ROOTS: An Open Source tool for authoring Behaviour Trees in a visual way inside of Unity

Eduardo Simón Picón

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

July 1, 2019

Supervised by: Luis Amable García Fernández

To Andrea.

For being there every time I was working during a Friday movie night.

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Luis Amable for his guidance in the most underestimated parts of the project. Also, I would like to thank him for some of the talks that we had in his office about what it meant to be a programmer and how to deal with the future.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report. Thank you for discovering me LaTex.

I would like to thank to my class mates: Ricardo, Oscar, Marc, Dani, Carlos, Sergio,Vicent, Kiko and Oliver. Without every good and bad experience that I have lived with them I wouldn't be the same nor learnt what I have learnt.

I would like to thank my parents for giving me the opportunity of studying whatever I wanted and at my own pace. Thank you for believing in me, and for teaching me that if you want to succeed in life, there's no other way than the hard-working one. Without your support and guidance I wouldn't be here.

I would also like to thank my beloved aunties Mamén and Lolita. Without their tremendous support throughout the years I wouldn't have been able to face the problems that life has presented me. They are a source of real kindness and support.

I would also like to thank my roots, specially the strongest one, Isabel. The only permanent constant across my life of variables. Thank you for being so supportive and grateful. Thank you for remembering me every day even though we are a little further than we were before.

Thank you to my beloved soulmate, for going choosing me to traverse the life's path next to you. Without your CONSTANT help and vitality I would have failed a long ago. You give me the peace and love that I need everyday, and I'll be grateful to that until the day I die.

Abstract

The current document tries to document the whole process of developing an editor tool for the Unity Game Engine, in particular, a node-based GUI for authoring Behaviour Trees which can be used for driving the decision making process of agents. The paper's main objective is to expose the inner working systems for people who are interested in extending and improving the different features of the tool.

Unreal Engine has a great native tool for authoring behavior trees but Unity doesn't, nevertheless, there are premium and feature-rich tools available in the asset store. However, none of them are open-source projects resulting in a difficult customization of the tool for the different project needs. Unity and Unreal are the most well-established and biggest game engines in the market, and as such, developing a non-existing solution for this problem could be a very successful movement. ROOTS, will be an open source project, and it's code will be publicly hosted at Github. Despite the nature of an open-source project, which implies that you don't get revenue from licensing the product, there are always other ways of profiting the development such as developing a proprietary extension for a client or via donations. Throughout the report the different achieved objectives and the future planed features will be discussed and reviewed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the following chapter the project's objectives and its driving force will be discussed and overviewed.

1.1 Work motivation

Artificial Intelligence is a crucial part of many games, from AAA games to indies. The most basic form of decision making algorithm used in video games is a Finite State Machine. By nature, FSMs are very simple to get up and running, however, they don't scale as nearly as good. In fact, some even claim that the era of FSMs in video games is over[22]. The games industry has adopted another system because of FSMs's poor scalability: Behavior Trees. In Unity, there is not a built-in way of authoring BTs, therefore, many asset creators have created paid plugins for creating these structures. However, none of them is open-source, and that's why I decided to start this project.

As a game developer focused in unity that doesn't have economic resources to afford high-end plugins for authoring behavior trees visually like Behavior Designer [2] or NodeCanvas [10], I feel frustrated. I did some research and found a plugin called BehaviorBricks [1], which is what I was looking for, except it doesn't work in the latest versions of unity and it hasn't received updates in a while, plus the editor code is compiled hence not open source. There are other free BT frameworks, but the authoring of the tree is not visual nor intuitive, like PandaBT Free [11].

In the end, I have decided to make my own editor for authoring BTs visually. A custom visual tool for authoring AI is a project that I have always wanted to start. I thought about a visual FSM editor and selling it for a few dollars, but the scope was to small to do it in 300 hours and I strongly believe that FSM are not the best solution for dealing with AI in real videogame productions. Then the BT idea came to my mind after my last project developed for the Artificial Intelligence subject. If only I had a tool like that for the project, everything would have been easier. When I finally decided to embark in a huge project like this one, I thought about monetizing it, but thanks to my tutor's advice I decided the right choice and make it open source.

1.2 Introduction to Behavior Trees

Behavior Trees are an artificial intelligence technique used for decision making, they are widely used across many triple-A games and indie games. They were introduced by Bungie Studios, creators of Halo 2 and have been the industry standard since. They are represented with a tree form composed of small tasks. Some of these tasks, can have children hence creating branches and making its shape more resemblant to a tree. Their nature is similar to a Hierarchical Finite State Machine, but the decision and the actions are decoupled. They are considered a reactive planning technique because they output a certain action given a world state without knowing the full state-space nor calculating a plan through it.

Behavior trees allow utilization in other projects, not like their counterparts. There are other approaches such as planners. These kinds of techniques also decouple the action logic from the decision logic, but they are less predictable. This takes away design power from designers, as they cannot simulate accurately their vision of intelligence.

One of the core features that behavior trees posses is the ability to run actions asynchronously. This feature is achieved by a simple yet a powerful concept: a task completion state.
The algorithm for traversing and determining its status is recursive and the lowest parts of the tree affect the highest parts. The evaluation traverses the tree from bottom left to top right and outsputs a state for each task. Depending on the type of task, the status can affect in different ways to its parent. The three main states are:

- **Failed:** The task didn't succeeded in its objective and its parent will act in consequence.
- **Succeeded:** The task achieved its goal.
- **Running:** The task is running and did not finish yet. This state is crucial to BTs due to the fact that most actions an agent can do in a game are performed over time, such as Steering Behaviors. This enables to resume the execution of the tree at that specific task during the next tick of the game thus, making the execution efficient because there's no need to keep checking other tasks.

A tree will finished its execution when its root task has a state returned different than running. If the status received is succeeded it is said that the tree has succeeded and the contrary when the return status is failed. Furthermore, the BT implementation can have other states to have a better control of the execution of each task, as an example, in my implementation, there are other states such as NonInitialized,Aborted or Invalid.

The tree is made up of 3 kinds of tasks:

- **Leaf Tasks.** These kind of tasks are the most common ones and there are 2 types.
  - **Conditionals** check if a specific condition is met at the current world state. They are equivalent to FMS's decisions but they are not coupled to an action.
  - **Actions** are the opposite of conditionals, instead of asking, they just perform a specified function such as Seeking, going to a world Position, Playing a sound or playing an animation.
- **Composites** are one of the biggest strengths of the BTs. They drive the flow of execution based on the status of its children. In practice, they are like visual Boolean operators such as AND and OR. The most common types of composites are:
  - **Sequences** behave like an AND operator. Returns Success when each one of its children have succeeded and it returns failed as soon as a child fails.
  - **Selectors** behave like an OR operator. Returns Success as soon as one of its children success. If a child fails it will execute the next one until one succeeds or all of them fails, hence, returning failed as the result of the Selector.
  - **Parallels** are a special type of composite. Their name implies that they run concurrently some task, but it doesn't have to be necessarily in a different thread or at the same execution time. The difference comes from the fact that parallels start all of their children at the same time and controls their status based on some parallel configuration called policy. As [12] mentions, its extremely important to correctly define the parallel policies, one for succeeding tasks and another one for failing ones. The Parallel is implemented using counters for its failing/succeeding tasks and based on the policies will return a certain state. The policies can interrupt the running children.
    For example, if you want to seek a target by calculating a movement vector and update a running animation parameter at the same time, you need a parallel. With a sequence or a selector, if the seek task is running, the setAnimationParam task won't be evaluated because the seek task hasn't failed nor succeeded. If we stablish a requiere one failing for the parallel's failing policy, as soon as the seek task fails, the parameter update will fail too and the running animation won't be updated resulting in a correct behavior. However, if we set the policy to require all for failing, when the seek task fails because it couldn't find the target, the animation task will keep updating, because it only fails if the animation is not present or the animator is deactivated. This will lead to the tree returning succeeded even tho the seek task has failed. To sum up, the correct use of the parallel task is crucial to achieve the desired results.

- **Decorators** come from the design pattern [3] and act as so. They can only have one child and they are mostly used as extra functionality for its children like waiting, inverting the result,etc. They provide special functionality to its child without the need of having to implement in the child. For example, if we want to have a log class that outputs a message to the console every 5 seconds you don't need to create a child class of Log extending its functionality with a timer. Instead, you can attach it to a wait decorator node and you've got a log which outputs every 5 seconds as you can see in the image. (See Figure 1.1)

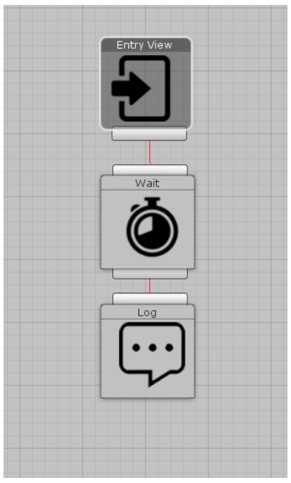


Figure 1.1: Example of a wait decorator with a log node. Instead of creating a new type of Log node that waits and log, we use a decorator to "add" functionality dynamically.

1.3 Objectives

The planned objectives are the following:

- **Implementing an Artificial Intelligence technique in Unity with the goal of driving NPCs that simulate intelligent behavior.** This technique is commonly known as Behavior Tree.
- **Implementing a library that lets us create behavior trees in a similar way as we operate with Unity's native tools.** The finality of the visual editor is to generate behavior trees with different visual elements and saving it to a file. The workflow will be very similar to the "Mecanim" animation tool, included in the engine by default. This tool, generates a level of abstraction in a visual way, making it easy to debug and create animation graphs. [4]
- **Creation of an API that allows developers to extend the library's** basic functionality in order to improve or adapt it to a certain project.

1.4 Environment and initial state

The project will be developed in its entirety by the author: Eduardo Simón Picón. The project will be available from the day after the version 1.0 is released to the public. The project will be open-source and public on Github[6] with a MIT License. The MIT License can be described as: "A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code".

Agile has been chosen as the software development methodology, being Scrum the workflow which will be used. In this Project, the author will be the Scrum Master, the team and the Product Owner, nevertheless, the roles could change in a future version of the tool. The Scrum board, the place where you keep track of the different task's status will be hosted on Trello[16], an amazing tool for creating and managing boards with cards[7]. In conjunction with Trello, there's a nice and free plugin for Google Chrome which shades the cards and makes the workflow even easier.

Chapter 2

Planning and resources evaluation

2.1 Planning

Firstly, I would like to mention that the following planning (see Figure 2.1) was an estimation of the different tasks and sub tasks needed to complete the project as well as the temporal cost of them. When developing software is very common to encounter different unexpected bugs and blockers along the project, in spite of this condition other tasks may be completed very easy or developed with less resources than it was originally expected. I would like to make clear that this is my first tool developed in the Unity Engine and its framework. While it's true that I have spent my last 4 years using the engine and making games with it, I have never attempted to create a big editor tool nor any project with this magnitude resulting in a huge challenge and source of motivation. In addition to the initial gantt diagram, I have used Trello as my main tool to follow the project progress. There were 3 planned sprints, 1 for every 100 hours of work. This was due to the nature of Scrum and the project complexity. It is mandatory to have a working prototype after every sprint. In Addition, a part-time job are 20 hours per week, so 5 weeks per sprint.

As such, I estimated that after:

- **The first 100 hours** I would have some kind of editor in which you could add nodes and search tasks.
- **The second 100 hours** the core would be implemented and a simple BT could be run.
- **The last 100 hours** would be for improving the UI, adding Blackboard support and inspector for the different task ending up with a small demo.

Table 2.1: Initial Sprint Planning

Sprint	StartDate	EndDate
1	4/02/2019	11/03/2019
2	11/03/2019	15/04/2019
3	15/04/2019	20/05/2019

To start planning the first thing I did was to split the work in different areas and established what had to be accomplished in each one of them. As such, there are 5 key parts in the project:

- **GUI Editor Programming** This part will consist on the development of the Editor Window. In this window the authoring of the BT will occur. Users will be able to manipulate nodes, create them or delete them. There will be a inspector window to inspect the different node's properties.
- **Behavior Tree internal Logic** This module of the module will control the core logic of the behavior tree. It will contain the behavior of the different tasks: Initialization, update and termination. It will be responsible of informing the run time Library if the tree has succeeded or needs to be ticked again.
- **Design and Implementation of an API for extending the tool** The nature of the tool will always encourage developers to extend it and adapt it to their specific needs, that's why a good design and versatile API is needed. An API is an Application Programming Interface, and exposes a set of classes and methods to operate with the tool.
- **Run-time Library** The run-time library is the bridge between the Core and the Unity Framework. It represents an agent's Brain Executor, being the brain the tree data. You can control the hyper-parameters of the execution via Inspector.
- **Demo Creation** A demo will be created to showcase a game where the tool is used.

- **Project Documentation** The documentation is crucial in a project like this. New developers must be able to obtain every piece of valuable information regarding the tool and its inner workings.

2.2 Resources evaluation

The costs of the work that is going to be undertaken must be estimated in advance. The human and equipment costs must be quantified so that the work can be assessed and so that, in a real case, the economic viability of the work could be evaluated.

The project has been planned with a unique resource in mind, working 4 hours a day from Monday to Wednesday. If the project was to be a real project inside a business the estimated cost, at 10 euros per hour worked, 3000 euros.

As far as I Know, Unity licensing system permits a free license for projects with a revenue under 100.000 dollars, besides, Visual Studio is included in the installation for free.

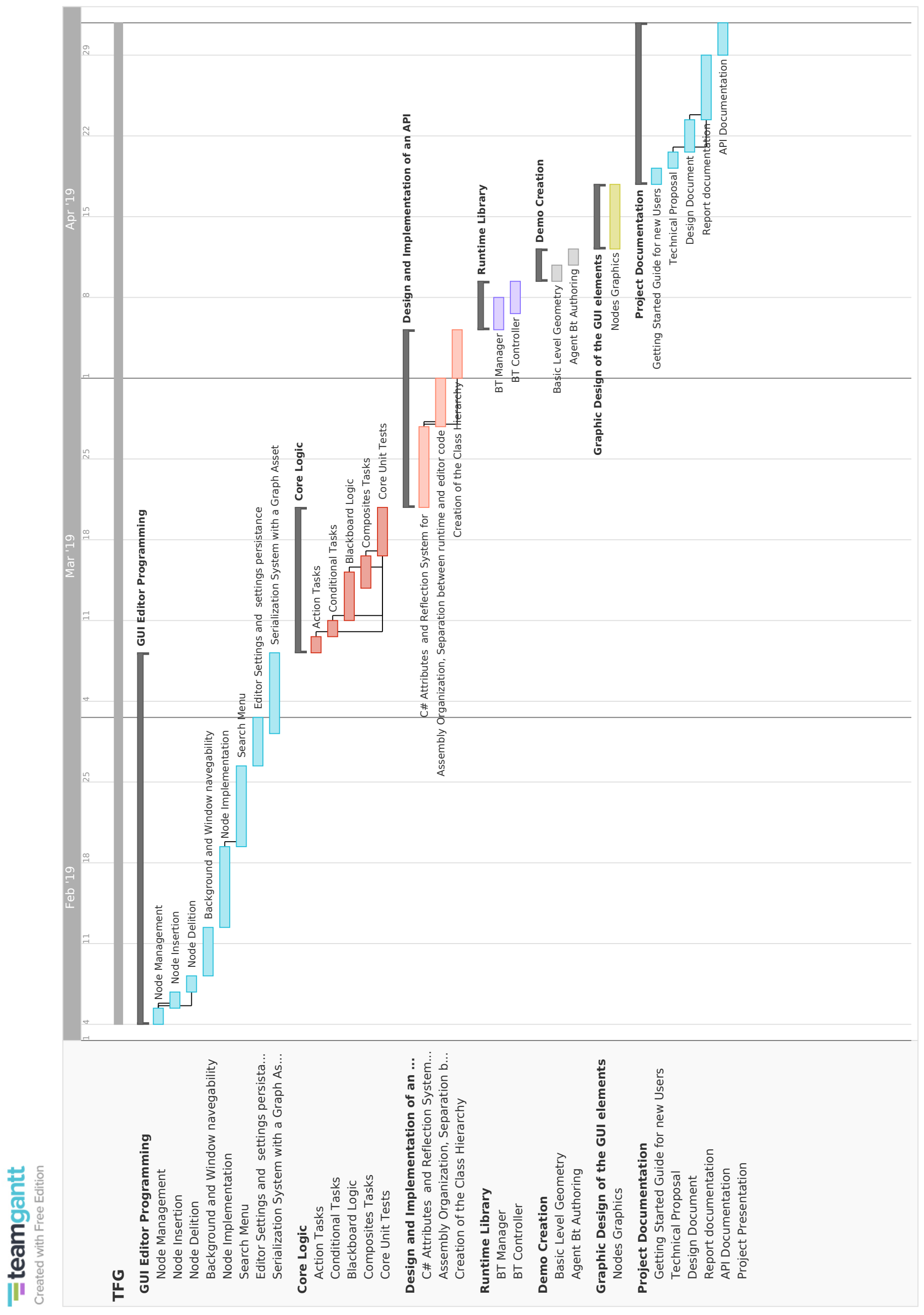


Figure 2.1: Planning of the project using GanttProject

Chapter 3

System analysis and design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirements analysis

3.1.1 Functional requirements

A functional requirement defines a function of the system that is going to be developed. This function is described as a set of inputs, its behavior, and its outputs. The functional requirements can be: calculations, technical details, data manipulations and processes, and any other specific functionality that defines what a system is supposed to achieve.

Input:	The user will press the right button in the project window and select the tree asset.
Output:	An asset will appear in the current project window directory.
A context menu will appear with the option to create a behavior tree graph asset. You will use this asset in the BTs editor to author your tree.	

Table 3.1: Functional requirement «Create Tree asset»

Input:	The user will select the BTEditor option in the Unity's toolbar.
Output:	An empty BT editor.
The editor will be prompting to select a tree in the tree graph field. It wont draw the workspace nor the inspector window.	

Table 3.2: Functional requirement «Open the BT Editor»

Input:	The user will select the tree in the tree field from the BTEditor.
Output:	The BTEditor will be loaded with the serialized data of the graph, if there's any.
The serialized data is stored in binary and will be converted into visual nodes and connections as well as restoring the different node's variables. Unity natively allows to drag assets to object fields, so the user can also drag the asset to the field	

Table 3.3: Functional requirement «Load Tree Graph»

Input:	A node selection in the createNodeWindow, right inside the BT editor
Output:	A displayed node in the editor of the selected type
Having the BTEditor opened and the tree selected, the system will provide a node of the type specified by the user and add it to tree. The Node will be serialized, so every time you enter and exit from the editor the information won't be erased.	

Table 3.4: Functional requirement «Add Node to selected Tree»

Input:	Right click a created node and select the Delete Option
Output:	The node wont show up in the editor and will be erased from the tree.
Having the BTEditor opened and the tree selected, the system must be able to process a delete command from the current selected node. The node will be destroyed along with its serialized data.	

Table 3.5: Functional requirement «Delete right clicked node»

Input:	Click in an exit socket and then click in an entry socket
Output:	A line or some graphical link between the two tasks.
Depending on the type of the task the some sockets may be available or not. The connection can only be between two tasks. You will only be able to create a connection if none of the sockets is hooked.	

Table 3.6: Functional requirement «Create connection when allowed»

Input:	Adjust variables in the tree inspector rect.
Output:	The variable adjusted and its drawer according to the variable type.
The inspector will be created dynamically based on the public field of the task associated with the current selected node. You will be able to edit them and commit its values when the tree is saved and compiled.	

Table 3.7: Functional requirement «Edit and Save node properties»

Input:	The user will click the toggle next to the field.
Output:	If there's a blackboard in the tree it will list the variables of the blackboard that share type with the inspected property. If there's no blackboard, an error message will prompt telling to select one.
When a tree is selected in the editor, a blackboard field will appear. That is the tree's blackboard. The available variables will come from it, and you will be able to modify them in the blackboard window.	

Table 3.8: Functional requirement «Link node property to tree blackboard»

Input:	The user will press the right button in the project window and select the blackboard asset option
Output:	A Blackboard asset in the current directory of the project window.
The blackboard will be created as an asset file. Like the tree graph, data will be serialized into the blackboard.	

Table 3.9: Functional requirement «Create a blackboard Asset»

Input:	The user will select the Blackboard Editor option in the Unity's toolbar.
Output:	A window will appear with a field telling you to select a blackboard.
The editor will contain a list, if a Blackboard is selected, of the different variables contained in the selected blackboard.	

Table 3.10: Functional requirement «Open Blackboard Editor window»

Input:	The user will interact with the list view by adding, removing or editing.
Output:	An updated state of the list of variables.
The list view contained in the Blackboard editor will let the user add variables of the different available types (retrieved dynamically), remove a selected variable and update a variable value using the drawer of that type. The user wont be able to assign scene references to variables in the blackboard, that will only be possible at run time.	

Table 3.11: Functional requirement «EditBlackBoardVariables»

Input:	Click the save button
Output:	A message in the console indicating the compiling status of the tree.
If there's no problem with the tree authored if will compile flawlesly, however, if for example we let a composite node without children it will throw an error. The save and compile of the tree may be called on editor close,on play mode or when a node is created/destroyed if the respective boolean is active in the editor toolbar. It must be possible to save the tree in play mode and update its parameters.	

Table 3.12: Functional requirement «Save and compile Tree.»

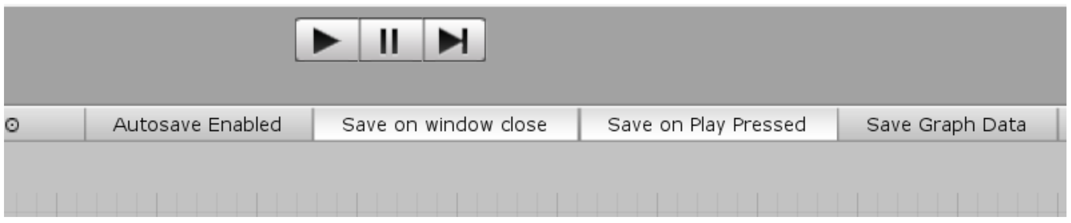


Figure 3.1: The different available save settings in the system.

3.1.2 Non-functional requirements

Non-functional requirements express how the system should be in overall, not what will be capable of. Some of them are:

- Every public function and performed action in the editor will be documented in a separate web page, in plain English and easy to read.
- The editor will behave fluently, without stutters or uncomfortable loading times.
- The run time core will be fast, allowing real time execution.
- The future user of the tool will have a good time extending it. It has been designed to be as easy to extend as possible. This was accomplished by the rule: If something needs an addition or modification, do it in just one step/place.
- The colors of the editor will be clear and not stressing. There wont be an element that will stand out on top of the others unless its intended to do so.
- The whole tool is written in English, its expected from the user to have minimal comprehensive skill in the language.
- The tool has been designed for artists,designers and programmers. It will be easy to understand and use with just a little knowledge on the behavior trees subject.
- The tool will be compatible with every unity version newer than 2019.1 and will be supported by every platform supported by unity.
- The code will be open for modification, distribution and use and will be available in a public repository online.

3.2 System design

When designing the system, I decided to represent the interaction with the system and the class structure using UML diagrams. The diagrams have been updated periodically and represent in essence the interactions of the system. I have omitted the details, specially in the class diagrams for the reader's convenience. In addition, I have split the diagrams for a better time reading it and reviewing them.

- Use case diagrams. These diagrams represent how the user will interact with the given systems.
- Class diagrams are UML diagrams that represent the code structure and its inner relationships in a visual way. I have split the diagrams into 3: The Node hierarchy diagram, the run time system diagram and the editor system diagram.
- Activities diagrams.

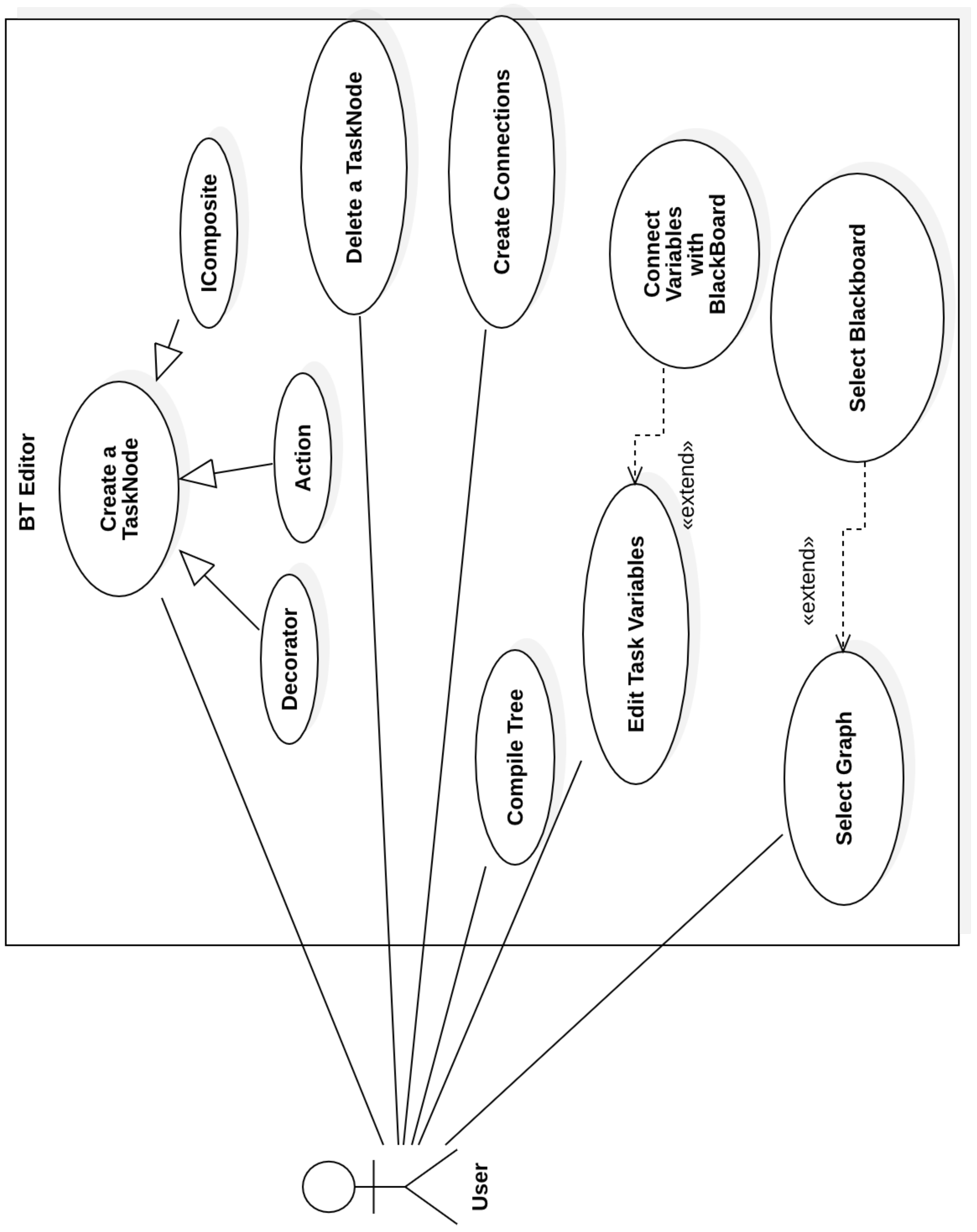
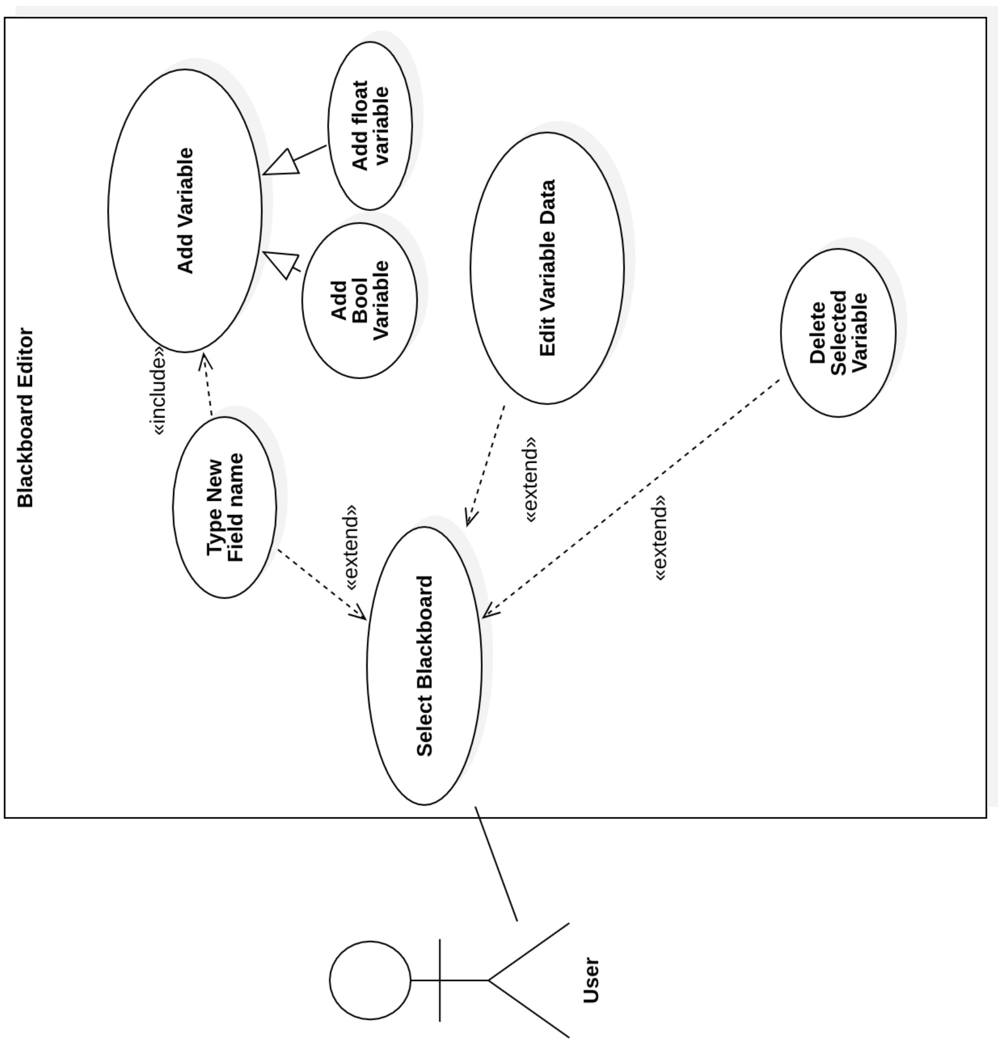


Figure 3.2: The Use case diagram of the BTEditor.



Use Case.png

Figure 3.3: The Use case diagram of the Blackboard Editor.

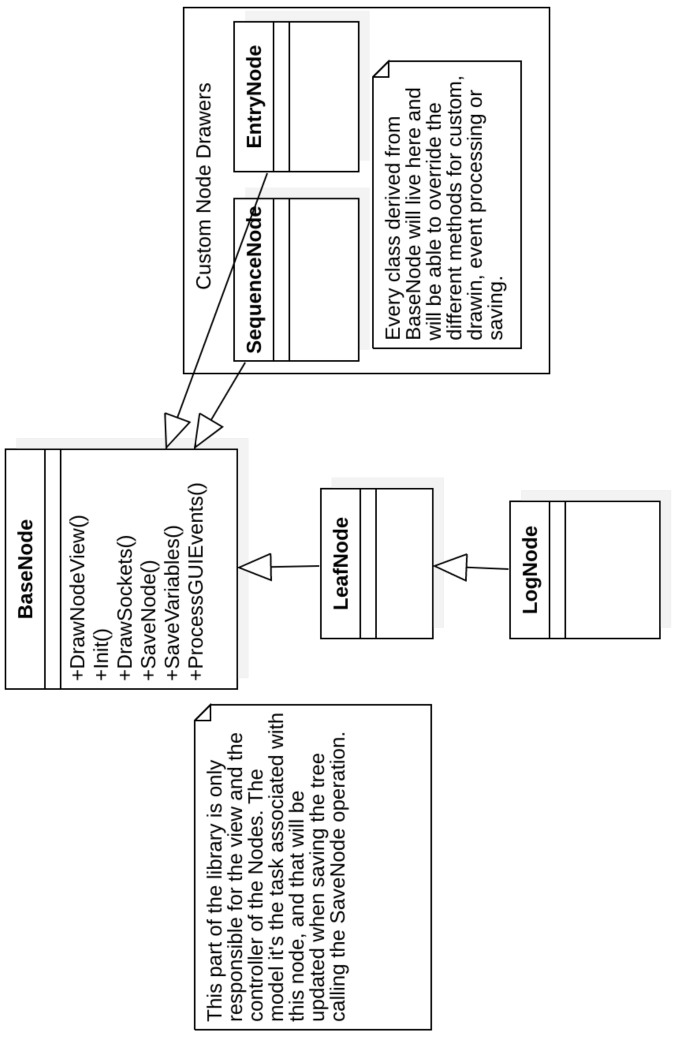

Custom Node Drawers
EntryNode
SequenceNode
Every class derived from BaseNode will live here and will be able to override the different methods for custom, drawin, event processing or saving.
BaseNode
+DrawNodeView()
+Init()
+DrawSockets()
+SaveNode()
+SaveVariables()
+ProcessGUIEvents()
LeafNode
LogNode
This part of the library is only responsible for the view and the controller of the Nodes. The model it's the task associated with this node, and that will be updated when saving the tree calling the SaveNode operation.

Figure 3.4: The Node class hierarchy.

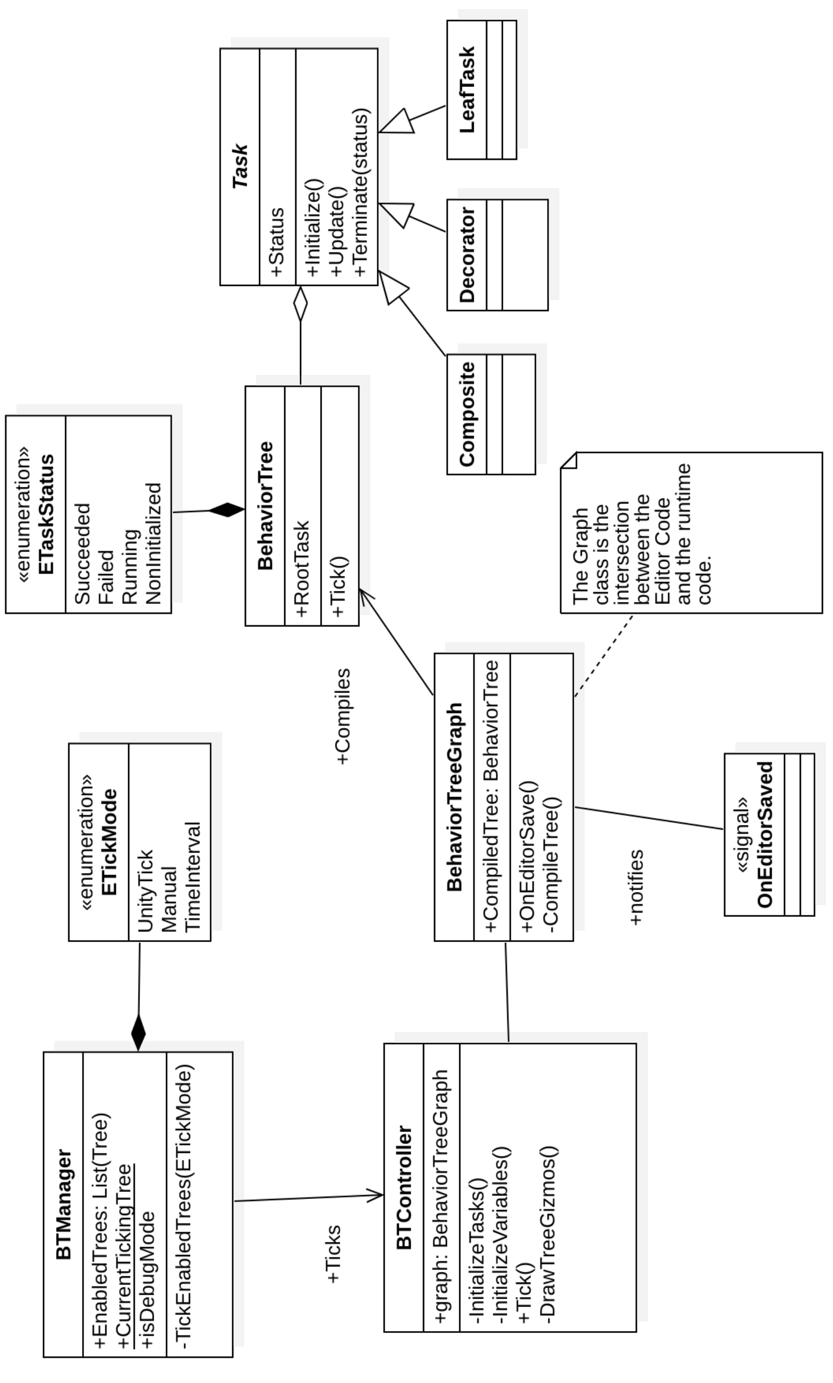


Figure 3.5: The run time class diagram. The classes and relationships depicted here occur when you hit play mode or in a build.

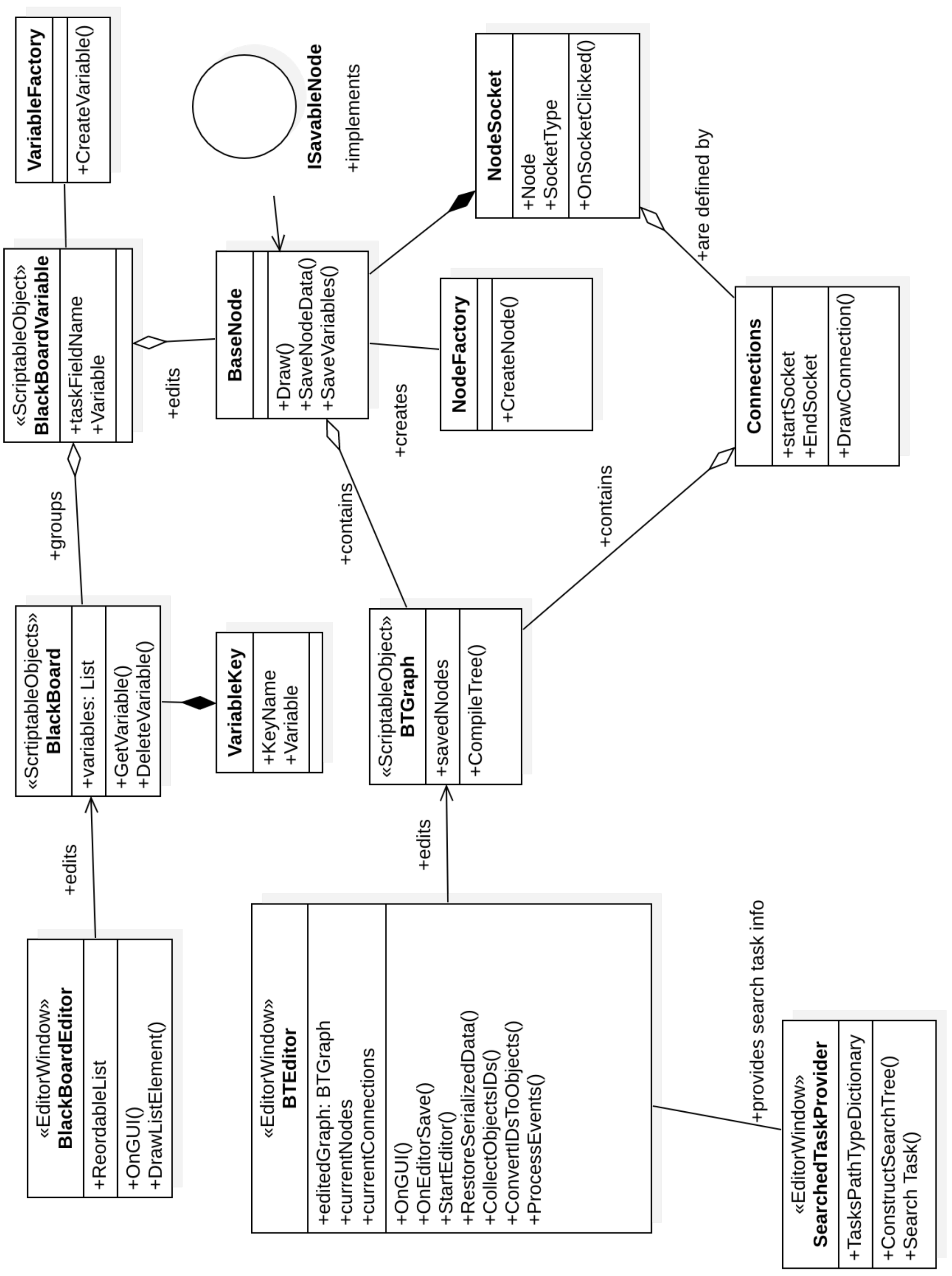


Figure 3.6: The Editor assembly class diagram, here are represented the relationships between classes used in the editor.

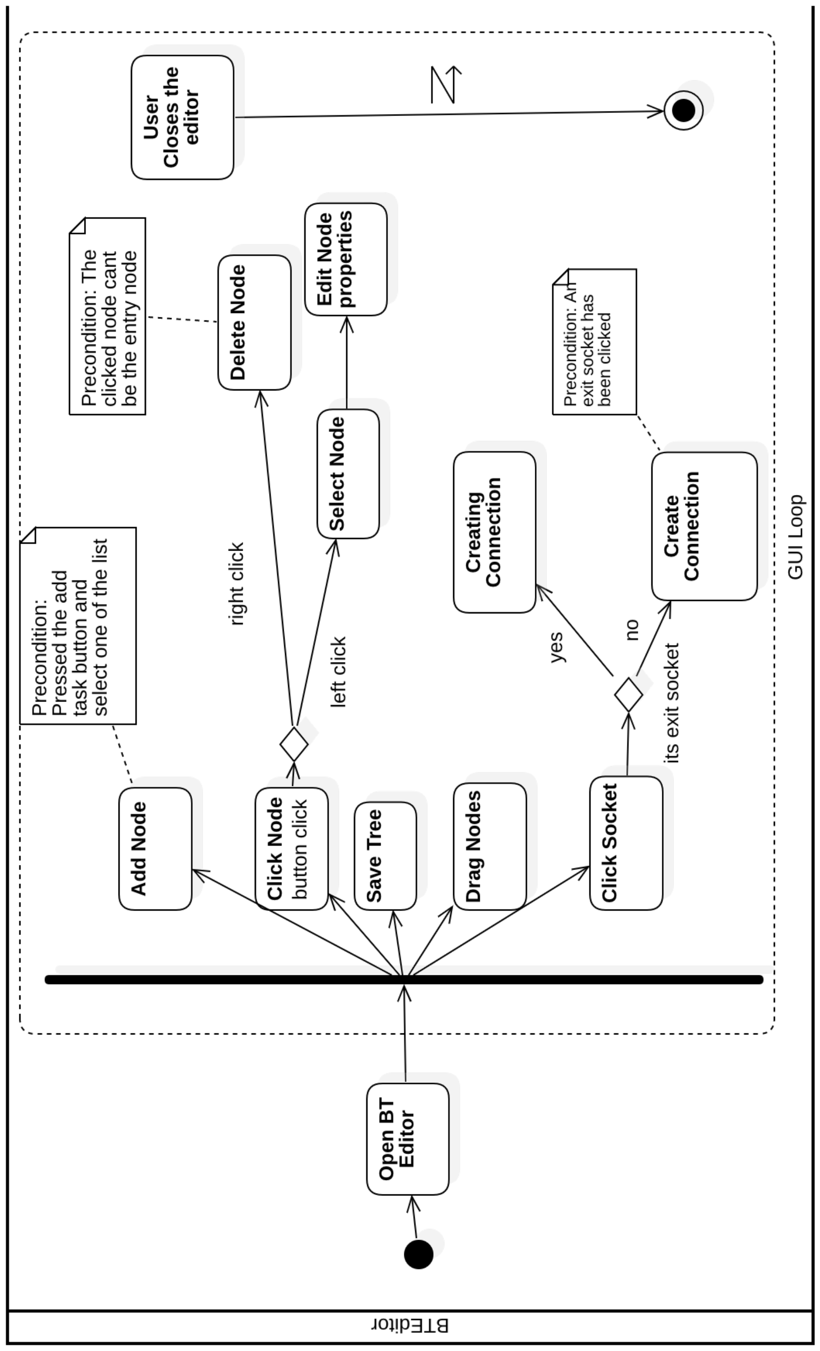


Figure 3.7: BTEditor activity Diagram.

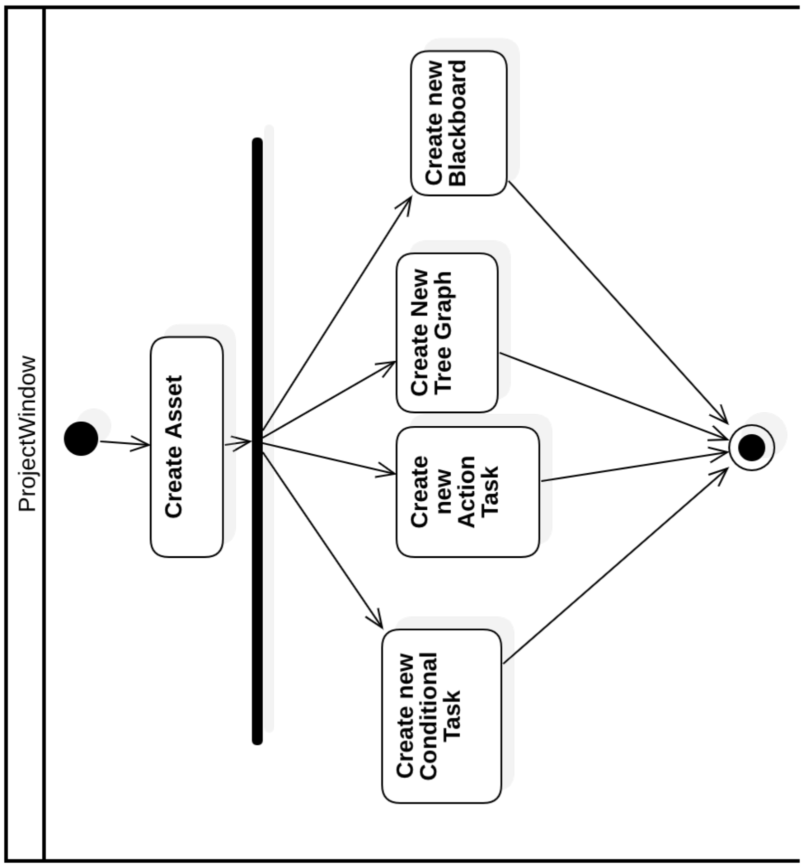
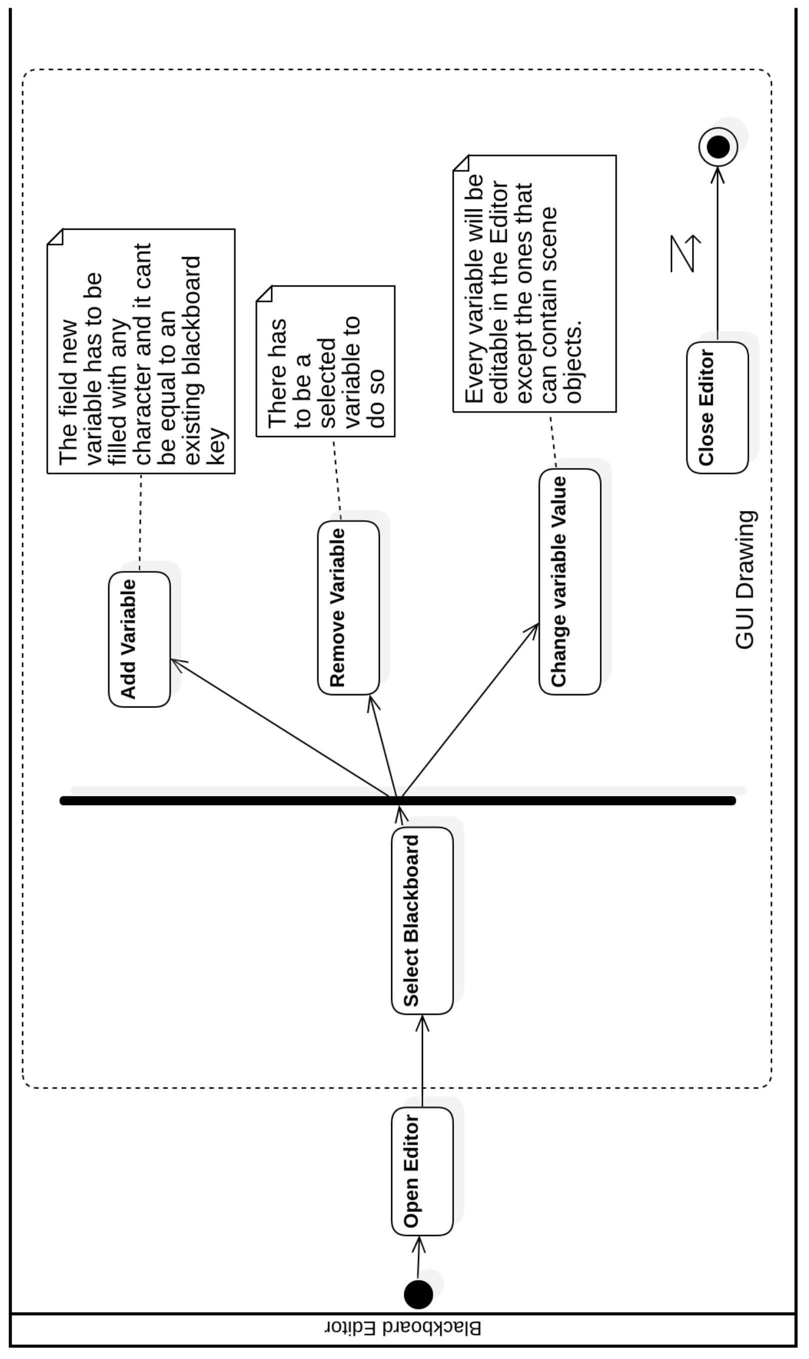


Figure 3.8: ProjectWindow Activity Diagram.


Blackboard Editor
Open Editor
Select Blackboard
Add Variable
Remove Variable
Change variable Value
Close Editor
GUI Drawing
The field new variable has to be filled with any character and it cant be equal to an existing blackboard key
There has to be a selected variable to do so
Every variable will be editable in the Editor except the ones that can contain scene objects.

Editoractivity.png

Figure 3.9: The Blackboard editor activity diagram.

3.3 Interface design

Firstly, as one of my objectives states: "Implement a library that lets us create behavior trees in a similar way as we operate with Unity's native tools" the library has to fit visually in the Unity Editor. For achieving the this the plan was set from the beginning: Using the Unity Editor API for creating the UI and utilizing the same Styles that the editor uses for the native Unity windows.

Unfortunately, Unity's UI is simple and looks old. It doesn't look like the modern and clean interfaces that every new mobile app or webpage implements nowadays. It wouldn't make much sense to spend many hours trying to imitate one of those webs or apps because, after that, my tool would not look like a native Unity window.

Secondly, I have to admit, that my UX skills haven't been developed much yet and the tool reflects that, however, the future plans of the tool include a big improvement regarding the UX and a whole new set of features to improve the usability, such as better selection and modification tools for the nodes. Better manipulation and visualization of the connections and a better and more appealing inspector.

Speaking of the inspector, I have designed the UI of the tool with a diagram that describes the key areas of the interface (See Figure 3.10):

- An **Inspector Window** for manipulating the selected node's properties. These properties will be tweaked by the users in play mode or in edit mode. This section of the UI has to be navigable with the keyboard input. Each type of variable will be drawn in a different way, and the field will have a toggle to connect the variable with a blackboard variable if selected.

- A **Searching Popup Window** for creating the different types of tasks. It will be displayed by the user using the Space key. The Space key has been chosen because it's a very big and comfortable key, and in many software products, it's used as an add menu.

- A **Toolbar** which modifies some editor's properties and some graph's properties. The toolbar will suffer many changes throughout the lifetime of the tool, because it takes a good amount of screen proportion and it needs to have the least possible amount of redundant information.

- A **Working Zoomable Area** where you will be able to create your tree structure. This will be the main area and the most used section. It will respond to mousewheel and click events. It will support dragging events for moving the nodes and connections around.

- A **Tooltip Rect** which will give information about the current selected node. The tool doesn't have a tutorial yet, however it has a high priority in the product backlog. To relieve it, the user will have some contextual information displayed in the rect and a button to hide the tooltip by default.

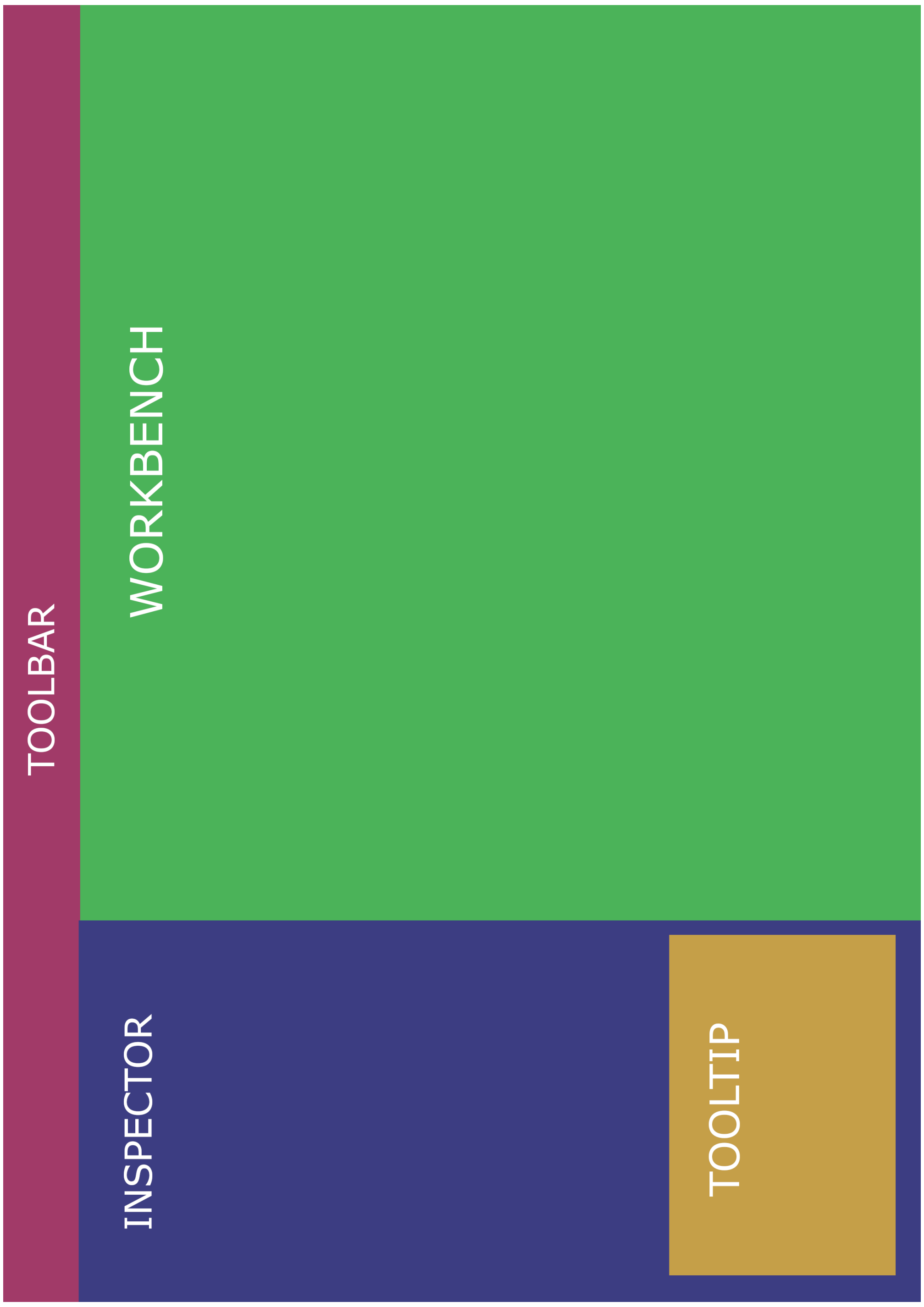


Figure 3.10: A Screenshot of the tool appearance.

Chapter 4

Work development and results

Contents

The initial planning wasn't going bad in terms of hours and expectations, however, there has been a 2 major inconveniences that changed the course of the development. In spite a month of inactivity, I have managed to finish everything in time. Without the two inconveniences I would have managed to finish the project as scheduled. The planning wasn't pessimistic nor optimistic. Needless to say, if the team were to be composed by two members, the project wouldn't have been completely stopped and there wouldn't be the huge delay that occurred during the second sprint.

4.1 Work development

4.1.1 Sprint 1

The state of the project in that sprint is shown in [19] and the appearance of the GUI (See Figures 4.2 and 4.1). The first sprint was done a week before than it was supposed to finish. It ended on the 3r of March. This gave me a small margin. At that time, I didn't how needed that time was for the upcoming sprint. During this sprint the following tasks were completed:

- **Core Tasks**
  - Task Parent Class Implementation

  - Selector Implementation
  - Action Implementation
  - Sequence Implementation
  - Decorator Implementation
  - Condition Implementation
  - Implemented some Tree Unit Tests

- **GUI**
  - Grid with movement on dragging
  - Robust Zooming system.
  - Context Menu For adding and deleting Nodes 4.1.

- **Node Creation System**
  - Creation of SearchTaskPath attribute. It is used in every task that wants to be in the Search Menu.
  - Custom Tree Data Structure for searching and displaying nodes based on a path.
  - Creation of a search basic window.

The are 3 very important features of C# that have been crucial to achieve this sprint:

- **Polymorphism with inheritance**. Every task inherits from ATask and implements 3 virtual methods for initialization, ticking and termination. Its crucial the virtual nature of this methods because the Tree class just ticks the current class and it's the current task responsibility to implement and define the behavior of tick for itself. In Addition, polymorphism allows to convert every task to its parent class, which makes us to write less code and creating a solid and extensible system.

- **Reflection**. It is a feature of C# which allows you to inspect code at run time by accessing the assembly information. In C#, the code is converted into an Intermediate Language and then interpreted by a Virtual Machine. The assembly is this IL and information about it. The Unity Editor when it's running, it's considered run time code which can be inspected in the Unity-Editor assembly. As I want an extensible system with just the creation of a single class, I need to rely on Reflection for inspecting the types that inherit from ATask and displaying them with if the class contains the appropriate attribute.

- **Attribute**. An attribute lets you define metadata for a specific class, for example, lets say that we have 2 different Task classes: Patrol and Seek. Both of them inherit from the same class, but we want to specify a different path in the task creation window for each one of them. A clean and well-tested way of doing this is using an attribute 4.3.

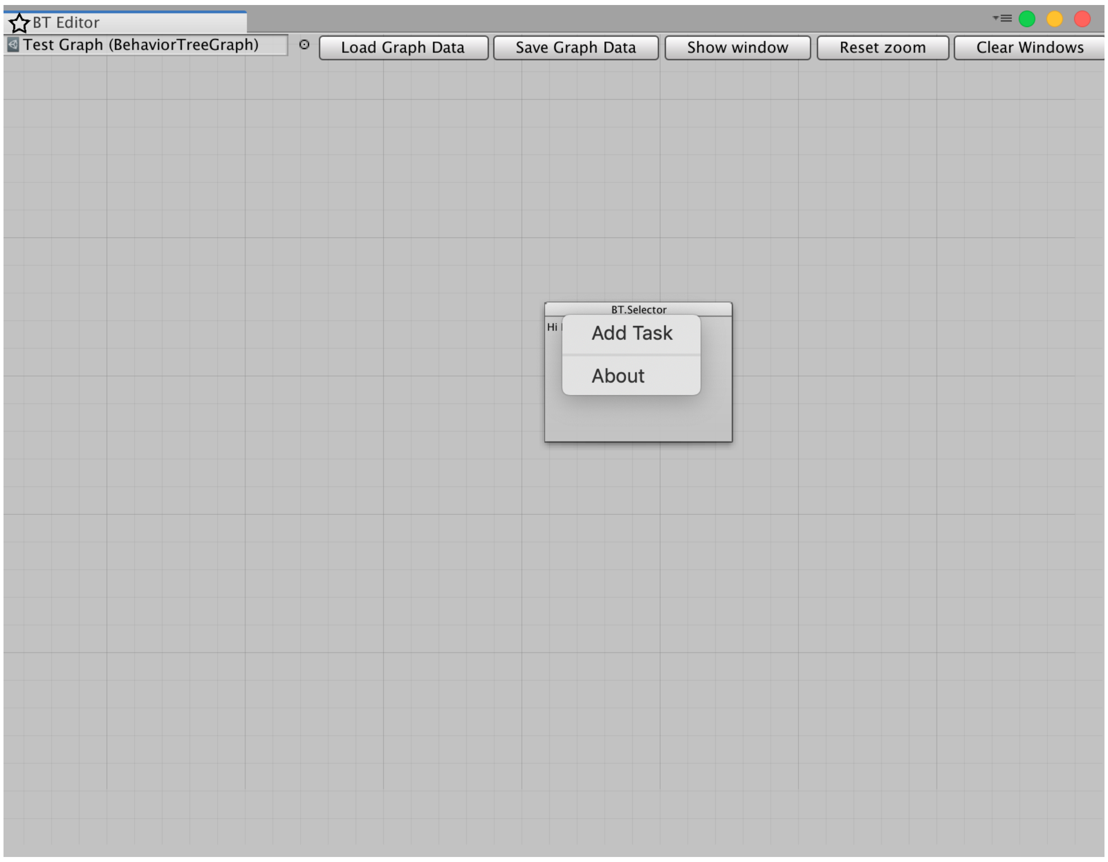


Figure 4.1: Basic context menu for future commands.

The attribute just defines metadata, but there has to be some place in the code to digest it. When we open the Search Tasks Window we call GetTypes (See Figure 4.4) and it gathers the different available classes in the assembly along with their attributes metadata. It stores all of this in a dictionary, a C# implementation of a hash map, for later processing and conversion into a custom Search Tree.

I have created a custom data structure for traversing the paths and drawing them using a recursive algorithm. The creation 4.5 of the tree is triggered right after the types have been retrieved. The tree implementation just holds the root node. Every node has a list of children and for traversing we just need to fetch the list from left to right. As with every tree structure, the traversing can be done using a recursive method (See Figure 4.5, making the code easier to read and maintain. The method is very simple, we receive as the main parameter the current examined node and follows:

– We check if the current node has children. If so, we make the call to traverse again with every element of the list.

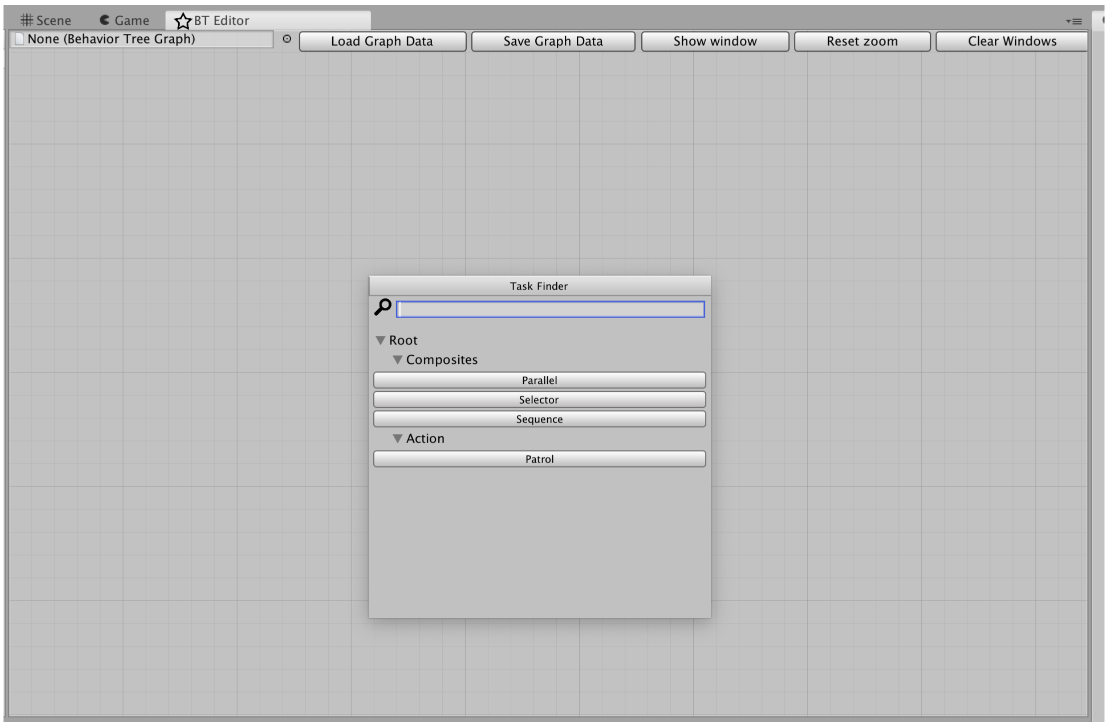
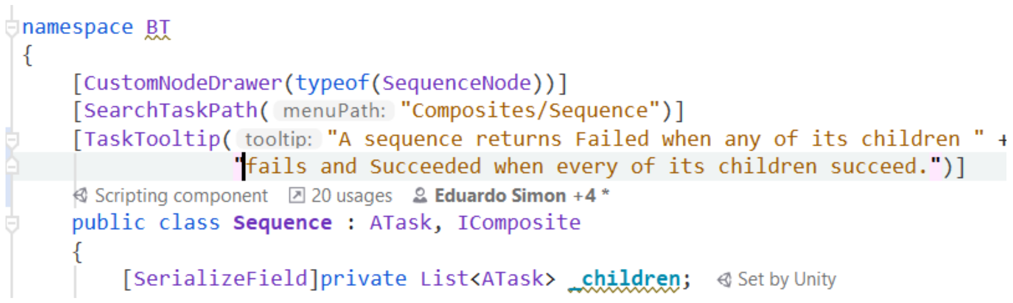


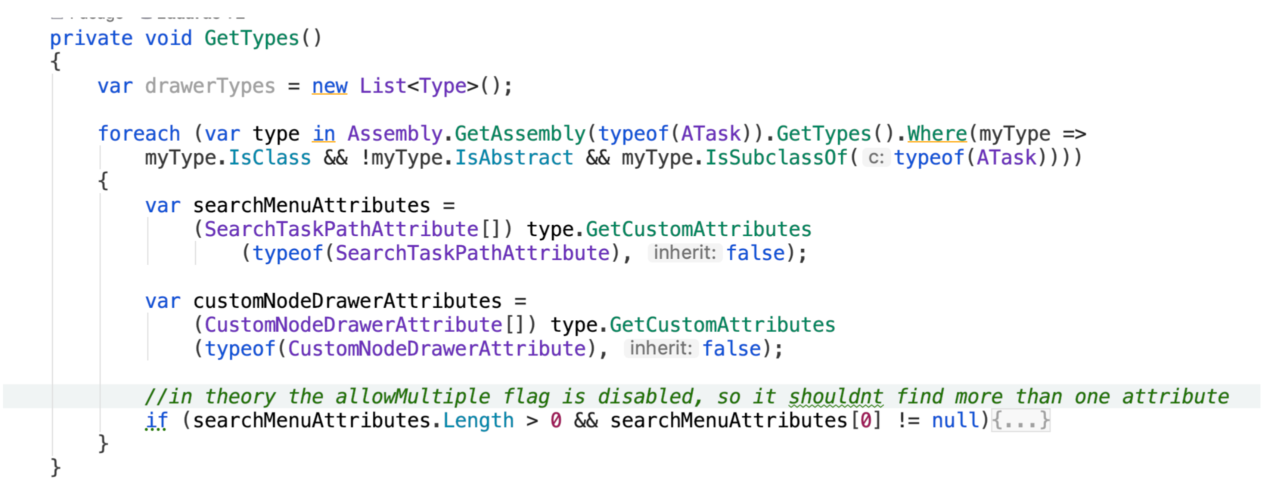
Figure 4.2: The state of the editor when finished the Sprint 1.



```
namespace BT
{
    [CustomNodeDrawer(typeof(SequenceNode))]
    [SearchTaskPath( menuPath: "Composites/Sequence")]
    [TaskTooltip( tooltip: "A sequence returns Failed when any of its children " +
                  "fails and Succeeded when every of its children succeed.")]
    Scripting component   20 usages   Eduardo Simon +4 *
    public class Sequence : ATask, IComposite
    {
        [SerializeField]private List<ATask> _children;   Set by Unity
```

Figure 4.3: An example of an attribute used to add metadata to the Sequence class. We only want this concrete class to have that search menu data. We could have created an abstract method and made every new task implement it, but attributes work better with reflection and are cope better with changes that inheritance.

– If the children list is empty we have arrived to a leaf node and to a stop condition of our recursive algorithm and we can do the needed operations. In this case, we need the name of the node and the types that holds. When reaching this point, the algorithm will start returning the previous calls to the function, resulting in a constructed tree



```
private void GetTypes()
{
    var drawerTypes = new List<Type>();

    foreach (var type in Assembly.GetAssembly(typeof(ATask)).GetTypes().Where(myType =>
        myType.IsClass && !myType.IsAbstract && myType.IsSubclassOf( c: typeof(ATask))))
    {
        var searchMenuAttributes =
            (SearchTaskPathAttribute[]) type.GetCustomAttributes
                (typeof(SearchTaskPathAttribute),  inherit: false);

        var customNodeDrawerAttributes =
            (CustomNodeDrawerAttribute[]) type.GetCustomAttributes
            (typeof(CustomNodeDrawerAttribute),  inherit: false);

        //in theory the allowMultiple flag is disabled, so it shouldnt find more than one attribute
        if (searchMenuAttributes.Length > 0 && searchMenuAttributes[0] != null){...}
    }
}
```

Figure 4.4: This method is called when we open the search window and stores their types and their metadata.

The node consists of a Name, a type and a Parent. When constructing the tree I just traverse the dictionary filled in the GetTypes method and filter the paths grouping their common parts.

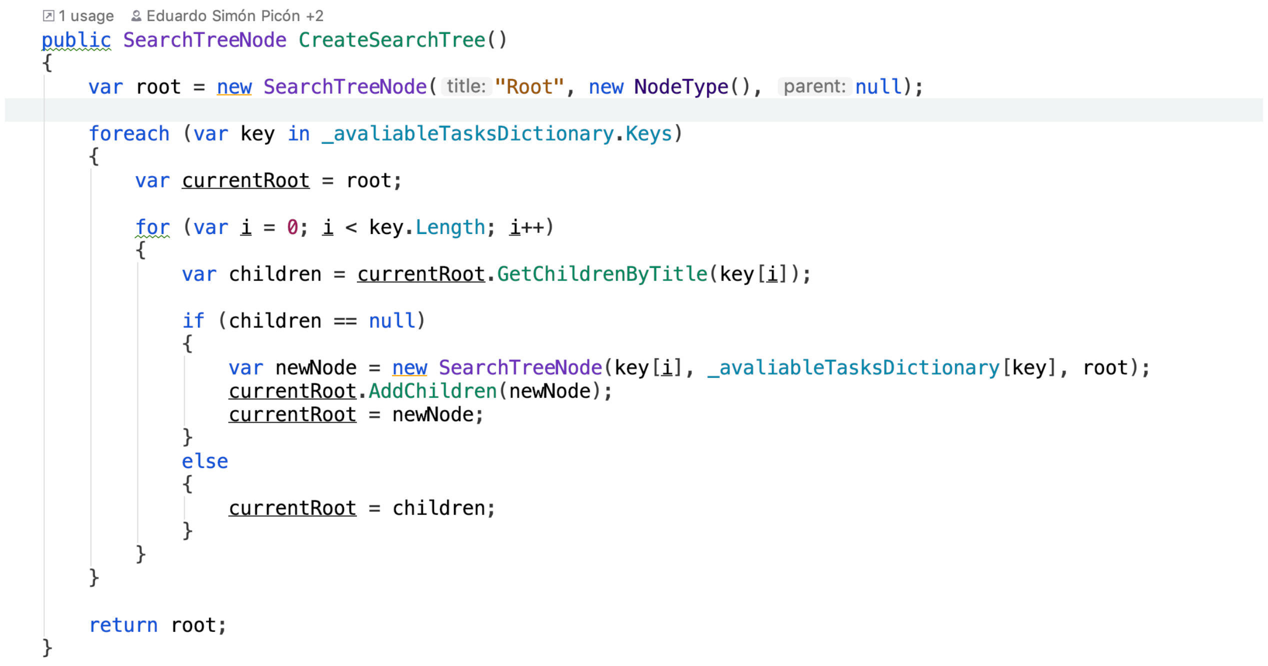
Having the paths filtered as a tree structure makes it very easy to draw it. Every child of a node in the tree will be in the same category allowing me to hide or collapse by category. I use the API from the UnityEditor to draw the search tree (See Figure 4.6) with buttons for the tasks and foldouts for the categories.

The implementation of the Tree core it has been based on this amazing implementation of BTs [5]. The implementation part was very easy, just translate the code into C# and think in the ad-dons that would need the library. The hard part was establishing the Task Base Type taking into account the unity serialization system. The fact that the only way to serialize reference to objects in Unity is by inheriting from Scriptable Object led to an enormous inconvenience which made the second sprint last 3 months.

4.1.2 Sprint 2

The second sprint has been a continuous struggle and made me think about restarting the project and coding it again from the ground up. There were 2 major issues that caused a huge deviation from the original planning. The sprint was active for 3 months, from the 3rd of March to the 5Th of June and the two main causes were:

- The fact that I got hired by a software development company. This reduced the amount of hours dedicated to the project weekly from 20 to 6 or 8.
- The little knowledge and the poor documentation of the Unity Serialization System. I ended up in a loophole because my tool was useless. It couldn't serialize a



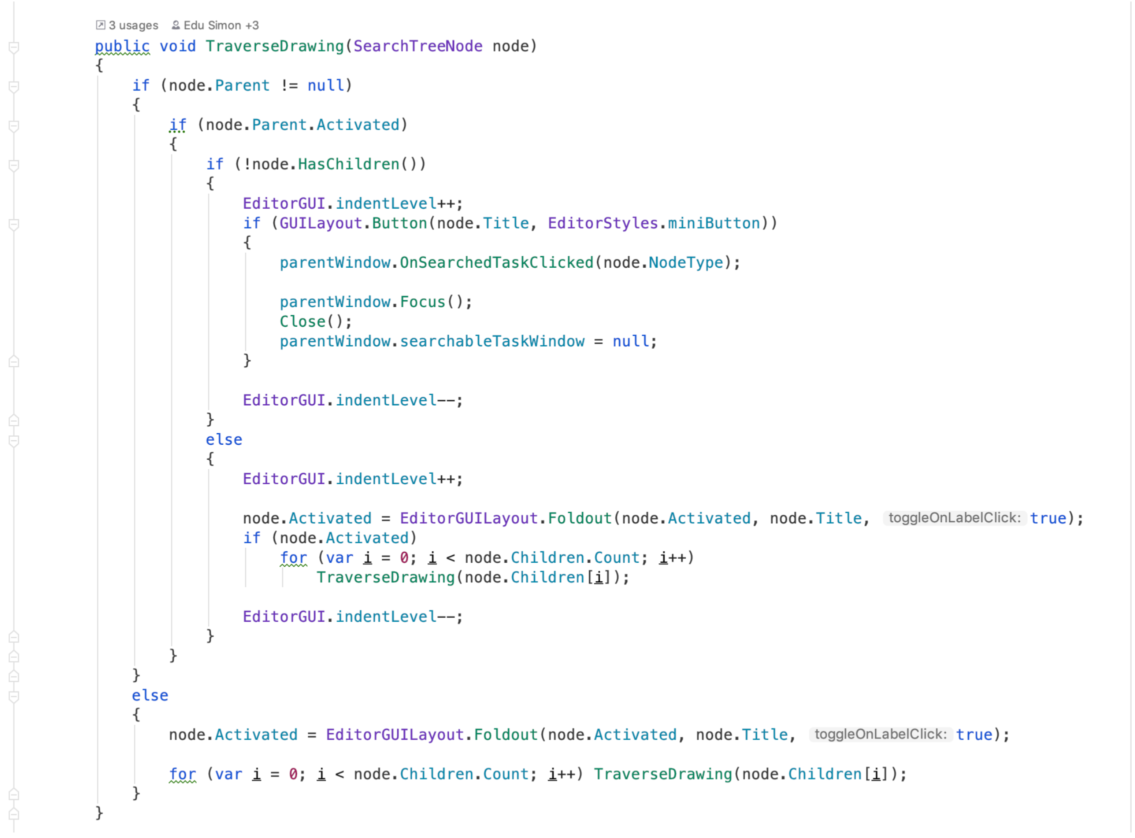
```
1 usage   Eduardo Simón Picón +2
public SearchTreeNode CreateSearchTree()
{
    var root = new SearchTreeNode( title: "Root", new NodeType(),  parent: null);

    foreach (var key in _avaliableTasksDictionary.Keys)
    {
        var currentRoot = root;

        for (var i = 0; i < key.Length; i++)
        {
            var children = currentRoot.GetChildrenByTitle(key[i]);

            if (children == null)
            {
                var newNode = new SearchTreeNode(key[i], _avaliableTasksDictionary[key], root);
                currentRoot.AddChildren(newNode);
                currentRoot = newNode;
            }
            else
            {
                currentRoot = children;
            }
        }
    }

    return root;
}
```

Figure 4.5: Creation of the tree using a recursive algorithm. The creation of each node is done when there are no more children to traverse. When this ocurrs, the function will start returning the stacked calls.

single bit of information and comply with the functional requirements at the same time. However, I managed to solve it using the trial and error method and by deep diving into the Unity forums where other developers have done similar things and have solved it. The implementation details of an editor like mine are not public, so I had to make adaptations to solutions that I found online.

The different accomplished tasks were:

- **GUI**
  - Custom inspector window dynamically created based on the tasks fields using reflection
  - Created sockets in the different nodes with different behavior based on the type of the task.
  - Delete Node Command implemented that removes connections
  - Created an entry point node which connects to the root task of the tree.
- **Run-time**
  - Behavior Manager Singleton for scheduling trees and ticking the trees based on its component variables such as as execute on enable.



```
3 usages   Edu Simon +3
public void TraverseDrawing(SearchTreeNode node)
{
    if (node.Parent != null)
    {
        if (node.Parent.Activated)
        {
            if (!node.HasChildren())
            {
                EditorGUI.indentLevel++;
                if (GUILayout.Button(node.Title, EditorStyles.miniButton))
                {
                    parentWindow.OnSearchedTaskClicked(node.NodeType);

                    parentWindow.Focus();
                    Close();
                    parentWindow.searchableTaskWindow = null;
                }

                EditorGUI.indentLevel--;
            }
            else
            {
                EditorGUI.indentLevel++;

                node.Activated = EditorGUILayout.Foldout(node.Activated, node.Title,  toggleOnLabelClick: true);
                if (node.Activated)
                    for (var i = 0; i < node.Children.Count; i++)
                        TraverseDrawing(node.Children[i]);

                EditorGUI.indentLevel--;
            }
        }
    }
    else
    {
        node.Activated = EditorGUILayout.Foldout(node.Activated, node.Title,  toggleOnLabelClick: true);

        for (var i = 0; i < node.Children.Count; i++) TraverseDrawing(node.Children[i]);
    }
}
```

Figure 4.6: The method that draws the tree in a recursive way. Following the same principle as the creation does, it traverses the tree looking for leafs, in this case, the leafs are the tasks that will be displayed as button in the seach task popup.

  – Behavior Tree controller component. This is the main interface between the run-time and the edited tree. To make a NPC run a BT you just need to add the BTManager to that scene and a BTController to the NPC along with your created tree.

- **Serialization**
  – Serializing Connections
  – Serializing Nodes
  – Loading and Saving in PlayMode as well as in edit mode.

- **Micro demo creation**
  – Authoring small seeking tree
  – Implemented seek action and is. target in range condition

Serialization

Serialization is the process of converting your objects and data structures to a format that is possible to store in a file. If the data is storable in a file, it can be shared throught any digital medium or saved and reused later for reconstructing the data of your application.

Serialization is heavily used in the web to transfer data between the database and the client. In my case, which is a little bit different than the web, the serializing system has two main functions:

- Store persistent data for its use in later sessions or in the build.
- Be the bridge between native land and managed land.

Unity's core is programmed in C++ and has a layer of Editor code and some extra functionality on top written in C#. With this, they can achieve great speeds for crucial operations and a better and more friendly syntax in the more high level systems.
Every time there's a recompile, unity has to reload the mono assemblies, the unity dlls [18]. All the serialized data will be load to the native side of unity creating a representation of it and every piece of the mono side of unity will be torn apart. Mono [8] is a framework for creating C# cross-platform applications. Unity uses mono to run you C# scripts every time you hit the play button or run the build. When there's nothing left in the managed side, Unity will recreate the data stored in C++ in the managed layer. The nature of this process constraints the serializer in that it has to be super fast, otherwise, the iteration time inside the unity editor will be huge. For achieving this, the folks at Unity had to limit its capabilities [17]. So, the serializable types need to follow the following rules:

- Be public, or have [SerializeField] attribute
- Not be static
- Not be const
- Not be readonly
- The fieldtype needs to be of a type that we can serialize.
- Custom non abstract classes with [Serializable] attribute.
- Custom structs with [Serializable] attribute. (new in Unity4.5)
- References to objects that derive from UntiyEngine.Object
- Primitive data types (int,float,double,bool,string,etc)
- Array of a fieldtype we can serialize

- List<T> of a fieldtype we can serialize

Serialized data is data stored in disk in a binary format or a text format depending on the user's settings and cant break any of the rules described above.
The thing that destroyed my planning was how Unity Serialized references. When you mark a regular C# class as Serializable, if you create references of that class they wont be serialized as a reference or pointer, but by value. Initially, my connections and tasks were plain classes, but I had to ditched them because when I was entering to edit mode from play mode all the changes weren't reflect. The changes had been applied to the copy kept in play mode hence the edit mode copy didn't notice the changes.

Furthermore, the only way to serialize classes is to inherit from UnityEngine.Object and there are 2 possible candidates here:

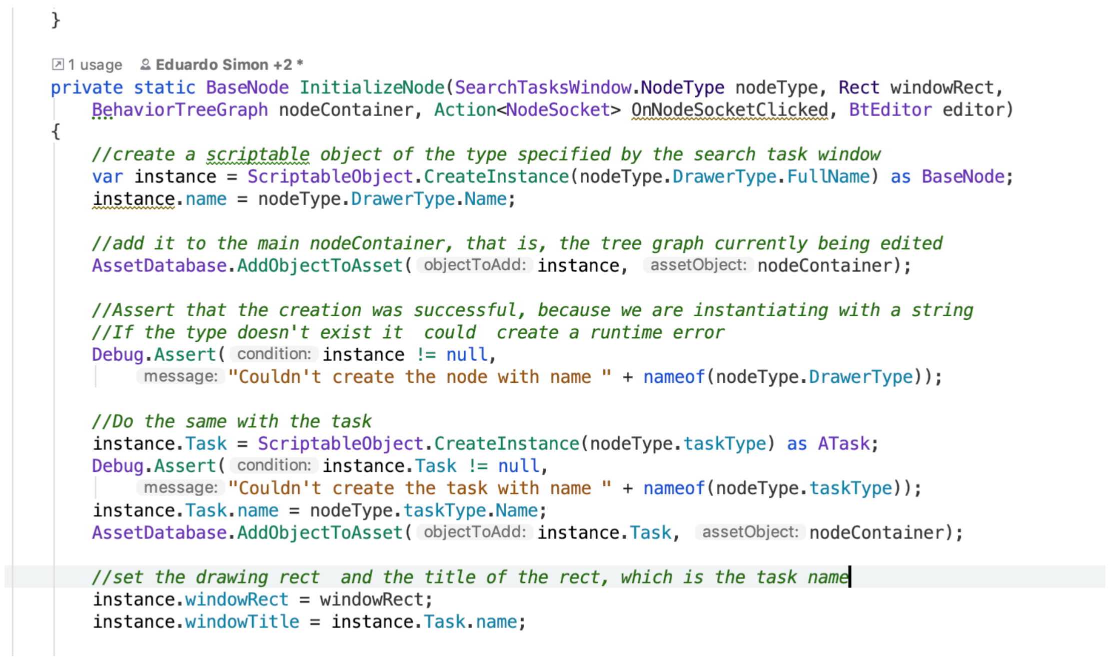
- MonoBehavious: The building block of every component in unity. They have a transform component, which means that they have a visual representation in the scene and as such, they can be placed in the virtual world. But they don't serve my purpose as you cannot edit its values in play mode and see the changes reflected in edit mode. In addition, a task shouldn't be a visual object, because is just logic. It doesn't seem right to say that a task IS-A MonoBehaviour because it lacks its fundamental property. This clashed with my functional requirements so I had to go with the other option.

- ScriptableObject: Sometimes called SO. They can be both Assets in the project window or serialized objects at run-time. As well as monobehaviors they can't be constructed using the new keyword and they received certain callbacks like OnEnable and OnDisable.

The Scriptable Object approach seemed promising, and for certain scenarios it worked. I created the instance whenever a new node was created and asigned to the reference field. If I didn't exit the editor worked like a charm, but as soon as you exit, everything disappeared.

Surfing the web looking for answers to my problem I found [15]. Summarizing, the trick was to add the created instance directly to an asset Scriptable Object stored in the project window, remember that SO can be assets or objects in memory at run-time. Thanks to that trick the data would be properly serialized and loaded whenever you reenter the editor. The downside to this approach is that I had to refactor the whole Editor and make every worth piece of data a Scriptable Object. The Creation Method got a little complicated and a cumbersome to deal with, but it finally worked. Thanks to this approach I could edit the tree in play mode and see the changes reflect in Edit mode.

One problem with this approach is that every asset added to the main one would show up in the project window and when creating big trees this can become very uncomfortable 4.9. However, every UnityEngine.Object has a field called HideFlags. This flags allow to add special treatment to that specific object regarding the garbage collection, the



```
}

1 usage   Eduardo Simon +2 *
private static BaseNode InitializeNode(SearchTasksWindow.NodeType nodeType, Rect windowRect,
    BehaviorTreeGraph nodeContainer, Action<NodeSocket> OnNodeSocketClicked, BtEditor editor)
{
    //create a scriptable object of the type specified by the search task window
    var instance = ScriptableObject.CreateInstance(nodeType.DrawerType.FullName) as BaseNode;
    instance.name = nodeType.DrawerType.Name;

    //add it to the main nodeContainer, that is, the tree graph currently being edited
    AssetDatabase.AddObjectToAsset( objectToAdd: instance,  assetObject: nodeContainer);

    //Assert that the creation was successful, because we are instantiating with a string
    //If the type doesn't exist it  could  create a runtime error
    Debug.Assert( condition: instance != null,
        message: "Couldn't create the node with name " + nameof(nodeType.DrawerType));

    //Do the same with the task
    instance.Task = ScriptableObject.CreateInstance(nodeType.taskType) as ATask;
    Debug.Assert( condition: instance.Task != null,
        message: "Couldn't create the task with name " + nameof(nodeType.taskType));
    instance.Task.name = nodeType.taskType.Name;
    AssetDatabase.AddObjectToAsset( objectToAdd: instance.Task,  assetObject: nodeContainer);

    //set the drawing rect  and the title of the rect, which is the task name
    instance.windowRect = windowRect;
    instance.windowTitle = instance.Task.name;
```

Figure 4.7: The node creation method. Every object that needs to be serialized has to follow the same procedure in order to be created and serialized.

hierarchy drawing or the including of the asset object in the build. With just a line of code I could hide every child asset of the tree 4.8.

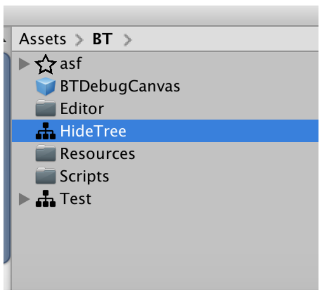


Figure 4.8: Every created ScriptableObject has the hideflags variable set to HideInHierarchy.

Another strange behavior that I experimented when building the tool, as well as the serialization problems due to the poor documentation and my little knowledge of the subject, was that whenever I entered play mode even though the references were serialized, the editor window would lose the reference. After several days of trials and errors and many visited pages I stumbled upon a small post which gave me the solution [14].

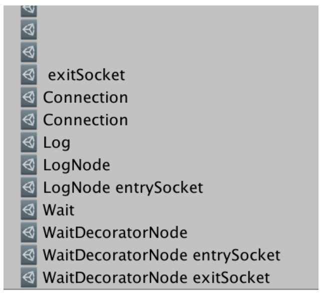
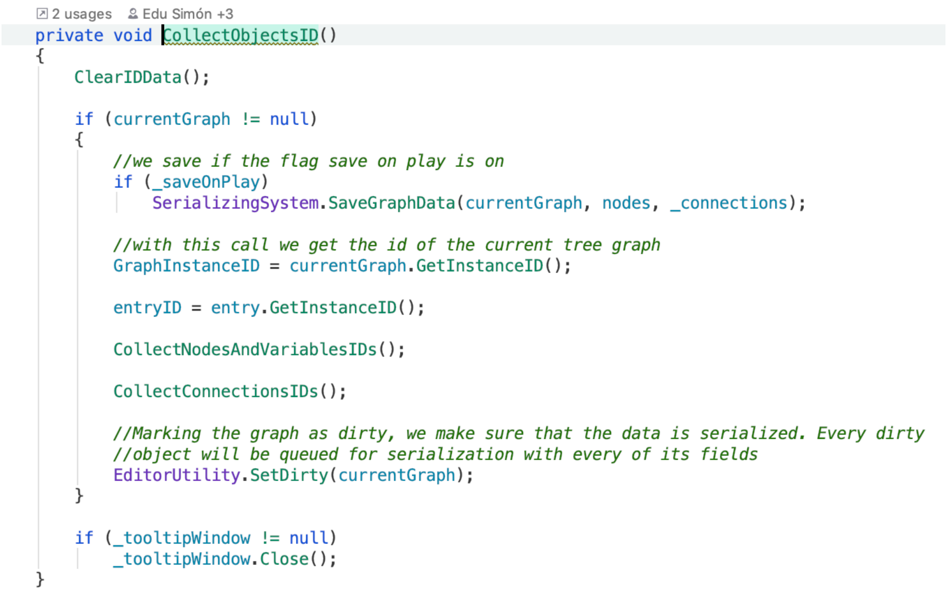


Figure 4.9: Every scriptable object has the default hideflags value.

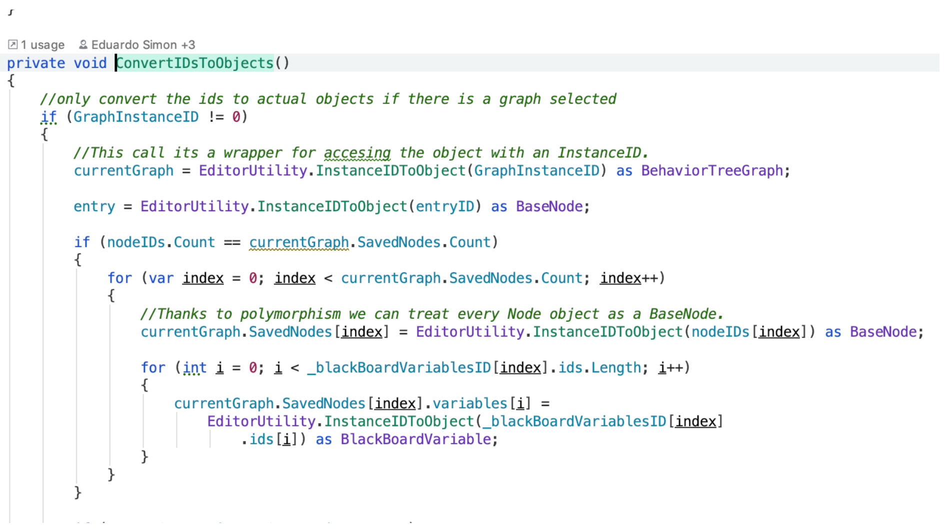
In Unity, every class derived from UnityEgine.Object has an integer that uniquely identifies it. This field is called InstanceID and is great for identifying objects in a build or in the editor. Unfortunately, it has a big caveat that makes its use very limited: the id is regenerated in every play/edit session. In Spite of this inconvenience, I can still use it for reloading the reference when entering Play mode, as I cache the ID reference when the user hits the play button of the selected tree graph along with all the objects that contains 4.10. When definitely in PlayMode I convert every ID to an object if there's a tree being edited with a call to ConvertIDToObject from the UnityEditor API. 4.11.



```
2 usages  Edu Simón +3
private void CollectObjectsID()
{
    ClearIDData();

    if (currentGraph != null)
    {
        //we save if the flag save on play is on
        if (_saveOnPlay)
            SerializingSystem.SaveGraphData(currentGraph, nodes, _connections);

        //with this call we get the id of the current tree graph
        GraphInstanceID = currentGraph.GetInstanceID();

        entryID = entry.GetInstanceID();

        CollectNodesAndVariablesIDs();

        CollectConnectionsIDs();

        //Marking the graph as dirty, we make sure that the data is serialized. Every dirty
        //object will be queued for serialization with every of its fields
        EditorUtility.SetDirty(currentGraph);
    }

    if (_tooltipWindow != null)
        _tooltipWindow.Close();
}
```

Figure 4.10: Collect every relevant ID from every object in the work space.

The state of the tool at the end of the second sprint in a video can be seen in [20].



```
1 usage   Eduardo Simon +3
private void ConvertIDsToObjects()
{
    //only convert the ids to actual objects if there is a graph selected
    if (GraphInstanceID != 0)
    {
        //This call its a wrapper for accesing the object with an InstanceID.
        currentGraph = EditorUtility.InstanceIDToObject(GraphInstanceID) as BehaviorTreeGraph;

        entry = EditorUtility.InstanceIDToObject(entryID) as BaseNode;

        if (nodeIDs.Count == currentGraph.SavedNodes.Count)
        {
            for (var index = 0; index < currentGraph.SavedNodes.Count; index++)
            {
                //Thanks to polymorphism we can treat every Node object as a BaseNode.
                currentGraph.SavedNodes[index] = EditorUtility.InstanceIDToObject(nodeIDs[index]) as BaseNode;

                for (int i = 0; i < _blackBoardVariablesID[index].ids.Length; i++)
                {
                    currentGraph.SavedNodes[index].variables[i] =
                        EditorUtility.InstanceIDToObject(_blackBoardVariablesID[index]
                            .ids[i]) as BlackBoardVariable;
                }
            }
        }
```

Figure 4.11: Convert every ID to its respective object.

4.1.3 Sprint3

The third sprint started on the 1st of June, 2 months after that it was supposed to start, however, the serialization system was working and the simplest tree that I could build was running, so that kinda cheered me up and motivated me to keep going with a positive attitude!.

During the third sprint, the following tasks were completed:

- **API Refactor**. The API refactor was crucial to keep up with the requirements of the tool. The API tool at the end of the second sprint was not bad, but it needed the creation and manipulation of:

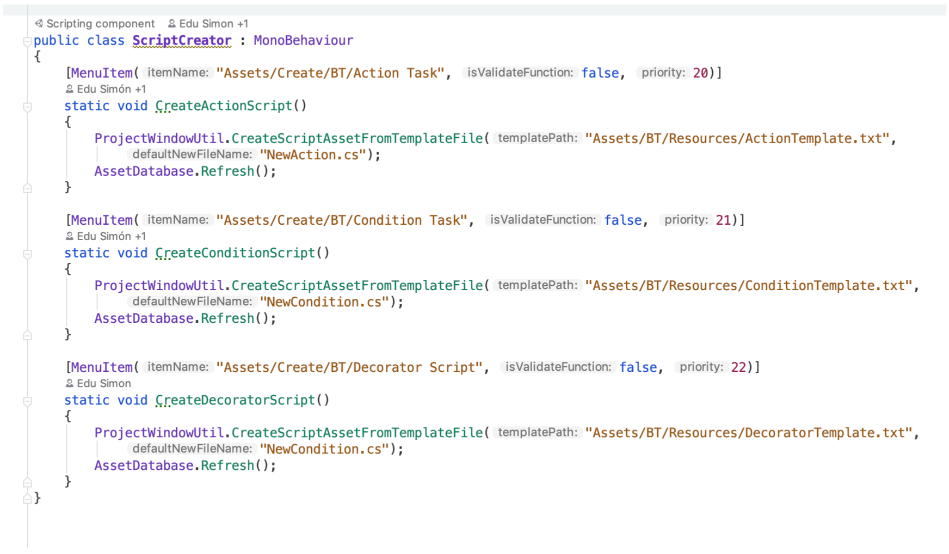
  - A new class inheriting from BaseNode
  - A new class inheriting from Decorator, Action or Conditional.
  - The override of a virtual C# property for getting the task
  - The override of the SaveNode along with syncing the task and the Node variables.

  This approach clearly wasn't optimal and didn't follow the rule: extending by just creating a new class in one place. Therefore, I decided to modify this approach. Firstly, I established my final goal in terms of extendability, if the user wants a new task, only a new class inheriting from Action,Decorator or Conditional had to be created. For achieving this, I needed to code from the ground up the SaveNode method, as it was the clear bottleneck here. The solution was to use reflection and inspect the fields of the task and use that to sync the data between the inspector and the compiled tree.

  At the point of writing this, they only things that needs to be done to create a new class is to go to the create menu in unity and select one of the three options: new Action, new Decorator or new Conditional. A script of the chosen type will be created like a normal unity script. The Attribute MenuItem on top of each of the static methods in the ScriptCreator class creates a new tab in the Unity Create Menu (See Figure 4.12).

  Secondly, I clearly had to put a layer of abstraction between the atomic types such as float or int and the variables of the inspector. That was needed for the following reasons:

  - I needed to serialize them as real references, so they had to inherit from ScriptableObject
  - They would need an special drawing method for each of them and have the data stored in a field with the same name to retrieve the data using Reflection
  - I needed polymorphism support and because of that, they all had to inherit from a base class (See Figure 4.13).



```
Scripting component   Edu Simon +1
public class ScriptCreator : MonoBehaviour
{
    [MenuItem( itemName: "Assets/Create/BT/Action Task",  isValidateFunction: false,  priority: 20)]
    Edu Simón +1
    static void CreateActionScript()
    {
        ProjectWindowUtil.CreateScriptAssetFromTemplateFile( templatePath: "Assets/BT/Resources/ActionTemplate.txt",
             defaultNewFileName: "NewAction.cs");
        AssetDatabase.Refresh();
    }

    [MenuItem( itemName: "Assets/Create/BT/Condition Task",  isValidateFunction: false,  priority: 21)]
    Edu Simón +1
    static void CreateConditionScript()
    {
        ProjectWindowUtil.CreateScriptAssetFromTemplateFile( templatePath: "Assets/BT/Resources/ConditionTemplate.txt",
             defaultNewFileName: "NewCondition.cs");
        AssetDatabase.Refresh();
    }

    [MenuItem( itemName: "Assets/Create/BT/Decorator Script",  isValidateFunction: false,  priority: 22)]
    Edu Simon
    static void CreateDecoratorScript()
    {
        ProjectWindowUtil.CreateScriptAssetFromTemplateFile( templatePath: "Assets/BT/Resources/DecoratorTemplate.txt",
             defaultNewFileName: "NewCondition.cs");
        AssetDatabase.Refresh();
    }
}
```

Figure 4.12: Extend the library functionality by creating a script.

When creating a task node, the task fields where inspected using reflection and a NodeFactory created the appropriate type of variable based on the type of the field inspected (See Figure 4.14).

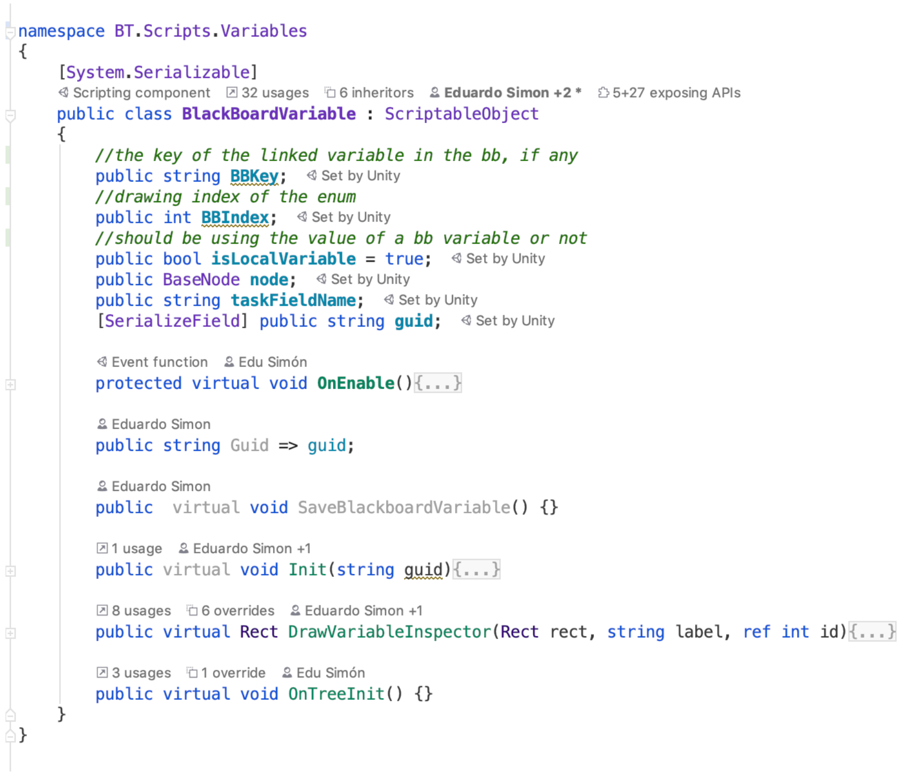
This method is very robust and versatile, however, it's not the fastest, but as we are just doing this once and its not at run time so we can take the risk.

So for saving and compiling my tree and pass the data from the node variables to the task fields I just need to use some reflection magic and iterate trough every variable in the node and set its corresponding task field (See Figure 4.16).

- **Blackboard System** The blackboard is an asset file associated with a tree graph. It is like a blank piece of paper for trees to store the data they want to make public. It's great for sharing information between agents. The implementation is very straightforward, the blackboard is just a container of keys. Every key has a name and a blackboard instance associated with it. We can retrieve, add or delete variables from the blackboard using the key name as the parameter.

- **Blackboard Editor** Developing the blackboard system I noticed that I needed some kind of system to edit a BlackBoard asset in the Editor and not just assign and retrieve variables at run time. I wanted to have a reordable list with the add and remove button. I researched and found that the UnityEditorInternal name space had just what I was looking for (See Figure 4.17). The editor was at that point was ready to create and use Blackboard Variables (See Figure 4.18).

- **Added support for linking inspector variables with blackboard variables** This feature was kind of a game changer for the tool, thanks to it, the user could



```
namespace BT.Scripts.Variables
{
    [System.Serializable]
    Scripting component  32 usages  6 inheritors  Eduardo Simon +2 *  5+27 exposing APIs
    public class BlackBoardVariable : ScriptableObject
    {
        //the key of the linked variable in the bb, if any
        public string BBKey;  Set by Unity
        //drawing index of the enum
        public int BBIndex;  Set by Unity
        //should be using the value of a bb variable or not
        public bool isLocalVariable = true;  Set by Unity
        public BaseNode node;  Set by Unity
        public string taskFieldName;  Set by Unity
        [SerializeField] public string guid;  Set by Unity

        Event function  Edu Simón
        protected virtual void OnEnable(){...}

        Eduardo Simon
        public string Guid => guid;

        Eduardo Simon
        public  virtual void SaveBlackboardVariable() {}

        1 usage  Eduardo Simon +1
        public virtual void Init(string guid){...}

        8 usages  6 overrides  Eduardo Simon +1
        public virtual Rect DrawVariableInspector(Rect rect, string label, ref int id){...}

        3 usages  1 override  Edu Simón
        public virtual void OnTreeInit() {}
    }
}
```

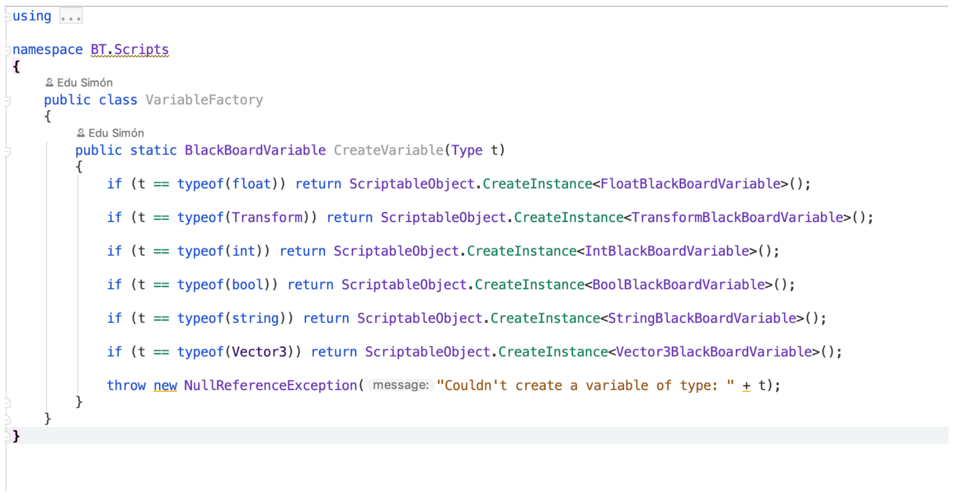
Figure 4.13: The common fields of every variable used in the editor.

connect variables from tasks with the blackboard for posting information or getting information. In addition, this was very useful for task that return some kind of data and need to be shared like a detect Player task. If the task detects the player, it can be posted to the Blackboard and known by every other agent in the scene. This system wasn't as trivial to implement as the blackboard itself, but thanks to my new acquired skills in the field of Reflection it wasn't too bad. The key aspects of this feature were:

– A base toggle drawer for every inspector variable that would activate the blackboard mode. When activated, a popup would appear instead of the value and letting you to select which blackboard variable you wanted to connect with. It has to be issued that only variables of the same type as the one selected would appear in the popup.

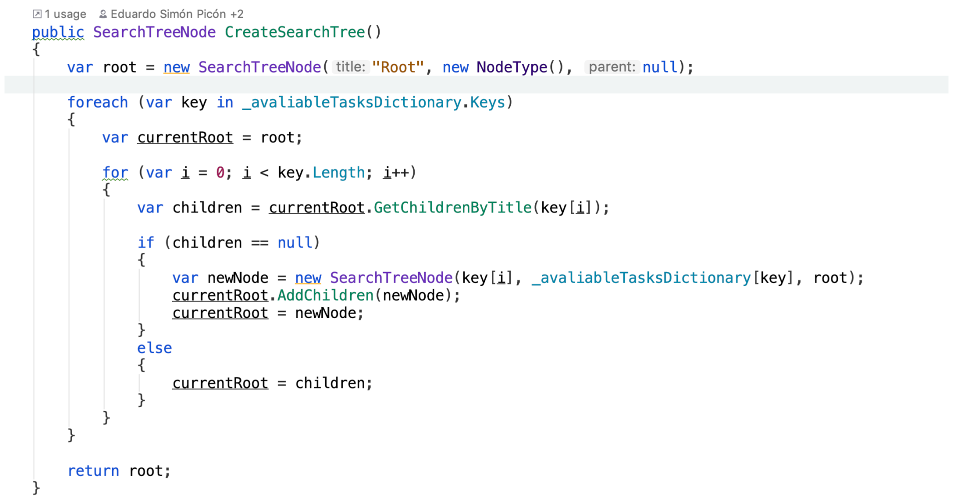
– Drawing an error when the user selects the blackboard mode but there is no blackboard selected.

– Dynamic creation of the variables popup when the variable is linked to the blackboard. Using reflection, inspects the current selected variable and store the index of the selected item in the popup. That index will be used to



```
using ...

namespace BT.Scripts
{
    Edu Simón
    public class VariableFactory
    {
        Edu Simón
        public static BlackBoardVariable CreateVariable(Type t)
        {
            if (t == typeof(float)) return ScriptableObject.CreateInstance<FloatBlackBoardVariable>();

            if (t == typeof(Transform)) return ScriptableObject.CreateInstance<TransformBlackBoardVariable>();

            if (t == typeof(int)) return ScriptableObject.CreateInstance<IntBlackBoardVariable>();

            if (t == typeof(bool)) return ScriptableObject.CreateInstance<BoolBlackBoardVariable>();

            if (t == typeof(string)) return ScriptableObject.CreateInstance<StringBlackBoardVariable>();

            if (t == typeof(Vector3)) return ScriptableObject.CreateInstance<Vector3BlackBoardVariable>();

            throw new NullReferenceException( message: "Couldn't create a variable of type: " + t);
        }
    }
}
```

Figure 4.14: The creation of the variable is delegated to the Variable Factory following the factory pattern.



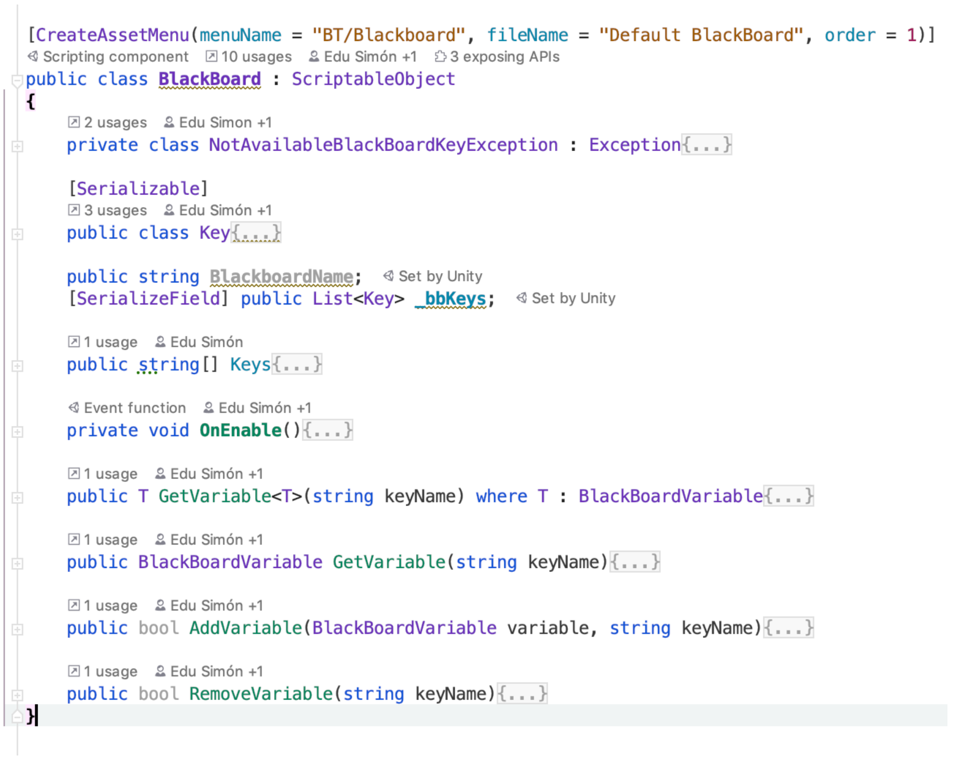
```
1 usage   Eduardo Simón Picón +2
public SearchTreeNode CreateSearchTree()
{
    var root = new SearchTreeNode( title: "Root", new NodeType(),  parent: null);

    foreach (var key in _avaliableTasksDictionary.Keys)
    {
        var currentRoot = root;

        for (var i = 0; i < key.Length; i++)
        {
            var children = currentRoot.GetChildrenByTitle(key[i]);

            if (children == null)
            {
                var newNode = new SearchTreeNode(key[i], _avaliableTasksDictionary[key], root);
                currentRoot.AddChildren(newNode);
                currentRoot = newNode;
            }
            else
            {
                currentRoot = children;
            }
        }
    }

    return root;
}
```

Figure 4.15: Set the value of the task field associated with this node variable.

retrieve the variable from the blackboard list.

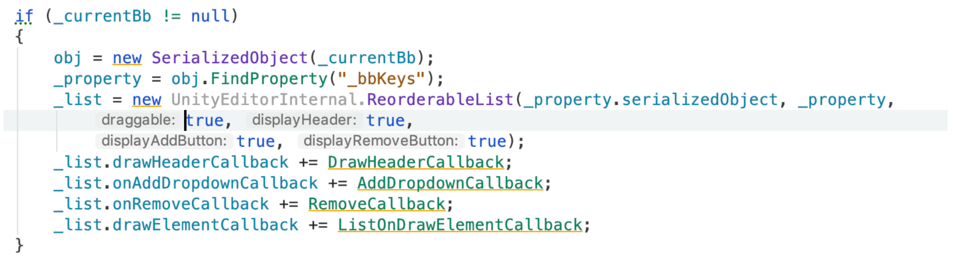
- **Added some visual debug capabilities to the tree.** One of the most important thing when building games is the power of the debug tools. Visual debugging is crucial to every tool and this one isn't an exception. I added the following debug features:

  – Visible world space canvas that outputs in the game view the current executing task and its status.



```
[CreateAssetMenu(menuName = "BT/Blackboard", fileName = "Default BlackBoard", order = 1)]
Scripting component   10 usages   Edu Simón +1   3 exposing APIs
public class BlackBoard : ScriptableObject
{
    2 usages   Edu Simon +1
    private class NotAvailableBlackBoardKeyException : Exception{...}

    [Serializable]
    3 usages   Edu Simón +1
    public class Key{...}

    public string BlackboardName;   Set by Unity
    [SerializeField] public List<Key> _bbKeys;   Set by Unity

    1 usage   Edu Simón
    public string[] Keys{...}

    Event function   Edu Simón +1
    private void OnEnable(){...}

    1 usage   Edu Simón +1
    public T GetVariable<T>(string keyName) where T : BlackBoardVariable{...}

    1 usage   Edu Simón +1
    public BlackBoardVariable GetVariable(string keyName){...}

    1 usage   Edu Simón +1
    public bool AddVariable(BlackBoardVariable variable, string keyName){...}

    1 usage   Edu Simón +1
    public bool RemoveVariable(string keyName){...}
}
```

Figure 4.16: The blackboard its just a list of keys, being each key a string and a blackboard variable instance.

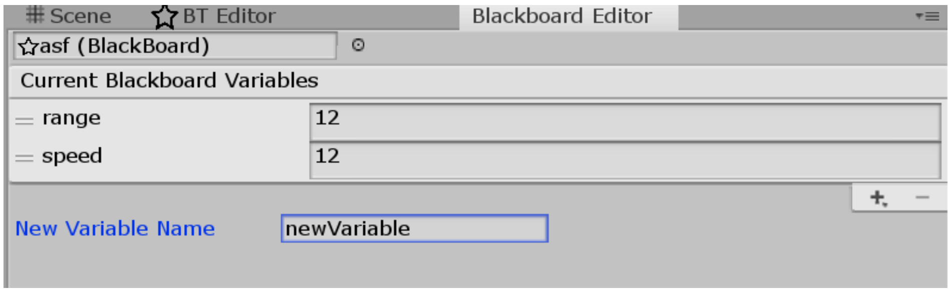


```
if (_currentBb != null)
{
    obj = new SerializedObject(_currentBb);
    _property = obj.FindProperty("_bbKeys");
    _list = new UnityEditorInternal.ReorderableList(_property.serializedObject, _property,
        draggable: true,  displayHeader: true,
        displayAddButton: true,  displayRemoveButton: true);
    _list.drawHeaderCallback += DrawHeaderCallback;
    _list.onAddDropdownCallback += AddDropdownCallback;
    _list.onRemoveCallback += RemoveCallback;
    _list.drawElementCallback += ListOnDrawElementCallback;
}
```

Figure 4.17: The reordable list has a nice wrapper with delegates. You just subscribe a method for every delegate that you want to be overridden.

– Created a class BTDebugGUI which draws a window with debug info from the runtime system. Currently it just outputs the current tree being executed and the gameobjects attached to it, but in the future I would like to output performance stats and run-time info.

– Implemented OnDrawGizmos as a base method for every task. OnDrawGizmos is a callback that offers Unity to every MonoBehaviour object in the



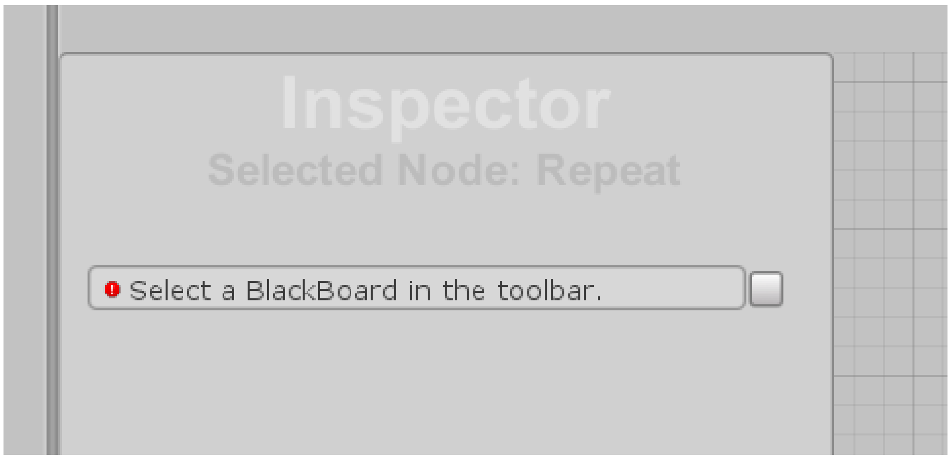
asf (BlackBoard)

Current Blackboard Variables

range	12
speed	12

New Variable Name newVariable

Figure 4.18: The look of the blackboard editor.



Select a BlackBoard in the toolbar.

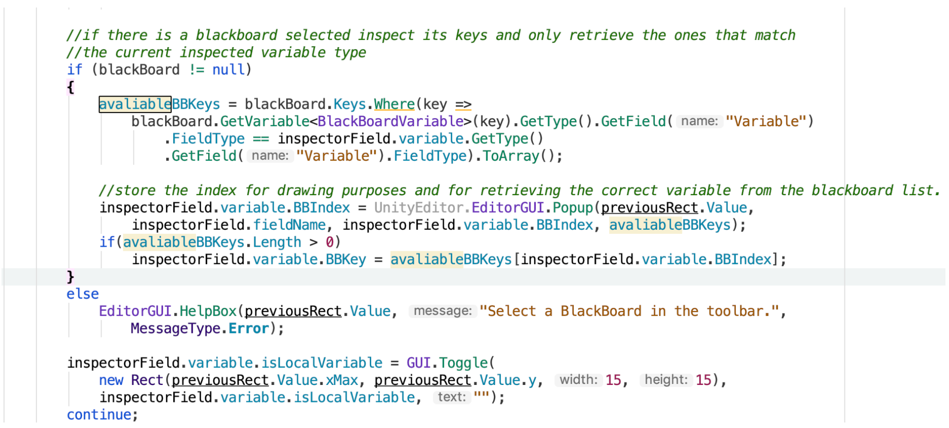
Figure 4.19: The error message drawn when there is no blackboard associated with the current graph.

scene. It allows to draw basic shapes like spheres and cubes for debugging purposes. The problem here is that the Task class inherits from ScriptableObject, not from MonoBehaviour. As a solution, I created a virtual method in the Task class called OnDrawGizmos. In the gameobject that runs the tree, whenever the tree is not null, I recursively traverse the tree calling the OnDrawGizmos of every Task. To keep the performance in production builds I decided to do it only when the BTManager has the debug flag activated.

– Changing the color of every node depending on its status: Red for failed, Yellow for running and green for suceeding.

See Figure 4.22 for a picture of the different debugging tools in action.

- **Improvements on the BT Controller and the BT Manager** Right after finishing the second sprint and experimenting a little bit with the tool I realised that there wasn't a lot of control in the execution scheduling side. I just had created a controller to run bts and a basic manager that gathered them all and



```
//if there is a blackboard selected inspect its keys and only retrieve the ones that match
//the current inspected variable type
if (blackBoard != null)
{
    avaliableBBKeys = blackBoard.Keys.Where(key =>
        blackBoard.GetVariable<BlackBoardVariable>(key).GetType().GetField( name: "Variable")
            .FieldType == inspectorField.variable.GetType()
            .GetField( name: "Variable").FieldType).ToArray();

    //store the index for drawing purposes and for retrieving the correct variable from the blackboard list.
    inspectorField.variable.BBIndex = UnityEditor.EditorGUI.Popup(previousRect.Value,
        inspectorField.fieldName, inspectorField.variable.BBIndex, avaliableBBKeys);
    if(avaliableBBKeys.Length > 0)
        inspectorField.variable.BBKey = avaliableBBKeys[inspectorField.variable.BBIndex];
}
else
    EditorGUI.HelpBox(previousRect.Value,  message: "Select a BlackBoard in the toolbar.",
        MessageType.Error);

inspectorField.variable.isLocalVariable = GUI.Toggle(
    new Rect(previousRect.Value.xMax, previousRect.Value.y,  width: 15,  height: 15),
    inspectorField.variable.isLocalVariable,  text: "");
continue;
```

Figure 4.20: The reflection code that gets the available keys of the same type as the inspected variable.

ticked them all. That's why I implemented a ticking system in which you could select the way of ticking each tree:

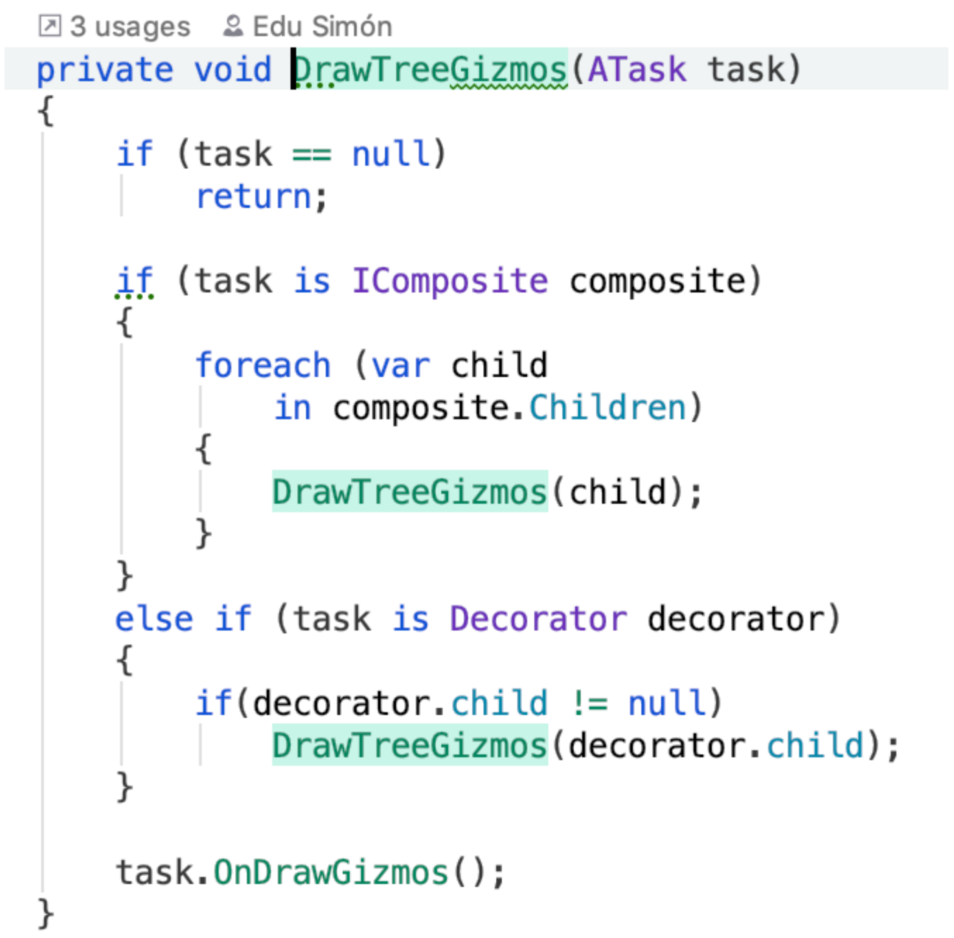
- UnityTick: The tree will be ticked every frame in the unity callback selected by each controller. It can be Update, FixedUpdate or LateUpdate.
- Tick interval. You can specify a time interval to tick the tree, making it more performant.
- Manual. You can specify when to tick the tree manually.

Moreover, I implemented a debug flag in the BTManager for debugging purposes. Along with that, the BTController has a public field in which you can select the highest log level, in case you need to know exactly whats going on, you can activate the verbose level.

- **Added more complex GUI Tests** Unit testing is a crucial part of this library. The are tests written for the core, and in this sprint I upgraded and added some GUi tests. These test are responsible for creating and closing the editor, adding nodes, connections and saving everything programmatically.

  Unity allows you to do unit tests using the NUnit library and Unity's Test Runner window (See Figure 4.23).

For creating a Test you just need to use the NUnit attributes (See Figure 4.24) and some methods from the Assert class. The idea is to code a behavior of the system and then assert that everything is working as expected. You just need to create a class and create methods inside of it. A method marked with the Test attribute will be read by the test runner and executed. You can setup special methods for setting up the environment or for destroying it. In my case I do so to open and close the editor. You can also use



```
private void DrawTreeGizmos(ATask task)
{
    if (task == null)
        return;

    if (task is IComposite composite)
    {
        foreach (var child
            in composite.Children)
        {
            DrawTreeGizmos(child);
        }
    }
    else if (task is Decorator decorator)
    {
        if(decorator.child != null)
            DrawTreeGizmos(decorator.child);
    }

    task.OnDrawGizmos();
}
```

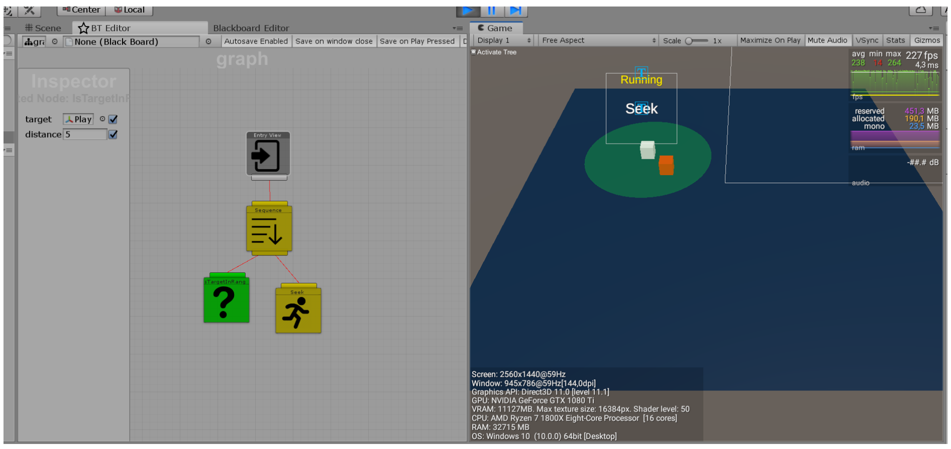
Figure 4.21: Recursively calling the onDrawGizmos method. Only Composites and decorators can have children, so I only call the method again when the task is a composite or a Decorator

the UseCase attribute to specify a specific UseCase of the test, for trying out different values passed to a test.

4.2 Results

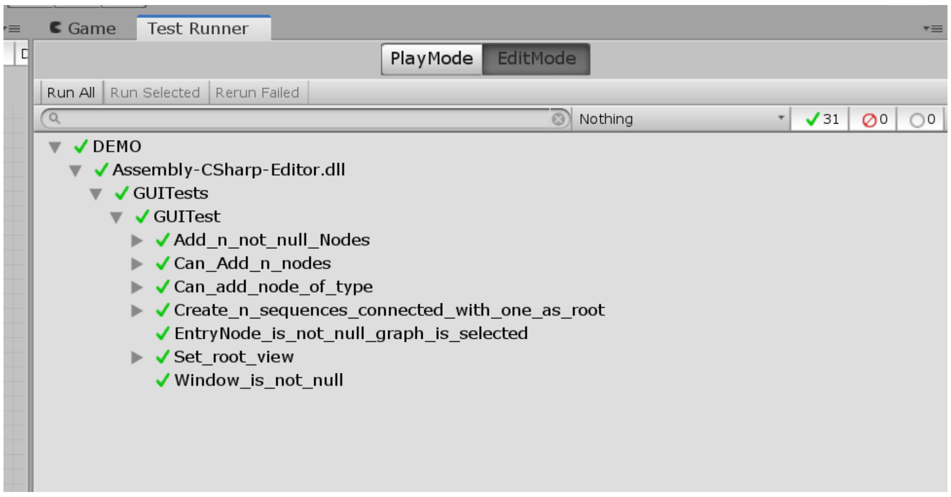
The results of all the sprint can be seen in the different videos [19] for Sprint 1, [20] and [21]. To view the demo and an overview of it check [**?**].

Following the objectives stated in the Introduction, it's clear that each one of them has been accomplished. Lets review them and see how.



```
Screen: 2560x1440@59Hz
Window: 945x786@59Hz[144,0dpi]
Graphics API: Direct3D 11.0 [level 11.1]
GPU: NVIDIA GeForce GTX 1080 Ti
VRAM: 11127MB. Max texture size: 16384px. Shader level: 50
CPU: AMD Ryzen 7 1800X Eight-Core Processor  [16 cores]
RAM: 32715 MB
OS: Windows 10  (10.0.0) 64bit [Desktop]
```

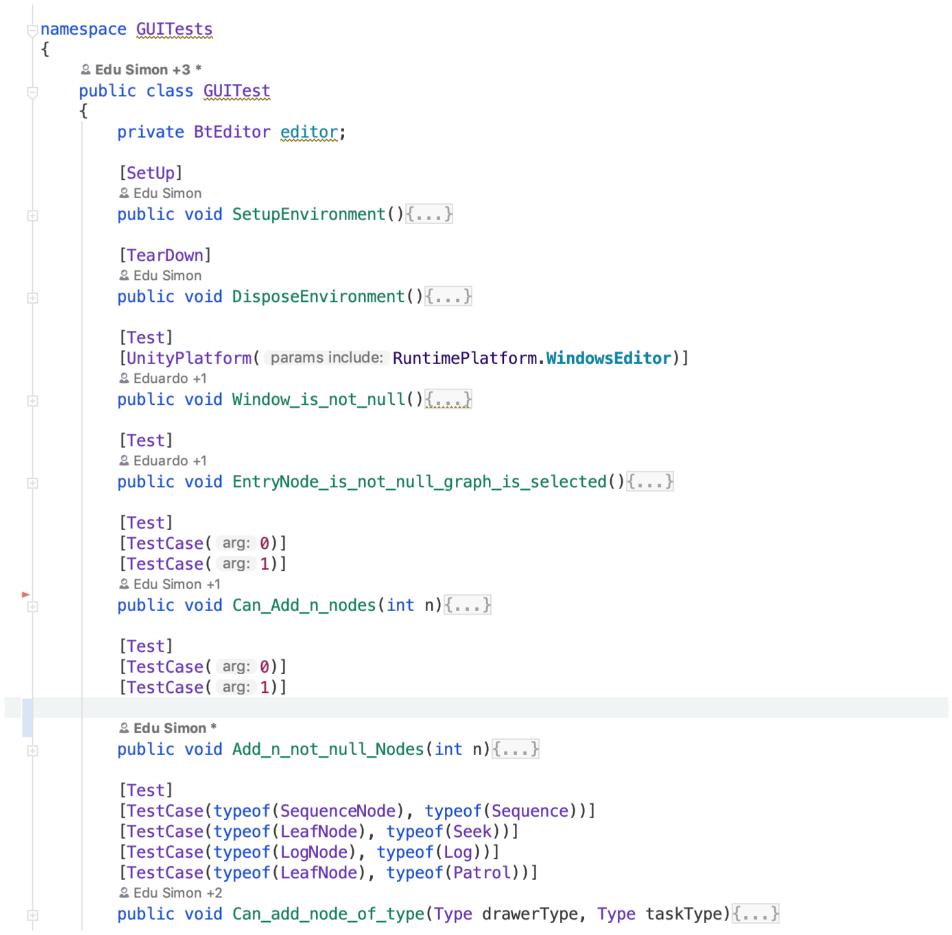
Figure 4.22: The debug system as a whole, is trying to give as much info to the user as possible.



```
▼ ✓ DEMO
  ▼ ✓ Assembly-CSharp-Editor.dll
    ▼ ✓ GUITests
      ▼ ✓ GUITest
        ▶ ✓ Add_n_not_null_Nodes
        ▶ ✓ Can_Add_n_nodes
        ▶ ✓ Can_add_node_of_type
        ▶ ✓ Create_n_sequences_connected_with_one_as_root
          ✓ EntryNode_is_not_null_graph_is_selected
        ▶ ✓ Set_root_view
          ✓ Window_is_not_null
```

Figure 4.23: The Test runner with all tests passed.

- **The first objective was to implement the BT technique in unity and drive a NPC (Non-playable Character) decision making process based on it**. This has been accomplished and its has been proven in the 2 available demos. Both demos have NPC driven by a BT and communicating, in the case of the second demos, with each other through a blackboard. This objective was accomplished in the second sprint, however just as a very early version.
- The second one was to **implement a plugin which has a very unity-friendly workflow**. This one can be marked as achieved by the following reasons.
  - All the work is done in an asset derived from the Scriptable Object class,



```
namespace GUITests
{
    Edu Simon +3 *
    public class GUITest
    {
        private BtEditor editor;

        [SetUp]
        Edu Simon
        public void SetupEnvironment(){...}

        [TearDown]
        Edu Simon
        public void DisposeEnvironment(){...}

        [Test]
        [UnityPlatform( params include: RuntimePlatform.WindowsEditor)]
        Eduardo +1
        public void Window_is_not_null(){...}

        [Test]
        Eduardo +1
        public void EntryNode_is_not_null_graph_is_selected(){...}

        [Test]
        [TestCase( arg: 0)]
        [TestCase( arg: 1)]
        Edu Simon +1
        public void Can_Add_n_nodes(int n){...}

        [Test]
        [TestCase( arg: 0)]
        [TestCase( arg: 1)]

        Edu Simon *
        public void Add_n_not_null_Nodes(int n){...}

        [Test]
        [TestCase(typeof(SequenceNode), typeof(Sequence))]
        [TestCase(typeof(LeafNode), typeof(Seek))]
        [TestCase(typeof(LogNode), typeof(Log))]
        [TestCase(typeof(LeafNode), typeof(Patrol))]
        Edu Simon +2
        public void Can_add_node_of_type(Type drawerType, Type taskType){...}
```

Figure 4.24: The attributes I used for testing.

very similar to the workflow used in the Unity Animator (See Figure 4.25) with the Unity Animator Controller asset. With a quick look, I can proudly say, that using my tools feels very close to what using a native window feels. Obviously, there are major and minor improvements to be made, however, I think that it's headed in the right direction.

– The life cycle of the different objects in the tree has been designed to mimic as much as possible to the ones native to unity. Awake, Start, Update and Terminate.

– The inspector has been designed to work as closely as possible to the Unity inspector, with drag and drop support, change focus on keyboard tab and dynamic creation of the inspector. In the near future custom drawers will be a thing, and the user will be able to modify the inspector fields.

– The window and its GUI elements have been coded using the GUI and Ed-

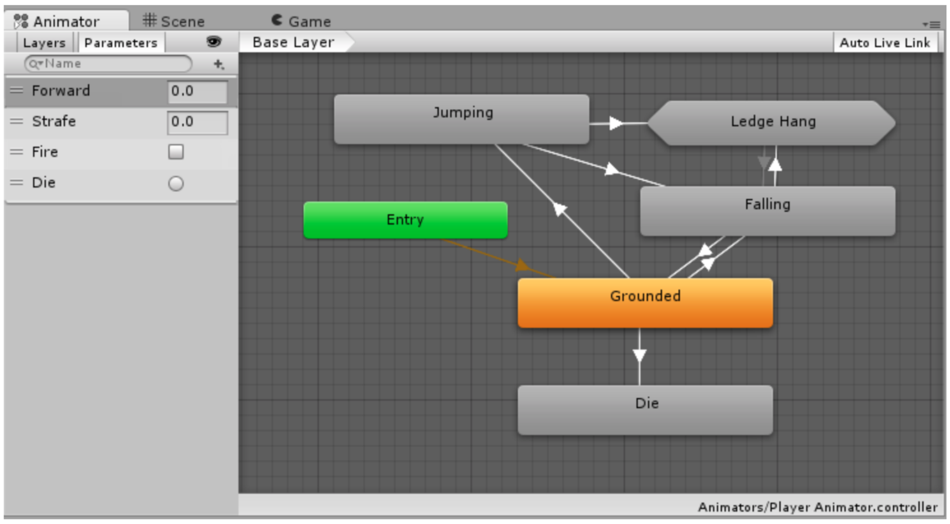


Figure 4.25: The UI of the Animator Window in Unity, my tool has been influenced by the Animator design principles.

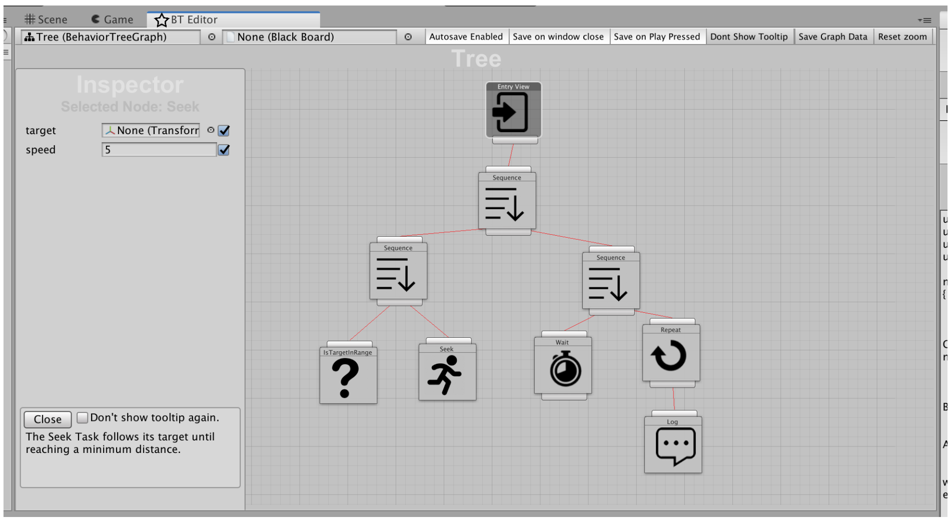


Figure 4.26: The look of my editor with a small tree. There is an inspector, a work area and a toolbar.

itorGUI name spaces and with the default styles available, improving the visual cohesion between the tool and the native tools in unity.

- **Creation of an easy to use API for extending the library's functionalities**. The design process has been long and costly. There were many possible approaches, and none of them seemed to satisfy this objective. However, thanks to reflection and some custom UnityEditor code, extending is super easy [4].

Chapter 5

Conclusions and future work

Contents

5.1 Conclusions

First of all, I must admit that I have struggled a lot during this project. There were weeks in which nothing was working, moments in which I thought I was the worst codebuilder alive. However, there were others in which everything worked out in the first try and flawlessly. I guess it's the charm of programming, living a constant struggle. Leaving internal reflections aside, the project has suprised me. I have liked it more than I thought I would. The DFP was like a testbench for the library, to see if I could build a foundation and try to continue from that.

Secondly, as a junior game developer, I have faced countless challengues and cornerstones. I have learnt a lot about Unity, about C# and about what it means to use a huge game engine like Unity. Unity is a huge black box, you write code and you hit run expecting a certain result. Most of the times, Unity behaves consistently, however, there are times that it doesn't. Those moments you may be thinking that the fault is yours and you might rewrite your code even it was correct in the first place. The only interface we have to this HUGE black box is the public API that they let you use and the avaliable documentation, blog post and forum threads in the Internet.

For common tasks like programming a character controller, you will have it easy and there will be a plenty of information online. However, when you try to deep dive into the engine or create a very niche tool/mechanic, the problem arises. You don't direct control over the engine and you have to rely on the native tools.

You could of course code everything from the ground up, but I want to create a tool for game developers for making their lives easier, not creating an engine from the ground up. I want my tool to live in the UnityEditor ecosystem and enforce the unity way of doing things, so I have no other option than sticking to it.

To conclude, time is limited. We can't reinvent the wheel every time, that's why we have to rely on pre-made tools like Unity or Unreal to create games, however there is something we can make about the black box problem. As developers we can opt to open source our projects and try to get the income from other sources like private and custom extension to the software or donations. If we enforce the use and creation of open source code, we can create better and bigger system on top of each other, because you have access to the whole internals of the lower levels and, in consequence, you can plan and design based on them.

Furthermore, In the DFP the foundations of the library have been built, along with a demo of how the plugin works, in spite of the great and successful work there are plans to continue developing the project. I have hired another developer, Ricardo Sanz [9], to arrive to the release date. The full 1.0 release will be on December of this year. There are many features that couldn't be added while planning the DFP. Some of them are:

- Support for event driven behavior trees
- Support for hint executed behavior trees
- New runtime that offers better performance with less cache misses.
- New GUI design with improved UX.
- Clear and concise documentation about the library inner workings and its API.
- An improved system, visually and programatically for creating tasks.
- Programming and testing of a huge number of tasks.
- Integrations with third party plugins such as Cinemachine, TextMeshPro, PlayMaker, Amplify Shader etc...
- Creation of more demos for covering different use cases.

If I had to start from the beginning, I would think twice if using the unity built-in serialization is a good idea. I could have used JSON and thus having more control of the serialization and deserialization process. This also implies a better compatibility in future and legacy versions. I would probably opted for the new UIELements system in the Unity Engine, which is very similar to how the web works, with css and view containers. I would have also made more frequent refactors and improve the planning phase. I have discovered some workflows using git like git flow, which would turn very handy in the beginning.

To conclude, I must say that I am really happy with the product and that it has been a pleasure to develop it. All the code will be hosted at Github publicly, and it will be free to use, modify and license. You can check out the code at [13].

If you have read until here, thank you very much for your patience and attention, everything that I have done here had the best of my intentions.

Bibliography

[1] Behavior bricks in the asset store. https://assetstore.unity.com/packages/tools/visual-scripting/behavior-bricks-74816.

[2] Behavior designer: Behavior trees for every one. https://assetstore.unity.com/packages/tools/visual-scripting/behavior-designer-behavior-trees-for-everyone-15277.

[3] Decorator design pattern. https://en.wikipedia.org/wiki/Decorator_pattern.

[4] Demo of a task creation. https://opacity.io/share#handle=50d02f0bae701c7d74404b56ab2f3131f4856b8d1242c211ebeaa8bf3b6d11baf0be3acf242103290a25f6da5c9be41

[5] Gameaipro: Behavior trees starter kit. http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf.

[6] Github website. http://www.github.com.

[7] Google chrome extension for coloring trello cards. https://chrome.google.com/webstore/detail/card-colors-for-trello/nodlpencjjlohojddhflnahnfpfanbjm.

[8] The mono project is a framework for creating cross-platform .net applications. https://www.mono-project.com.

[9] My new partner twitter profile. https://twitter.com/ricardosanz97.

[10] Nodecanvas in the asset store. https://assetstore.unity.com/packages/tools/visual-scripting/nodecanvas-14914.

[11] Panda btfree in the asset store. https://assetstore.unity.com/packages/tools/ai/panda-bt-free-33057.

[12] Parallels info. https://books.google.es/books?id=jDb6AwAAQBAJ&lpg=PA82&ots=YQIqsBSSm3&dq=example%20use%20case%20of%20parallels%20in%20bt%20games&hl=es&pg=PA83#v=onepage&q=example%20use%20case%20of%20parallels%20in%20bt%20games&f=false.

[13] The public repository of the tool: Roots. https://github.com/EduardoSimon/Roots.

[14] Retrieve reference in custom editor window when entering play mode. https://www.pixelcrushers.com/maintain-references-entering-playmode-in-custom-unity-editor-windows/.

[15] Solution for scriptable objects when reloading editor. http://antondoe.blogspot.com/2016/08/serialization-with-polymorphism-and.html.

[16] Trello website. https://trello.com/.

[17] Unity serialization. https://blogs.unity3d.com/es/2014/06/24/serialization-in-unity/.

[18] Unity serialization best practices. https://forum.unity.com/threads/serialization-best-practices-megapost.155352/.

[19] Video showing sprint1 progress. https://youtu.be/_yNAJdlq_44.

[20] Video showing sprint2 progress. https://youtu.be/kdLa-E00OC4.

[21] Video showing sprint3 progress. https://youtu.be/yfxN-x-8lFQ.

[22] Alex J. Champandard. 10 reasons the age of finite state machines is over. http://aigamedev.com/open/article/fsm-age-is-over/. December 28, 2007.