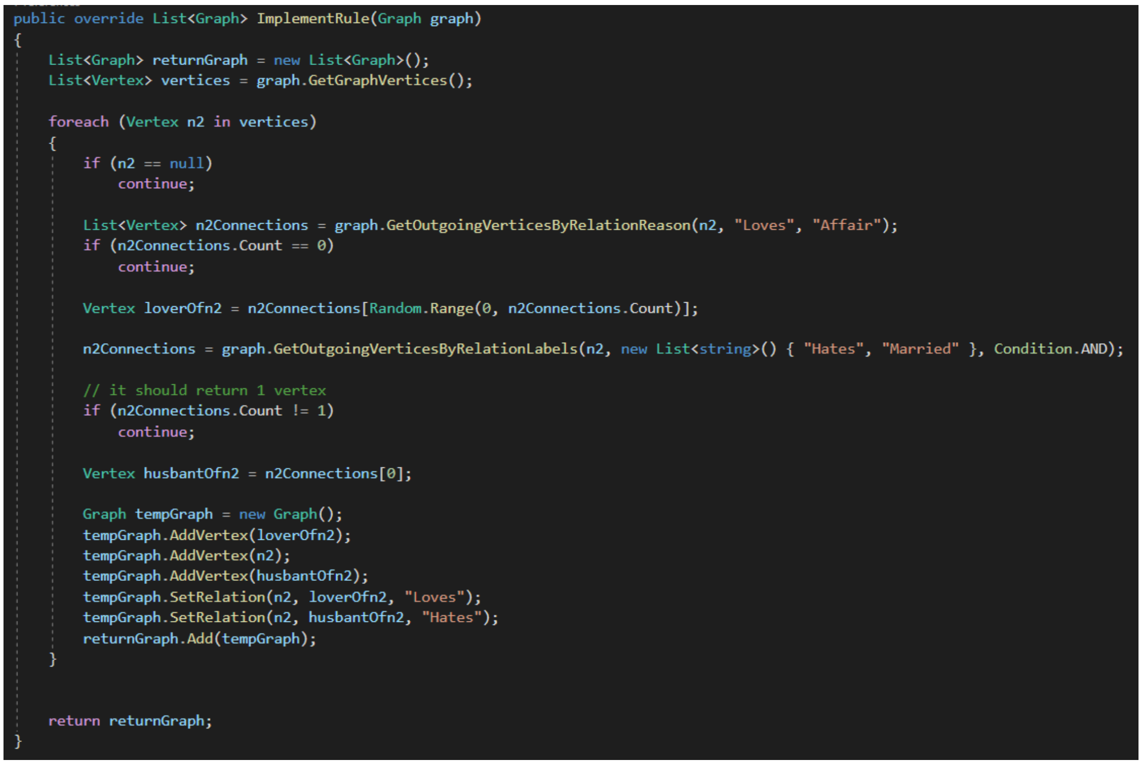
```
19 references
public abstract class Rule
{
    5 references
    public float RuleMultiplier { get; protected set; }

    protected string ruleName;
    0 references
    public Rule()
    {
        ruleName = null;
    }
    0 references
    public Rule(string ruleName)
    {
        this.ruleName = ruleName;
    }
    0 references
    public void SetRuleName(string ruleName)
    {
        this.ruleName = ruleName;
    }
    9 references
    public string GetRuleName()
    {
        return ruleName;
    }

    8 references
    public abstract List<Graph> ImplementRule(Graph graph);

    7 references
    public abstract Quest GenerateQuestFromRule(Graph graph,
                                                List<InteractableCharacter> characters,
                                                List<InteractableResource> resources,
                                                List<InteractableObject> objects,
                                                List<InteractableEnemy> enemies);

    7 references
    public abstract float GetAverageCost(Graph graph);
}
```

Figure 36: The Rule superclass definition.



```
public List<Vertex> GetGraphVertices()...

public List<Vertex> GetIncomingVertices(Vertex vertex)...

public List<Vertex> GetOutgoingVertices(Vertex vertex)...

public List<Vertex> GetAllConnectedVertices(Vertex vertex)...

public List<Vertex> GetIncomingVerticesByRelationLabel(Vertex vertex, string relationLabel)...

public List<Vertex> GetOutgoingVerticesByRelationLabel(Vertex vertex, string relationLabel)...

public List<Vertex> GetOutgoingVerticesByRelationLabels(Vertex vertex, List<string> relationLabels, Condition condition)...

public List<Vertex> GetIncomingVerticesByRelationLabels(Vertex vertex, List<string> relationLabels, Condition condition)...

public List<Vertex> GetAllVerticesByRelationLabel(Vertex vertex, string relationLabel)...

public List<Vertex> GetIncomingVerticesByRelationReason(Vertex vertex, string relationLabel, string reasonLabel)...

public List<Vertex> GetOutgoingVerticesByRelationReason(Vertex vertex, string relationLabel, string reasonLabel)...

public List<Vertex> GetAllVerticesByRelationReason(Vertex vertex, string relationLabel, string reasonLabel)...
```

Figure 37: An example of functions that of the Graph class, that are used to search nodes (defined as Vertex) according to their relationships.



```
public override List<Graph> ImplementRule(Graph graph)
{
    List<Graph> returnGraph = new List<Graph>();
    List<Vertex> vertices = graph.GetGraphVertices();

    foreach (Vertex n2 in vertices)
    {
        if (n2 == null)
            continue;

        List<Vertex> n2Connections = graph.GetOutgoingVerticesByRelationReason(n2, "Loves", "Affair");
        if (n2Connections.Count == 0)
            continue;

        Vertex loverOfn2 = n2Connections[Random.Range(0, n2Connections.Count)];

        n2Connections = graph.GetOutgoingVerticesByRelationLabels(n2, new List<string>() { "Hates", "Married" }, Condition.AND);

        // it should return 1 vertex
        if (n2Connections.Count != 1)
            continue;

        Vertex husbantOfn2 = n2Connections[0];

        Graph tempGraph = new Graph();
        tempGraph.AddVertex(loverOfn2);
        tempGraph.AddVertex(n2);
        tempGraph.AddVertex(husbantOfn2);
        tempGraph.SetRelation(n2, loverOfn2, "Loves");
        tempGraph.SetRelation(n2, husbantOfn2, "Hates");
        returnGraph.Add(tempGraph);
    }


    return returnGraph;
}
```

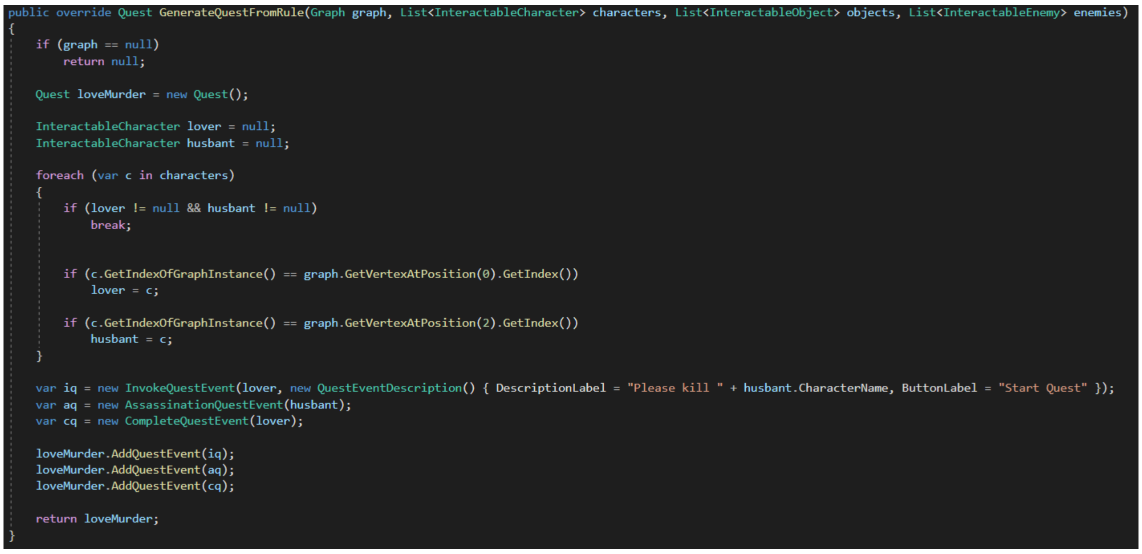
Figure 38: Implementation of LoveMurderRule. This rule searches for nodes that are are married but also in an affair. The rule will then give an assassination side quest to the player.

a QuestEventDescription, which is used to pass information about the quest. The QuestEventDescription will mostly be used by the UI when the player interacts with the character, so that the UI element that appears which gives a brief description on the quest, will take that information from the QuestEventDescription.

The other most important QE is the CompleteQuestEvent. This QE is responsible for completing its respective quest. Like the InvokeQuestEvent, this QE takes as input a target InteractableCharacter, which is the entity that the player must go to, to complete the quest, and a QuestEventDescription. The QuestEventDescription is used here, for the UI elements that appear when the player interacts with said entity. In this QE, the developer can implement a reward system for the player, which is triggered when they complete a quest. For simplicity, in this demo a reward system was not implemented. Those two QEs, must be included by the developer when creating a new Quest. They represent the start and the end events of the Quest.

### 4.2.4 Creating new Quest Events

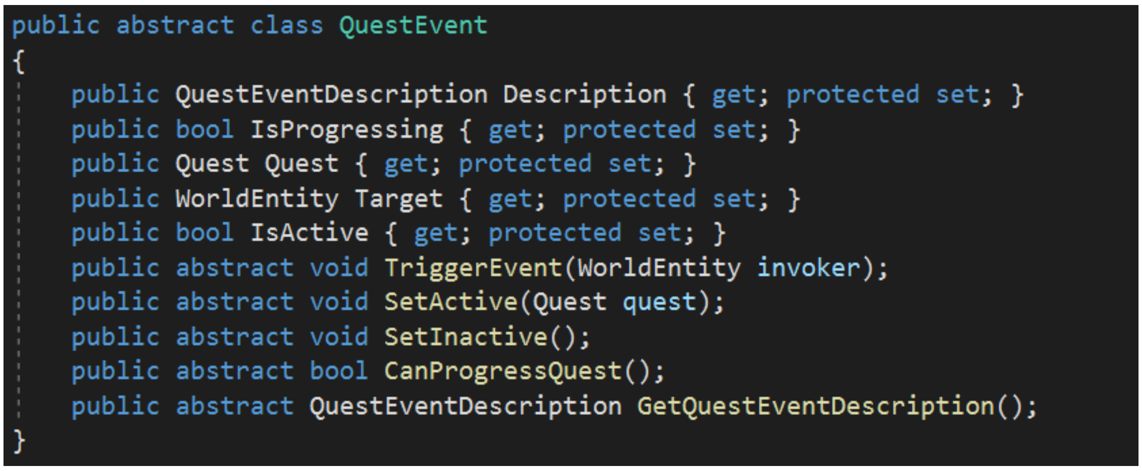
Having described how a Quest is implemented and focusing on the importance of Quest Events, since they are the building blocks of the Quest, the workflow of implementing those building blocks will now be presented.



```
public override Quest GenerateQuestFromRule(Graph graph, List<InteractableCharacter> characters, List<InteractableObject> objects, List<InteractableEnemy> enemies)
{
    if (graph == null)
        return null;

    Quest loveMurder = new Quest();

    InteractableCharacter lover = null;
    InteractableCharacter husbant = null;

    foreach (var c in characters)
    {
        if (lover != null && husbant != null)
            break;


        if (c.GetIndexOfGraphInstance() == graph.GetVertexAtPosition(0).GetIndex())
            lover = c;

        if (c.GetIndexOfGraphInstance() == graph.GetVertexAtPosition(2).GetIndex())
            husbant = c;
    }

    var iq = new InvokeQuestEvent(lover, new QuestEventDescription() { DescriptionLabel = "Please kill " + husbant.CharacterName, ButtonLabel = "Start Quest" });
    var aq = new AssassinationQuestEvent(husbant);
    var cq = new CompleteQuestEvent(lover);

    loveMurder.AddQuestEvent(iq);
    loveMurder.AddQuestEvent(aq);
    loveMurder.AddQuestEvent(cq);

    return loveMurder;
}
```

Figure 39: Generating a side quest for the LoveMurderRule.

When creating a new quest event, the developer must extend the abstract Quest Event class. This class has a set of necessary parameters that are passed to its subclasses, along with a set of abstract methods, that must also be implemented by the developer. Below we shall discuss what those methods do.

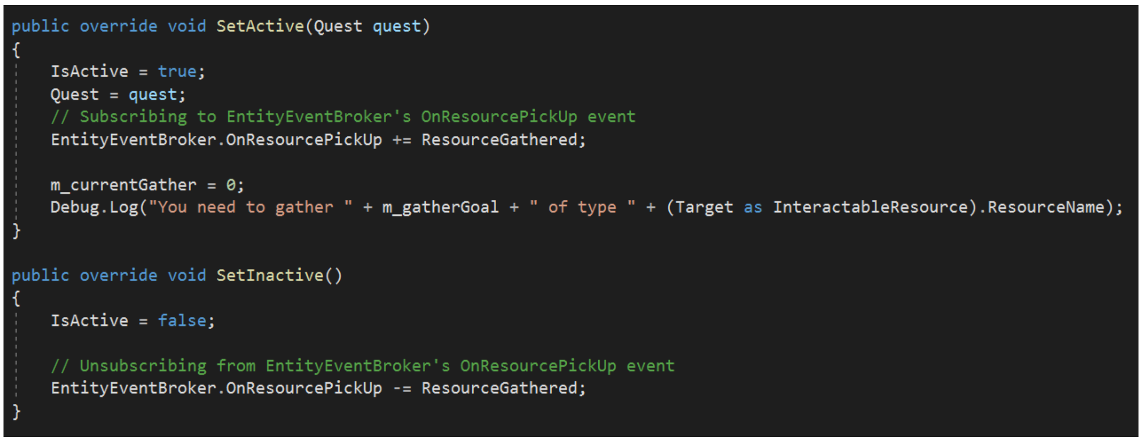


```
public abstract class QuestEvent
{
    public QuestEventDescription Description { get; protected set; }
    public bool IsProgressing { get; protected set; }
    public Quest Quest { get; protected set; }
    public WorldEntity Target { get; protected set; }
    public bool IsActive { get; protected set; }
    public abstract void TriggerEvent(WorldEntity invoker);
    public abstract void SetActive(Quest quest);
    public abstract void SetInactive();
    public abstract bool CanProgressQuest();
    public abstract QuestEventDescription GetQuestEventDescription();
}
```

Figure 40: The QuestEvent superclass. All custom quest events must extend that class.

The first methods that the developer must work on, are the **SetActive** and **SetInactive**. As their name implies, those are responsible for activating and deactivating the current QuestEvent, respectively. The SetActive method takes as input a Quest parameter, which signifies the Quest on which the QE is part of. This function is mostly used to instantiate the QuestEvent and to subscribe to the EntityEventBroker's events, that it should listen to. The SetInactive method, is used to end the quest event and clear all residual data that reside in that. It is also used to unsubscribe to the events, the SetActive subscribed to. An example of using those functions is shown in Figure 41.

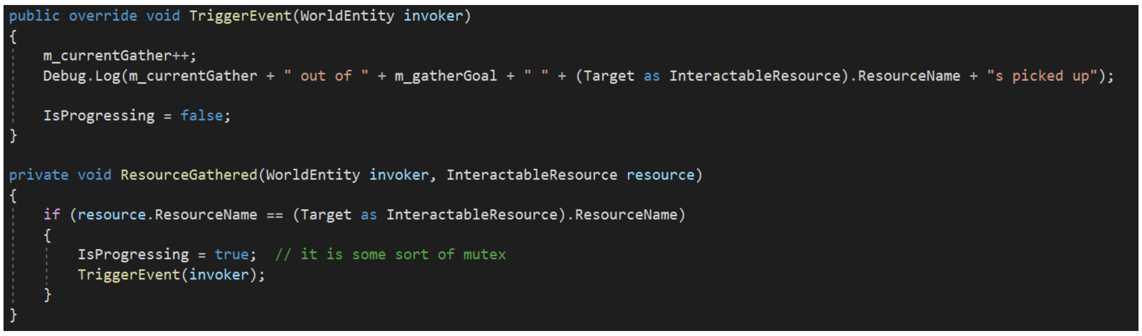
The next method to be implemented is the **TriggerEvent**. This function is



```
public override void SetActive(Quest quest)
{
    IsActive = true;
    Quest = quest;
    // Subscribing to EntityEventBroker's OnResourcePickUp event
    EntityEventBroker.OnResourcePickUp += ResourceGathered;

    m_currentGather = 0;
    Debug.Log("You need to gather " + m_gatherGoal + " of type " + (Target as InteractableResource).ResourceName);
}

public override void SetInactive()
{
    IsActive = false;

    // Unsubscribing from EntityEventBroker's OnResourcePickUp event
    EntityEventBroker.OnResourcePickUp -= ResourceGathered;
}
```

Figure 41: Implementation of SetActive and SetInactive methods on the Resource Gather quest event.

responsible for what is happening the when the current event is triggered. It is set as public, to give the ability to the developer to call it from another class. It can also be called from another method inside the QuestEvent class which is mostly the case for this demo. The approach followed, was to subscribe to the EntityEventBroker, using a private function which checks if the event should be triggered or not. If it should, then call the TriggerEvent function from inside the QuestEvent class, as shown in Figure 42.

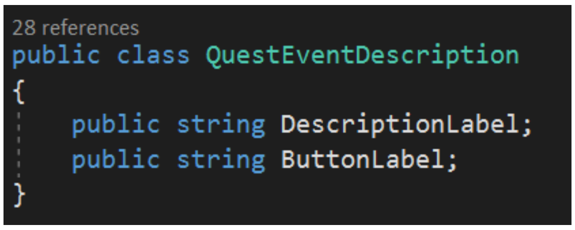


```
public override void TriggerEvent(WorldEntity invoker)
{
    m_currentGather++;
    Debug.Log(m_currentGather + " out of " + m_gatherGoal + " " + (Target as InteractableResource).ResourceName + "s picked up");

    IsProgressing = false;
}

private void ResourceGathered(WorldEntity invoker, InteractableResource resource)
{
    if (resource.ResourceName == (Target as InteractableResource).ResourceName)
    {
        IsProgressing = true;  // it is some sort of mutex
        TriggerEvent(invoker);
    }
}
```

Figure 42: Triggering the Resource Gather quest event from an inside private function.

The **CanProgressQuest** method is used in a mutex style of way, to allow the event to fully trigger before it is progressed by the QuestManager. When a world event is triggered, both the QuestEvent and the QuestManager are triggered. That means that the QE could be potentially triggered **after** the Quest has progressed, which could lead to consistency issues. So, before the QuestManager can progress, it will call the CanProgressQuest method to check if it can do so. The developer, can implement a condition where the QuestManager should wait before it moves to the next QE, or they could just set it to true, if there is no need to do that check.

Finally, the **GetQuestEventDescription** function returns a brief description of the quest. It was used for the demo so that the UI pop up can have a brief description of the quest event, so that the player will know how to progress. The QuestEventDescription class has two variables, the DescriptionLabel, which is the body of the description, and the ButtonLabel, which is the text that appears in the interaction button of the UI.



```
28 references
public class QuestEventDescription
{
    public string DescriptionLabel;
    public string ButtonLabel;
}
```

Figure 43: The QuestEventDescription class. It is a helper class for the demo's purposes.

With all those tools at their disposal, both a designer and a developer can introduce this system to their own project.

# 5 Implementation

In this section, a more detailed description of the overall structure of this thesis will be provided. Specifically, the first part will give an overview of how the Publisher Subscriber Pattern was implemented, introduce all respective components and how they are connected together, as well as show how events flow from the Interactables to the Quest management system and to the world graph and vice versa.

At the end of this section, there will be a more detailed description of the custom World Graph Editor, as well as a showcase of the demo quests that were created for this thesis' demo. Lastly, we will see how a completion of a quest influences the world graph.

## 5.1 Overview of overall structure

This section is dedicated to the detailed description of the system's structure, the interaction between individual components of the system via the Entity Event Broker as well as a brief description of those components.

The structure of this system contains three distinct parts: the Quest Scheduler, the World Graph and the Interactables, all of which are connected by the Entity Event Broker as shown in Figure 44.

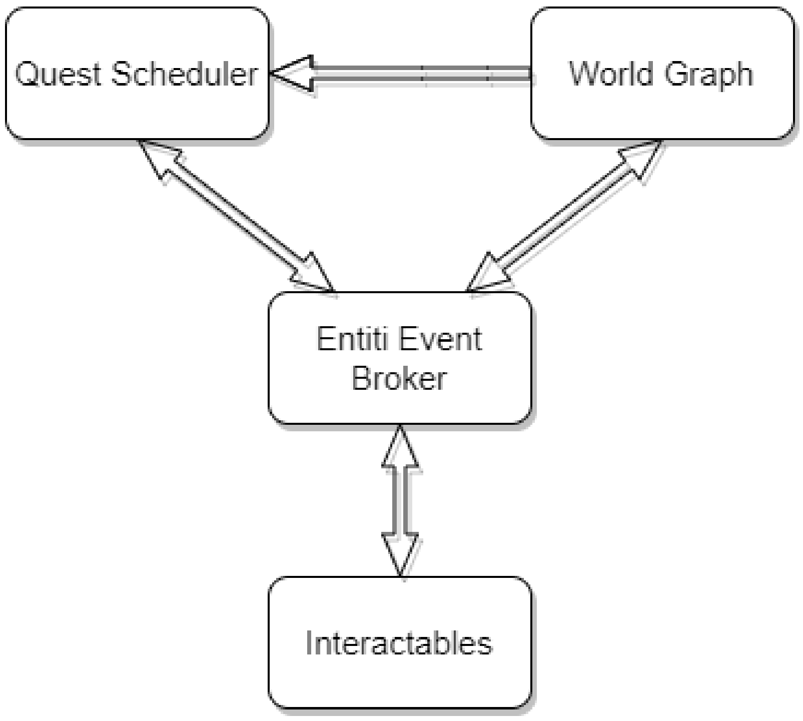


Figure 44: Basic overview of the structure of the system.

The basic flow of information between the components is:

- Interactables send events to the Entity Event Broker
- The Entity Event Broker splits the event between the Quest Scheduler and the World Graph, depending on who needs it
- Either the Quest Scheduler or the World Graph returns an event back to the Interactable
- The Quest Scheduler requests information from the World Graph in order to create a Quest

On the next subsections we will look more on each of the aforementioned components and describe what kind of information they pass between them.

## 5.2 World Graph and Graph Handler

### 5.2.1 Graph Handler

In this section, we will explore more on how the World Graph component is structured, how it works internally and how it shares information with the other components.

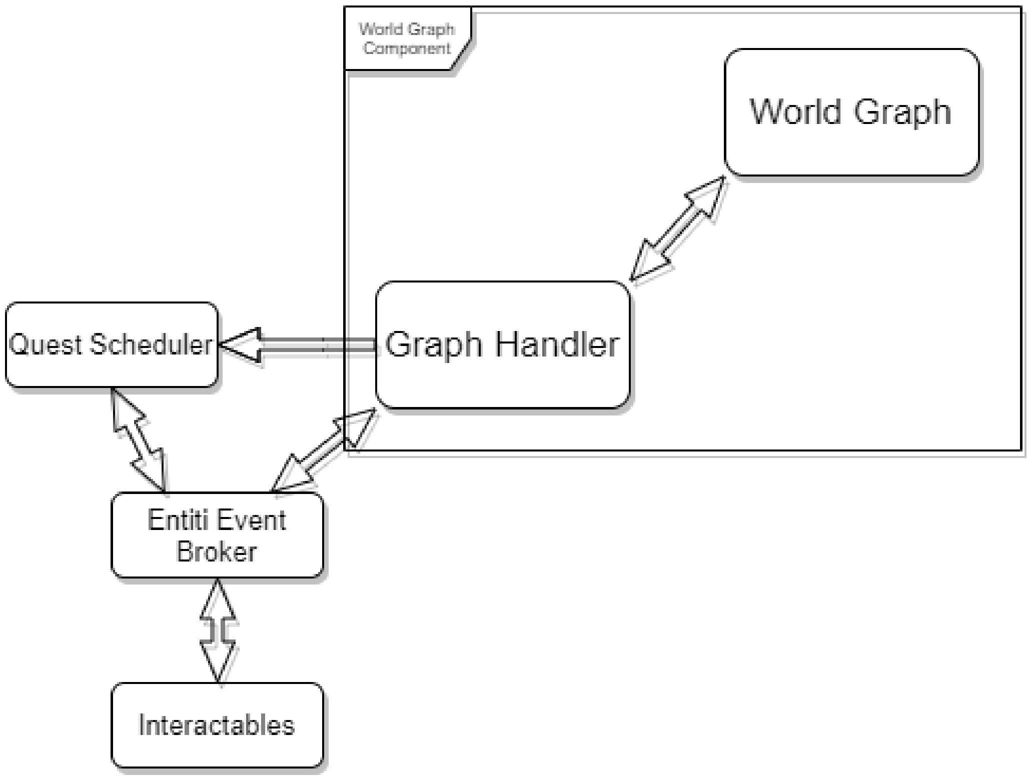


Figure 45: The structure of the world graph component. It is comprised of the Graph Handler and the World Graph classes.

Internally, the World Graph component, consists of two main classes, the World Graph, which shall be mention as **Graph** from now on, and the Graph Handler which extends Unity's MonoBehaviour class, as shown in Figure 45. Having in mind a key principle in OOP, which is Separation of Concerns, the Graph Handler acts as a mediator between other components and the main graph.

The reasoning behind using the mediator design pattern, for that component is to restrict any direct access to the Graph class, which is solely used to search in a graph structure. In addition, it would also introduce the need to for the Graph to be able to translate its search results appropriately the fit the needs of the calling component, which would then introduce additional complexity to the class. That created the need to have a class which could handle all incoming requests to the World Graph and translate the results appropriately to fit the needs of the invoker, hence the implementation of the Mediator pattern.

First, let's analyze the Graph Handler class. Since this class extends the MonoBehaviour class, it has to be added to a component. As it was described in previous sections, the Graph Handler class has one public variable of type `SG_Graph`. That class is used in the custom World Graph Editor where, it the editor, we create such a graph and pass it as input in the public variable of the Graph Handler.

The Graph Handler also contains two private member variables:

- `m_graph` of type **Graph** which keeps a reference to the games world graph instance.
- `m_interactables` of type **Dictionary<int, Interactable>** which keeps a reference of all the Interactables and their indexes.

Those variables are instantiated when the Awake function is being called from Unity. The Awake function is one of the many callback functions of Unity's API. It is called during the initialization of each class that extends the MonoBehaviour class - which the Graph Handler does - and implements the Awake function. Another reason that Awake was used instead of Start, which is another callback used for initialization, is that Start is used by most game objects in Unity including Interactables. In Figure 46, we can see that the execution order is calling first the Awake function and then the Start function. So using Awake instead of Start in the Graph Handler, makes sure that the world graph would be created before initializing the Interactables, so they can be referenced correctly.

Inside the Awake function, two functions are being called, the **InitGraph()** and the **SubscribeToEvents()**. The InitGraph function is using the public variable Space Graph to create an internal world graph that will be used during runtime. It will check first if the variable is populated and if there are nodes inside the graph given, and then it will start initializing the world graph. The SubscribeToEvents function is used to subscribe to all necessary events on the Entity Event Broker. The implementation of those two functions is shown in Figure 47.

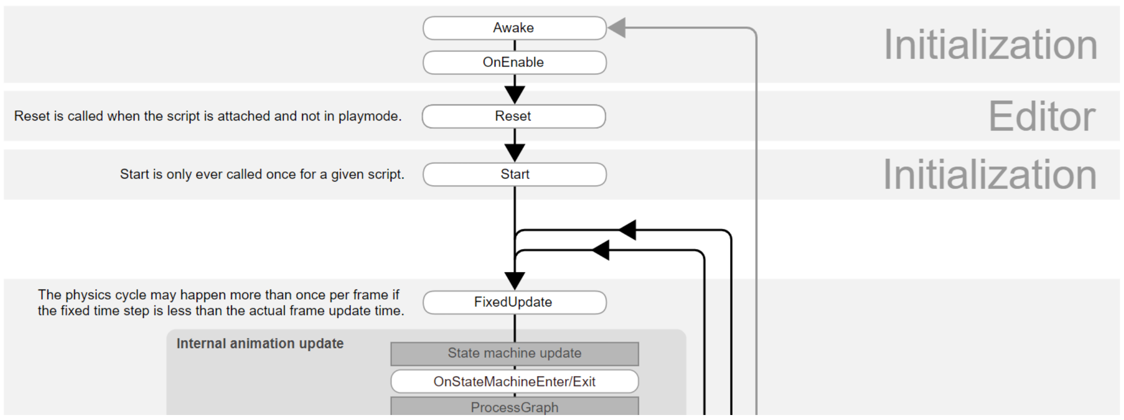
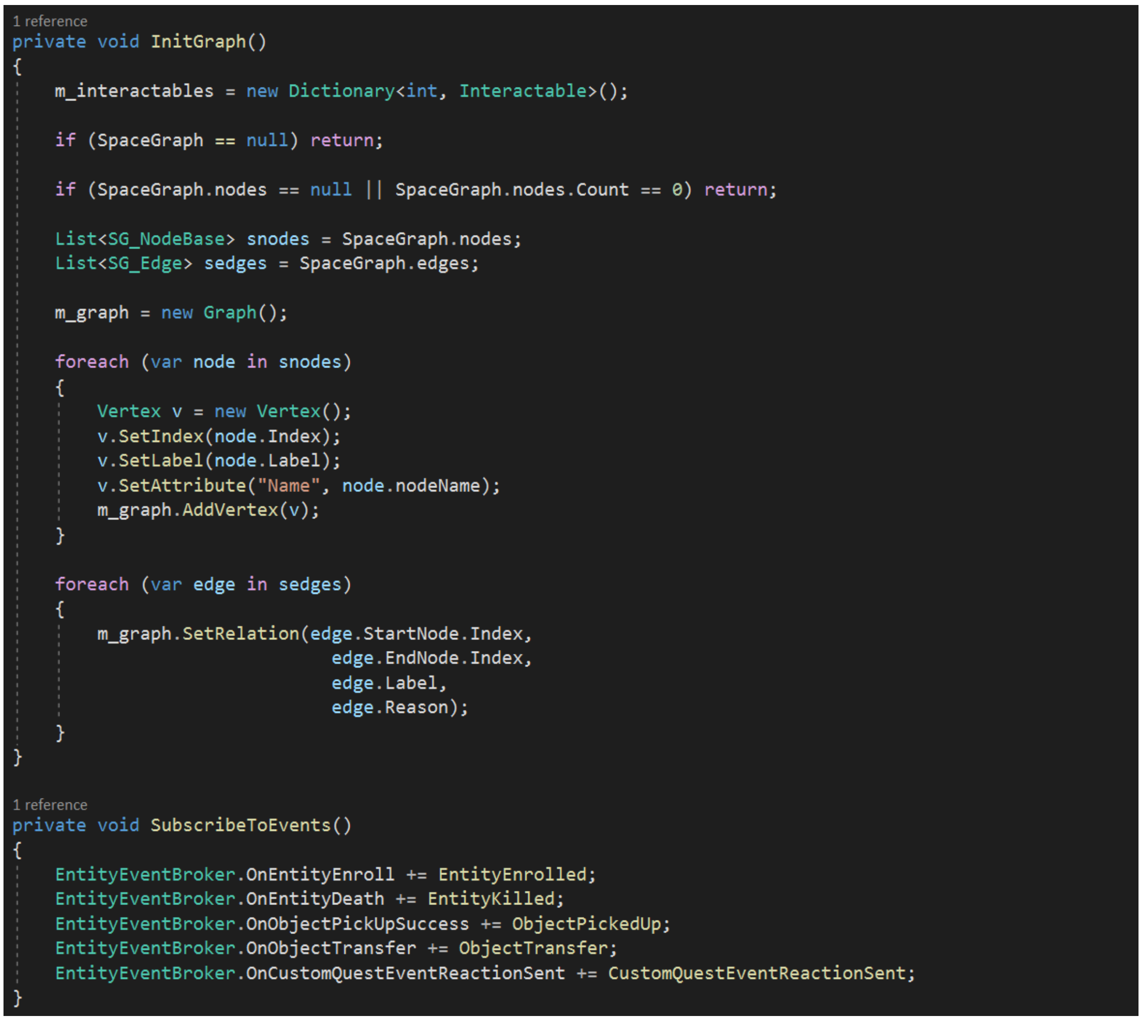

Initialization
Editor
Initialization
Awake
OnEnable
Reset is called when the script is attached and not in playmode.
Reset
Start is only ever called once for a given script.
Start
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph

Figure 46: Part of the callback functions of Unity's API and their execution order. For a complete list visit https://docs.unity3d.com/Manual/ExecutionOrder.html

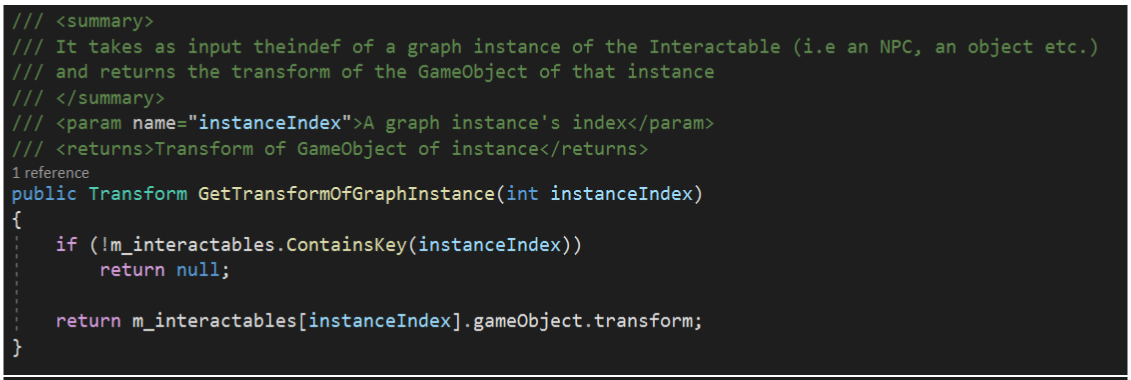


```
1 reference
private void InitGraph()
{
    m_interactables = new Dictionary<int, Interactable>();

    if (SpaceGraph == null) return;

    if (SpaceGraph.nodes == null || SpaceGraph.nodes.Count == 0) return;

    List<SG_NodeBase> snodes = SpaceGraph.nodes;
    List<SG_Edge> sedges = SpaceGraph.edges;

    m_graph = new Graph();

    foreach (var node in snodes)
    {
        Vertex v = new Vertex();
        v.SetIndex(node.Index);
        v.SetLabel(node.Label);
        v.SetAttribute("Name", node.nodeName);
        m_graph.AddVertex(v);
    }

    foreach (var edge in sedges)
    {
        m_graph.SetRelation(edge.StartNode.Index,
                            edge.EndNode.Index,
                            edge.Label,
                            edge.Reason);
    }
}

1 reference
private void SubscribeToEvents()
{
    EntityEventBroker.OnEntityEnroll += EntityEnrolled;
    EntityEventBroker.OnEntityDeath += EntityKilled;
    EntityEventBroker.OnObjectPickUpSuccess += ObjectPickedUp;
    EntityEventBroker.OnObjectTransfer += ObjectTransfer;
    EntityEventBroker.OnCustomQuestEventReactionSent += CustomQuestEventReactionSent;
}
```

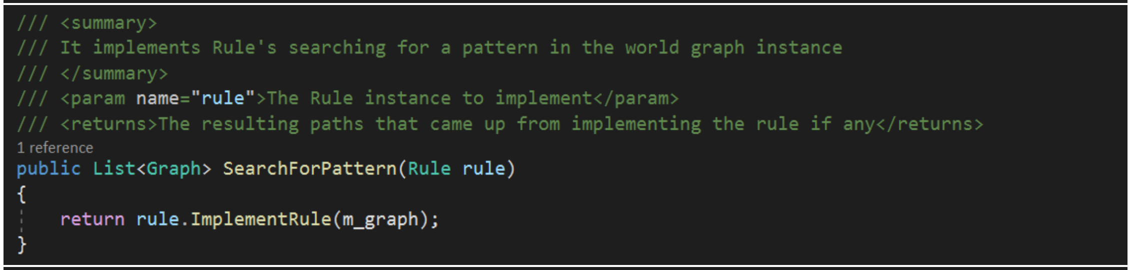
Figure 47: The InitGraph and SubscribeToEvents functions of the Graph Handler.

Most of the events that the Graph Handler is subscribed to, are created for the demo's purposes. The most important one is the OnEntityEnroll which is the one used by the Interactables when they get initialized. When the event is invoked, it is listened by the Graph Handler, it gets the Interactable's object and index and adds them to the `m_interactables` dictionary. The rest of the events are called in case something happens to an Interactable i.e it gets killed, it picks something up etc.

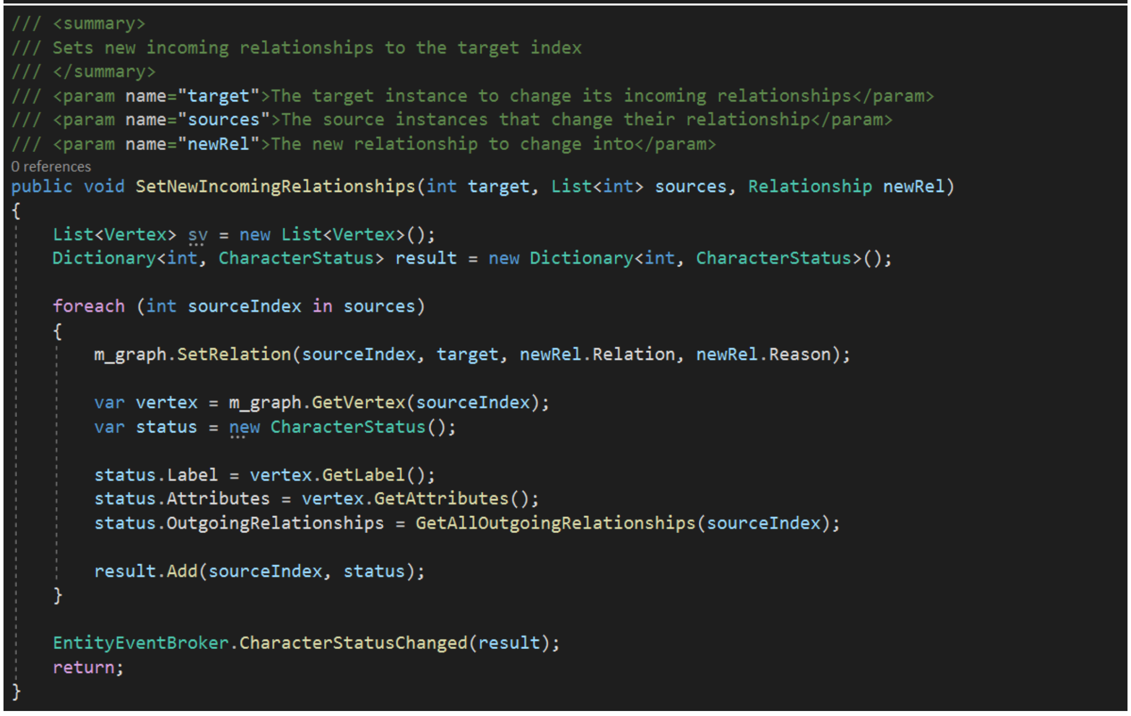
Once all Interactables are enrolled to the Graph Handle, the class provides multiple public functions that are purposed on providing information on the world graph, modify it and traverse it in order to find suitable patterns. A comprehensive list of those functions and their inputs and outputs will be provided below.



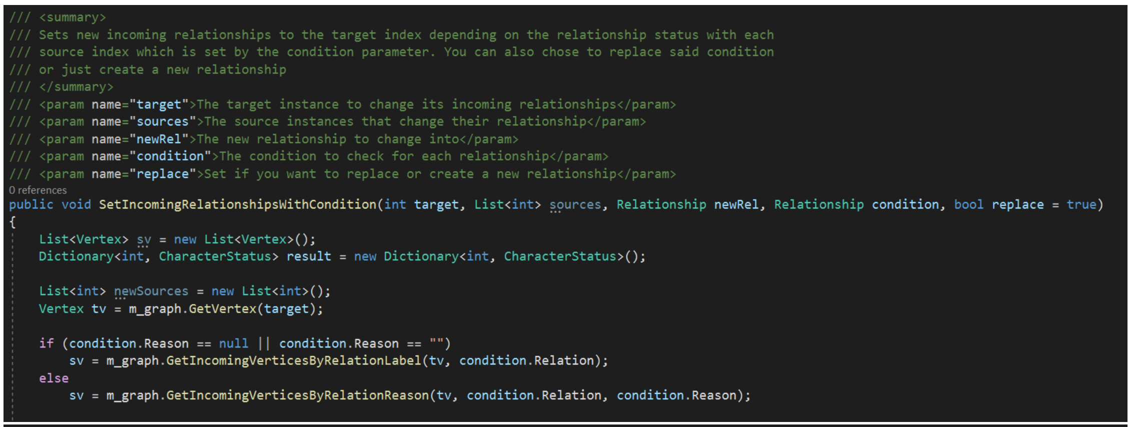
```
/// <summary>
/// It takes as input theindef of a graph instance of the Interactable (i.e an NPC, an object etc.)
/// and returns the transform of the GameObject of that instance
/// </summary>
/// <param name="instanceIndex">A graph instance's index</param>
/// <returns>Transform of GameObject of instance</returns>
1 reference
public Transform GetTransformOfGraphInstance(int instanceIndex)
{
    if (!m_interactables.ContainsKey(instanceIndex))
        return null;

    return m_interactables[instanceIndex].gameObject.transform;
}
```



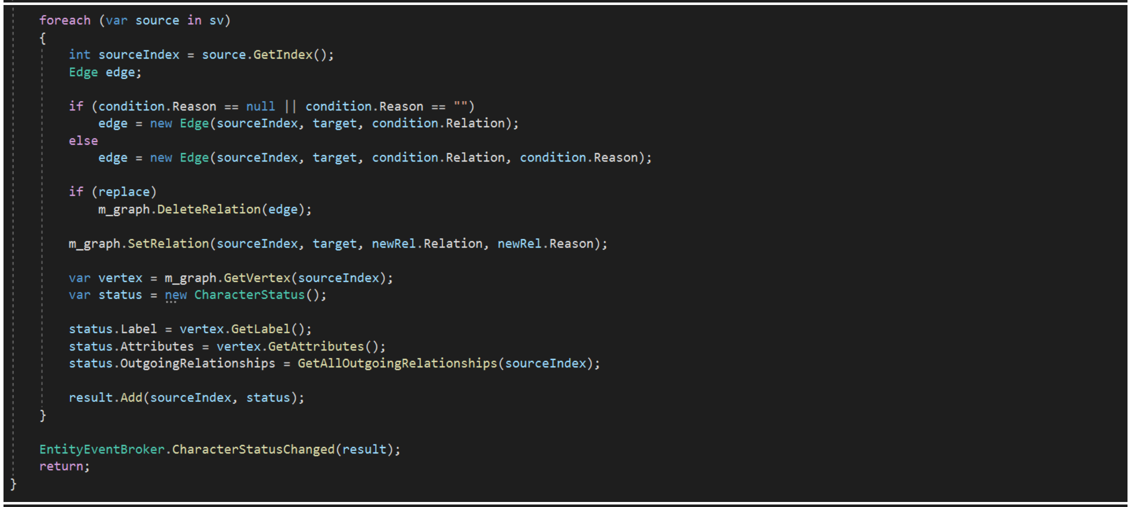
```
/// <summary>
/// It implements Rule's searching for a pattern in the world graph instance
/// </summary>
/// <param name="rule">The Rule instance to implement</param>
/// <returns>The resulting paths that came up from implementing the rule if any</returns>
1 reference
public List<Graph> SearchForPattern(Rule rule)
{
    return rule.ImplementRule(m_graph);
}
```



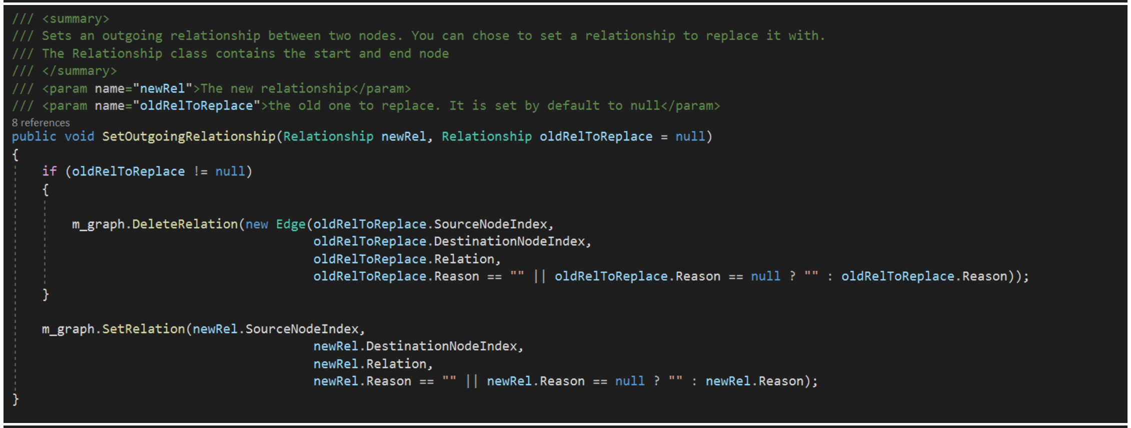
```
/// <summary>
/// Sets new incoming relationships to the target index
/// </summary>
/// <param name="target">The target instance to change its incoming relationships</param>
/// <param name="sources">The source instances that change their relationship</param>
/// <param name="newRel">The new relationship to change into</param>
0 references
public void SetNewIncomingRelationships(int target, List<int> sources, Relationship newRel)
{
    List<Vertex> sv = new List<Vertex>();
    Dictionary<int, CharacterStatus> result = new Dictionary<int, CharacterStatus>();

    foreach (int sourceIndex in sources)
    {
        m_graph.SetRelation(sourceIndex, target, newRel.Relation, newRel.Reason);

        var vertex = m_graph.GetVertex(sourceIndex);
        var status = new CharacterStatus();

        status.Label = vertex.GetLabel();
        status.Attributes = vertex.GetAttributes();
        status.OutgoingRelationships = GetAllOutgoingRelationships(sourceIndex);

        result.Add(sourceIndex, status);
    }

    EntityEventBroker.CharacterStatusChanged(result);
    return;
}
```



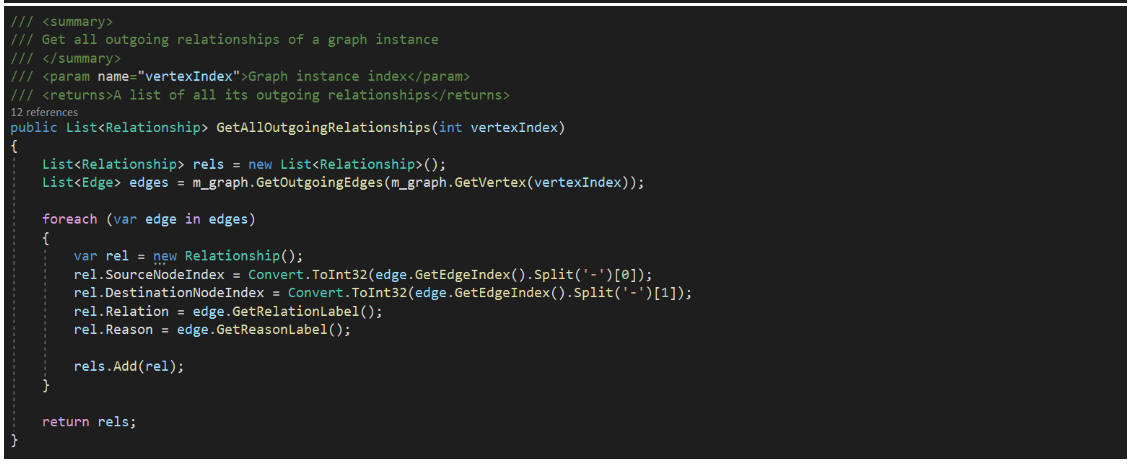
```
/// <summary>
/// Sets new incoming relationships to the target index depending on the relationship status with each
/// source index which is set by the condition parameter. You can also chose to replace said condition
/// or just create a new relationship
/// </summary>
/// <param name="target">The target instance to change its incoming relationships</param>
/// <param name="sources">The source instances that change their relationship</param>
/// <param name="newRel">The new relationship to change into</param>
/// <param name="condition">The condition to check for each relationship</param>
/// <param name="replace">Set if you want to replace or create a new relationship</param>
0 references
public void SetIncomingRelationshipsWithCondition(int target, List<int> sources, Relationship newRel, Relationship condition, bool replace = true)
{
    List<Vertex> sv = new List<Vertex>();
    Dictionary<int, CharacterStatus> result = new Dictionary<int, CharacterStatus>();

    List<int> newSources = new List<int>();
    Vertex tv = m_graph.GetVertex(target);

    if (condition.Reason == null || condition.Reason == "")
        sv = m_graph.GetIncomingVerticesByRelationLabel(tv, condition.Relation);
    else
        sv = m_graph.GetIncomingVerticesByRelationReason(tv, condition.Relation, condition.Reason);
```



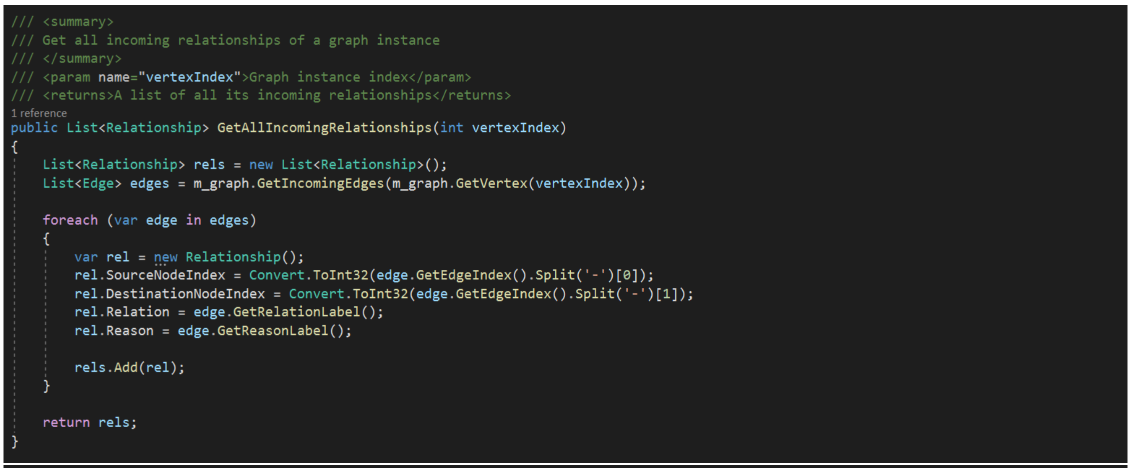
```
    foreach (var source in sv)
    {
        int sourceIndex = source.GetIndex();
        Edge edge;

        if (condition.Reason == null || condition.Reason == "")
            edge = new Edge(sourceIndex, target, condition.Relation);
        else
            edge = new Edge(sourceIndex, target, condition.Relation, condition.Reason);

        if (replace)
            m_graph.DeleteRelation(edge);

        m_graph.SetRelation(sourceIndex, target, newRel.Relation, newRel.Reason);

        var vertex = m_graph.GetVertex(sourceIndex);
        var status = new CharacterStatus();

        status.Label = vertex.GetLabel();
        status.Attributes = vertex.GetAttributes();
        status.OutgoingRelationships = GetAllOutgoingRelationships(sourceIndex);

        result.Add(sourceIndex, status);
    }

    EntityEventBroker.CharacterStatusChanged(result);
    return;
}
```



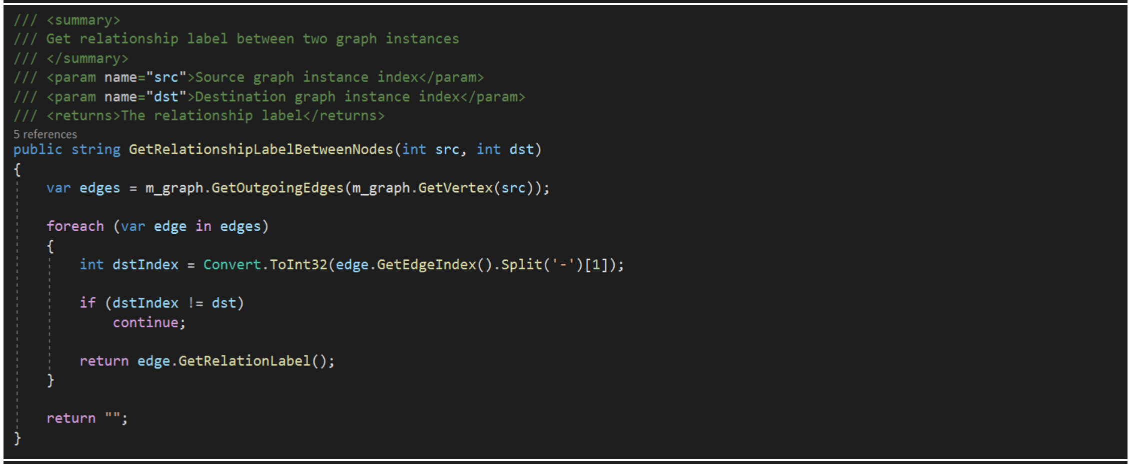
```
/// <summary>
/// Sets an outgoing relationship between two nodes. You can chose to set a relationship to replace it with.
/// The Relationship class contains the start and end node
/// </summary>
/// <param name="newRel">The new relationship</param>
/// <param name="oldRelToReplace">the old one to replace. It is set by default to null</param>
8 references
public void SetOutgoingRelationship(Relationship newRel, Relationship oldRelToReplace = null)
{
    if (oldRelToReplace != null)
    {

        m_graph.DeleteRelation(new Edge(oldRelToReplace.SourceNodeIndex,
                                        oldRelToReplace.DestinationNodeIndex,
                                        oldRelToReplace.Relation,
                                        oldRelToReplace.Reason == "" || oldRelToReplace.Reason == null ? "" : oldRelToReplace.Reason));

    }

    m_graph.SetRelation(newRel.SourceNodeIndex,
                                        newRel.DestinationNodeIndex,
                                        newRel.Relation,
                                        newRel.Reason == "" || newRel.Reason == null ? "" : newRel.Reason);
}
```



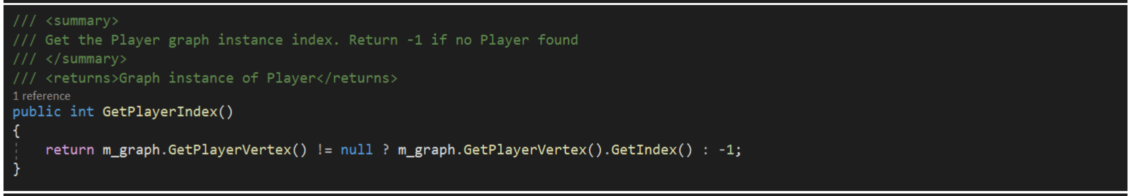
```
/// <summary>
/// Get all outgoing relationships of a graph instance
/// </summary>
/// <param name="vertexIndex">Graph instance index</param>
/// <returns>A list of all its outgoing relationships</returns>
12 references
public List<Relationship> GetAllOutgoingRelationships(int vertexIndex)
{
    List<Relationship> rels = new List<Relationship>();
    List<Edge> edges = m_graph.GetOutgoingEdges(m_graph.GetVertex(vertexIndex));

    foreach (var edge in edges)
    {
        var rel = new Relationship();
        rel.SourceNodeIndex = Convert.ToInt32(edge.GetEdgeIndex().Split('-')[0]);
        rel.DestinationNodeIndex = Convert.ToInt32(edge.GetEdgeIndex().Split('-')[1]);
        rel.Relation = edge.GetRelationLabel();
        rel.Reason = edge.GetReasonLabel();

        rels.Add(rel);
    }

    return rels;
}
```



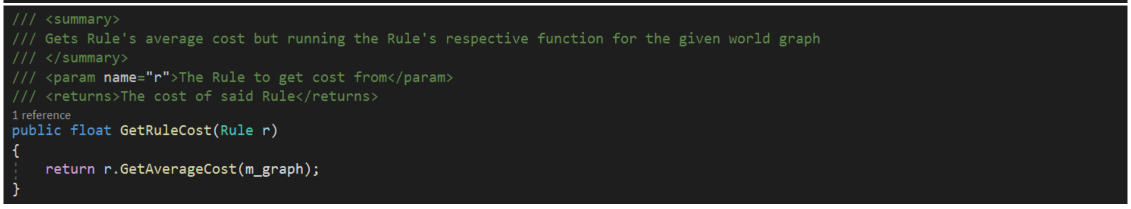
```
/// <summary>
/// Get all incoming relationships of a graph instance
/// </summary>
/// <param name="vertexIndex">Graph instance index</param>
/// <returns>A list of all its incoming relationships</returns>
1 reference
public List<Relationship> GetAllIncomingRelationships(int vertexIndex)
{
    List<Relationship> rels = new List<Relationship>();
    List<Edge> edges = m_graph.GetIncomingEdges(m_graph.GetVertex(vertexIndex));

    foreach (var edge in edges)
    {
        var rel = new Relationship();
        rel.SourceNodeIndex = Convert.ToInt32(edge.GetEdgeIndex().Split('-')[0]);
        rel.DestinationNodeIndex = Convert.ToInt32(edge.GetEdgeIndex().Split('-')[1]);
        rel.Relation = edge.GetRelationLabel();
        rel.Reason = edge.GetReasonLabel();

        rels.Add(rel);
    }

    return rels;
}
```



```
/// <summary>
/// Get relationship label between two graph instances
/// </summary>
/// <param name="src">Source graph instance index</param>
/// <param name="dst">Destination graph instance index</param>
/// <returns>The relationship label</returns>
5 references
public string GetRelationshipLabelBetweenNodes(int src, int dst)
{
    var edges = m_graph.GetOutgoingEdges(m_graph.GetVertex(src));

    foreach (var edge in edges)
    {
        int dstIndex = Convert.ToInt32(edge.GetEdgeIndex().Split('-')[1]);

        if (dstIndex != dst)
            continue;

        return edge.GetRelationLabel();
    }

    return "";
}
```



```
/// <summary>
/// Get the Player graph instance index. Return -1 if no Player found
/// </summary>
/// <returns>Graph instance of Player</returns>
1 reference
public int GetPlayerIndex()
{
    return m_graph.GetPlayerVertex() != null ? m_graph.GetPlayerVertex().GetIndex() : -1;
}
```

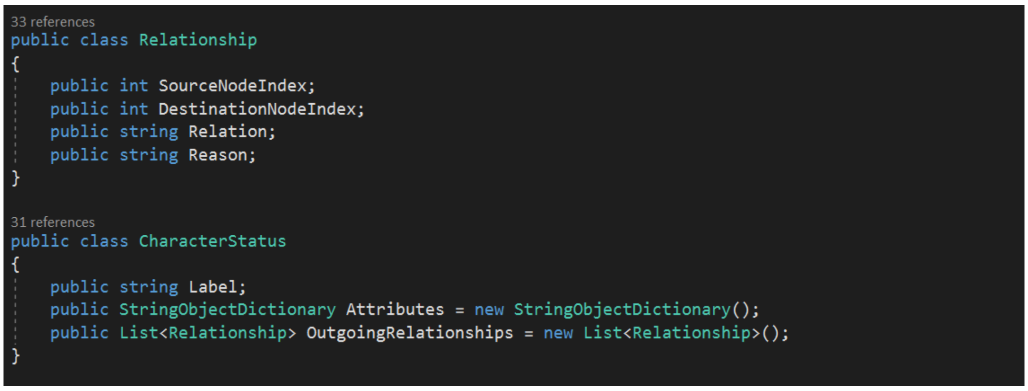


```
/// <summary>
/// Gets Rule's average cost but running the Rule's respective function for the given world graph
/// </summary>
/// <param name="r">The Rule to get cost from</param>
/// <returns>The cost of said Rule</returns>
1 reference
public float GetRuleCost(Rule r)
{
    return r.GetAverageCost(m_graph);
}
```

All functions that are shown above are calling the API provided in the Graph class of which the world graph is an instance of. All of them will be explained in detail in the next section. Additionally, for the previous functions, there were two additional helper classes that are shown in Figure 51. In the next section, the Graph class, of which the world graph is an instance of, will be given a detailed description on how it operates and its provided API.

### 5.2.2 Graph class - World Graph

In the previous section, the Graph Handler was described and how it interacted with outside components i.e Interactables, and how it called the API the Graph class provided, to search for patterns. In this section, we shall describe how the said

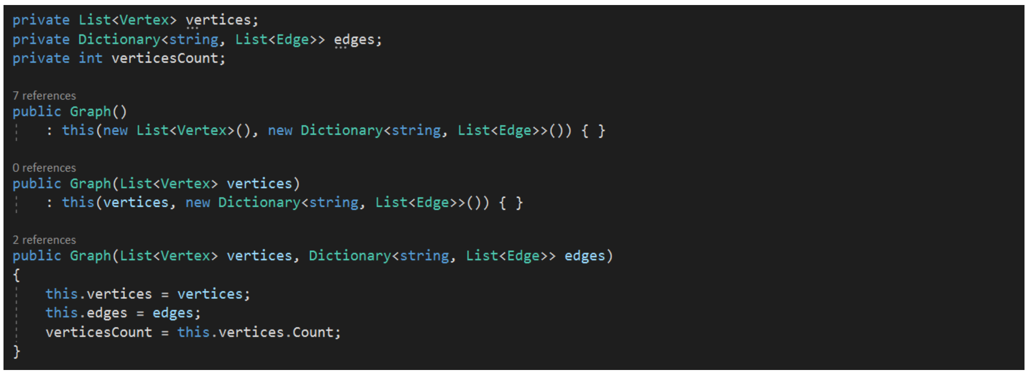


```
33 references
public class Relationship
{
    public int SourceNodeIndex;
    public int DestinationNodeIndex;
    public string Relation;
    public string Reason;
}

31 references
public class CharacterStatus
{
    public string Label;
    public StringObjectDictionary Attributes = new StringObjectDictionary();
    public List<Relationship> OutgoingRelationships = new List<Relationship>();
}
```

Figure 51: Helper classes of Graph Handler.

API works internally and how the Graph class stores data. The terms Graph class and world graph are interchangeable since the later is an instance of the former.

The concept of this thesis dictates to have a graph like form of a world and to have that graph being traversed in search for patterns inside it. So, first of all the Graph class should store a list of nodes and a list of edges, keeping data for all entities and the relationships between them, respectively. Those members and the respective constructors, initializing those member variables, are shown in Figure 52.

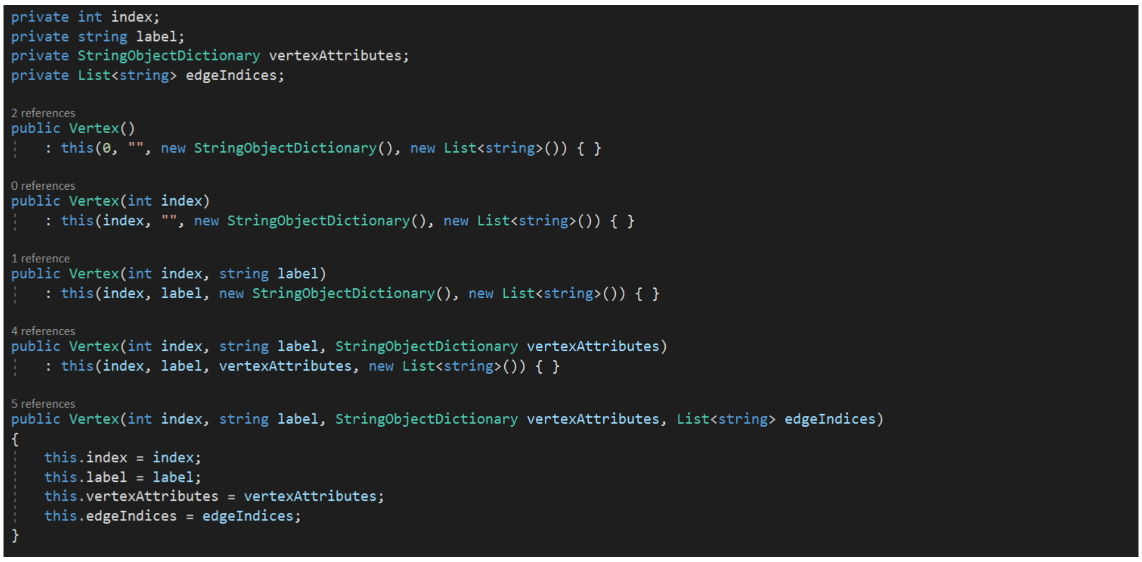


```
private List<Vertex> vertices;
private Dictionary<string, List<Edge>> edges;
private int verticesCount;

7 references
public Graph()
    : this(new List<Vertex>(), new Dictionary<string, List<Edge>>()) { }

0 references
public Graph(List<Vertex> vertices)
    : this(vertices, new Dictionary<string, List<Edge>>()) { }

2 references
public Graph(List<Vertex> vertices, Dictionary<string, List<Edge>> edges)
{
    this.vertices = vertices;
    this.edges = edges;
    verticesCount = this.vertices.Count;
}
```

Figure 52: Member variables and constructors of Graph class.

As shown in Figure 52, the Graph class has three member variables, **vertices** which is a list of type **Vertex**, **edges** which is a dictionary with a key of type **string** and a value of a list of type **Edge** and a **verticesCount** which is an integer that keeps the count on vertices. As for the constructors, they are invoking an overloaded constructor that initializes the aforementioned member variables depending on its input.

Before describing the inner workings of the Graph class, it is best if the Vertex and Edge classes are described in more detail. Starting with the Vertex class, as displayed in Figure 52, the member variables store information on the index of the vertex, its label - whether it is an NPC, Object etc. -, its attributes and a list of Edge indices that are connected to the Vertex whether they are incoming or outgoing. As for its constructors, they are all invoking the main overloaded constructor that initializes the member variables depending on its inputs. Lastly, all of its functions are its getters and setters, meaning they are only getting and setting said member variables.



```
private int index;
private string label;
private StringObjectDictionary vertexAttributes;
private List<string> edgeIndices;

2 references
public Vertex()
    : this(0, "", new StringObjectDictionary(), new List<string>()) { }

0 references
public Vertex(int index)
    : this(index, "", new StringObjectDictionary(), new List<string>()) { }

1 reference
public Vertex(int index, string label)
    : this(index, label, new StringObjectDictionary(), new List<string>()) { }

4 references
public Vertex(int index, string label, StringObjectDictionary vertexAttributes)
    : this(index, label, vertexAttributes, new List<string>()) { }

5 references
public Vertex(int index, string label, StringObjectDictionary vertexAttributes, List<string> edgeIndices)
{
    this.index = index;
    this.label = label;
    this.vertexAttributes = vertexAttributes;
    this.edgeIndices = edgeIndices;
}
```

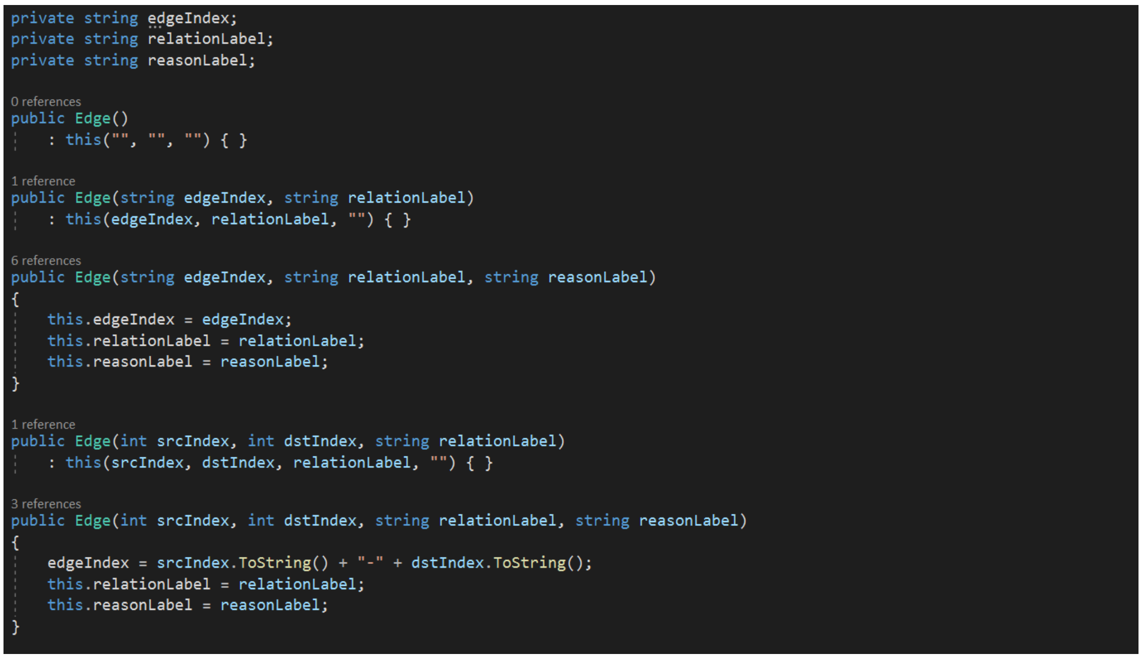
Figure 53: Member variables and constructors of Vertex class.

The vertexAttributes, are of type StringObjectDictionary, which, as the name of the class implies, is a dictionary with a string type key and an object type value, which is the superclass of all objects in C#. The StringObjectDictionary class, extends the SerializableDictionary class which is a class taken from the Unity Asset Store, in order to be able to serialize a dictionary. Unity does not have yet the ability to serialize the dictionary type, which is something that the custom graph editor needs in order to store data to the file system. It behaves like a regular dictionary with the addition of being serializable.

The Edge class, following the same structure as the Vertex class, has three member variables, storing its index, the relation label and reason. The edge index is of type string and contains the information on what the source and target vertices are in the form of "(source index)-(target index)". The constructors invoke two main overloaded constructors, with the main difference between them is having the ability to either give the edge index as a string, or give the source and target indices as input and create the edge index inside the constructor. Additionally, as in the Vertex class, all of its functions are simple getters and setters. Lastly, the edge index explains why the edge list of the Graph class, is a dictionary with a key of type string and a value of type list of edges. Two vertices can have more than one edge which all share the same index but different relationship label and reason.

Having described both Vertex and Edge classes, we can now return to analyzing the Graph class. Having all vertices and edges available, the API mainly modifies and reads them. More specifically, the functions that the Graph class provide are to create a new vertex or an edge, to edit their attributes and general information and it also searches the graph for specific vertices or edges and relationships between them, given a set of parameters i.e the relationship label or reason.

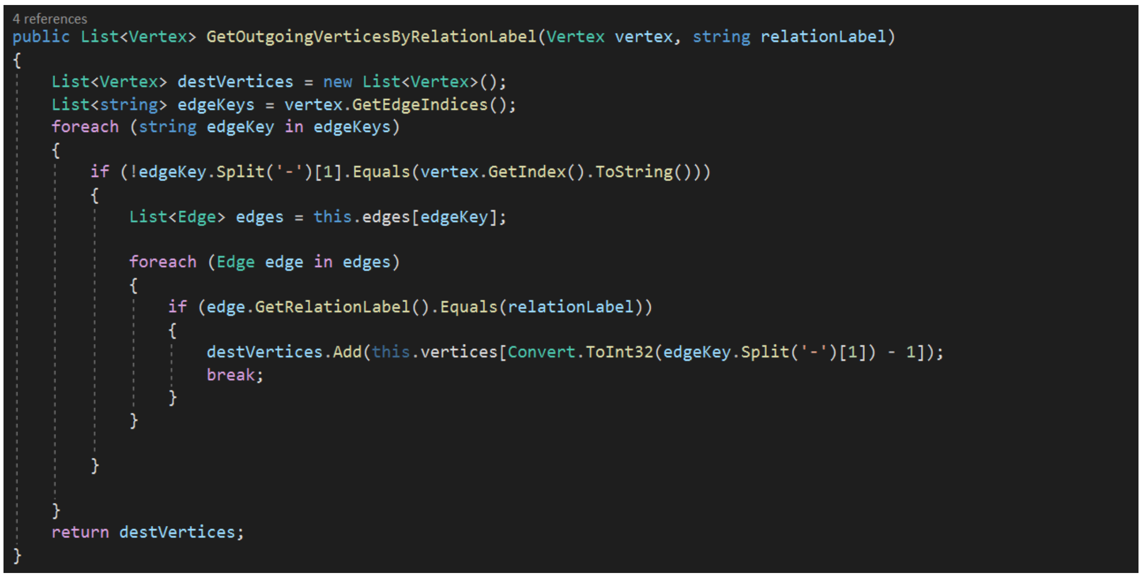
As an example we shall see the public function **GetOutgoingVerticesByRe-**



```
private string edgeIndex;
private string relationLabel;
private string reasonLabel;

0 references
public Edge()
    : this("", "", "") { }

1 reference
public Edge(string edgeIndex, string relationLabel)
    : this(edgeIndex, relationLabel, "") { }

6 references
public Edge(string edgeIndex, string relationLabel, string reasonLabel)
{
    this.edgeIndex = edgeIndex;
    this.relationLabel = relationLabel;
    this.reasonLabel = reasonLabel;
}

1 reference
public Edge(int srcIndex, int dstIndex, string relationLabel)
    : this(srcIndex, dstIndex, relationLabel, "") { }

3 references
public Edge(int srcIndex, int dstIndex, string relationLabel, string reasonLabel)
{
    edgeIndex = srcIndex.ToString() + "-" + dstIndex.ToString();
    this.relationLabel = relationLabel;
    this.reasonLabel = reasonLabel;
}
```

Figure 54: Member variables and constructors of Edge class.

**lationLabel** which returns all vertices that are connected to the target vertex by an outgoing relationship. Then search for the given type of relation label and return the list of vertices that it found. Most other functions, use the same type of implementation, but to do other things like search for outgoing vertices, delete said vertices or create them.



```
4 references
public List<Vertex> GetOutgoingVerticesByRelationLabel(Vertex vertex, string relationLabel)
{
    List<Vertex> destVertices = new List<Vertex>();
    List<string> edgeKeys = vertex.GetEdgeIndices();
    foreach (string edgeKey in edgeKeys)
    {
        if (!edgeKey.Split('-')[1].Equals(vertex.GetIndex().ToString()))
        {
            List<Edge> edges = this.edges[edgeKey];

            foreach (Edge edge in edges)
            {
                if (edge.GetRelationLabel().Equals(relationLabel))
                {
                    destVertices.Add(this.vertices[Convert.ToInt32(edgeKey.Split('-')[1]) - 1]);
                    break;
                }
            }

        }

    }
    return destVertices;
}
```

Figure 55: The implementation of the GetOutgoingVerticesByRelationLabel function.

In conclusion, the Graph class in conjunction with the GraphHandler class, provide an API that allows Rules to search through a world graph, return matching patterns and change the world graph to represent the impact of the player's actions to the world.

## 5.3 Interactables

Having described the world graph component which includes both the GraphHandler and Graph classes, the next step is to describe the Interactables and how they are connected with the Entity Event Broker. Building up on the current structure of the system shown in Figure 45, we have the structure shown in Figure 56, which shows how the Interactables component is structured.

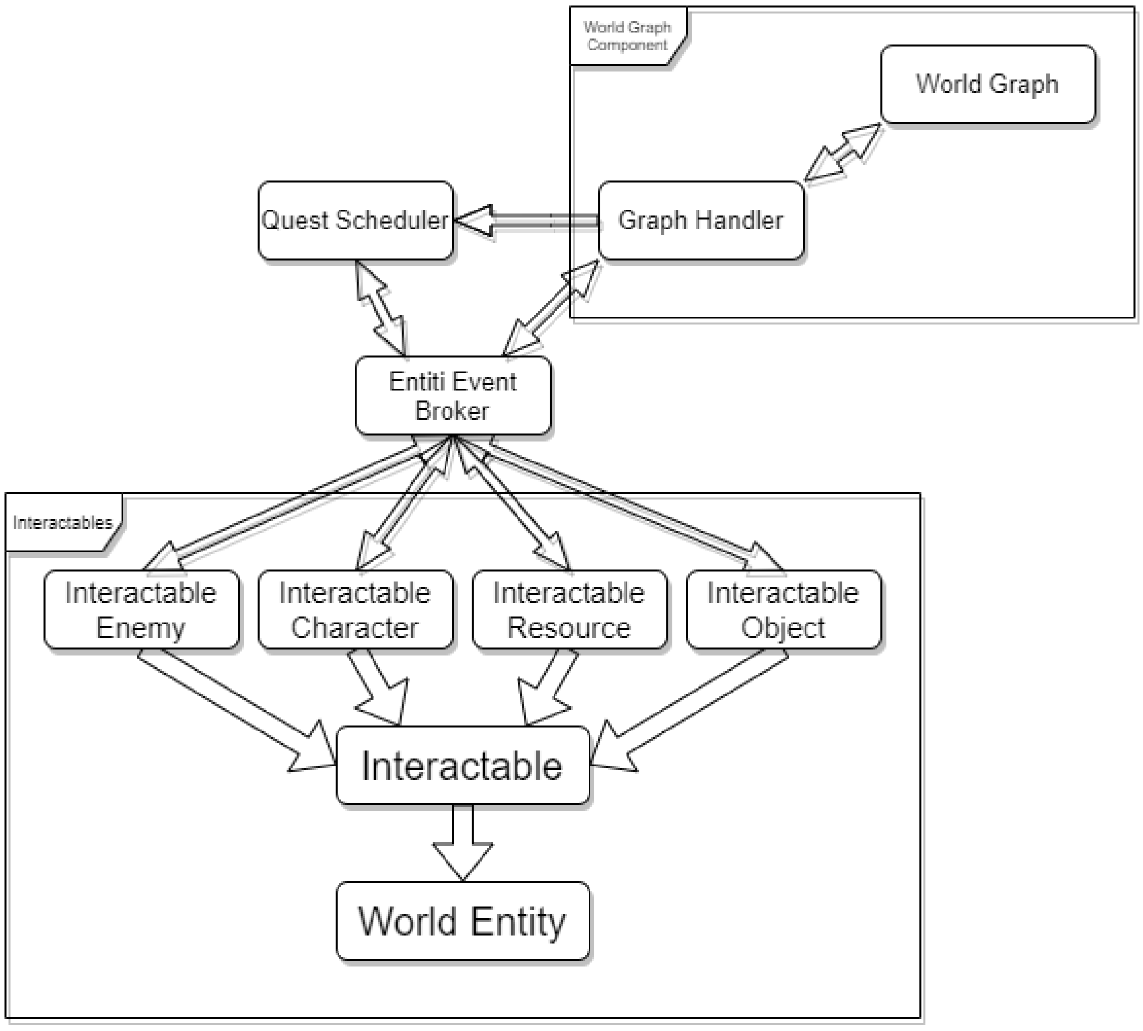
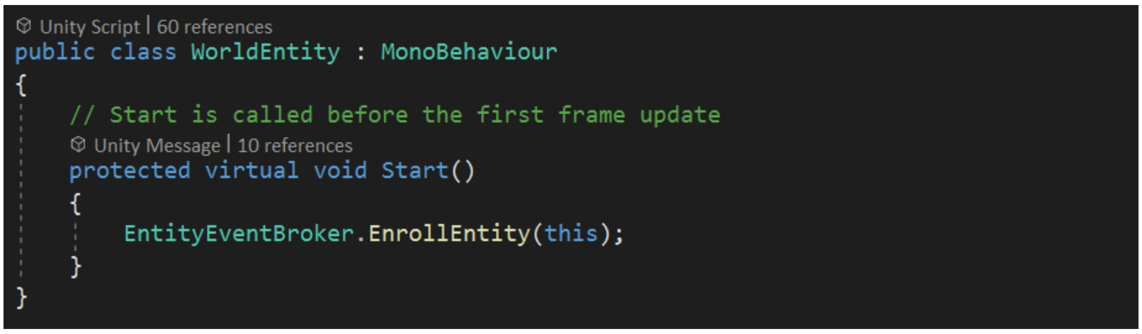


Figure 56: The structure of the Interactables' component.

As shown in the Interactables' component structure, there are three levels of abstraction. The first and highest one is the **World Entity** class which is the superclass of all the Interactables. This class was created to mainly future proof the system in case more graph components were created. That is why the main purpose of this class is to enroll the subclass to the world graph. In Figure 57 we can see the implementation of the World Entity class which comprises of a virtual version of the Start function of Unity which enrolls the entity via the EnrollEntity event of Entity Event Broker. All subclasses that extend the World Entity need to override and call the base Start function.

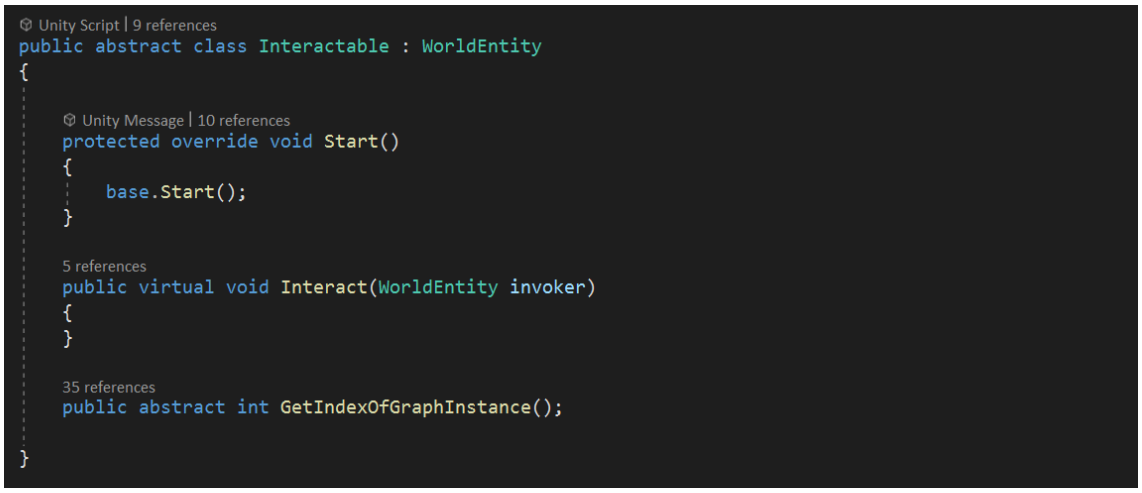
The next part to analyze is the Interactable class which is the base of all the



```
Unity Script | 60 references
public class WorldEntity : MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 10 references
    protected virtual void Start()
    {
        EntityEventBroker.EnrollEntity(this);
    }
}
```

Figure 57: World Entity implementation.

interactable components. This class is as simple as the World Entity class, but it does add more functionality to all classes that extend it. First of all, on Start, it calls the base class which enrolls the Interactable to the World Graph. Next, it adds two more functions, the **Interact**, which is virtual so it can be overridden, and the **GetIndexOfGraphInstance**, which is abstract so it forces all subclasses to implement it. The former function allows the Interactable to be accessible to other game objects while the latter retrieves the graph instance from the world graph.



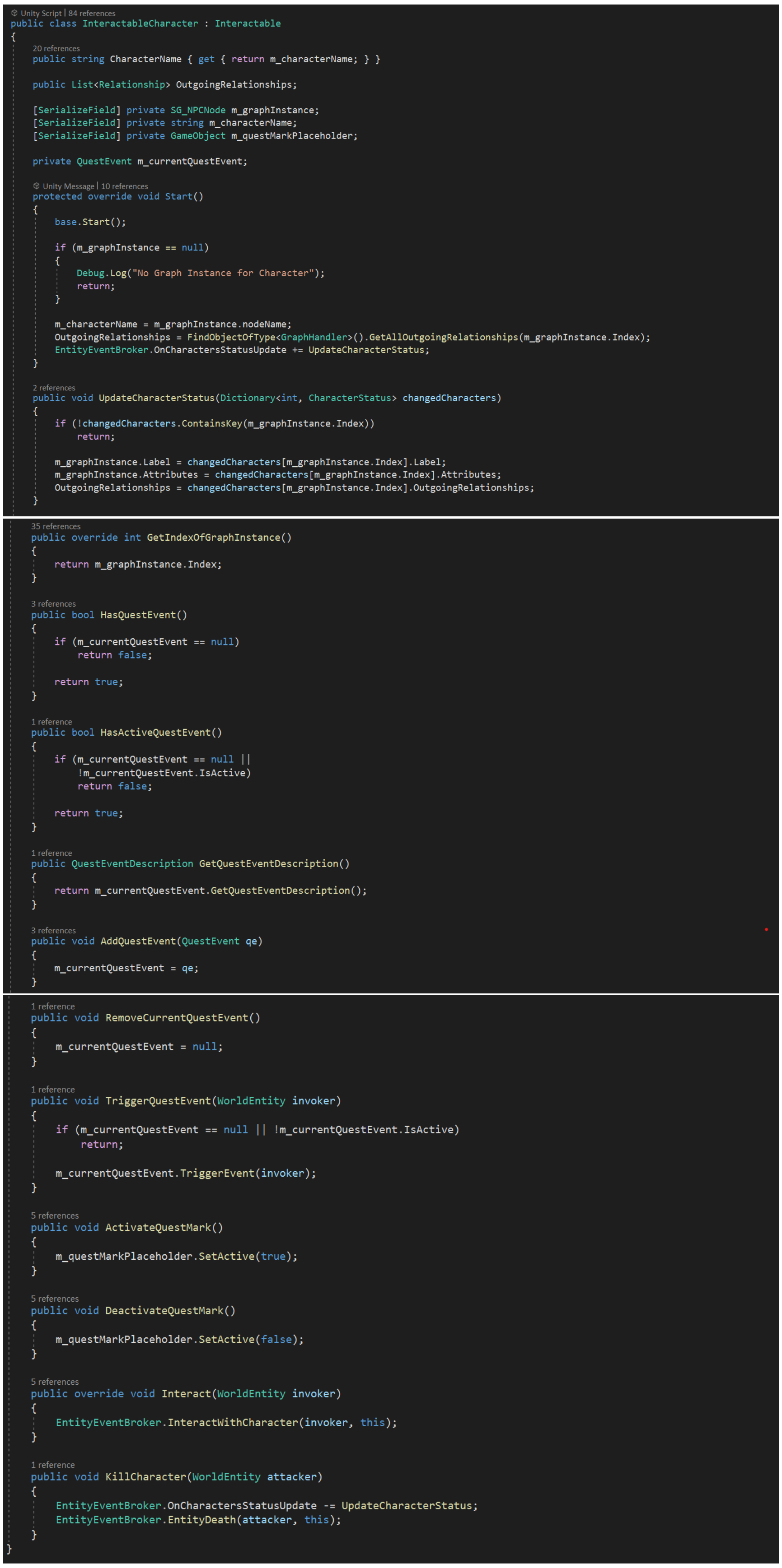
```
Unity Script | 9 references
public abstract class Interactable : WorldEntity
{

    Unity Message | 10 references
    protected override void Start()
    {
        base.Start();
    }

    5 references
    public virtual void Interact(WorldEntity invoker)
    {
    }

    35 references
    public abstract int GetIndexOfGraphInstance();

}
```

Figure 58: Interactable superclass implementation.

As the base classes have been described, we can now start describing the concrete classes which are the **InteractableCharacter**, **InteractableEnemy**, **InteractableObject** and **InteractableResource**. Those four classes are responsible for interacting with the game world and are almost identical in functionality, except on some slight variations and in the case of Interactable Characters.

In the screenshots below in Figure 59, the implementation of the InteractableCharacter class is shown, since it is the most important one and has the most variations compared to the rest of the classes. The InteractableCharacter class has two main concerns to handle, one is the way it interacts with the rest of the world and two, is the way it handles the quests that are assigned to them.

Regarding the first concern, which is how it interacts with the other world objects, it implements the functions passed by the superclass and it also adds some



```
Unity Script | 84 references
public class InteractableCharacter : Interactable
{
    20 references
    public string CharacterName { get { return m_characterName; } }

    public List<Relationship> OutgoingRelationships;

    [SerializeField] private SG_NPCNode m_graphInstance;
    [SerializeField] private string m_characterName;
    [SerializeField] private GameObject m_questMarkPlaceholder;

    private QuestEvent m_currentQuestEvent;

    Unity Message | 10 references
    protected override void Start()
    {
        base.Start();

        if (m_graphInstance == null)
        {
            Debug.Log("No Graph Instance for Character");
            return;
        }

        m_characterName = m_graphInstance.nodeName;
        OutgoingRelationships = FindObjectOfType<GraphHandler>().GetAllOutgoingRelationships(m_graphInstance.Index);
        EntityEventBroker.OnCharactersStatusUpdate += UpdateCharacterStatus;
    }

    2 references
    public void UpdateCharacterStatus(Dictionary<int, CharacterStatus> changedCharacters)
    {
        if (!changedCharacters.ContainsKey(m_graphInstance.Index))
            return;

        m_graphInstance.Label = changedCharacters[m_graphInstance.Index].Label;
        m_graphInstance.Attributes = changedCharacters[m_graphInstance.Index].Attributes;
        OutgoingRelationships = changedCharacters[m_graphInstance.Index].OutgoingRelationships;
    }

    35 references
    public override int GetIndexOfGraphInstance()
    {
        return m_graphInstance.Index;
    }

    3 references
    public bool HasQuestEvent()
    {
        if (m_currentQuestEvent == null)
            return false;

        return true;
    }

    1 reference
    public bool HasActiveQuestEvent()
    {
        if (m_currentQuestEvent == null ||
            !m_currentQuestEvent.IsActive)
            return false;

        return true;
    }

    1 reference
    public QuestEventDescription GetQuestEventDescription()
    {
        return m_currentQuestEvent.GetQuestEventDescription();
    }

    3 references
    public void AddQuestEvent(QuestEvent qe)
    {
        m_currentQuestEvent = qe;
    }

    1 reference
    public void RemoveCurrentQuestEvent()
    {
        m_currentQuestEvent = null;
    }

    1 reference
    public void TriggerQuestEvent(WorldEntity invoker)
    {
        if (m_currentQuestEvent == null || !m_currentQuestEvent.IsActive)
            return;

        m_currentQuestEvent.TriggerEvent(invoker);
    }

    5 references
    public void ActivateQuestMark()
    {
        m_questMarkPlaceholder.SetActive(true);
    }

    5 references
    public void DeactivateQuestMark()
    {
        m_questMarkPlaceholder.SetActive(false);
    }

    5 references
    public override void Interact(WorldEntity invoker)
    {
        EntityEventBroker.InteractWithCharacter(invoker, this);
    }

    1 reference
    public void KillCharacter(WorldEntity attacker)
    {
        EntityEventBroker.OnCharactersStatusUpdate -= UpdateCharacterStatus;
        EntityEventBroker.EntityDeath(attacker, this);
    }
}
```

Figure 59: InteractableCharacter class implementation.

class specific functions. Firstly, it overrides the Start method while calling the base, which will enroll the interactable to the world graph. Additionally, it overrides both the virtual and the abstract methods of the Interactable class. Finally, the next function added to the script, is the **UpdateCharacterStatus** function, which takes as input a dictionary with the Interactable index as key and a CharacterStatus instance as value. This function is called whenever a change happens on the world graph that affects InteractableCharacters. The input parameter holds the information on all the characters that need to change their status. If the current InteractableCharacter index is contained in the dictionary, then apply the changes.

Regarding the quest handling of the interactable characters, the class provides a set of methods which check whether the character has a quest event assigned to them, if that quest event is active, the ability to add or remove a quest event and to trigger it. These functions are mainly used by the Quest Scheduler, which will be described in detail in later sections, to control how quest events are distributed and triggered. Lastly, it also provides methods for activating or deactivating quest marks, which are given as an input from the script with the name of **Quest Mark Placeholder**.

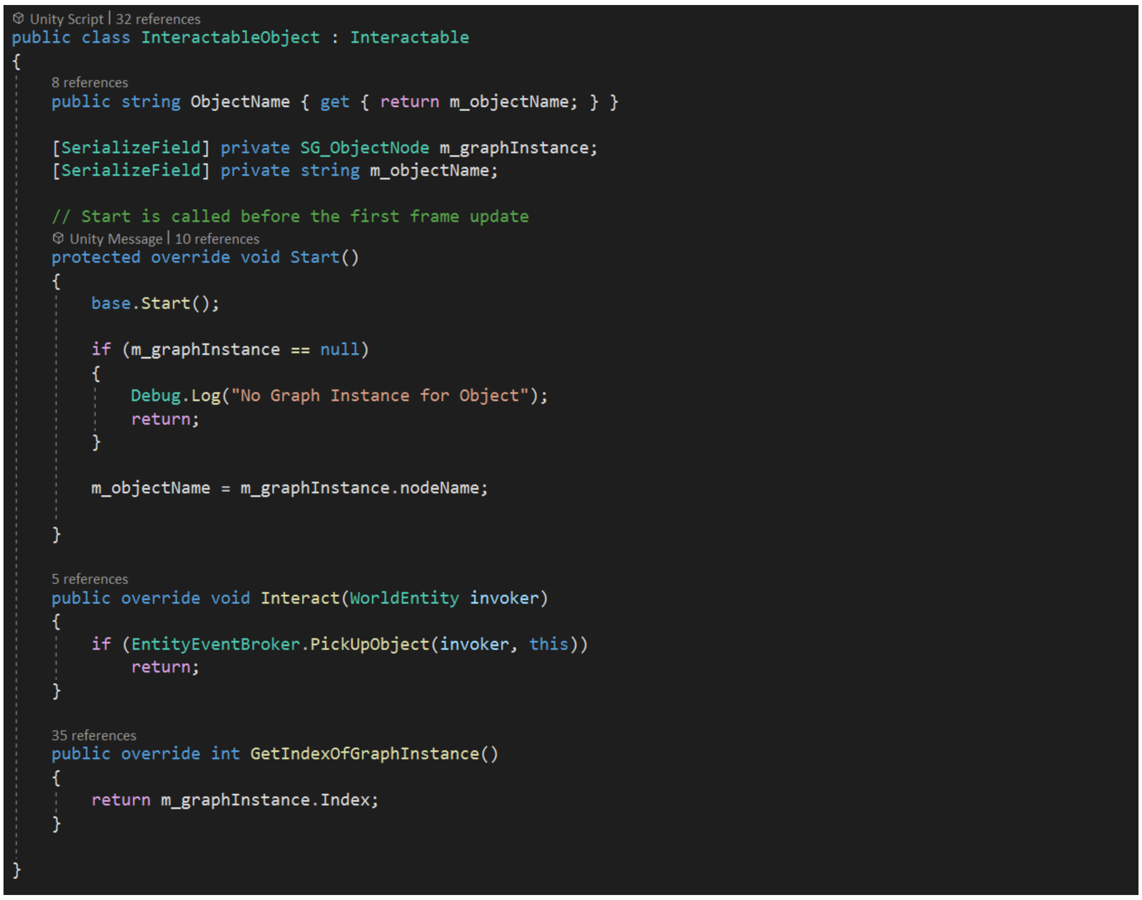
Having described the InteractableCharacter class and the way it functions, we can also describe the rest of the Interactable classes. As they are highly similar, we can describe one and extrapolate the rest. The class that will be analyzed is the **InteractableObject** the implementation of which is shown in Figure 60. What this class basically does, is to override the Start function and call the base function at the start to enroll the entity to the world graph, and to implement the Interact and GetIndexOfGraphInstance functions. The difference between those three classes is the name variable which is named after the respective interactable i.e ResourceName for InteractableResource, and the way the Interact function is implemented.

## 5.4 Quests and QuestEvents

Seeing how the Interactables work internally and before analyzing the Quest Scheduler, we need to get provide more details on how Quests and Quest Events work. This section is dedicated to analyzing these two components. It is important to note that, those two are not shown in the structure model in Figure 60. This is because they are not considered as an independent component, rather a set of information that is stored in the Quest Manager and the respective Interactable.

On a more abstract level, a Quest instance has a stack of Quest Events that are tied to Interactables. The Quest class can add new Quest Events depending on how we want the Quest to progress. It can also trigger Quest Events, start a new Quest, complete it and end it by setting all of its Quest Events as inactive. The implementation of the Quest class is shown in Figure 61.

Regarding the Quest Events, the way they are structured is by extending an abstract base class and implementing the the desired functionality of the respective



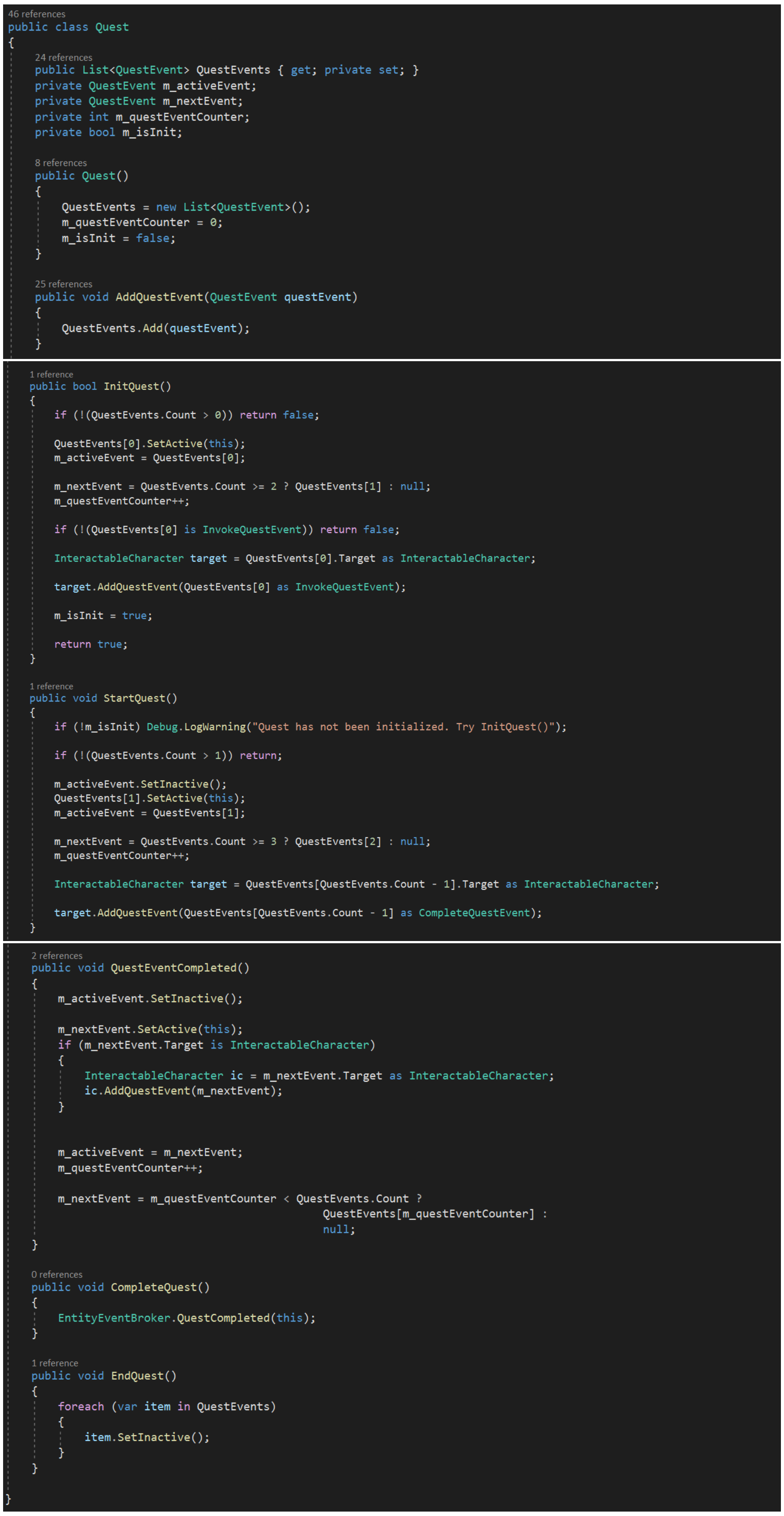
```
Unity Script | 32 references
public class InteractableObject : Interactable
{
    8 references
    public string ObjectName { get { return m_objectName; } }

    [SerializeField] private SG_ObjectNode m_graphInstance;
    [SerializeField] private string m_objectName;

    // Start is called before the first frame update
    Unity Message | 10 references
    protected override void Start()
    {
        base.Start();

        if (m_graphInstance == null)
        {
            Debug.Log("No Graph Instance for Object");
            return;
        }

        m_objectName = m_graphInstance.nodeName;

    }

    5 references
    public override void Interact(WorldEntity invoker)
    {
        if (EntityEventBroker.PickUpObject(invoker, this))
            return;
    }

    35 references
    public override int GetIndexOfGraphInstance()
    {
        return m_graphInstance.Index;
    }

}
```

Figure 60: InteractableObject class implementation.

Quest Event, while also overriding the abstract base functions. The base class, provides an API which holds information on the parent Quest, the target of the event, a brief description - which was added for the demo purposes - and a set of public abstract functions to trigger the event and activate or deactivate it. Those functions are necessarily set as abstract, as to force the developer extending the base class to implement them.

Having touched on the base class of the Quest Event, we can take a look at a concrete class implementation. We can take as an example the **KillEnemiesQuestEvent** class, which is a Quest Event for killing Interactable Enemies. This QE would be used in a quest where we want the player to kill random spawnable world entities, or what's called in the gaming world "do some farming". The implementation of this class is shown in Figure 63.

Concluding this section, we have seen the Quest class and its implementation as well as the Quest Event and its implementation with an example of one of its concrete classes. A Quest is comprised of multiple Quest events, which depending on what the developer wants to accomplish, they dictate the flow of that quest.

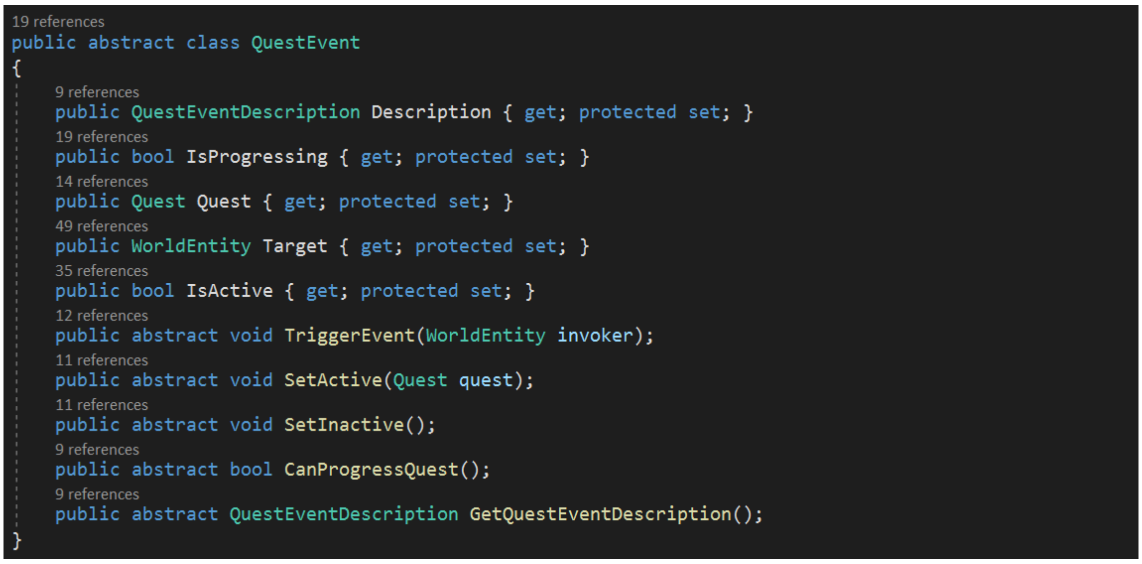


```
46 references
public class Quest
{
    24 references
    public List<QuestEvent> QuestEvents { get; private set; }
    private QuestEvent m_activeEvent;
    private QuestEvent m_nextEvent;
    private int m_questEventCounter;
    private bool m_isInit;

    8 references
    public Quest()
    {
        QuestEvents = new List<QuestEvent>();
        m_questEventCounter = 0;
        m_isInit = false;
    }

    25 references
    public void AddQuestEvent(QuestEvent questEvent)
    {
        QuestEvents.Add(questEvent);
    }
```

```
    1 reference
    public bool InitQuest()
    {
        if (!(QuestEvents.Count > 0)) return false;

        QuestEvents[0].SetActive(this);
        m_activeEvent = QuestEvents[0];

        m_nextEvent = QuestEvents.Count >= 2 ? QuestEvents[1] : null;
        m_questEventCounter++;

        if (!(QuestEvents[0] is InvokeQuestEvent)) return false;

        InteractableCharacter target = QuestEvents[0].Target as InteractableCharacter;

        target.AddQuestEvent(QuestEvents[0] as InvokeQuestEvent);

        m_isInit = true;

        return true;
    }

    1 reference
    public void StartQuest()
    {
        if (!m_isInit) Debug.LogWarning("Quest has not been initialized. Try InitQuest()");

        if (!(QuestEvents.Count > 1)) return;

        m_activeEvent.SetInactive();
        QuestEvents[1].SetActive(this);
        m_activeEvent = QuestEvents[1];

        m_nextEvent = QuestEvents.Count >= 3 ? QuestEvents[2] : null;
        m_questEventCounter++;

        InteractableCharacter target = QuestEvents[QuestEvents.Count - 1].Target as InteractableCharacter;

        target.AddQuestEvent(QuestEvents[QuestEvents.Count - 1] as CompleteQuestEvent);
    }
```

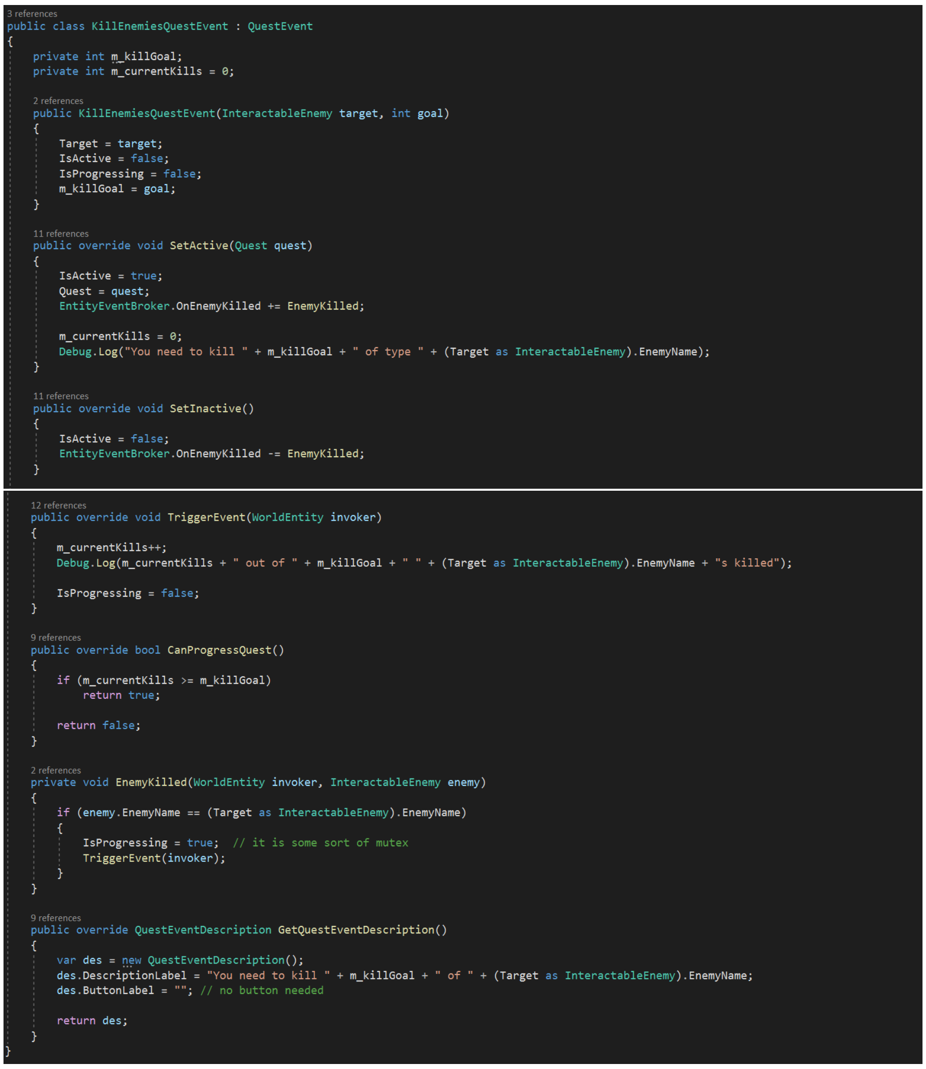
```
    2 references
    public void QuestEventCompleted()
    {
        m_activeEvent.SetInactive();

        m_nextEvent.SetActive(this);
        if (m_nextEvent.Target is InteractableCharacter)
        {
            InteractableCharacter ic = m_nextEvent.Target as InteractableCharacter;
            ic.AddQuestEvent(m_nextEvent);
        }


        m_activeEvent = m_nextEvent;
        m_questEventCounter++;

        m_nextEvent = m_questEventCounter < QuestEvents.Count ?
                                            QuestEvents[m_questEventCounter] :
                                            null;
    }

    0 references
    public void CompleteQuest()
    {
        EntityEventBroker.QuestCompleted(this);
    }

    1 reference
    public void EndQuest()
    {
        foreach (var item in QuestEvents)
        {
            item.SetInactive();
        }
    }
}
```

Figure 61: Quest class implementation.



```
19 references
public abstract class QuestEvent
{
    9 references
    public QuestEventDescription Description { get; protected set; }
    19 references
    public bool IsProgressing { get; protected set; }
    14 references
    public Quest Quest { get; protected set; }
    49 references
    public WorldEntity Target { get; protected set; }
    35 references
    public bool IsActive { get; protected set; }
    12 references
    public abstract void TriggerEvent(WorldEntity invoker);
    11 references
    public abstract void SetActive(Quest quest);
    11 references
    public abstract void SetInactive();
    9 references
    public abstract bool CanProgressQuest();
    9 references
    public abstract QuestEventDescription GetQuestEventDescription();
}
```

Figure 62: Quest Event base class implementation.



```
3 references
public class KillEnemiesQuestEvent : QuestEvent
{
    private int m_killGoal;
    private int m_currentKills = 0;

    2 references
    public KillEnemiesQuestEvent(InteractableEnemy target, int goal)
    {
        Target = target;
        IsActive = false;
        IsProgressing = false;
        m_killGoal = goal;
    }

    11 references
    public override void SetActive(Quest quest)
    {
        IsActive = true;
        Quest = quest;
        EntityEventBroker.OnEnemyKilled += EnemyKilled;

        m_currentKills = 0;
        Debug.Log("You need to kill " + m_killGoal + " of type " + (Target as InteractableEnemy).EnemyName);
    }

    11 references
    public override void SetInactive()
    {
        IsActive = false;
        EntityEventBroker.OnEnemyKilled -= EnemyKilled;
    }

    12 references
    public override void TriggerEvent(WorldEntity invoker)
    {
        m_currentKills++;
        Debug.Log(m_currentKills + " out of " + m_killGoal + " " + (Target as InteractableEnemy).EnemyName + "s killed");

        IsProgressing = false;
    }

    9 references
    public override bool CanProgressQuest()
    {
        if (m_currentKills >= m_killGoal)
            return true;

        return false;
    }

    2 references
    private void EnemyKilled(WorldEntity invoker, InteractableEnemy enemy)
    {
        if (enemy.EnemyName == (Target as InteractableEnemy).EnemyName)
        {
            IsProgressing = true;  // it is some sort of mutex
            TriggerEvent(invoker);
        }
    }

    9 references
    public override QuestEventDescription GetQuestEventDescription()
    {
        var des = new QuestEventDescription();
        des.DescriptionLabel = "You need to kill " + m_killGoal + " of " + (Target as InteractableEnemy).EnemyName;
        des.ButtonLabel = ""; // no button needed

        return des;
    }
}
```

Figure 63: KillEnemiesQuestEvent class implementation.

## 5.5 Quest Scheduler

Having described in the previous section how a Quest works internally, as well as the Quest events, in this section we shall explore how a quest is scheduled. Scheduling a quest requires multiple actions to be takes. First, the system must decide whether it has to generate a new quest or not, depending on the events in the game world and an internal countdown. Additionally to that, it also has to keep a track of all pending and active quest that are currently in the game world. The overall structure of the Quest Scheduler component, and that of the whole system, is shown in Figure 64.

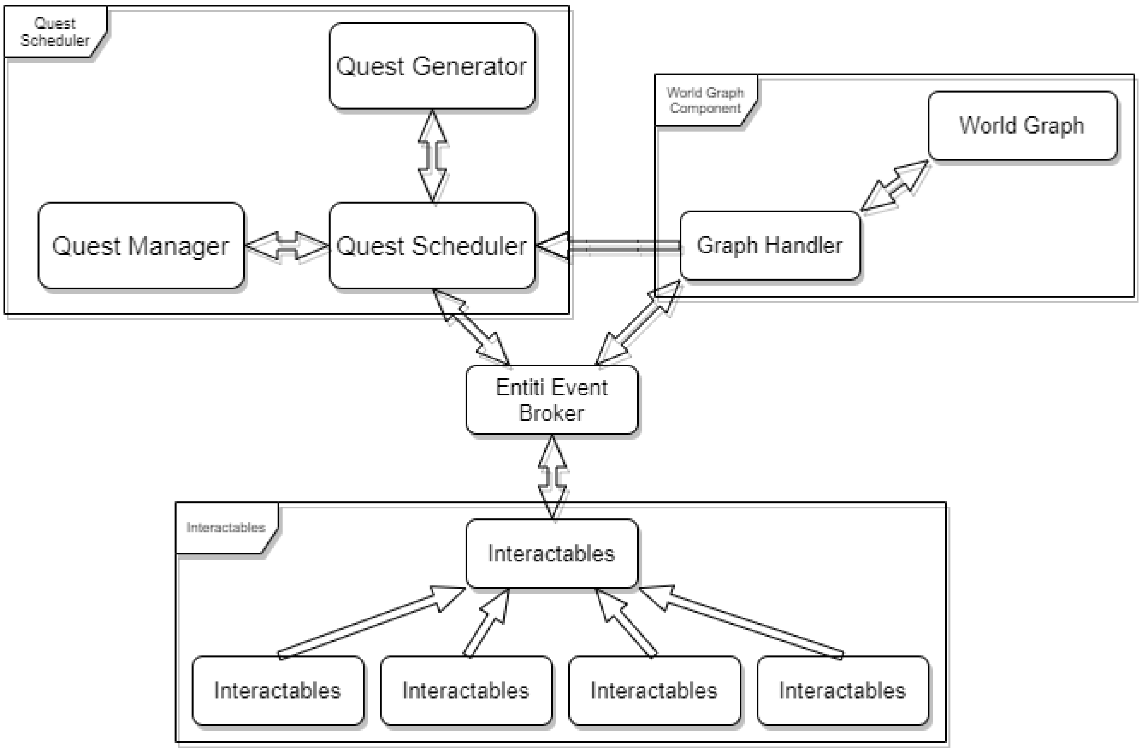


Figure 64: Quest Scheduler component structure. Following the Separation of Concerns principle, to differentiate between generating and managing the quests, two separate classes have been implemented with the respective names.

Since the Quest Scheduler extends the MonoBehaviour class, it should be added as a component to a Game Object, preferably on a persistent scene to avoid losing all the progress. When added to a component, the user is presented with a set of public variables to modify to their needs, their use being described in previous section.

The Quest Scheduler's purpose is to act as a mediator between incoming events from the Interactables or World Graph, and the Quest Generator and Manager. Whenever an event is activated by an Interactable, the Quest Scheduler checks to see if the Quest Manager has an active quest on that and if it does, it triggers a new coroutine to progress that quest.

The second part of the scheduler's purpose is to trigger the Quest Generator to generate a new quest. That happens in two ways. The first way is by the variables set by the user in the editor. On every Update, it keeps a counter from the time passed since the last quest generation, and if it reaches a specified time it will call the Quest Generator on a coroutine, as to not disrupt the rest of the gameplay. That specified time is measured in seconds, and it is a randomly generated number, called **goal time** between **MinQuestGenerationThreshold** and **MaxQuestGenerationThreshold**, which are public variables of the editor.

The second way to trigger a quest generation is on another quest's completion. When a quest is completed, it sets the counter to the goal time, which means that on the next update, it will try and generate a quest.

### 5.5.1 Quest Manager

The Quest Manager, as the name implies, is the class that manages all the quests of the system. This class is responsible for managing all active and pending quests in the game world. It has internal lists that contain all of the active and pending quests as well as a list with a reference to all of the quest events.

The available functions of this class offer the ability to add a new pending quest, to activate a quest, to progress it and to eventually complete it. It is also responsible for checking whether an Interactable that has just triggered an event, is part of an active quest event, which will subsequently trigger a progression of the parent quest.

To clarify though on the progression of a quest, as mention in the section on Quests, the actual progression is implemented in the Quest class. But, since the Quest Manager is aware of when an Interactable has triggered an event, and not the Quest itself, the manager can check if the Interactable is part of a quest, and then call the Quest's public function responsible for progressing to the next quest event.

### 5.5.2 Quest Generator

Next, we will discuss on the last and most important part of the Quest Scheduler, the Quest Generator. The Quest Generator class is responsible for the whole process of generating a new Quest. After receiving a new request from the Quest Scheduler, the generator will go through a process to try and decide which rule to choose from, try and find a valid sequence of interactables that match said rule's pattern and lastly, decide whether the player can get that quest or not, depending on their relationship with the entities of the game world.

Upon receiving a new request from the scheduler, the generator will first decide whether it can create a new quest depending on the quest pool size and the Quest Generation Multiplier from the scheduler, which introduces a random component in the first step of the process.

The second step of the process is to decide on which rule to try and use. This is an iterative process that goes through every rule and derives a fitting value from a cost function and whichever has the greatest value, is selected as the best rule. The cost function is shown below:

$$v = rulem(costm\frac{1}{1+cost} + countm\frac{1}{1+count})rand \tag{1}$$

where

| | |
|---|---|
| $v$ | = value of cost function |
| $rulem$ | = rule multiplier |
| $costm$ | = rule cost multiplier |
| $cost$ | = cost of a rule |
| $countm$ | = rule count multiplier |
| $count$ | = rule occurrence count |
| $rand$ | = random number at range [0,1) |

The cost function described above is a simple one for the purposes of this project. The output $v$ is compared through all the rules and whichever one is the highest is considered as the *best rule*. The *rulem* is a multiplier that each rule has hardcoded in it. This is decided by the developer when deciding on which rules to give more chance of appearing.

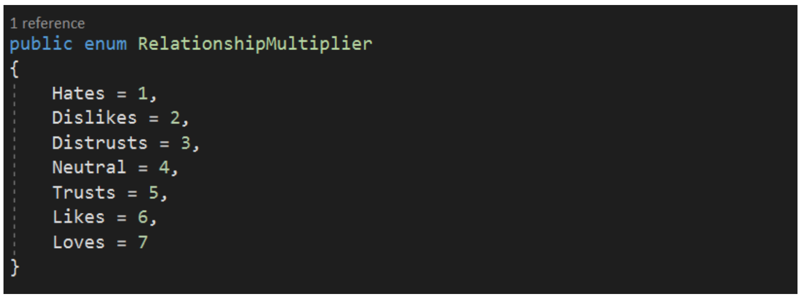
The *costm* and *countm* are multipliers that are set on the scheduler from the unity editor, to make them accessible to the designer, to bias over the cost or the number of occurrence of each rule respectively.

The *cost* parameter is the calculated cost of the implementation of the respective rule. In other words, the system sums up the number of altered relationships if the rule were to be implemented. The higher the cost the lower the chance for this rule to be selected as to avoid big changes in the game world. The *count* parameter represents the amount of times the specific rule has been used so far. Like cost, the higher the occurrence the lower the chance for the rule to be selected, to avoid repetition.

Lastly, the *rand* parameter is added as a variance to the rule output. The randomized aspect is there to reduce repetition even further, since under the right circumstances one rule could be selected multiple times due to high value. By multiplying it with a random number, it gives a chance to other rules as well to be selected. Having both random elements and more concrete values, like cost and count, in the function, addresses the exploitation/exploration dilemma which promotes variety in the output of quests.

The third step of the process, after selecting the best rule, is to search for that rule's pattern in the world graph. If it finds potential paths, it will proceed to the next step. If not, it will exit the process and try again later. The next step is to check whether the paths returned are not currently being used for other quests.

If there are available paths, the system will select an appropriate InteractableCharacter to initiate the quest, depending on the relationship between said character and the player. In the Quest Generator there is a list with acceptable relationship statuses, each with its own multiplier. The system will try and select an appropriate character, starting by the highest available multiplier and going down if one is not selected. The list of acceptable relationship statuses is shown in Figure 65.



```
1 reference
public enum RelationshipMultiplier
{
    Hates = 1,
    Dislikes = 2,
    Distrusts = 3,
    Neutral = 4,
    Trusts = 5,
    Likes = 6,
    Loves = 7
}
```

Figure 65: The list of acceptable relationship statuses. The multiplier is shown next to its respective enumeration.

When all of the aforementioned steps are completed and a rule was selected and implemented on an InteractableCharacter, the quest is generated and passed to the Quest Manager through the Quest Scheduler. And thus the generation process in complete and it will sit idle until being called by the scheduler.

## 5.6 Entity Event Broker

Having described all the individual components of the Publisher Subscriber Pattern, i.e the Interactables, the World Graph and the Quest Scheduler, we can now analyze the message broker between them which is the Entity Event Broker. As the pattern suggests, there needs to be a class that implements the events that need to exist in a system, and have them accessible to all appropriate classes.

While Unity has its own messaging system, it is commonly preferred to use the .NET event system for such structures. As it was also mentioned in a previous section on creating new events, the .NET **event** keyword was used to create delegates that would transfer information between components. Every event is also paired with its calling function which invokes the respective event. This also provides the ability to add conditions for which to decide whether to invoke said event or not. Lastly, every event was made **static**, as to make is accessible without having to create a new instance of the Entity Event Broker.

## 5.7 UI Controller

In order for the player to be able to interact with the game world, a certain UI had to be implemented. With this UI the player could pick up objects, talk to InteractableCharacters etc. For simplicity reasons, this UI is a modified version of the one provided by Unity's RPG Creator Kit.

It is also important to point out, that the UI Controller designed for this system, was done so as a proof of concept to be able to demonstrate how it works. It is not part of the system and a new one should be implemented if this system was to be used elsewhere.

For the UI Controller to be able to work with the system it had to be able to do three things, interact with the world around it, show the player's inventory and show quest markers. Showing quest markers is done through the InteractableCharacter script whenever a character has a new quest. In this demo it's shown as a question mark over the character as shown in Figure 66.

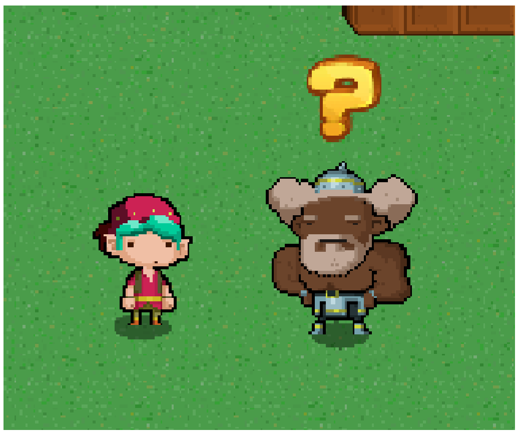


Figure 66: A question marker showing that this character has a quest available.

Showing the inventory was implemented on Unity's kit and whenever the player goes over an interactable object or resource, it would pick it up and show it on the left side of the screen.

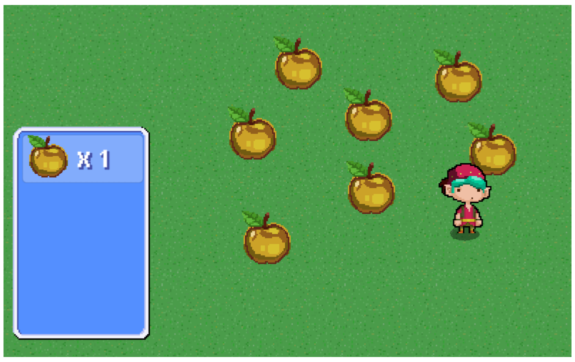


Figure 67: UI showing inventory after a resource is picked.

Lastly, interacting with the game world only requires walking up to an InteractableCharacter and a pop up will appear showing information to the player. Depending on whether the player has an action available, i.e taking on a quest, an appropriate button will appear that will trigger the invoking of that action.

## 5.8 Custom Graph Editor Tool

As it was mentioned in previous sections, in order to assist the designer in creating a game world graph, a custom tool was created in which the designer can create the nodes of the graph and mark them as Interactables and add relationships between those nodes, providing them with labels and optional reasons for those labels.

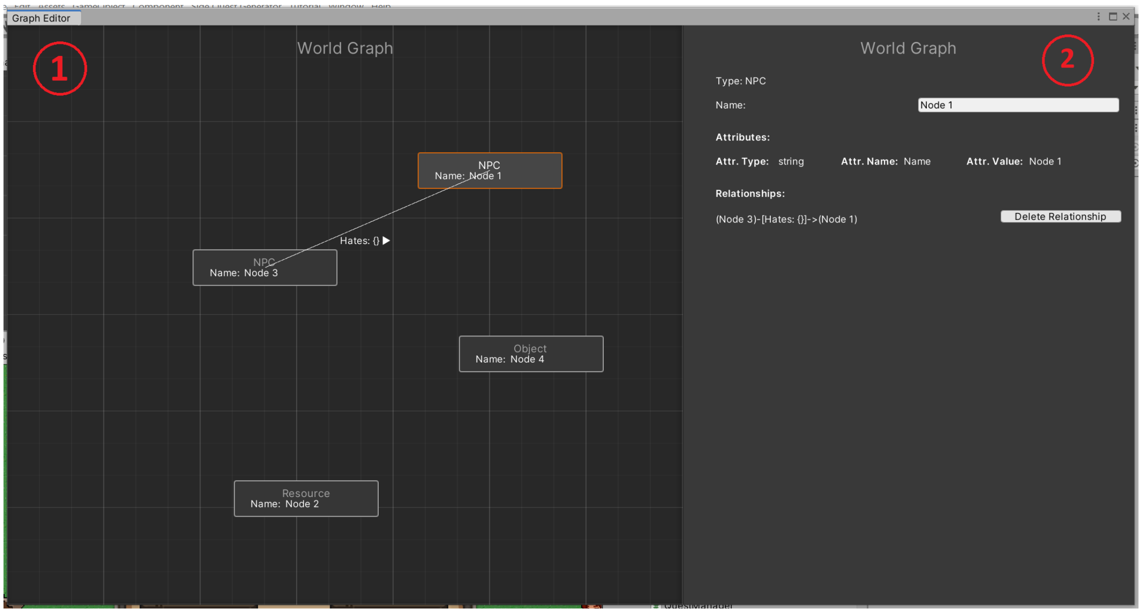
This custom editor was created using Unity's Editor library which provides the developer the ability to create custom inspectors and editor windows. When we select the Side Quest Generator ribbon and Launch Graph Editor the **SG_Graph EditorWindow** class is called by the static InitEditorWindow method, to create an instance of the window. Upon creating the window, two distinct views are created, the **Graph View** on the left and the **Attribute View** on the right. In the Graph View we can see and edit the world graph and all its relationships, while in the Attribute View we can see the attributes of each individual node and delete its relationships.

Once the window is instantiated, the user can create or load a word graph by pressing *right click* on the Graph View and selecting the respective option. Upon loading a new graph, the **OnGui** method is called on the SG_GraphEditorWindow class which updates the windows UI. It first collects all the events that occurred on that window like mouse clicks and goes through each view to update its respective UI.

First on the Graph View, it draws the rectangle of the view, and upon it draws a grid from a custom made texture. Next, it goes through every node of the graph and draws its respective rectangle. The SG_NodeBase has a virtual method which is called by the subclasses of each child node like InteractableObject etc. which holds all necessary information the node needs for the UI like its position on the grid and its texture. When all nodes are drawn, the relationships between them are drawn as lines with labels. The SG_Edge class which stores said relationships, holds the information of the start and end nodes, so it also has access to the positions of said nodes.

The Attribute View shows the attributes of a selected node. It shows the type of interactable the node is - with the NPC type representing the InteractableCharacter class - the name of the node, its list of attributes with the first one being the name and lastly a list of all the relationships that are associated with that node, along with a button to delete them. If no node is selected then nothing is shown.

One last thing to point out is that any change made to the world graph, is immediately set as dirty and since all classes are serializable, said changes are saved immediately.



Type: NPC

Name: Node 1

Attributes:

Attr. Type: string Attr. Name: Name Attr. Value: Node 1

Relationships:

(Node 3)-[Hates: {}]->(Node 1)

Figure 68: Interface of Custom Graph Editor. On 1) we see the Graph View with nodes and relationships between them. On 2) we see the Attribute View with attributes of a selected node showing in a list.

## 5.9 Demo Quests

For the purposes of this thesis, four different quest rules were implemented. Those quest rules search the world graph for their respective patterns and generate the appropriate quests when triggered. Additionally, each rule uses a different set of Interactables, so that all of them may be demonstrated in the project.

### 5.9.1 Steal Rule

In the Steal Rule, the system will try and search for a pattern where an InteractableCharacter **(IC1)** dislikes another InteractableCharacter **(IC2)** and if the latter **owns** -which is the label condition- an InteractableObject **(IO1)**, (IC1) will give a quest to the player to go and steal (IO1) and bring it to them.

Upon stealing the object and subsequently completing the quest, changes will occur on the world graph to reflect those actions. Specifically, (IC2) will dislike both the Player and (IC1), (IC1) will like the Player and (IO1) will now be owned by (IC1). Below, in Figure 69, there is a graphic representation of the search pattern, the resulting quest and the changes in the game world upon completion.

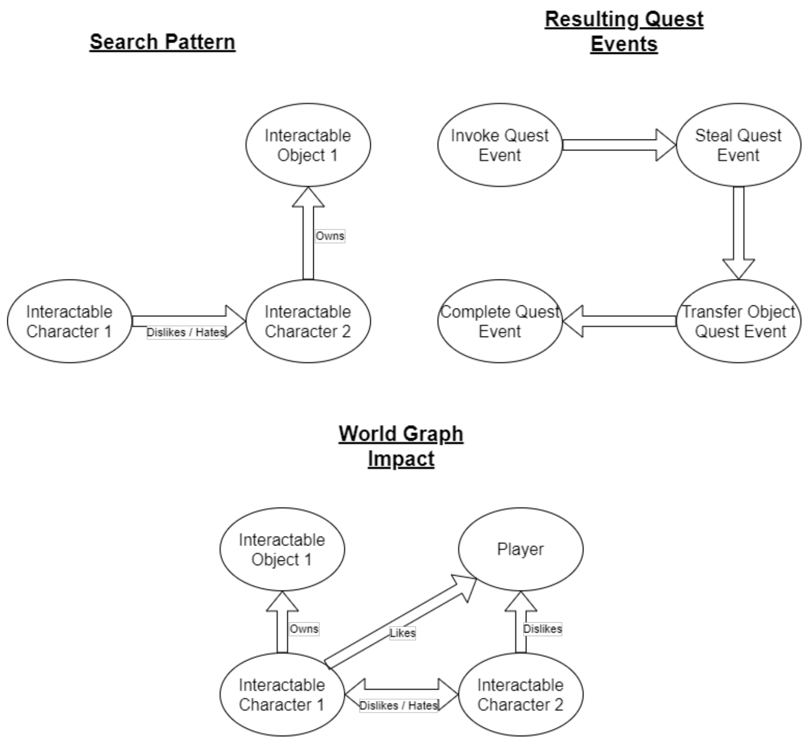


Figure 69: The Steal Rule overview.

### 5.9.2 Love Murder Rule

In the Love Murder Rule, the system will search for a pattern where an InteractableCharacter **(IC1)** has a Love connection with an Affair reason - as to avoid any regular love relationships - with another InteractableCharacter **(IC2)** and also has a pair of Married and Hates relationship with another InteractableCharacter **(IC3)**. If it acquires one such pattern, (IC1) will give a quest to the Player to go and kill (IC3).

This quest is a potentially costly one, because upon murdering (IC3), everyone that had a Like relationship with that character will acquire a hate relationship with the Player and everyone that disliked or hated that character will gain a like relationship with the Player. In addition, as to avoid having (IC3) involved in any future quests, the character will lose all of its relationships when killed. In Figure 70, there is a graphic representation on the quest's search pattern, resulting quest and impact.

### 5.9.3 Resource Gather Rule

The Resource Gather Rule is a simple one so that the Player not only has very impactful quests, but also simpler ones to explore the world graph. In this rule, the system searches for InteractableResources **(IR1)**, chooses one at random and produces a quest for a random InteractableCharacter **(IC1)** where the player has to gather a number of the selected resources. Upon completion of the quest, the

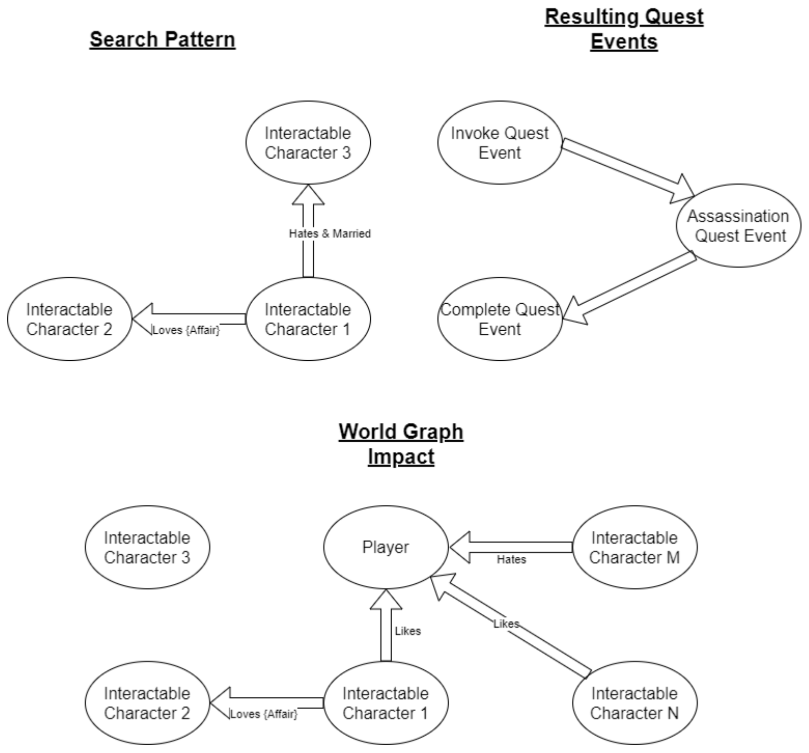


Figure 70: The Love Murder Rule overview.

(IC1) will like the Player more. In Figure 71, there is a graphic representation on the quest's search pattern, resulting quest and impact.

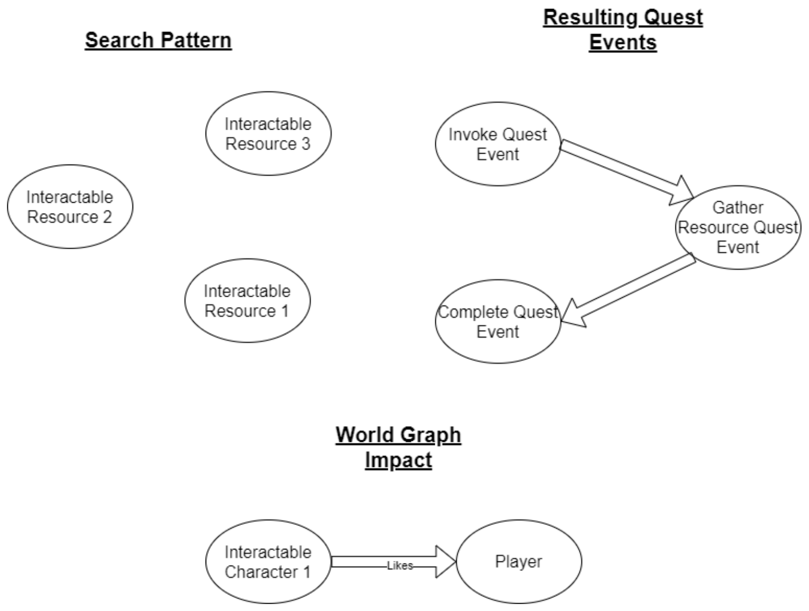


Figure 71: The Resource Gather Rule overview.

### 5.9.4 Enemy Kill Rule

Like the Resource Gather Rule, the Enemy Kill Rule finds a random InteractableEnemy **(IE1)** and produces a quest for a random InteractableCharacter **(IC1)** where the player has to kill a number of the selected enemies. Upon completion of the quest, the (IC1) will like the Player more. In Figure 71, there is a graphic representation on the quest's search pattern, resulting quest and impact.

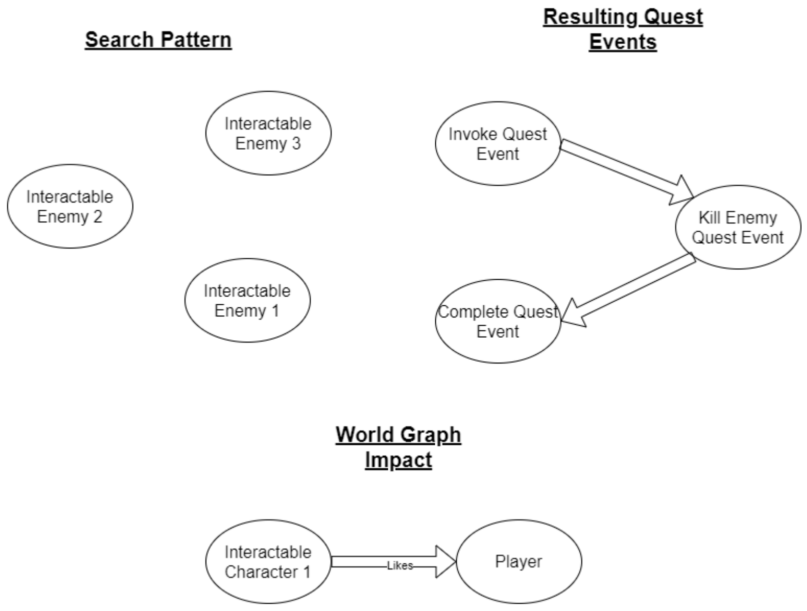


Figure 72: The Resource Gather Rule overview.

## 5.10 Demonstration of world impact on quest completion

Having described how the overall system is interconnected and finally generates a new quest, a demonstration of one such quest and its world impact is necessary for a proof of concept. To better show the impact to the game world, a project-specific addition was made to the scene view, to show the names of the Interactables as well as the relationships between them. That way any changes made to the world graph, would also show on the scene view.

In the first two screenshots, the demo world is shown both in the scene view - the actual game - and in the graph view, which is the underlying world graph. The rule selected for this demonstration, was the *Love Murder Rule*. The InteractableCharacter *Lover* which has an affair relationship with the InteractableCharacter *Wife* who is married to the InteractableCharacter *Husband*. The relationships of all associated ICs and the Player are shown in Figure 74. Notice that the relationships between the ICs and the Player are initially set to Neutral.

The quest is introduced to the player by the Lover and the player needs to approach him to invoke said quest. A pop up will appear prompting the player to start the quest and then the player needs to go the Husband and assassinate him to progress the quest. As shown in Figure 75, the Husband now has a marker to show the player that this IC has a quest associated with it.

After killing the Husband the player can go back to the Lover - which has now gained a quest marker - and finish the quest. Approaching the Lover a pop up will appear prompting the player to get a reward (which for simplicity reasons there isn't one) and finish the quest.

At this stage it is important to point out that the relationships between the player, the Lover and the Wife have now been changed to Like and Love respectively with a *Killed Husband* reason as shown in Figure 76. It is also important to point out that this change was triggered because of the death of the Husband and not because the completion of the quest since the quest management system is not tied to the game world.

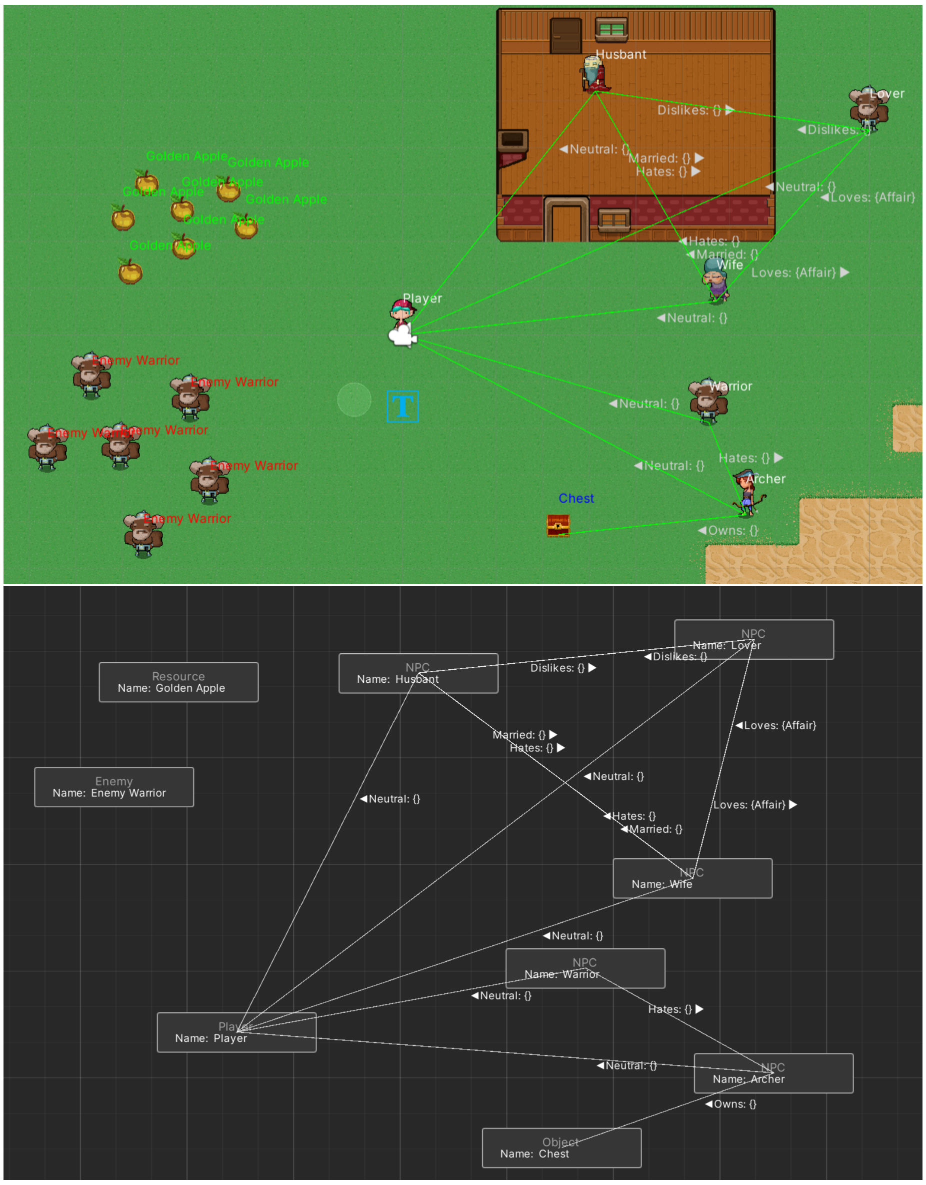


Figure 73: The top screenshot shows the game world from the scene view of Unity's interface. The bottom screenshots shows the underlying game world that the system is using.

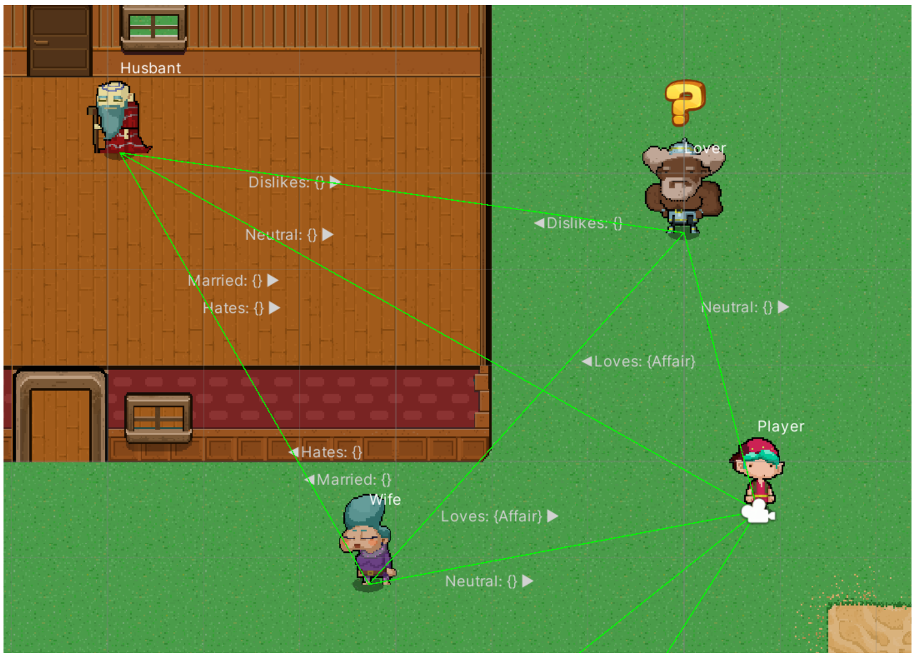


Figure 74: All relationships between InteractableCharacters and Player that triggered the Love Murder Rule.

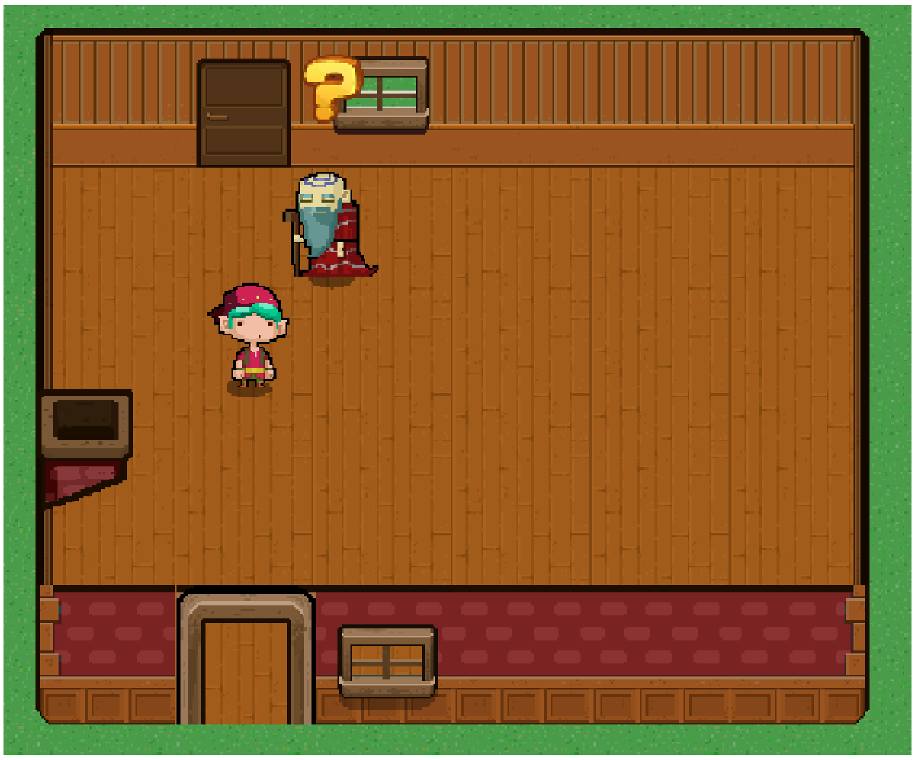


Figure 75: A marker above the Husband InteractableCharacter to indicate that there is a quest associated with him.

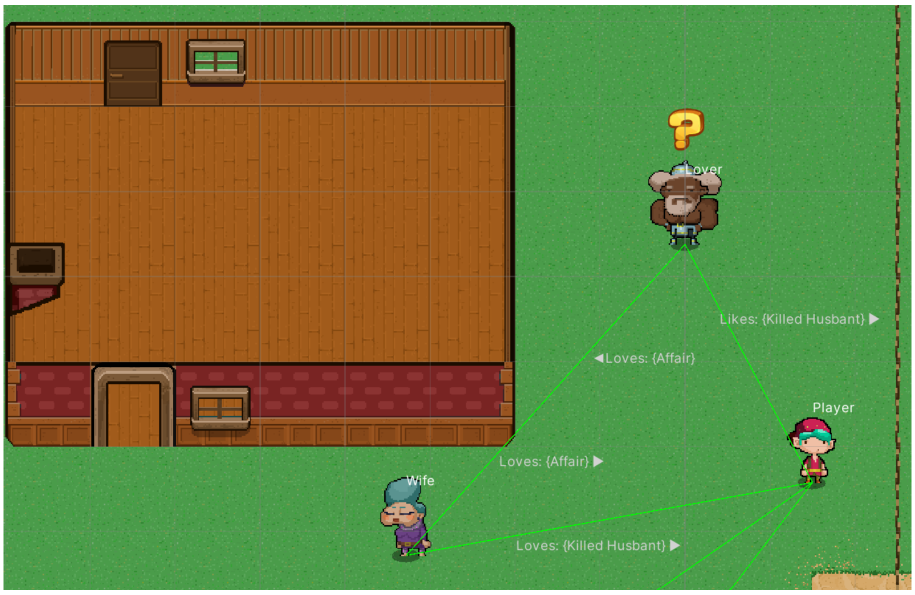


Figure 76: After killing the Husband, the relationships between the player, the Lover and the Wife have changed.

# 6 Conclusion

## 6.1 Summary

In the system described in this thesis, we tried to implement a dynamic side-quest generation tool, in Unity 3D which would take into account the current state of the game world and provide the player with more gameplay content. While regular systems use strings to generate a sequence of events in a quest, those strings do not provide any other information other that the event itself which adds a restriction to what could be generated, therefore does not provide with much variety.

The goal of the described system was to not use strings, but graphs. Using graphs meant that each quest event could potentially have all necessary information to add as much complexity as the developer needs. While the idea of using graphs to generate quests is not new, this system was a practical application on a widely used game engine, showcasing how such a system could be implemented in a game.

## 6.2 Limitations

While this system was briefly tested mostly to see if it works as expected, it must be pointed out that it was not tested by designers or other developers to test it's ease of use and expected behaviour.

More specifically, from a designer's point of view, the both the system and the tool should be intuitive enough to use. Additionally, the parameters set to the system regarding best rule selection, should produce the desired variety and number of side quests. This was not sufficiently tested during the implementation of the system.

From a developer's point of view, the system should be expandable enough to be able to easily add new rules for quest generation. Also, additional world impact effects should easily be added to the graph handler with the use of the Graph class API. This was also not tested enough, as no other developers used this system during its implementation.

## 6.3 Future Work

As stated above, this implementation was a simple practical showcase of a graph-based side-quest generation system, which demonstrated how such a system would work in its basic form. Thus, more improvements could be added to make it more versatile and increase the complexity enough, to be able to generate a variety of new side-quests. Such potential improvements are mentioned below.

### 6.3.1 Creating more Side-Quests

For this instance of the system, four rules were implemented that could generate quests. To increase the complexity of the system, a first step would be to add more rules that use the provided API. One would have to adjust the necessary multipliers, so that they are not preferred much more over the others, but that is a trial and error process, which would lead to a better system.

### 6.3.2 Creating Custom Quest Creation Tool

One other big improvement would be to create a custom quest creation tool, like the custom graph editor that was implemented in this project. Since quests are comprised of a sequence of quest events, it makes them modular enough for such a tool to be viable.

Quests are considered as directed graph structures that have quest events as nodes. The search patterns of each rule is represented as another graph which specifies what sort of pattern to search for. The final impact of the rule on the world graph, could also be represented as a form of graph that instructs the graph to make certain changes. This means that a similar tool like the graph editor can also be created to generate new rules, that the user could design said graphs and have the system translate them appropriately.

### 6.3.3 Implementing a better cost function for Quest selection

One final but more significant improvement in the system, would be to add a better cost function which would decide which rule to use, whenever the scheduler calls the generator for a new quest. The current cost function, is a rather simple one which only takes into account the occurrence frequency or each rule and the amount of changes that would occur if said rule was to be generated. Multipliers have also been added to introduce another level of variety to the generation process.

A better cost function would also try and restrict the player from making significant changes to the game world by his actions. Also, finally, a heat map could also be introduced to which would show where the player has been, and have the system generate quests in places the player has not visited, to promote better exploration of the map. Introducing these changes to the system, it would significantly increase the repeatability of the game.

An improved cost function could be inspired by the Upper Confidence Bound for Trees (UCT) algorithm [2] which is in a way similar to the one used, but adds a significant amount of complexity, and thus variety, in the rule selection method. Following the UCT algorithm, a tree could be generated with the Player as a root node and the branches being all the possible quest paths that could be generated at each time. Every node represents an action tied to a different Interactable, and the cost of each action could be calculated by simulating it in the game world. This increases the exploration element of the algorithm, since it takes into account all possible nodes, not just the whole paths, which guarantees that every Interactable will be selected at some point.

### 6.3.4 Testing expected diversity of generated side quests

As mentioned before, the system was not sufficiently tested to explore the diversity of generated quests by having different parameters and also taking into account the world impact. To evaluate the quality of the output quests, simulations of the generation system with different parameters on the same game world could be run, to see the distribution of output quests as well as the frequency.

### 6.3.5 Evaluating the usability of the Graph Editor tool

As it was pointed out in the Limitations section, the Graph Editor tool was not used by other designers to test its usability. One future improvement on the overall system, would be to have designers use the tool and make suggestions on how to provide a better user experience and then implement those changes for a better tool.

# References

[1] Selmer Bringsjord and David Ferrucci. *Artificial intelligence and literary creativity: Inside the mind of brutus, a storytelling machine.* Psychology Press, 1999.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

[3] V. Bui, H. Abbbass, and A. Bender. Evolving stories: Grammar evolution for automatic plot generation. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[4] Benjamin N Colby. A partial grammar of eskimo folktales1. *American Anthropologist*, 75(3):645–662, 1973.

[5] Pablo Gervás. Propp's Morphology of the Folk Tale as a Grammar for Generation. In Mark A. Finlayson, Bernhard Fisseni, Benedikt Löwe, and Jan Christoph Meister, editors, *2013 Workshop on Computational Models of Narrative*, volume 32 of *OpenAccess Series in Informatics (OASIcs)*, pages 106–122, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-57-6. doi: 10.4230/OASIcs.CMN.2013.106. URL http://drops.dagstuhl.de/opus/volltexte/2013/4156.

[6] Dieter Grasbon and Norbert Braun. A morphological approach to interactive storytelling. In *Proc. CAST01, Living in Mixed Realities. Special issue of Netzspannung.org/journal, the Magazine for Media Production and Inter-media Research*, pages 337–340. Citeseer, 2001.

[7] K. Hartsook, A. Zook, S. Das, and M. O. Riedl. Toward supporting stories with procedurally generated game worlds. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 297–304, 2011.

[8] Ben Kybartas and Rafael Bidarra. A survey on story generation techniques for authoring computational narratives. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(3):239–253, 2017.

[9] Ben Kybartas and Clark Verbrugge. Analysis of regen as a graph-rewriting system for quest generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):228–242, 2014.

[10] Boyang Li and Mark O Riedl. A phone that cures your flu: Generating imaginary gadgets in fictions with planning and analogies. In *Intelligent narrative technologies*, 2011.

[11] Lyn Pemberton. A modular approach to story generation. In *Proceedings of the fourth conference on European chapter of the Association for Computational Linguistics*, pages 217–224. Association for Computational Linguistics, 1989.

[12] Julie Porteous, Marc Cavazza, and Fred Charles. Narrative generation through characters' point of view. In *AAMAS*, pages 1297–1304. Citeseer, 2010.

[13] Vladimir Propp. Morphology of the folktale, trans. *Louis Wagner, 2d. ed.(1928)*, 1968.

[14] David E Rumelhart. Notes on a schema for stories. In *Representation and understanding*, pages 211–236. Elsevier, 1975.

[15] Ivo Swartjes, Edze Kruizinga, and Mariët Theune. Let's pretend i had a sword. In *Joint International Conference on Interactive Digital Storytelling*, pages 264–267. Springer, 2008.

[16] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[17] Josep Valls-Vargas, Jichen Zhu, and Santiago Ontañón. From computational narrative analysis to generation: a preliminary review. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, page 55. ACM, 2017.

[18] Pengcheng Wang, Jonathan P Rowe, Wookhee Min, Bradford W Mott, and James C Lester. Interactive narrative personalization with deep reinforcement learning. In *IJCAI*, pages 3852–3858, 2017.

[19] Albin Zehe, Martin Becker, Lena Hettinger, Andreas Hotho, Isabella Reger, and Fotis Jannidis. Prediction of happy endings in german novels based on sentiment information. In *3rd Workshop on Interactions between Data Mining and Natural Language Processing, Riva del Garda, Italy*, page 9, 2016.