UNIVERSITY OF CRETE
SCHOOL OF SCIENCES AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

Autonomia: A knowledge-based framework for realistic agent behaviours in dynamic video game environments

Zacharias Pervolarakis

Thesis submitted in partial fulfillment of the requirements for[1] the
Masters' of Science degree in Computer Science and Engineering

Thesis Advisor: Constantine Stephanidis

[1]This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.
The work has been supported by the Foundation for Research and Technology Institute of Computer Science (ICS).

UNIVERSITY OF CRETE
SCHOOL OF SCIENCES AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

Autonomia: A knowledge-based framework for realistic agent behaviours in dynamic video game environments

by **Zacharias Pervolarakis**

In partial fulfillment of the requirements for the
Master of Science degree in Computer Science

APPROVED BY:

[illegible]

Author: Zacharias Pervolarakis, University of Crete

[illegible]

Thesis Supervisor: Constantine Stephanidis, Professor, University of Crete

[illegible]

Committee Member: Kostas Magoutis, Associate Professor, University of Crete

[illegible]

Committee Member: Xenophon Zabulis, Research Director, FORTH

[illegible]

Director of Graduate Studies: Polyvios Pratikakis, Professor, University of Crete

Heraklion, November 2023

This thesis is dedicated to my family
to my father Lefteris who taught me practicality and wisdom,
my mother Katerina who taught me how to love and dream,
and my brother Manos who taught me how to understand and learn.

Abstract

Video games are a popular form of entertainment that offer interactive and immersive experiences to the players. A key element of these experiences is the presence of non-player characters (NPCs), which are autonomous agents that populate the game world and interact with the player and the environment. NPCs can enhance the realism and diversity of the game scenarios by exhibiting human-like behaviours that are consistent, adaptive and believable. However, creating such behaviours is a complex and challenging task that requires a combination of artificial intelligence (AI) techniques and game design principles. Current methods and frameworks for NPC decision-making often rely on predefined scripts or rules that limit the NPC's capability to adapt to dynamic situations. Moreover, NPCs usually lack autonomy, as they are unable to pursue their own goals, as well as to interact with other NPCs or the player. Therefore, there is a need for novel approaches that can improve the credibility and adaptability of NPC behaviours in video games.

This work introduces Autonomia, an innovative knowledge-based framework for realistic agent behaviours in dynamic video game environments. Autonomia is deeply rooted in the Theory of Mind (ToM), leveraging a knowledge graph to depict the world's state, with each NPC possessing a replica of this world state in its "memory". This "memory" is designed to support higher orders of ToM while constantly evolving as the NPC perceives the world around it and interprets events. Autonomia uses a modular system to define the functionality and behaviour of different types of nodes in the graph, such as physical objects, animals or people. The framework as structured, allows NPCs to dynamically react to changes in the environment purely based on its ability to perceive and hold memory. In this context, Autonomia introduces a new way to model behaviours and goals, enabling them to be treated as knowledge that can be communed, discovered or even forgotten just like any other part of the NPC's "memory". Basing everything on their acquired knowledge, NPCs utilize a Goal-Oriented Action Planning (GOAP) algorithm to come up with plans in any dynamic environment.

Lastly, an implementation of Autonomia is provided for the Unity game engine, including the "Prometheus Tavern" case study, on which a two-part expert-based evaluation was conducted. The first part confirmed that the provided features and the architecture of the Autonomia framework deliver solutions that can improve the credibility of NPC behaviours, whereas the second showed that the agents of the system have the capability to adapt to their environment and behaviour in a realistic manner.

Keywords: Game AI, Autonomous Agents, NPC, NPC Behaviours

Περίληψη

Τα βιντεοπαιχνίδια είναι μια δημοφιλής μορφή ψυχαγωγίας και μπορούν να προσφέρουν διαδραστικές και καθηλωτικές εμπειρίες στους παίκτες. Βασικό στοιχείο αυτών των εμπειριών είναι η παρουσία χαρακτήρων τύπου Non-Playing Character (NPC) δηλαδή πρακτόρων που δεν ελέγχονται από τον παίκτη ή τους παίκτες του παιχνιδιού. Οι πράκτορες NPC είναι αυτόνομοι χαρακτήρες που κατοικούν στον κόσμο του παιχνιδιού και αλληλεπιδρούν με τον παίκτη και το περιβάλλον. Οι πράκτορες αυτοί μπορούν να ενισχύσουν τον ρεαλισμό και την ποικιλομορφία των σεναρίων του παιχνιδιού, παρουσιάζοντας συμπεριφορές που θυμίζουν ανθρώπινες, είναι συνεπείς, αληθοφανείς και προσαρμόζονται ανάλογα με το δυναμικό τους περιβάλλον. Ωστόσο, η δημιουργία τέτοιων συμπεριφορών είναι ένα πολύπλοκο και δύσκολο έργο που απαιτεί μεθοδολογία, καλό σχεδιασμό και συνδυασμό πολλαπλών τεχνικών τεχνητής νοημοσύνης (AI). Οι τρέχουσες μέθοδοι και προγραμματιστικά πλαίσια για την υλοποίηση της ικανότητας των πρακτόρων NPC να παίρνουν αποφάσεις, συχνά βασίζονται σε προκαθορισμένα σενάρια ή κανόνες που περιορίζουν την προσαρμοστικότητα τους σε δυναμικά μεταβαλλόμενες καταστάσεις. Επιπλέον, συχνά οι πράκτορες NPC χαρακτηρίζονται από έλλειψη αυτονομίας, καθώς δεν έχουν την ικανότητα να επιδιώξουν τους δικούς τους στόχους ή ακόμα και να αλληλεπιδράσουν με άλλους πράκτορες NPC ή με τον ίδιο τον παίκτη. Ως εκ τούτου, υπάρχει ανάγκη για καινοτόμες λύσεις που να μπορούν να βελτιώσουν την αξιοπιστία και την ικανότητα των συμπεριφορών των πρακτόρων NPC στα βιντεοπαιχνίδια.

Η παρούσα μεταπτυχιακή εργασία εισάγει το σύστημα Autonomia, ένα καινοτόμο προγραμματιστικό πλαίσιο βασισμένο στην γνώση, σχεδιασμένο να προσδίδει ρεαλιστικές συμπεριφορές πρακτόρων NPC σε δυναμικά περιβάλλοντα βιντεοπαιχνιδιων. Το σύστημα Autonomia θεμελιώνεται στη Θεωρία του Νου (ΘτΝ) και χρησιμοποιεί γραφήματα γνώσης (knowledge graphs) για την απεικόνιση της κατάστασης του κόσμου. Ο κάθε πράκτορας NPC διαθέτει ένα αντίγραφο αυτής της κατάστασης στη “μνήμη” του. Συγκεκριμένα, η σχεδίαση της “μνήμης” επιτρέπει τόσο την συνεχή εξέλιξη της, όσο και την υποστήριξη υψηλότερων επιπέδων ΘτΝ καθώς ο πράκτορας αντιλαμβάνεται το περιβάλλον και σχηματίζει προσωπική εικόνα για τον κόσμο. Ένα σύστημα δομοστοιχείων (modular system) χρησιμοποιείται για να ορίζεται η λειτουργικότητα και η συμπεριφορά διαφορετικών τύπων κόμβων του γράφου, όπως αντικείμενα, ζώα ή άνθρωποι. Η σχεδίαση του συστήματος Autonomia επιτρέπει στους πράκτορες NPC να αντιδρούν δυναμικά στις αλλαγές του περιβάλλοντος με βάση την ικανότητά τους να αντιλαμβάνονται και να ερμηνεύουν γεγονότα στη “μνήμη” τους. Επίσης, εισάγουμε έναν νέο τρόπο μοντελοποίησης των συμπεριφορών και των στόχων, ο οποίος επιτρέπει την χρήση τους ως γνώση η οποία μπορεί να μεταφερθεί μέσω διαλόγου, να ανακαλυφθεί ή ακόμα και να ξεχαστεί όπως κάθε άλλο κομμάτι "μνήμης" του πράκτορα NPC. Βασίζοντας τα πάντα στην επίκτητη γνώση του, ο πράκτορας NPC χρησιμοποιεί έναν Goal-Oriented Action Planning (GOAP) αλγόριθμο για να μπορεί να "σκαρφιστεί" αλυσίδες συμπεριφορών σε κάθε δυναμικό περιβάλλον.

Τέλος, παρέχεται μια υλοποίηση του συστήματος Autonomia στην μηχανή παιχνιδιών Unity η οποία περιέχει την μελέτη περίπτωσης "Ταβέρνα του Προμηθέα" με την οποία πραγματοποιήθηκε μια αξιολόγηση με εμπειρογνώμονες την οποία χωρίζουμε σε δύο μέρη. Τα αποτελέσματα του πρώτου μέρους επιβεβαίωσαν ότι η προσφερόμενη λειτουργικότητα και η αρχιτεκτονική του προγραμματιστικού πλαισίου Autonomia προσφέρουν λύσεις για την βελτίωση της αξιοπιστίας και της ικανότητας των συμπεριφορών των πρακτόρων, ενώ τα αποτελέσματα του δεύτερου έδειξαν ότι οι πράκτορες του συστήματος έχουν την ικανότητα να προσαρμόζονται στο περιβάλλον τους και να συμπεριφέρονται με ρεαλιστικό τρόπο.

Keywords: Game AI, Autonomous Agents, NPC, NPC Behaviours

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

In this chapter, we delve into the motivation and background behind the development of the Autonomia Framework. We explore the challenges associated with creating believable [1]–[3] and intelligent non-player characters (NPCs) in the context of video games and interactive simulations. Additionally, we provide an overview of the objectives and structure of this thesis, setting the stage for a comprehensive examination of Autonomia's architecture, implementation, and contributions to the field of artificial intelligence in gaming.

1.1 Motivation

Video games, as a widely enjoyed form of entertainment and art, have the potential to deliver captivating experiences [4], [5]. A central part within these games is the interaction with non-player characters (NPCs)—computer-controlled entities that assume roles of allies, adversaries, or neutrals [2] and NPCs hold the ability to craft those immersive and lifelike scenarios by simulating emotions, personalities, motivations, and social dynamics akin to human beings. Yet, crafting truly convincing NPCs remains a formidable challenge, necessitating a blend of technical skills, artistic ingenuity, and psychological understanding. Unfortunately, the current state of most NPCs falls short of authenticity, often adhering to scripted, predictable, or inconsistent behaviours that shatter the illusion of reality and reduce or even diminish players' enjoyment [6]–[8]. The lack of NPCs' believability often becomes evident when they encounter intricate situations that haven't been anticipated by the designer or programmer. For the most part, NPCs depend on rigid, pre-defined rules or scripts dictating their responses to specific in-game situations. Unfortunately, these rules often lack flexibility, sophistication, and fail to capture the dynamic nature of the game world and player actions. Furthermore, most NPCs lack a unified model encompassing their perceptions, memories, goals, and plans. This absence impedes their capacity to reason over actions, predict outcomes, or collaborate effectively with other NPCs and players. Consequently, NPCs often manifest as superficial, artificial entities, lacking the depth of human-like intelligence or agency [9].

To illustrate this problem further, we can examine some examples of games that have attempted to create believable NPCs, and discuss their strengths and limitations. For instance, The Sims [10] is a popular life simulation game that allows players to create and control virtual people with various personality traits. The game uses a complex system of needs, motives, skills, and relationships to determine the behaviour and emotions of the NPCs. However, some critics [11], [12] have argued that the NPCs in The Sims are still too simplistic and deterministic, lacking the ability to form meaningful bonds or exhibit moral agency. Similarly, Mass Effect [13] is a sci-fi role-playing game that features a rich cast of NPC companions with distinct backgrounds, personalities, and moral alignments. The game allows players to interact with these NPCs through dialogue choices and influence their loyalty and romance. However, some reviewers [14], [15] have noted that the NPCs in Mass Effect are still constrained by predefined scripts and branching paths, limiting their autonomy and responsiveness to player actions. Another example is Detroit: Become Human [16], a narrative-driven game that explores the themes of artificial intelligence and androids. The game features multiple playable characters that can make choices and face moral dilemmas that affect the story's outcome. The game also uses advanced facial animation and voice acting to convey the emotions and expressions of the NPCs. However, again some critics [17], [18] have pointed out that the NPCs in Detroit: Become Human [16] are still influenced by clichés and stereotypes, lacking the subtlety and complexity of human psychology.

These examples underscore the intricate nature of crafting convincing NPCs in the realm of game development, revealing persistent challenges and unmet aspirations. It's worth noting that these examples originate from the game development industry, where talented programmers have long pursued the elusive goal of achieving lifelike game AI. Yet, even with their dedicated efforts, the industry has not fully realized this ambition.

In this thesis, we aim to contribute to this field by proposing a novel approach for creating believable NPCs based on a realistic NPC memory representation grounded in acquired knowledge, an extendable modular architecture, a novel way of defining behaviours and lastly, encapsulate everything in an open-source framework which could serve as common ground for game AI research.

1.2 Thesis Structure

The remainder of this master's thesis is divided into seven chapters, as indicated in the table of contents. Below, a summary of each chapter is provided:

- Chapter 2: This chapter provides a comprehensive literature review of previous works that have greatly contributed to the field of game AI. It traces the evolution of AI techniques, from early finite state machines to contemporary state-of-the-art behavior modeling methods.

- Chapter 3: The third chapter delves into the theoretical foundations of Autonomia, explaining the architectural decisions that underpin this thesis' work.

- Chapter 4: In this section, the implementation of the theoretical framework outlined in Chapter 3 is described. It covers the essential components and design patterns employed in Autonomia.

- Chapter 5: This chapter focuses on the integration of Autonomia into the Unity game engine. It also elucidates the various designer tools developed in this thesis and presents the Prometheus Tavern case study, offering insights into its exploration.

- Chapter 6: The sixth chapter details the heuristic evaluation, including its methodology and its findings.

- Chapter 7: This final chapter provides a summary of the work undertaken in this thesis and offers a discussion of potential future directions and aspirations for Autonomia.

Chapter 2

2 Background Theory and Related Work

In this chapter, we explored the existing body of research and development in the fields of game AI, human-computer interaction (HCI), and computer science. We conducted a comprehensive review of relevant literature to inform the design and development of Autonomia. This exploration of related work was a vital process in shaping the framework's features and capabilities. Our related work process involved a systematic approach to gathering and synthesizing information from various sources. We employed a combination of academic journals, conference papers, books, and online resources to ensure a comprehensive review of the subject matter.

First, we delve into the foundational concepts of our research field, which form the basis of our Background Theory. Subsequently, we examine existing works that align with or partially address our objectives concerning Autonomia. This structured approach offers readers a coherent journey, starting with the essential theoretical framework and culminating in a comprehensive understanding of Autonomia's relevance in the broader research context.

2.1 Background Theory

In this section, we delve into the foundational principles and theories that underpin the field of research relevant to Autonomia. We explore the core concepts and pillars that form the basis of intelligent NPC behaviour and game AI. This provides readers with a solid understanding of the theoretical framework upon which Autonomia is built.

2.1.1 The importance of Video Games

In the early 2000s, there was a prevailing belief that video games had a detrimental impact on the mental health of young people. The media often launched verbal attacks on gaming culture, though rarely with concrete evidence. While some earlier studies did suggest potential negative effects of arising from gaming, this has created a somewhat unjust stigma around video games. Even today, accusations of video games being a harmful habit persist.

However, it's important to note that more recent research has uncovered a multitude of benefits associated with gaming. These studies have shed light on both the positive aspects of gaming and the valuable insights we can gain regarding the learning process through video games and interactive experiences. This evolving body of research challenges the notion that gaming is inherently harmful and highlights the potential for constructive and educational outcomes from video game engagement.

Isabela et al. [19] conducted an overview of the benefits of video gaming. They summarize that games can improve the cognitive, motivational, emotional and social skills of a person. More specifically, cognitive improvements were researched on various aspects; from better spatial cognition [20] to better attention allocation control [21]. Among other benefits for motivation, Isabela et al. consider games to be "an ideal training ground" for acquiring an incremental theory of intelligence [22]. Games can also make people feel in the "zone" [23], increase their overall happiness and relaxation and even provoke a sense of "intense pride" [24]. In their work, they also highlight that contrary to stereotypes, the average gamer is not socially awkward, nor does he enjoy being locked up in his room alone [teens and something]. Most gamers prefer to play games with friends, either cooperatively or competitively. Cooperative games usually reward effective cooperative and supportive actions, promoting prosocial behaviours to the players [19].

Moreover, games possess an inherent ability to sustain user engagement, as evidenced by research [25], [26]. Serious gaming is a field of research that tries to capitalize from this engaging quality of games by trying to adapt non-game contexts, such as education and training, into gaming experiences. The aim is to engage the player in such a way that they unwittingly acquire knowledge and comprehension in areas they might not typically have the patience to learn about.

2.1.2 Artificial Intelligence in Video Games

Artificial intelligence (AI) represents a field within computer science dedicated to the creation of machines and systems capable of executing tasks that typically demand human intelligence. These tasks encompass activities like reasoning, learning, planning, decision making, perception, and natural language processing. AI finds application across diverse domains, addressing challenges in areas such as robotics, medicine, education, finance, and entertainment [27]–[35].

Within the broader scope of AI, "Game AI" stands as a distinct subfield [36]. It concentrates on the development of intelligent agents and systems capable of interacting with or simulating games. Games, defined as formal systems with rules, goals, challenges, and feedback mechanisms, also function as environments to assess the skills of both human and artificial agents [37], [38]. Game AI serves various purposes, primarily enhancing the gameplay experience of the human player but it can also contribute to content generation, game design testing and balancing mechanics, as well as serve as a research platform for other AI techniques [36].

One of the most formidable challenges encountered in game AI is performance optimization. To create games that feel realistic and immersive, multiple systems must operate in tandem, all within the constraints of an extremely tight timeframe. Game development typically allocates approximately 16.67 milliseconds per frame to achieve the requisite 60 frames per second (FPS) performance, which is considered the common acceptable standard [39]. Within this limited timeframe, a game must handle tasks ranging from rendering graphics and animations to managing NPC AI and numerous other functions.

To achieve such performance, many functions in a game become approximations trying to oppose an optical illusion to the player as of to what is real. As technology continues to advance, games become increasingly impressive, raising player expectations with each release. The player's willingness to suspend disbelief becomes harder to satisfy, emphasizing the importance for the game development community to continually push boundaries and explore new techniques to deliver captivating and cutting-edge gaming experiences.

2.1.3 Suspension of Disbelief

Suspension of disbelief is a term coined by Samuel Taylor Coleridge to describe the willingness of a reader or a viewer to accept the fictional premises of a story, even if they are implausible or contradictory to reality [40]. It is a crucial concept for understanding the immersive and emotional effects of narrative media, such as literature, film, and games.

In the context of games, suspension of disbelief can be seen as a skill that players use to construct narrative coherence from the often dissonant elements of gameplay and story [8]. For example, players may ignore the unrealistic aspects of game mechanics, such as health bars, inventory systems, or save points, and focus on the narrative aspects, such as characters, dialogue, or plot. Alternatively, players may integrate the game mechanics into their interpretation of the story, such as by rationalizing them as part of the game world or the protagonist’s abilities.

However, suspension of disbelief in games is not a passive or automatic process. It requires active participation and engagement from the players, who have to balance their attention between the game rules and the game fiction [40]. Moreover, suspension of disbelief in games is not a binary or stable state. It can vary depending on the player’s preferences, expectations, and mood, as well as on the game’s design, genre, and mode. Suspension of disbelief can also be challenged or broken by various factors, such as bugs, glitches or inconsistencies [8].

2.1.4 Believability of Non-Player Characters

For the continuation of this thesis, “believable” non-player characters (NPCs) are those system agents that behave in ways that are consistent, realistic, and respond with expected ways to the player’s actions or the game’s events [1]. NPCs that do not follow this narrative immersion, as termed by Adams [41] can break the player’s immersion and suspension of disbelief by creating a sense of disconnect between the game world and the player’s expectations. For example, if an NPC repeats the same dialogue over and over, ignores the player’s presence or questions, or reacts inappropriately to the game’s situations, such as being calm during a crisis or hostile during a peaceful encounter, the player may feel that the NPC is not a living being, but a scripted object. This can reduce the player’s emotional involvement and identification with the game’s story and characters, as well as undermine the game’s credibility and coherence. Realistic NPCs can also maintain narrative coherence by supporting the game’s theme, genre, and mode. For example, realistic NPCs can follow the conventions and expectations of the game’s genre, such as being heroic in an action-adventure game or being mysterious in a horror game. Believable NPCs can also match the tone and mood of the game’s mode, such as being humorous in a casual game or being serious in a simulation game. Furthermore, such NPCs can also disrupt narrative coherence in a positive way by introducing conflict, tension, or surprise in the game story. For example, realistic NPCs can betray, deceive, or challenge the player, creating a sense of drama and intrigue.

2.2 Related Work

After establishing the background theory, we shift our focus to existing works that align with the objectives of Autonomia. We examine research efforts and projects that share either the overarching aim or specific goals similar to those pursued by Autonomia. This comparative analysis helps position Autonomia within the broader context of the field, highlighting its unique contributions and areas of innovation.

2.2.1 Behaviour Models

In the field of artificial intelligence and computer science, understanding and modeling human or agent behaviour is a pivotal aspect of designing intelligent systems. This section delves into the realm of behaviour models, which serve as fundamental ground for orchestrating the actions and decision-making processes of agents, whether they are autonomous robots, video game characters, or other AI-driven entities. By examining a range of behavioural modeling techniques, including Finite State Machines (FSM), Fuzzy Finite State Machines (FUFSMs), behaviour Trees, Stanford Research Institute Problem Solver (STRIPS), Goal-Oriented Action Planning (GOAP) and Hierarchical Task Networks (HTN), we explore the rich landscape of methods that enable machines to exhibit complex behaviours, adapt to changing environments, and interact effectively with the world around them. Through this exploration, we gain valuable insights into the underlying theories and practical applications of these models, which are essential for the development of intelligent and responsive AI systems.

2.2.1.1 Finite State Machine

FSM (Finite-State Machines) is a technique used to generate decisions for agents within games or simulations [42]. This method employs a state-centric approach, aiming to simplify the process of creating agent behaviours based on states and transitions. Rooted in the theory of computation, FSMs are designed to cater to the demands of low-level and reactive behaviours, such as movement, animation, or combat.

FSMs consist of a set of states and transitions between them, where each state represents a distinct behaviour or action, and each transition is triggered by a condition or event. FSMs are easy to implement and understand, but they can also become complex and unwieldy when the number of states and transitions grows.

The history and development of FSMs can be traced back to the early days of computer science and game development. FSMs are based on the concept of automata, which are abstract machines that can recognize patterns or perform computations. Automata theory was developed by mathematicians and logicians such as Alan Turing, Alonzo Church, and John von Neumann in the 1930s and 1940s [43]. Automata theory provided the foundation for the fields of computation, programming languages, and artificial intelligence.

FSMs were first applied to games in the 1950s and 1960s, when computer games were still in their infancy. One of the earliest examples of FSMs in games was Nimrod[44], a machine that played the game of Nim against human opponents. Nimrod used an FSM with four states to determine its moves based on the number of remaining pieces. Another early example of FSMs in games was Spacewar! [45], one of the first video games ever created. Spacewar! used an FSM with three states to control the behaviour of the enemy spaceship.

FSMs became more popular and widespread in games in the 1970s and 1980s, when arcade games and home consoles emerged. Many classic arcade games used FSMs to create simple but engaging behaviours for their characters and enemies.

FSMs continued to be used in games in the 1990s and 2000s, when games became more complex and realistic. Many genres of games used FSMs to create diverse and dynamic behaviours for their agents, such as shooters, strategy, simulation, or role-playing games. For example, Half-Life [46] used an FSM with six states (idle, alert, combat, scripted, dead, and prone) for each enemy soldier.

FSMs have some notable strengths, such as providing agents with robustness and versatility in decision making. They allow agents to select different actions based on the context at hand. FSMs are also among the cheapest behaviour models in terms of computational resources allocation, and they are simple to design and implement. However, FSMs have some limitations as well. The main drawback is their limited expressiveness and difficulty in modeling complex game scenarios. In such cases, a system would have too many states and transitions, which would make the FSM hard to read and configure [46].

2.2.1.2 FuSM

FuSM (Fuzzy State Machines) [42] is a technique used to generate decisions for agents within games or simulations. This method employs a fuzzy logic approach, aiming to handle the uncertainty and ambiguity in the game environment. Instead of having binary transitions between states, FuSMs have fuzzy transitions that are weighted by a degree of membership, which represents how much a state is active or applicable at a given moment. FuSMs can produce more smooth and natural behaviours than FSMs, as they allow for blending and mixing of multiple states. FuSMs are often used for high-level or strategic behaviours, such as decision making, planning, or learning.

However, external factors can lead to utility fluctuations or unexpected changes, resulting in outcomes that are hard to anticipate. Debugging and testing can also pose challenges, as agent behaviour can be influenced by numerous variables and conditions, while utility scores can be difficult to visualize and comprehend. The applications and extensions of FuSMs are wide-ranging, with many games and simulations integrating or adapting the technique for their agents. Notable examples include The Sims, Clone Combat 2, S.W.A.T. 2 [47] and many more.

2.2.1.3 Utility AI

Utility AI [48], or Utility-based Artificial Intelligence, emerges as a technique employed to facilitate decision-making for agents within gaming and simulations. This method revolves around optimizing agent action selection based on their inherent benefits. Rooted in the concept of utility from the economic and psychology sciences, UtilityAI is designed to cater to the demands of real-time and dynamic environments, using numerical values, formulas, and scores to quantify the relative utility of potential actions, streamlining the decision-making process. Within this framework, a decision system identifies the action with the highest utility or employs probabilistic methods based on utility scores for action selection.

UtilityAI rests on the premise that agents act rationally to maximize their utility—a measure of their preference or valuation of outcomes or states. Utility's definition is contextual, with factors such as health, hunger, happiness, safety, or wealth influencing its formulation. Mathematical functions or curves capture the changes in utility concerning various inputs or variables. These functions represent proportional relationships (linear), diminishing returns (exponential), increasing returns (logarithmic), threshold (sigmoidal), or custom-made complexities. By embracing these functions, Utility AI captures agents' nuanced preferences and behaviours, adding depth to their decision-making process.

UtilityAI boasts strengths in providing agents with robust and flexible decision-making capabilities. It simplifies code maintenance and enhances believability, as agents showcase a wider array of actions that are also transparent as of why they occur, making them easy to debug. However, utility-based AI requires careful handcrafted values for it’s actions and a large amount of developing will be allocated to testing and configuring. Kevin Dil et al. who have served as experts in the field of computer science have provided with design patterns and ways to configure a utility-based AI [48].

2.2.1.4 Behaviour Trees

Behaviour trees (BTs) are a powerful and popular technique for creating game AI [49], as they allow for complex and dynamic behaviours to be composed of simple and modular tasks. Behaviour trees are also easy to design, test, and debug, as they provide a clear and intuitive graphical representation of the AI's decision-making process. A behaviour tree is a directed tree that consists of three base types of nodes: root, control flow, or execution. The root node is the starting point of the tree, and it has only one child node. The control flow nodes are the inner nodes of the tree, and they determine how the tree is traversed. The execution nodes are the leaf nodes of the tree, and they perform the actual actions or conditions that control the AI entity.

The control flow nodes are then commonly classified into four types: sequence, selector, parallel, or decorator. A sequence node runs each of its child nodes in order until one fails, or all succeed. A selector node runs each of its child nodes in order until one succeeds, or all fail. A parallel node runs all of its child nodes simultaneously until a certain condition is met. A decorator node modifies the behaviour or outcome of its single child node.

The execution nodes can be further classified into two types: action or condition. An action node performs a specific task or behaviour, such as moving, attacking, or speaking. A condition node checks a certain state or variable, such as health, distance, or visibility.

The behaviour tree is executed by traversing from the root node to the active node every frame, following the logic of the control flow nodes and the status of the execution nodes. The status of a node can be one of three values: running, success, or failure. A running status means that the node is still performing its task or checking its condition. A success status means that the node has completed its task or satisfied its condition. A failure status means that the node has failed to complete its task or satisfy its condition.

Behaviour Trees are a well-defined structure that can provide readable, performant, and self-contained behaviours. Such behaviours can also include control flow logic and be easy to debug. Unfortunately, behaviour trees start to fail when the behaviour begins to scale, becoming unreadable when they have many nodes and branches. Furthermore, BTs are tightly coupled with their specific agent or system, making them difficult to reuse. Finally, they are not great either when dealing with dynamic environments since they have limited to no capabilities of adapting and dynamically changing their structure.

The applications and extensions of behaviour trees are wide-ranging, with many games and simulations integrating or adapting the technique for their agents. Notable examples include "Halo" [50], a sci-fi shooter featuring enemies with realistic and adaptive behaviours based on utility functions and curves. "DEFCON" is another commercial game that found success basing its implementation on behaviour trees [49]. In this game, a cold-war scenario is simulated where the player assumes the role of an army general hidden in a bunker, in hold of heavy weaponry and attempts to destroy the enemy is psychological warfare.

2.2.1.5 STRIPS

The “Stanford Research Institute Problem Solver” or STRIPS [51], [52], was initially an automated planner but was later known as a formal language for describing planning tasks, which consists of an initial and goal condition formed by conjunctions of propositional atoms and a set of actions made up by a precondition, add and delete lists. STRIPS planning is one of the most studied problems in artificial intelligence, and it has many applications in games, simulations, robotics, and other domains.

The complexity of STRIPS planning was first analyzed by Bylander et al. [51], who showed that the problem is PSPACE-complete in general, and NP-complete for some restricted classes. Bylander also identified some tractable subclasses of STRIPS planning, such as those with bounded plan length, bounded number of actions, or acyclic causal graphs.

One of the most successful approaches to finding plans for STRIPS tasks is to use search algorithms that explore the space of possible states or actions. There are two main types of search: forward search and backward search. Forward search starts from the initial state and applies actions until a goal state is reached, while backward search starts from the goal condition and regresses over actions to produce sub goals until a subgoal satisfied by the initial state is obtained. Forward search is also called progression, while backward search is called regression. Kautz and Selman [53] proposed one of the first forward search algorithms for STRIPS planning, called SATPLAN, which encodes the planning problem as a SAT formula and uses a SAT solver to find a satisfying assignment that corresponds to a plan. SATPLAN was later improved by Kautz et al., who introduced several techniques to reduce the size and complexity of the SAT encoding, such as action ordering constraints, mutex constraints, and relevance analysis. On the other hand, Bonet and Geffner [54] proposed one of the first backward search algorithms for STRIPS planning, called HSPr, which uses heuristic functions to guide the search and select the best actions to regress over. HSPr was later extended by Bonet et al., who introduced several techniques to improve the quality and efficiency of the heuristic functions, such as relaxed plans, additive heuristics, and landmarks.

Another way to approach STRIPS planning is to extend or modify the language to capture more expressive or realistic features of planning tasks. For example, Fikes and Nilsson [55] introduced conditional effects, which allow actions to have different effects depending on some conditions.

2.2.1.6 GOAP

GOAP (Goal-Oriented Action Planning) is a technique used to generate plans for agents within games or simulations. This method employs a goal-centric approach, aiming to streamline the process of generating agent behaviours based on objectives. Rooted in the STRIPS formalism, GOAP is designed to cater to the demands of real-time and dynamic environments, adapting the STRIPS concept for more practical use.

Jeff Orkin's contributions mark a significant milestone in the history and development of GOAP. Orkin introduced GOAP [56] while working on the game F.E.A.R. at Monolith Productions. He was inspired by the STRIPS planning system [52], which was developed in the 1970s as a general problem solver for automated planning. Orkin adapted STRIPS for real-time control of autonomous character behaviour in games, by using a simplified representation of the world state, a heuristic search algorithm to find the optimal plan, and a flexible action execution system that can handle dynamic changes in the environment. Orkin also added some features, such as action weighting, interruptibility, relevance pruning, plan monitoring, and plan blending, to make GOAP more efficient and user-friendly.

The advantages and disadvantages of GOAP are closely tied to its design choices and trade-offs. Notable strengths of GOAP include its ability to provide agents with flexibility and adaptability in behaviour, granting them the capacity to select different plans based on the context and goals at hand. This approach also reduces code complexity and maintenance efforts, as each action is encapsulated and independent, allowing for easy addition or removal of actions. Moreover, this modular structure increases code modularity and reusability, enabling actions to be shared among various agents or goals, while new agents or goals can be formed by combining existing actions. This, in turn, contributes to elevating the realism and believability of agents, as they can exhibit a wider range of actions, intelligent responses, and adapt to changes in their environment or state.

However, GOAP also presents certain limitations. One such drawback is the requirement for a higher level of design effort and domain knowledge. Each action necessitates well-defined preconditions and effects, while every goal needs a clearly defined criterion for satisfaction. This demands a deep understanding of the game mechanics and context. Additionally, GOAP can be prone to inefficiency and unpredictability. The process of finding a plan can involve navigating a large search space, coupled with a complex heuristic function. External factors can lead to plan failures or unexpected changes, resulting in outcomes that are hard to anticipate. Debugging and testing can also pose challenges, as agent behaviour can be influenced by numerous variables and conditions, while plans can be intricate to visualize and comprehend.

The applications and extensions of GOAP are wide-ranging, with many games and simulations integrating or adapting the technique for their agents. Notable examples include "F.E.A.R," a first-person shooter featuring enemies with coordinated attacks and dynamic behaviours, “Transformers: War for Cybertron” [57], [58] a third-person shooter were the player fights in a war of robots , “Assassin’s Creed Odyssey” [59] a large scale open-world game with hundreds of autonomous NPCs living their daily life.

GOAP stands as a robust planning technique, rooted in a goal-oriented perspective that generates plans for agents in dynamic, interactive environments. Despite its successes, challenges remain in this domain, such as optimizing plans, managing uncertainty, integrating planning with learning or reasoning, and developing user-friendly tools for plan creation and editing.

2.2.1.7 HTN

HTN (Hierarchical Task Network) planning [60] is a technique used to generate plans for agents based on hierarchical decomposition of tasks. This method employs a task-centric approach, aiming to exploit the structure and knowledge of the domain to guide the planning process. Rooted in the AI programming languages, HTN planning is designed to handle complex and expressive planning problems that go beyond the capabilities of STRIPS-like planners. HTN operators are similar to STRIPS actions but can have complex preconditions and effects. Methods are rules that define how to decompose abstract tasks into subtasks, which can be either primitive or compound. A solution to an HTN problem is then a sequence of operators that can be derived from the initial task network by applying methods recursively.

As for the previous techniques, the advantages and disadvantages of HTN planning are closely tied to its design choices and trade-offs. Notable strengths of HTN planning include its ability to provide agents with domain-specific and customized plans, leveraging the expert knowledge encoded in the methods. This approach also increases efficiency and scalability, as the search space is reduced by focusing on relevant tasks and operators. Moreover, this modular structure enhances modularity and reusability, enabling methods and operators to be shared among various domains or problems, while new domains or problems can be formed by adding or modifying methods or operators. This, in turn, contributes to elevating the expressiveness and flexibility of HTN planning, as it can handle complex goals, temporal constraints, preferences, uncertainty, and other features that are challenging for classical planners.

HTN planning also presents certain limitations. One such drawback is the difficulty of acquiring and maintaining domain knowledge [61]. Each method requires well-defined preconditions and subtasks, while each operator needs clearly specified preconditions and effects. This demands a high level of expertise and domain analysis. Additionally, traditional HTN planning assumes a fully predictable path, which may not hold in real-world scenarios. This can lead to plans that are not robust or flexible enough [62].

SHOP2 is an extension of HTN, an acronym for Simple Hierarchical Ordered Planner 2, which is an automated planning system that can generate plans for various domains and problems [63]. SHOP2 is an extension of the original SHOP planner, which was developed by the University of Maryland [64]. SHOP2 uses a domain-independent planning algorithm that can handle hierarchical task networks (HTNs), conditional effects, axioms, and durative actions and supports temporal and metric domain planning. Lastly SHOP2 has been used for various applications, such as web service composition [60], information gathering and practical planning such as evacuation scenarios [65].

2.2.2 Machine Learning in Game AI

Machine learning (ML) falls under the umbrella of artificial intelligence and revolves around the use of algorithms and statistical models to enable machines to act without explicit programming. It allows non-player characters (NPCs) to learn from data, experiences, or rewards, allowing them to enhance their performance over time.

Machine learning techniques garnered significant recognition with landmark achievements such as AlphaGo, DeepMind's AI, defeating the world champion in Go, an intricate game demanding profound intuition. This breakthrough illustrated the immense potential of machine learning in tackling complex challenges [66].

Another remarkable instance of machine learning's capabilities pushed to the extreme can be seen in the "Dota 2" team developed by OpenAI [67]. This AI system achieved the unprecedented feat of defeating world champions in an e-sport game. Notably, the system underwent rigorous training, processing approximately two million frames every two seconds over a training period spanning ten months. These monumental successes highlight the remarkable power of machine learning in mastering and excelling in tasks that demand high-level strategic thinking and decision-making.

Kunanusont et al. [68] have proposed a General Video Game Artificial Intelligence (GVG-AI) framework based on deep learning, to allow systems to play games learned through screen-captured video.

Joon Sung Park et al. [69] in their recent work, surprised the research community by making a video game simulation of 25 instances of ChatGPT, each role-playing as its own person, all living in the same community. Those ChatGPT personas, could even self-reflect and showed in general great social interactions.

2.2.3 Social NPCs Model

Social NPCs are non-player characters that can interact with the player and other NPCs in a game world, using social cues, emotions, relationships, and goals. Social NPCs can enhance the immersion, realism, and narrative of a game, as well as provide more opportunities for gameplay and exploration. Several approaches have been proposed to model social NPCs in games, using different techniques and frameworks.

2.2.3.1 Comme il-Faut (CiF)

One of the most influential works in this domain is Comme il-Faut (CiF) [70], a social agent architecture that represents rich social interactions between agents that include emotions, social and relationship contexts, and longer term mood. CiF was applied to the inaugural game “The Prom”, which is an interactive narrative experience centered around a clique of high school students, mainly from the counter-culture scene, as they navigate the final week leading up to their prom night. In this game, players assume the role of guiding these characters in making social choices. They must decide from a range of options, such as flirting, sharing interests, or cracking jokes at someone's expense, based on the characters' current thoughts and feelings. These interactions unfold as detailed dialogues between the characters. The game utilizes CiF's algorithms to generate social action lists for each character, taking into account their unique personalities, existing relationships, and past social experiences.

CiF-CK is a social agent architecture developed by Guimaraes et al. [71] and is based on CiF. This work elevated CiF and created a mod for the successful game title “The Elder Scrolls V: Skyrim” to apply and evaluate their architecture, having the player himself interacting with those social agents through Skyrim’s first-person perspective gameplay.

2.2.3.2 FAtiMA Modular

The FAtiMA modular [72] is an agent model architecture that encapsulates the minimum set of functionalities, considered by the authors, to build emotional agents. Their approach allows them to quickly and easily build various social agent models in order to compare them and evaluate them. Seven years later, Mascarenhas et al. [73] assembled a collection of diverse open-source tools specifically tailored for emotional agents, each possessing a degree of decision-making capacity. These tools also feature an integrated dialogue system closely aligned with the common industry technique of dialogue trees. To showcase the practicality of their work, they undertook various use case scenarios.

For instance, "Space Modules Inc" serves as an illustrative example. In this game, players take on the role of customer service representatives on behalf of a spaceship part manufacturer. Each customer in this virtual world exhibits a distinct emotional profile, demanding the player to employ unique social strategies or tactics in handling each situation effectively.

Another intriguing project they embarked upon is "Police Interrogation" a virtual reality game where players assume the role of a police officer. Their objective is to extract as much information as possible from subjects without letting the situation spiral out of control. These practical applications of emotional agents and dialogue systems underscore the versatility and real-world relevance of their open-source tools.

2.2.4 World State Representation

In his work [74], Jeff Orkins highlights the importance of a symbolic representation of the world state based on two observations; a) today's expectations of game AI are beyond a simple finite state machine, and b) planning algorithms like GOAP are computationally expensive if left unchecked. Various optimizations need to take place, and it is mandatory for the algorithm to be able to connect goals and behaviours through their preconditions and effects. In addition, he speaks of context (or procedural) preconditions and effects, which represent a piece of code that will run upon the execution of logic, and that it is mostly used for pruning the search tree.

There is also a plethora of works that highlight the importance of modeling a game's world state in a semantic way. Kessing et al. [75] iterates over the key benefits of having a semantic world and they build a tool named Entika to facilitate the deployment of such mechanisms in a game. Afonso's and Prada's work [76] was also inspiring for this work as they provide a model of agents that can relate having as a basis a dominant psychological theory regarding personal agency, the Theory of Mind [77], [78].

2.3 Progress beyond the state of the art

This section discusses the progress beyond the state of the art of the work presented in this thesis. The Autonomia Framework introduces novel concepts and approaches that break new ground and surpass the current state of the art in several key aspects.

1. **A World State that replicates a Theory of Mind:** The world state in Autonomia is modeled as a Memory class which is purely based upon the Theory of Mind and knowledge graphs. This allows the system to have recursive representations of various micro-world states, depicting the personal perspectives each NPC has for the world and the people around it. This architectural decision allows Autonomia's world state to have a multi-ordered [79] theory of mind representation.

2. **Behaviours and goals reside in Memory**: Everything an NPC knows in

Autonomia is extracted either by its Memory or Perception module. This allows the execution and evaluation of behaviours and goals to be made in a realistic manner with knowledge accessible only from their own, unique theory of mind. In addition, behaviours and goals themselves are part of this world state representation, they are modeled and used in a way that allows them to be treated as first class citizens of the Memory class and in this way, they can be communicated, forgotten or even discovered. Lastly, other behaviour models can be encapsulated in Autonomia's behaviours to enrich them with the first-class citizen attribute.

3. **Planning through expressive and procedural preconditions and effects**: In Autonomia, plans are devised using a Goal-Oriented Action Planning (GOAP) algorithm. What sets this approach apart is the use of Expressions for both behaviour and goal preconditions and effects. This elevates the algorithm by infusing it with procedural expressivity while still supporting state matching. In addition, the Unity implementation of Autonomia simplifies the process with: a) a simple design pattern for the creation of user expressions, and b) a graph node editor for authoring Expression graphs.

4. **Intended Use Optimization for GOAP**: This thesis introduces a new optimization for GOAP-based algorithms that enables better control over the formulation of plans and improves performance by narrowing the dynamic search space of behaviours.

5. **Common ground for Research**: The problem of NPC believability is a multifaceted problem spanning from visual fidelity to behavioural and emotional authenticity. The Autonomia Framework, as an open-source and extensible project, offers a collaborative platform for researchers to contribute their expertise. The ultimate goal is to collectively work towards crafting realistic NPCs, making it a valuable and unifying endeavor for research in the field.

Chapter 3

3 Framework Architecture

Game development is renowned as one of the most demanding fields in software engineering. It continually presents new challenges and higher expectations. To keep pace with this evolving landscape, the game development industry recognizes the paramount importance of having the right tools for the job. This thesis places its primary emphasis on the Autonomia Framework as a tool designed to aid fellow developers. The framework serves as a foundational structure that can be extended, allowing developers to concentrate on specific tasks and problems that suit their expertise. For instance, a future implementation of the system could include emotional AI libraries running in parallel with ML trained animation systems, whilst having graphical tools for game designers to freely express their creativity. It's crucial to note that the framework, as presented here, is not intended as a final nor a complete solution. Instead, it is an invitation to the research community and developers to explore, build upon, and refine this framework further.

3.1 Overview

In this chapter, we delve into the architectural decisions that form Autonomia, presenting the specific definitions and classes that are heavily used in the core of our framework. We will explore essential components that underpin Autonomia's functionality, providing a comprehensive understanding of its inner workings. These fundamental components include:

- **World Representation Based on the Theory of Mind:** We explain how Autonomia utilizes the Theory of Mind to construct a rich world representation that facilitates NPCs' understanding of their environment and interactions.

- **Module System:** This section elaborates on how our modular system enriches the nodes of the knowledge graph within the framework, empowering them with extended functionality and flexibility. We present in this section the core modules of Autonomia.

- **Expression, Behaviour and Goal (EBG) System:** Definitions for the interconnected systems of expression, behaviour and goals and how they synergize to drive and plan NPC actions, reactions and plans.

Together, these components form the foundation of Autonomia, enabling the creation of intelligent NPCs. As we delve into the specifics, readers will gain insights into how Autonomia leverages these elements to enhance the authenticity and complexity of NPC interactions in the context of video games. Yet, the core of the framework is not enough to run on its own, since it only provides the basic motif and tries to enforce specific patterns. It is up to the developers to implement and extend the framework based on their own development needs.

3.2 Methodology

The development and design of the Autonomia Framework followed a structured methodology that combined a detailed literature review, iterative prototyping, and a strong commitment to ambitious research. The approach taken in this project differed from conventional game development, which tends to prioritize safety and predictability due to industry demands. Research, on the other hand, allows for greater creativity and exploration of unconventional ideas, even if they carry a risk of failure. As illustrated in the following subsections, the methodology used in creating the Autonomia Framework involves literature review and theoretical foundation as well as iterative prototyping and development.

3.2.1 Literature Review and Theoretical Foundation

The initial phase of the framework's development commenced with an extensive literature review, which spanned a wide range of sources. These sources included academic papers, books, online documentation, and industry standards. The primary objective of this review was to acquire a comprehensive understanding of existing game AI frameworks, AI theories, and software engineering best practices.

Building upon this knowledge, the theoretical foundation for the Autonomia Framework was laid. This involved synthesizing relevant AI concepts, such as the Theory of Mind, major behavior models, and programming design patterns that facilitate code scalability. These theoretical insights served as the basis for making architectural decisions and establishing core design principles for the framework.

3.2.2 Iterative Prototyping and Development

The development of Autonomia followed an iterative and agile approach. This methodical process commenced with a significant amount of time dedicated to designing the overarching concept. The primary focus during this phase was on bridging the gaps within existing methodologies and techniques, as well as identifying innovative ways to enhance NPC believability.

Following the initial design phase, multiple prototype versions of the framework were created, each building upon the insights gained from the previous iteration. These prototypes served as experimental platforms for exploring different architectural structures, algorithms, and features. Feedback collected from prototype testing played a pivotal role in refining the final architectural design.

Throughout the development process, a strong emphasis was placed on adhering to software engineering best practices. This included the implementation of version control, issue tracking, and coding standards.

3.3 Theory of Mind & World Representation

3.3.1 Theory of Mind

Theory of Mind [77], [78] is a term used in phycology to describe the ability of one's self to understand the mental state of others. Its definition extends to being able to define and determine different emotional states, feelings, desires, beliefs or even thoughts of others. A person using his Theory of Mind (ToM) should be able to extend, predict and explain the behaviour of others. For example, if person A, notices person B crying, person A could explore his current model of the world, his theory of mind, in order to understand why person B is having this reaction. Using common knowledge, person A can assume that person B is for some reason sad. Then by delving deeper and extending his ToM through perception, person A might narrow down the reasons person B is crying and is sad, or maybe realize those tears are tears of joy.

Our theory of mind allows us to interact with other social beings in meaningful ways; to empathize, communicate and even understand different perspectives and interpretations of events. Each one's theory of mind is gradually developed from infancy. Babies begin paying attention to facial expressions, voice alterations and gestures. From there, people begin realizing their own emotions, realize that other people have other beliefs and perspectives and sooner or later develop more complex skills such as sarcasm, humor or even deception. It is important to highlight, that even thought to some degree all people are able to construct their own theory of mind, and each one can vary based on the person, situation and culture. Also, it is not a unique skill to humans. Some animals, such as apes, dolphins, elephants, dogs, and crows, have shown evidence of having some form and capability of theory of mind.

3.3.2 World State in Video Games

The world state is a term that refers to the current condition and status of the game world and its elements, such as the environment, the characters, the objects, the events and others. The world state can change dynamically based on the actions and choices of the player and other agents, as well as random or even scripted events. The model of our world state will define the strengths and weaknesses of our engine. For example, in a role-playing game, the world state might include the level, health, inventory, and reputation of the player character, as well as the quests they have completed or failed, the allies and enemies they have made, and the locations they have visited or unlocked. The world state might also include the weather, time of day, seasons, political situation, and cultural events and rules of the game world. These factors can influence how the player interacts with the game world and how the game world reacts to the player.

3.3.3 Autonomia World State

The Theory of Mind (ToM) is a multifaceted concept comprising various interconnected mechanisms that collectively enable us to comprehend, experience, and respond to the world around us. It is only rational for ToM to serve as the foundation for Autonomia. In our attempt to address this issue, we directed our attention to developing a world state representation capable of mirroring the nature of ToM. After researching the literature, brainstorming sessions, and testing, we arrived at the conclusion that a knowledge graph structure would best align with our objectives. Although relatively uncommon in game development, the adoption of knowledge graphs represents an emerging trend that offers significant potential advantages.

A knowledge graph is a structure for representing information in the form of a network of nodes and edges. Nodes represent entities or concepts, while edges are the relations among them and each of those may contain labels or properties of any type. A knowledge graph inherently has the ability to capture semantic meaning and context of information, thereby enabling reasoning and inference based on the data it contains. Furthermore, this approach opens the door to future possibilities, including natural language interactions within the game world, such as querying, narrating, or even engaging in conversations. Additionally, it facilitates the integration of external data and knowledge sources into the game world, enriching the realism, diversity, and relevance of game content with minimal effort. However, it's important to acknowledge that these benefits do introduce increased complexity and challenges, particularly concerning real-time performance optimization in the game environment.

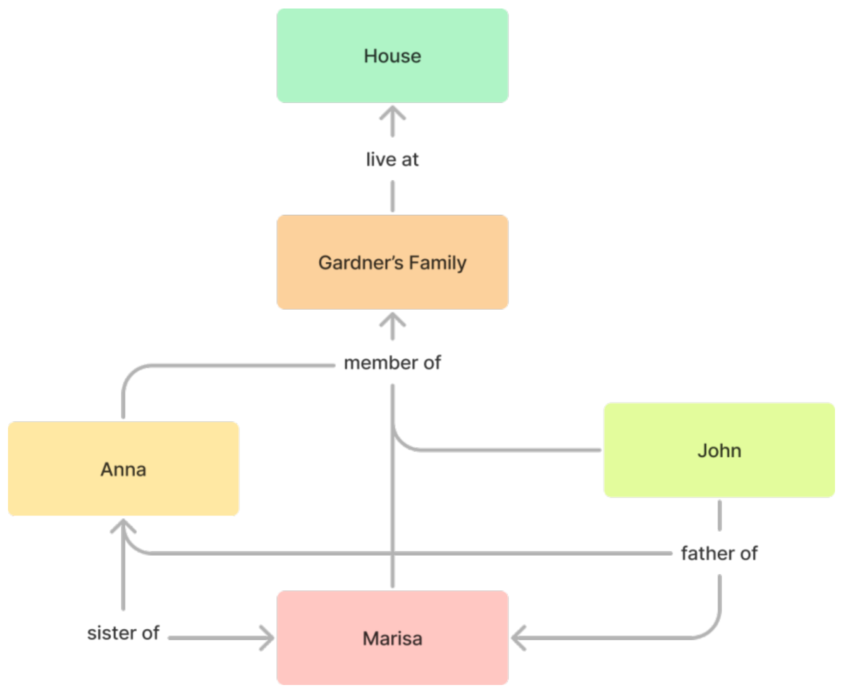


Figure 1: Example of a knowledge graph

Of course, each node has the potential to represent any concept in the game world. To attach meaning and functionality, we allow each node to have modules. By attaching modules to a node we can classify it into different conceptual categories. For example, a node with a Perception, Memory and Needs module can represent a simple NPC. We elaborate regarding modules in the next section.

3.4 Module System

Since Autonomia is a framework designed to be extended for any need and platform, it was crucial to implement a modular system that will efficiently decouple different functionalities. Modules have a node owner to whom they provide their features. With a specialized function that will be discussed later, there may also exist various copies of a module for different layers of memory.

In addition, Modules may implement methods derived from the base Module class to fulfill their functionality or even serve as plain data containers. They also have the option to serialize or deserialize their data to be persistent throughout sessions. In the following subsections we describe the main Modules used in the core version of Autonomia.

3.4.1 Active Events Module

During the early stages of Autonomia's development, we recognized the necessity of implementing an event system. In our context, an event signifies any observable action, such as eating, walking, or conversing, and is defined by an actor, a type, and a subject. Another way to view events is as temporary relations actively being caused by some action. For example, the action of drinking water is a relational fact as much as an action, but it is due to last for a brief moment.

In order to represent a currently running event in the world, we created a Module named ActiveEvents. Any component in the framework that will begin an effect has the responsibility to access the ActiveEvents module, add the newly created event and remove it to signal the event's conclusion. So, the ActiveEvents module is our way of exposing actions to our perception system which will be discussed later.

3.4.2 Event Interpreter Module

In addition, we dictate that events are nothing more than plain data. On their own they do not carry any meaning. Thus, we created another Module named *EventInterpreter*. This module assumes a crucial role within Autonomia, as it focuses on updating an NPC's Memory, specifically its relational memory. By isolating the responsibility for updating relational memory, we enable the system to potentially generate context-aware assumptions and interpretations of events. For instance, consider a scenario where two individuals engage in a physical fight; this event can be interpreted in multiple ways. It could signify hatred between them, or it might be a friendly sparring match. Alternatively, one individual could be a law enforcement officer apprehending the other for reasons known or unknown to the virtual agent. This approach reinforces our assertion that events, by themselves, lack inherent meaning.

So, each *EventInterpreter* may have multiple interpretations for the same type of event, but each can be characterized by a “matching score”. The interpretation with the highest score gets to alter the memory of an NPC when the need arises.

Lastly, we have facilitated the ability for interpretations to be transferred from NPC to another NPC. This addition allows us to have a newborn child agent that cannot make sense of the world, but as it grows older it begins to understand, be taught, and eventually teach others the ability to interpret.

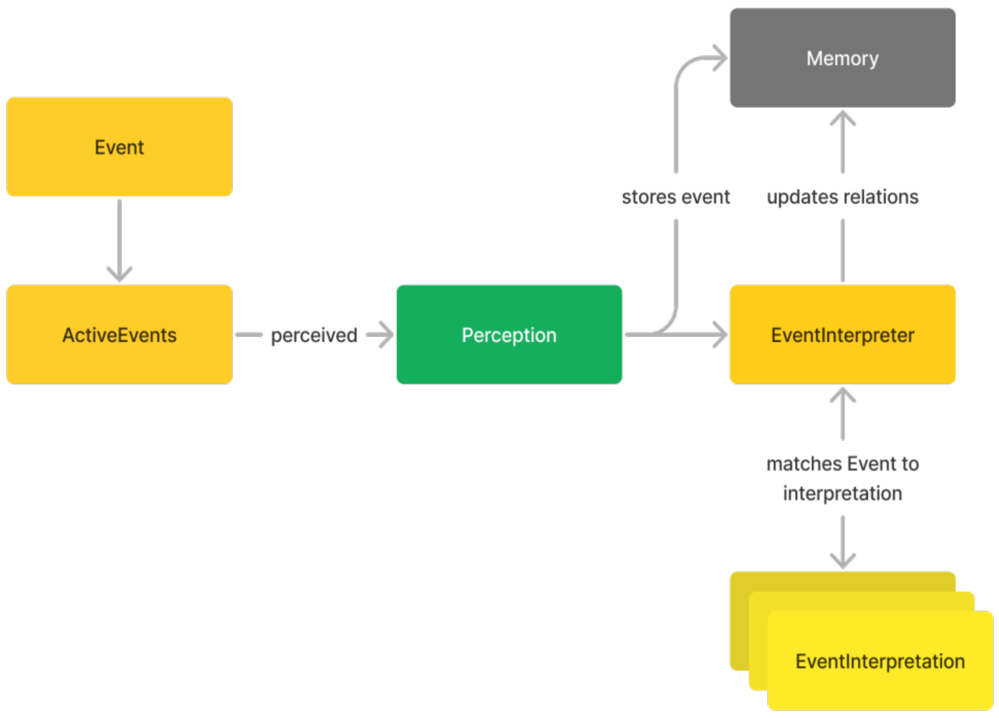


Figure 2: ActiveEvents, Perception and EventInterpreter

3.4.3 Perception Module

We have chosen to model Autonomia's perception system after the principles of human perception. This system comprises three key layers as shown in Figure 3. Detectors being implementation-specific; assume the responsibility of identifying and storing current *Events* or Nodes within their designated stimuli. At any given point in time, the *Perception* module can access all available stimuli, enabling it to retrieve related information. By combining various sensory modalities, including visual, auditory, and potentially supernatural senses, we can enable our agents to respond dynamically to their environment.

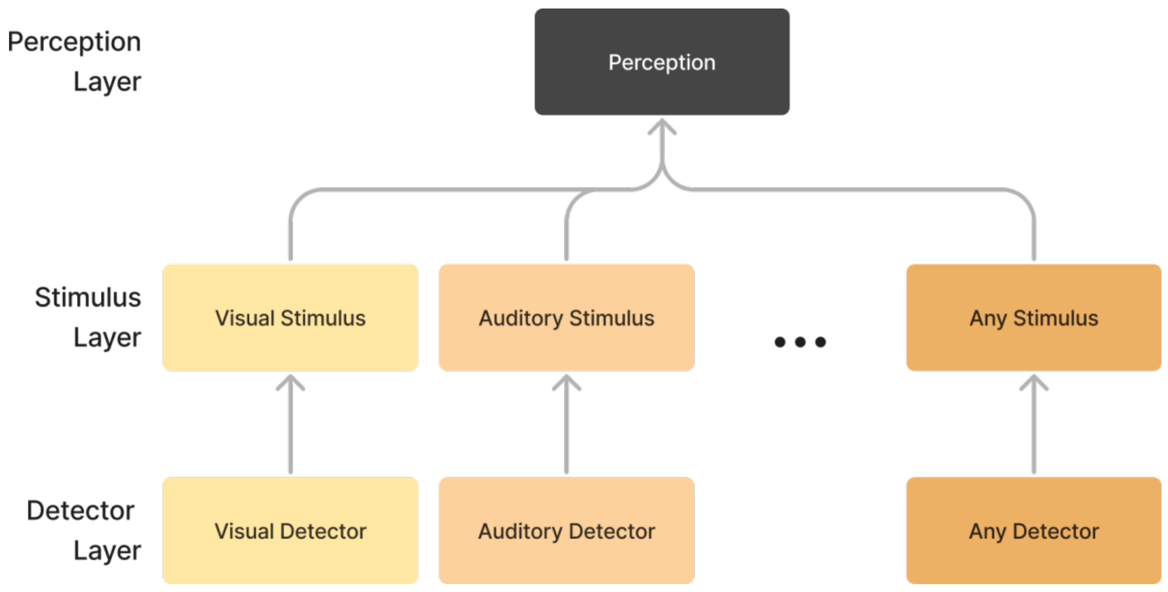


Figure 3: Perception system structure of Autonomia

The *Perception* module holds a pivotal role within the Autonomia system, as it is responsible for actively searching for detected events and subsequently forwarding them to the *EventInterpreter* module. Furthermore, the perception operates on a publish-subscribe (pub/sub) basis, enabling other modules to request specific event notifications from the perception system.

It is crucial to emphasize that, at this point, the perception system serves as the sole conduit of communication between an NPC's memory and the external world. This design decision allows us to introduce a filtering mechanism within the perception system, referred to as what we term a "memory-local node". That means, that every module subscribing to an event through perception, will always receive nodes in their "memorized version". For instance, in a scenario where Node A visually perceives Node B, Node A's knowledge about Node B should be restricted to what it already possesses in its memory. Node A should not gain access to information about any hidden objects behind Node B's back unless it has prior knowledge of this fact or chooses to interpret and assume such knowledge. This represents a fundamental concept within Autonomia's framework, as every other module providing functionality to an NPC is strictly constrained by the NPC's existing knowledge base through the Perception module. Implementation details regarding Perception can be found in section 4.2.6.

3.4.4 Memory Module and the Theory of Mind

The *Memory* module is composed of two distinct types of data structures:

- *Relations*: This structure takes the form of a knowledge graph, representing all the relational knowledge of the nodes of the world,
- *Event Index*: This structure is used as an efficient way for swiftly retrieving stored events.

In our implementation, we have utilized interfaces to allow for custom implementations of both graph structures and event indices. Additionally, we have introduced a class named *MemoryQuery*. This class serves as a library of method calls designed to streamline the traversal and manipulation of these structures. For instance, we have implemented functions such as "get neighbors," "get relation by type," and even support for breadth-first searches to facilitate graph traversal. Lastly, *MemoryQuery* supports a simple type of non-nested string queries.

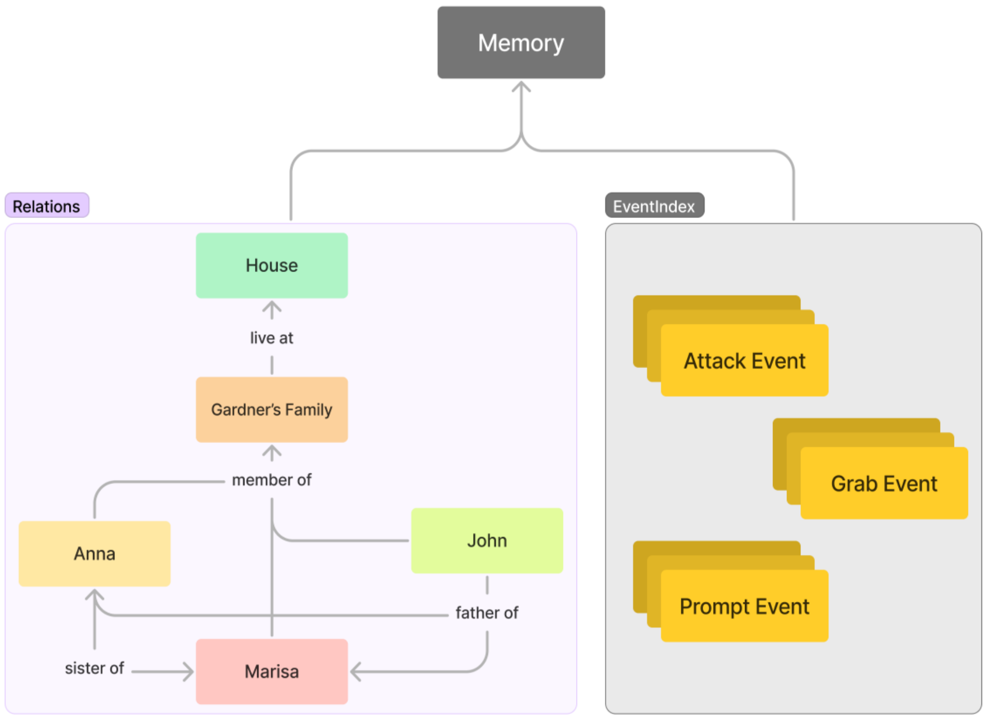


Figure 4: Memory Module representation structurally

An essential design principle characterizing the *Memory* module revolves around the concept of exclusively returning the "memory-local node" at any given point in time. This design choice plays a pivotal role when coupled with the perception system, as it enables agents to exhibit realistic behaviour based uniquely on their knowledge of the world.

To illustrate this principle, consider three nodes within our scenario: Node A and Node B, both residing in the same house, and Node C, representing a food resource in the fridge. If Node B were to wake up early and consume Node C, it is logical that Node A remains unaware of this occurrence. Reasonably, the expected behaviour for Node A would have him waking up, planning to remember where Node C is, proceeding to its location, only to realize that it is no longer present. This mechanism ensures that Node A's actions align with its knowledge, promoting realistic and immersive agent behaviour within the system.

The most important contribution of this module is yet to be explained, but in its current state it has already singlehandedly achieved, as defined, a multilayered Theory of Mind. The key takeaway is that nodes are defined with their modules included. A Node in the relational part of a *Memory* module can contain a *Memory* module which as well can contain other Memory modules of other NPCs and so on. This leads to a scenario in which Node A contains a version of Node B's knowledge as he perceives it, and vice versa, which in itself creates a never-ending recursive loop of “if he knows-they know he knows” etc. This is further illustrated in Figure 5. To solve the infinite recursion problem, we assign nodes with a *simDepth* (simulation or simulacrum) variable and define a *maxSimDepth* in our system. The larger the *maxSimDepth* the more accurate theory of mind we can achieve, but we are also bound to use more resources and greatly increase complexity.

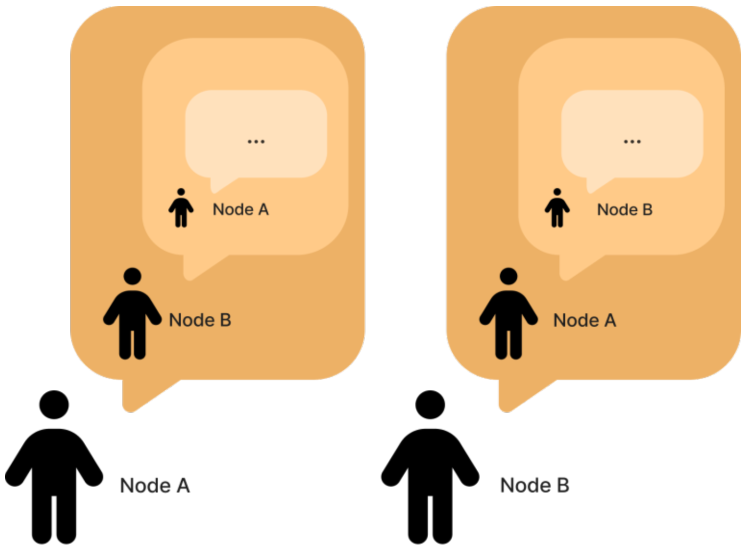


Figure 5: Illustration showcasing the infinite recursion

3.4.5 Behaviour Controller Module

Behaviours have not been defined yet at this point yet, but for simplicity's sake let us assume behaviours as simple actions, for instance walking, running, eating, sitting etc. Various modules which can also be completely agnostic to each other, may want at some point, to initiate some behaviour. But, it is easy for behaviours to contradict with each other, for example a person cannot walk while sitting. Thus, we created the *BehaviourController* module which is tasked with deciding which behaviour should run at which point.

This problem required a lot of careful thinking and planning. Some ideas we experimented with were CPU scheduling algorithms like round-robin but they do not necessarily make sense in our human behavioural context. Also, we brainstormed ideas of separate body limb declarations for each behaviour. For example, we may be walking down the street going to work, which is a behaviour that mostly occupies our legs. This does not stop us from greeting someone, a behaviour which would require the head and hand, but this thought process substantially increases complexity. In the end, we decided that simple is better and our solution to the problem follows next.

The *BehaviourController* allows competitors to have a ticket granted to them, and each ticket is paired with an importance value which starts at zero. Then, each competitor may "try" their ticket with an importance value. If the importance they declared is higher than the running ticket's importance, the behaviour controller allows them to "switch" the current behaviour to what they dictate. This of course comes with credibility issues. A competitor may declare the highest possible value simply to take control, but this is fine. In a game scenario the programmer wants this control to enforce story elements to take place.

To explain our algorithm's logic, first we need to understand that it is common and reasonable for many mechanisms in the conscious mind to want something done. It is truly simple; in the end, we will do the things that we care about the most. So, the brain's ability to value goals is what drives us to behave in any specific way. For instance, let us imagine Node A is at bar and that node is talking to his love interest. He may notice his friends are also at the bar but continues talking to his interest. At some point Node A may also feel the need to use the bathroom and we have multiple conflicting behaviours that want control of Node A's actions. The part of his brain that wants to appeal to his love interest wants to continue talking but the need to use the bathroom will gradually increase. It is only natural that when the importance of that need becomes greater that it will take control. To extend this example further, let us imagine that while our subjects are talking a robbery may take place in the bar and both NPCs would ideally turn to their survival instincts.

3.4.6 Intended Uses Module

This is a module intended purely for optimizing the A* algorithm in our GOAP search and giving game designers more control. More details can be found in section 4.2.14.2, yet this serves as a great example of the versatility and flexibility of modules.

3.5 Expressions, Behaviours and Goals

It has been exhaustively discussed in the literature and is reasonable to agree that a realistic and believable NPC has agency. There should be purpose behind his actions and he should have dreams and goals he strives to achieve. Every action should be supported by a reason and this concept has given rise to most successful behavioural models that were discussed in their respective related work chapter (2.2.1), and the games that adopted them have proved their worth.

None can doubt the complexity of the problem at hand and it is a challenge that should not be looked down upon. It was clear that every decision in the framework's architecture should be made to complement this exact aspect.

In this section, we discuss our Expression-Behaviour-Goal (EBG) model. The purpose of this model is to allow easy authoring of behaviours and goals and allowing the system to match them through their declared expressions. Each concept will be broken down individually, and we talk about their synergy as a completed model in section 3.5.5. Not by any means does our model try to replace traditional or custom behavioural models. Instead, we view it as a way to encompass what already exists and further complement it by using our model, which allows everything to be treated as knowledge that can be passed along between agents.

3.5.1 Expressions

In most STRIPS-based planning algorithms like GOAP and HTNs, actions or behaviours must declare preconditions and effects. This declaration enables the algorithm to determine which actions can be executed under specific conditions. For instance, an agent cannot execute an "Attack with Sword" action if he is not already in hold of a sword. To address this, a hypothetical action like "Grab Sword" would establish the precondition for "Attack with Sword," potentially setting a variable like "isHoldingSword" to true. Then, the preconditions for "Attack with Sword" would evaluate to true and the agent could execute that action.

While methods like shared blackboards are commonly employed for precondition-effect algorithms due to their speed and efficiency, they have limitations in capturing all relative information or knowledge needed for agents to plan and act effectively. This limitation compromises the expressiveness and realism of the system [74].

Our solutions to this problem are Expressions. Drawing inspiration from Abstract Syntax Trees (AST) [80] in compiler design, we define expressions as abstract nodes within a simplified syntax tree, allowing users to create custom expressions that can carry and process information in a structured manner. Ultimately, the root of the expression tree can be evaluated, triggering a cascading evaluation throughout the tree. If the tree successfully evaluates, the final value can be retrieved from the expression. Detailed technical information regarding expression methods is discussed in section 4.2.12.

We have defined two primary derived classes for expressions that automate the matching and evaluation logic for user-defined classes:

1. Producer Expression: These expressions serve as the leaves of the tree and can generate a value, such as *StringExpression* or *NumberExpression*.

2. Processor Expression: Expressions as such can have children whose values they utilize to produce a new value. For instance, a *MathExpression* would require two children evaluating to numbers, and another child representing an arithmetical operation like addition or subtraction.

3.5.2 Behaviours

In our system, behaviours are an abstraction to commonly referred actions, “the process of doing something”. Autonomia’s behaviours can refer to nonphysical actions as well, for example thinking or planning your next action. Each behaviour consists of two sets of expressions: preconditions, which determine the conditions that must be met for the behaviour to execute, and effects, which specify the desired outcomes upon the behaviour's completion.

We define the "actor" as the node executing the behaviour, and the "owner" as the node exposing or providing the behaviour, akin to an affordance. To illustrate this concept further, consider a "Sit Chair" behaviour. In this case, the person intending to sit in the chair serves as the actor, while the chair itself is the owner.

Moreover, each behaviour includes methods that describe its cost or effectiveness and an estimate of the required time based on the current actor's knowledge. These attributes, namely the cost and required time of a behaviour, wield considerable influence over the planner's decision-making process when selecting behaviours as part of a plan. We will elaborate into the impact of these factors on goal planning in the upcoming sections.

It is also important to mention, that nodes do not have any direct reference to their behaviours. The framework has an injection mechanism, that allows new nodes to be created in its runtime, called *BehaviourNodes*. Those nodes are then connected to their related nodes in the graph and become in a sense accessible through them. By having Behaviour Nodes be related to nodes that expose them, we allow them to be treated as a piece of knowledge. Node A is now able to teach Node B the existence of a behaviour contained in BehaviourNode C, he was previously unaware of. This is a novel addition to our system, in effect turning behaviours as first-class citizens of the framework.

Last but not least, Autonomia's behaviours are not created to replace previous techniques. The purpose of the behaviour system is to allow discovery and planning in a way that is always based on the knowledge of the actor. A behaviour could be a Behaviour Tree [49] and then seamlessly switch to a Utility AI [48] implementation whilst completely decoupling them.

3.5.2.1 Complex Behaviour

To facilitate more advanced actions, we introduce a specialized class known as *Complex Behaviour* within our framework. *Complex Behaviours* are designed to streamline the execution of abstract or high-level behaviours. For instance, consider the "Tavern Waiter" behaviour, which is inherently intricate. It involves multiple steps, distinct phases, and requires dynamic planning. Complex Behaviours offer several advantages, including:

- Behaviour Queues: They enable the creation of behaviour queues, allowing for the sequential execution of multiple behaviours. This is particularly useful for orchestrating complex sequences of actions.

- Automated Planning/Replanning: *Complex Behaviours* incorporate the ability to automatically replan in response to a behaviour not meeting its preconditions. This ensures adaptability in the face of unexpected obstacles or changes in the environment.

- Nested Complexity: *Complex Behaviours* can contain other *Complex Behaviours*, fostering a hierarchical structure. In the case of "Tavern Waiter," it may encompass behaviours like "Take Order," "Serve Order," and "Take Bill," each of which can in turn, contain their own recursive behaviours.

This hierarchical approach to behaviour design empowers our framework to handle intricate, multi-step tasks efficiently and flexibly.

3.5.3 Goals

In crafting a believable agent, the presence of goals that steer its actions is imperative. The agent should possess an awareness of these goals and the capability to formulate plans to achieve them. While existing literature provides extensive insights, on the modeling of goals, in the core version of Autonomia, we have opted for a simplified representation to accommodate future extensions. Our model of goals comprises three key components: a) name, b) a set of expressions that signify when the goal is satisfied, and c) a method that returns their current value or importance. This metric assists the agent in prioritizing and selecting goals for planning and execution.

Exactly like *BehaviourNodes*, we allow the existence of *GoalNodes*, enabling them the same benefits discussed previously. Mainly, allowing goals to be treated as knowledge that can be passed along. This streamlined approach to modeling goals in Autonomia lays the foundation for the inclusion of more sophisticated goal-related functionalities in future iterations.

3.5.4 Abstract Nodes and Wildcard values

In the early stages of Autonomia's development, it became evident that there was a necessity for defining a node type that abstracts the specific requirements we seek. For instance, a user might wish to declare a goal like "Sit Goal," but this goal could be satisfied by any chair or even any object allowing a person to sit. To address this need, we introduced the concept of Abstract Nodes.

Abstract Nodes serve as specialized nodes designed to establish a superset of other nodes by leveraging the native labels derived from the knowledge graph structure. Essentially, they allow for a higher level of abstraction, enabling users to define goals and conditions in a more generalized manner, while retaining the flexibility to encompass a wide range of specific instances. Now, when creating a "Sit Goal," the process involves crafting an event expression of type "sit," utilizing an abstract node labeled as "Chair" as the subject. It is important to note that this approach necessitates that all chair nodes in the system be consistently labeled with the "Chair" label.

For more intricate goals that involve multiple criteria or conditions, a thoughtful design process is required in advance. This process ensures that the labeling and abstraction of nodes align with the specific goals and objectives defined within the system, allowing for the effective representation of complex behaviours and objectives.

Another tool that was required to be created was the wildcard values. These wildcards serve as placeholders that can be matched to any data type, effectively indicating that a value can take on any form. For instance, in the context of a "Walk behaviour," one of its effects could be an event of the type "move" with a subject designated as a wildcard node. This signals to the planner algorithm that the wildcard node can represent any other node in the system.

Wildcard values enhance the flexibility and adaptability of our framework, allowing it to accommodate a wide range of potential scenarios and conditions where the exact identity of a subject or value may vary.

3.5.5 EBG Model

Combining expressions, behaviours, and goals, we have a completely expressive behaviour model paired with meaning that is also easily extendable by other programmers. The EBG model is a way of encapsulating any form of AI whilst giving it the ability to be treated as a piece of knowledge within the game world, essentially making behaviours and goals first-class citizens in the entirety of the framework. NPCs can potentially discover, teach and compare behaviours residing within the world and seamlessly be able to use them for their plans and goals.

Expressions serve as a flexible bridge between behaviours and goals. Since expressions are based upon abstract syntax trees [80], an extended implementation of this system could be considered its own micro-programming language that is also easy to extend with a few lines of code. At the same time, due to their strict syntax they can allow state matching for the GOAP [57] algorithm making them a powerful and versatile tool.

3.6 Goal Planning

Planning is an essential aspect of our daily lives, whether carried out consciously or unconsciously. For any intelligent life form, the ability to plan, replan in response to obstacles and prioritize tasks based on current objectives is fundamental. Within Autonomia, our agents are equipped with two primary planning algorithms, and the framework allows for the incorporation of additional algorithms in future iterations. First, we have a Goal-Oriented Action Planning (GOAP) algorithm, to enable complex behaviour chains to be formed. In addition, Autonomia introduces a scheduling algorithm that allows NPCs to plan their entire day proactively. This algorithm utilizes the agent's prior knowledge of the world and projects the expected state for each moment in time. This proactive approach enables agents to make informed decisions and efficiently allocate their time and resources to accomplish their tasks and priorities.

3.6.1 Autonomia's GOAP A* Search

Our applied algorithm combines two core techniques: A* search and Goal-Oriented Action Planning (GOAP) but alters them just enough to fit the broader context and dynamic world of Autonomia. Here's an overview of how our implementation of algorithm operates:

- **Behaviour Set:** The agent queries all potential behaviours from his memory, each characterized by preconditions, effects, and associated costs. For example, in our agent's memory there may exist a kitchen with a plethora of tools. All of the exposed behaviours are added to the behaviour set.
- **Agent's Goal:** The agent also has a defined goal, representing a desired state or expressions it aims to satisfy. For instance, a goal might be to "Eat Food" which is satisfied when the agent succeeds on eating a food resource.
- **A Search*:** The agent employs the A* search algorithm to identify the optimal sequence of behaviours leading to the goal. In our context, the nodes within the search state represent states of the world as sets of expressions, while the edges represent the most recent behaviour that brought that state. More information can be found in section 4.2.14.2. The A* algorithm can be optimized by using a heuristic function that enables us to prioritize the exploration more promising routes. In our example, to satisfy the "Eat Food" goal, an agent could calculate the following plan: "WalkBehaviour" to get to the fridge, "OpenDoorFridgeBehaviour" exposed by the fridge to open the door, "RetreiveFridgeItemBehaviour" to get the apple from the fridge.

The A* GOAP algorithm empowers agents to intelligently plan their actions, adapting to environmental changes to achieve their goals. However, as discussed in the related work section, GOAP can be computationally expensive, especially when applied to a large number of NPCs simultaneously calculating plans in real-time. Contrasting conventional implementations of GOAP, instead of using a dictionary of string keys and boolean values, we have a completely freeform and dynamic world representation through expressions.

In Autonomia, we've implemented several strategies to ensure the efficiency of this algorithm. We use the C# Task library to run the GOAP algorithm as an asynchronous task, allowing concurrent execution and avoiding the blocking of the main thread for more costly calculations. Our integration of cancellation tokens gives us the ability to stop planning tasks at will, providing control over the planning process. We've included a configurable maximum number of steps for each instance of the GOAP algorithm to prevent excessive resource consumption. Lastly, we minimize the search space and improve the heuristic function of the A* search by using an optimization we term “GOAP with Intended Uses”, for more information read section 4.2.14.2

In a specific iteration of Autonomia, we used an NPC's intelligence score to dynamically adjust the number of steps allocated to the A* GOAP algorithm. This added practical intelligence to NPCs, allowing them to allocate computational steps based on their perceived intelligence level.

These strategies collectively enhance the efficiency and adaptability of the A* GOAP algorithm within Autonomia, making it suitable for managing a dozen of NPCs in dynamic game environments.

3.6.2 Schedule

In our daily lives, we often follow a routine, with a predefined idea of how our day will unfold. This routine typically involves waking up, tending to morning rituals, commuting to work, putting in a nine-to-five shift, returning home, and perhaps enjoying some leisure activities before bedtime. However, there are days when our schedules vary, influenced by work commitments, health appointments, or unexpected emergencies. We possess the ability to manage our daily schedules and have knowledge of our plans.

Recognizing the significance of this aspect in shaping the believability of NPCs, we drew inspiration from the highly successful game title Red Dead Redemption 2 (RDR2) [81]. In RDR2, NPCs lead detailed lives, adhering to daily routines while also accommodating dynamic changes that can disrupt their schedules. To mirror this level of realism and adaptability, we introduce a class of goals known as "ScheduledGoals."

ScheduledGoals enable Autonomia's NPCs to incorporate schedule-based objectives into their behaviour, besides general purpose goals. This addition not only enhances the authenticity of the NPCs' actions but also allows for flexible adjustments in response to changing circumstances, contributing to a more immersive and dynamic game world.

Furthermore, the implementation of a pre-planned schedule for our agents serves as an effective strategy to reduce the need for real-time planning, resulting in significant performance savings. Essentially, this schedule acts as a form of "baked" plans, predefining the agents' activities and behaviours during their daily routines in a meaningful way. Using *ScheduledGoals*, NPCs can make social appointments, work at consistent hours, etc. This proactive approach not only enhances computational efficiency but also contributes to the seamless and immersive execution of agent behaviours within the game world.

3.6.3 The PlanBehaviour

In a realistic game world, it's reasonable to anticipate the frequent need for planning. Players and agents within the game often face context-specific options and decisions that require careful consideration. To facilitate this, we've introduced a behaviour called *PlanBehaviour* into our framework.

The *PlanBehaviour* takes a goal as its parameter and utilizes the default planner, which in our case is the GOAP algorithm and initiates the planning process within behaviour instance. Once the algorithm completes its calculations, the behaviour attempts to execute the generated plan. For example, consider a waiter who needs to find a clean plate to serve customers. In a dynamic game scenario, this task may require planning because clean plates may not always be readily available and might need cleaning first. This feature enables users of the framework to easily incorporate sub-behaviours based on specific goals within their own custom behaviours, allowing for more intricate, readable, and context-sensitive agent behaviours.

3.7 Compatibility with other AI models

In the ever-evolving landscape of AI and game development, the compatibility of AI frameworks with existing models holds immense significance. Autonomia, boasting a versatile architecture and a comprehensive array of features, is purposefully designed to seamlessly interface with other AI models while enabling them to be treated as a piece of knowledge within the game world. This compatibility empowers game developers to harness a blend of AI techniques, leading to the creation of more immersive and intricate gaming experiences.

Autonomia's commitment to openness and extensibility, we envision a future where collaboration within the game AI research community flourishes. Researchers and developers can build upon or even change for the better the Autonomia framework, specializing in their respective areas of AI expertise, and contribute to its open-source development. This collaborative approach fosters a vibrant ecosystem where each individual can make their unique contributions, ultimately benefiting the broader community of game developers and AI enthusiasts.

3.8 Scalability and Performance

The scalability and performance of a game's AI system are inherently tied to the algorithms in use and the complexity of the game world. Autonomia acknowledges this relationship and offers a versatile framework that adapts to the specific demands of each game. We can theorize of ways to make the system's most demanding mechanisms scalable by using cloud computing to separate each agent instance in different machines but Autonomia is far from that in its current implementation, so we will not elaborate more on that specific topic.

In the end, Autonomia is but a tool. It is designed as an open-ended framework, providing developers with a high degree of customizability to tailor it to their specific needs. In essence, Autonomia's flexibility and adaptability makes it a valuable tool for game development teams, allowing them to search for their desired balance between AI sophistication and performance optimization, ultimately delivering a compelling gaming experience.

3.9 The Autonomy Paradox

To deliver an immersive gaming experience, it's imperative to have non-player characters (NPCs) that exhibit realism, dynamic worlds that evolve based on unique gameplay, and a narrative that unfolds through scripted NPC actions. However, this presents a challenging paradox in video game development, which we named the "Autonomy Paradox."

The Autonomy Paradox encapsulates the dilemma faced by game developers: on one hand, they strive for NPCs to behave realistically and autonomously, responding to the player's actions and creating dynamic game worlds. On the other hand, there's a need for NPCs to adhere to scripted behaviours and specific narratives, limiting their "free-will" to ensure the progression and narrative of the game's storyline.

This paradox is at the heart of the framework's title, Autonomia ("*Αυτονομία*" in Greek). It symbolizes the delicate balance that Autonomia seeks to achieve by providing developers with the tools to create NPCs that can exhibit autonomy when required, yet also allow predetermined narratives and behaviours when essential for the game's storyline.

Chapter 4

4 Framework Implementation

The implementation of Autonomia commenced in November of 2022, and has evolved into two primary components to date. The two main parts are: a) the Autonomia.Core library, and b) the Unity Integration.

Chapter 3 explained the theoretical point of view of the system's architecture, where Chapter 4 aims to provide insights on its inner workings and technical details. Then we follow with Chapter 5, which explains the Unity Integration of Autonomia, which serves as: a) a use case for the framework's integration, and b) another starting point for fellow researchers to join our work. To ensure future collaboration and understanding, we've adhered to suitable design patterns in the codebase, promoting maintainability and readability.

Autonomia is openly available as an open-source project on GitLab [82], [83], fostering transparency and community engagement. We welcome and value constructive criticism and feedback from the research and development community.

4.1 Third-Party Tools

4.1.1 C#

The choice of programming language plays a pivotal role in the development of any software framework, and Autonomia is no exception. In selecting C# [84] as the foundational language for Autonomia.Core, we considered several key factors that align with our goals and objectives.

4.1.1.1 Seamless Unity Integration

C# is the primary programming language used within the Unity game development platform. Given our aspiration to seamlessly integrate Autonomia with Unity, adopting C# as the core language was a natural choice. This alignment enables Autonomia to operate harmoniously within Unity, simplifying the implementation process for game developers.

4.1.1.2 Versatility of C#

One of the foremost reasons for considering C# as a programming language for a new framework is its versatility. C# is a statically-typed, object-oriented language that can be employed in a variety of application domains. Whether the framework is intended for web development, desktop applications, mobile apps, and of course game development, C# can seamlessly adapt to meet these diverse requirements. With the advent of .NET Core, C# has transcended its Windows-centric roots and become a cross-platform language. Developers can build applications and frameworks that run on Windows, macOS, and Linux, making it an ideal choice for ensuring broad compatibility and reaching a wider audience. Also, C# boasts a robust standard library, the .NET Framework (or .NET Core/.NET 5+), which offers a comprehensive set of APIs for various tasks such as file I/O, networking, and data manipulation. This extensive library support accelerates framework development by reducing the need to reinvent the wheel, saving time and effort.

4.1.1.3 Strong Developer Community

Another compelling reason to choose C# for a new framework is the vibrant and engaged developer community that surrounds it. No one can doubt that C# developers benefit from a wealth of learning resources, including official Microsoft documentation, online courses, tutorials, and active forums. This wealth of knowledge facilitates the onboarding of new developers to the framework and aids in solving complex challenges. To further extend this point, the C# ecosystem is teeming with third-party libraries and tools that extend its capabilities. These resources can be leveraged to enhance the functionality of the framework and expedite development.

4.1.1.4 Strong Language Features

C# offers several language features that can greatly benefit framework development. C# enforces strong type checking and supports modern programming paradigms like object-oriented and functional programming. This leads to code that is more reliable, maintainable, and less error-prone—a crucial factor for framework longevity. Furthermore, C# features a sophisticated asynchronous programming model that simplifies concurrent operations, a key requirement for high-performance frameworks handling multiple tasks concurrently. To extend our point further, C# has evolved over the years with the introduction of features like pattern matching, local functions, and expression-bodied members. These additions make code more concise and expressive, enhancing developer productivity.

4.1.2 Neo4j

In our attempts to populate our world with information while developing and testing, we used Neo4j desktop [85] [86]. Neo4J is a popular and powerful graph database management system. It's designed to store, manage, and query data in the form of a graph, which is a data structure consisting of nodes and relationships which carry labels and properties. Since Autonomia is designed to work with graph databases, Neo4j saves us the trouble of implementing the entire system ourselves. For better performance, Autonomia does have its own internal graph database system but the disk storage for persistence happens through Neo4J.

In addition, Neo4j Desktop comes with powerful tools to help you create and examine your data. Following, we outline the tools included with Neo4J that we consider important for Autonomia.

4.1.2.1 Cypher Querying Language

Cypher is a powerful and expressive query language specifically designed for working with graph databases, with Neo4j being one of its primary implementations. It provides a way to interact with graph data by specifying patterns and operations on nodes and relationships within the graph. Cypher's syntax and semantics are tailored to the unique structure of graph data, making it efficient and intuitive for querying and manipulating graph databases. In Cypher, the most dominant feature at the core of its functionality is pattern matching. Patterns are defined using an expressive, readable syntax, representing nodes, relationships, and their associated properties. For example, a pattern like:

$$(user{:}\,User) - [{:}\,FRIENDS\_WITH] \rightarrow (friend{:}\,User)$$

would match a user node connected to another user node through a "FRIENDS_WITH" relationship. Nodes and relationships are the fundamental building blocks in Cypher. Nodes are enclosed in parentheses, and relationships are enclosed in square brackets. These nodes and relationships can have labels to categorize them and properties to store key-value data.

Cypher queries allow you to specify conditions for matching patterns and filtering results. You can use WHERE clauses to filter nodes or relationships based on their properties and employ various predicates such as "=" or "CONTAINS" to compare values. Queries in Cypher return data in a tabular format, making it easy to work with the results. You can specify which parts of the matched patterns you want to retrieve using the RETURN clause, enabling you to extract specific information from the graph.

One of Cypher's strengths is its ability to find paths in the graph, representing sequences of nodes and relationships that match certain criteria. This is particularly useful for traversing and analyzing complex graph structures.

4.1.2.2 Neo4j Bloom

Neo4j Bloom is an intuitive and visually driven data exploration and visualization tool designed to work seamlessly with Neo4j graph databases. It empowers users to interact with and gain insights from complex graph data without needing to write complex queries or code. With Neo4j Bloom, you can create interactive visualizations of your graph, explore relationships, and discover patterns in a user-friendly and intuitive manner. It's a valuable asset for both technical and non-technical users who want to harness the power of graph databases for data analysis and decision-making. In Autonomia, we value this tool as an inspector and canvas for the World State. A game designer can easily enrich the world of the game simply by using this tool, creating nodes, relationships and set handcrafted goals to the agents of the implementation.

4.1.2.3 Neo4j Driver for C#

For our C# - Neo4j communication, we used the Neo4j.Driver package. Neo4j.Driver for C# is a dedicated driver that enables C# developers to connect their applications with Neo4j databases seamlessly. It serves as a bridge between C# code and the Neo4j database server, allowing developers to perform various operations, such as querying the database, creating, or updating nodes and relationships, and retrieving results. It manages the database connection, enables Cypher Queries through C# with parameterization and all the above can be executed asynchronously using C#'s Tasks library.

4.2 Autonomia.Core

The Autonomia Framework has been developed under the Autonomia.Core namespace. By using namespaces, we aim to ensure flexibility for potential future implementations, offering improved clarity and versioning control. For instance, should an extension centered around emotional AI emerge, it could be neatly encapsulated within a hypothetical Autonomia. *EmotionalAI* namespace is maintaining a structured and organized codebase.

Moreover, the project is made available in two distinct formats: a) a library project containing the entire source code, and b) a prebuilt .DLL file (Dynamic Link Library). The latter plays a pivotal role in promoting decoupling of code, fostering modularity, and enhancing the overall efficiency of the Autonomia framework. The provision of dual-format availability empowers developers with the freedom to select the option that aligns best with their preferences and project requirements. Given that Autonomia is an ongoing project, it welcomes active participation and customization from the developer community.

Anyone is encouraged to modify the source code as needed to tailor it to their specific needs. Furthermore, should a developer create valuable enhancements or extensions to the framework, there is an open invitation to request integration into the public Autonomia repository. Contributions that bring substantial value to the Autonomia ecosystem are welcomed and can be considered for inclusion, promoting collaborative development, and fostering a robust framework for intelligent and autonomous systems.

In the following sections, we analyze and elaborate on technical details regarding multiple components that form Autonomia.

4.2.1 Graph

In the development of Autonomia, we employed an external graph database to store our data. However, during runtime, we recognized the importance of having a local graph database implementation for faster read/write operations. To ensure seamless compatibility, the Graph class has been designed to implement the *IDatabaseClient* interface and incorporates two dictionaries (or HashMaps) for nodes and edges, respectively. You can find the class diagrams for Graph, Node, and Edge in Figure 6.

It's worth noting that each element within the graph possesses a unique string value serving as its identifier, referred to as an "id". This unique id enables us to enhance the graph's performance by utilizing dictionaries not only for the graph itself but also for the nodes and memory queries. This optimization is especially beneficial, as graph databases may entail a multitude of edges for each node.

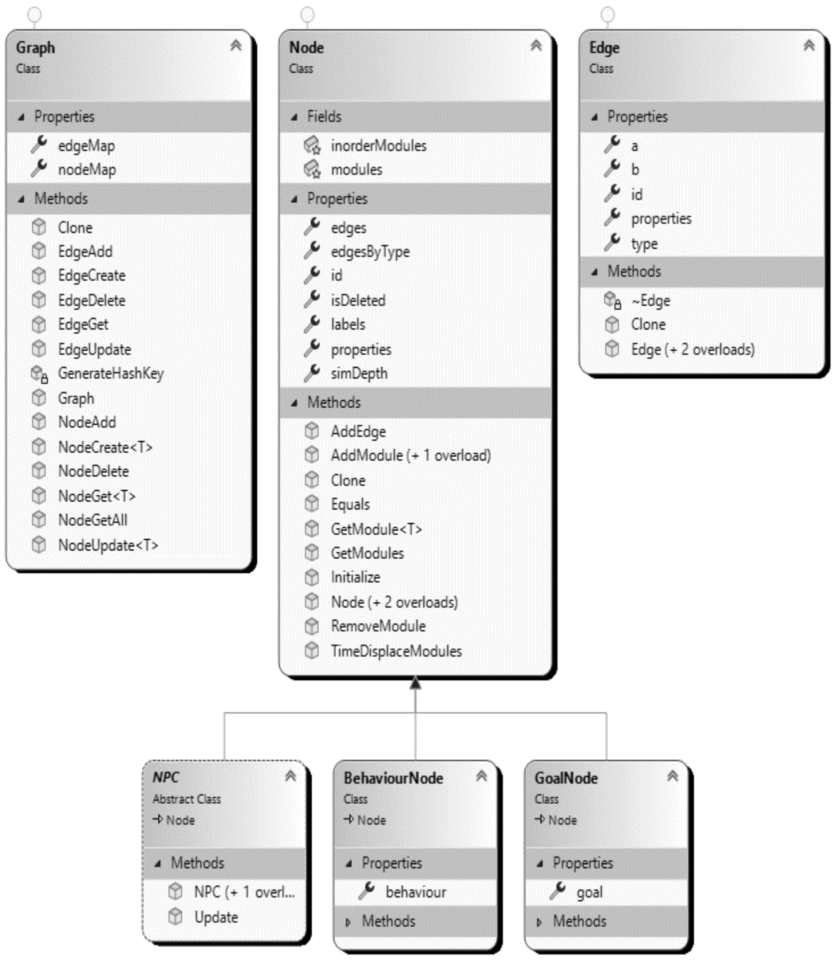


Figure 6*: Graph, Node and Edge class diagrams.*

Furthermore, it's noteworthy that we have integrated modules within the *Node* class, creating a tight coupling between them. This coupling enables us to include modules when cloning nodes, and it's essential to emphasize that the cloning of classes will be a common practice throughout Autonomia. Given our approach of treating each piece of knowledge as unique, cloning is a typical operation to preserve knowledge at the stage it was acquired. This will become increasingly evident as we proceed with our exploration of Autonomia's functionality.

For the cloning implementation, we predominantly utilized a variation of the prototype pattern that is designed to be compatible with inheritance. To elaborate, the process involves implementing a copy constructor for each subclass and overriding the Clone method to return a new instance of the subclass, passing as a parameter the current instance into its own copy constructor. This approach ensures that inheritance is seamlessly maintained while enabling the effective cloning of objects within Autonomia.

4.2.2 Injector and Injectables

To simplify the assignment of specific elements, we created an Injection mechanism. To achieve this, we created an Injector class who uses C#'s reflection mechanism, to find class types that contain custom Injectable attributes. Then, we use that injectable to find nodes which the overridden method "ShouldInject" returns true. Lastly, for those nodes we call the overridden "Inject" method.

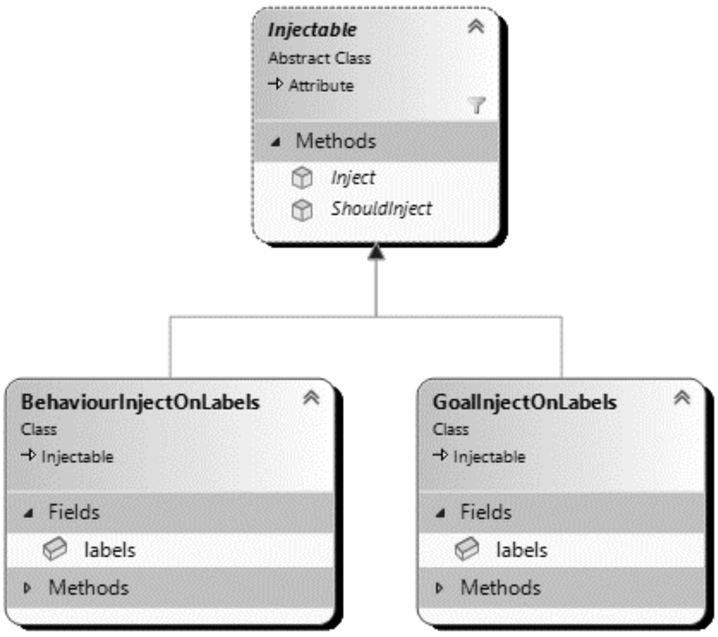
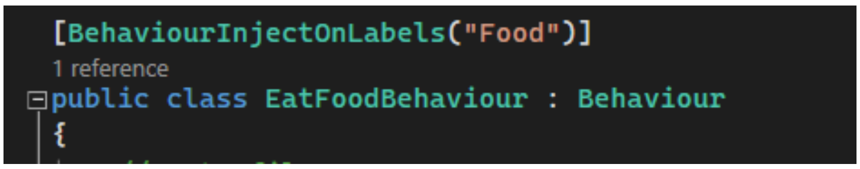


Figure 7: Injectable attribute class diagram

In Autonomia, we employ two primary Injectable attributes: a) *BehaviourInjectOnLabels*, and b) *GoalInjectOnLabels*. Both attributes accept string values as parameters and compare them with the labels associated with each node. When there is a match, these attributes inject themselves into the respective node.

In our implementation, we utilize behaviours as affordances, meaning that behaviours should be associated with nodes that provide them. For instance, an 'EatFoodBehaviour' should be injected into every node containing the 'Food' label. Thanks to our injection system, achieving this is straightforward and efficient, requiring only a single line of code, as illustrated in Figure 8.



```
[BehaviourInjectOnLabels("Food")]
1 reference
public class EatFoodBehaviour : Behaviour
{
```

Figure 8: Specific behaviour being injected using node labels

The implementation and extension of Injectable attributes are notably straightforward. For instance, we can create a special behaviour that should be exclusive to a particular NPC by defining a new injectable and using the node's ID or name as parameters.

Furthermore, this extensibility can be taken a step further by incorporating randomness into injectables, allowing us to introduce an element of uniqueness to our virtual world. Consider a scenario where a game designer wishes to distribute a specific piece of knowledge to a random subset of NPCs. This objective can be effortlessly accomplished using our injection mechanism. Importantly, this approach maintains a high level of decoupling, ensuring that it remains independent of, for example, the behaviour or goal class.

4.2.3 Node Factory

The *Node Factory* is an important singleton class in our framework. Although the entire functionality of our nodes are dictated by their labels, properties and modules, we still deemed it important to allow users of the framework to create classes that will handle initialization for each of their conceptual node types. This is important to consider when we consider modules that may have complex dependencies with each other. Furthermore, by having strictly defined types for our Nodes we can more easily classify them and group their behaviours. For instance, in Autonomia we use a common NPC Manager singleton to handle NPC updates. For this, we have the NPC class deriving from the Node class and implement in its initialization a) registering to the NPC manager and b) declaring the proper order of initialization for each module. Its class diagram is visible in Figure 9.

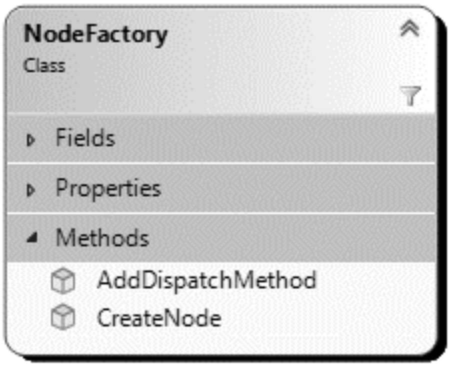
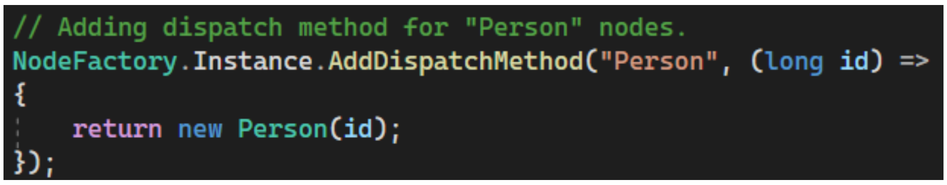


Figure 9: Node Factory class diagram

Node Factory is allowed to receive upon initialization, dispatch methods, namely callbacks, with label descriptors to enable users to imbue the factory with their own custom node types. An example of this can be seen Figure 10.



```
// Adding dispatch method for "Person" nodes.
NodeFactory.Instance.AddDispatchMethod("Person", (long id) =>
{
    return new Person(id);
});
```

Figure 10*: Example of adding custom dispatch methods to the Node Factory*

4.2.4 Modules

Modules are components that are attached to nodes and are used as data containers or offer additional functionality. First, we explain the base Module class and then move on to the implementation of some core modules. The Module class diagram and its derived classes belonging to the current version of Core are shown in Figure *11*.

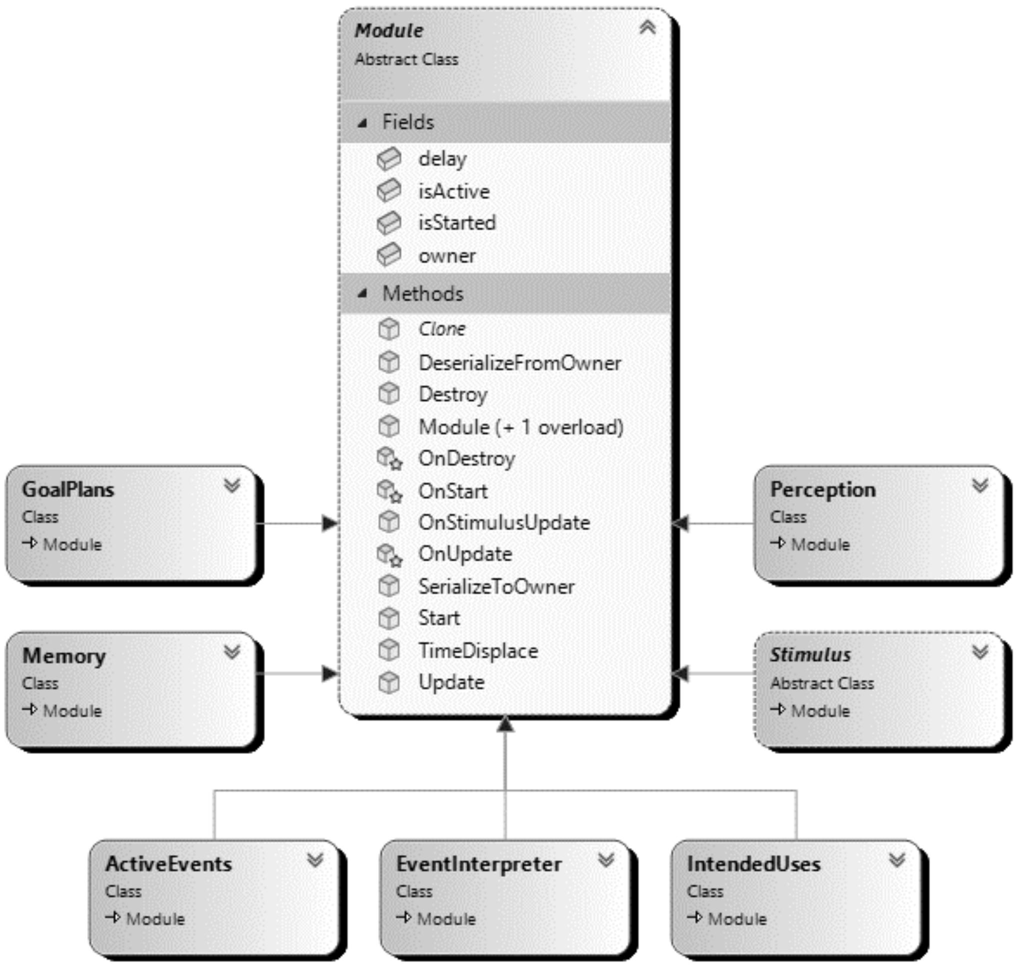


Figure 11*: Class diagram of the Module class and derived classes*

Each module within Autonomia follows a standardized structure, containing three key functions: *OnStart*, *OnUpdate*, and *OnDestroy*. These methods are called by the *ModuleManager* when appropriate. To enhance performance, it's possible to assign a specific delay value measured in milliseconds, thus preventing redundant per-frame calls of the Update function. Additionally, several other methods are important to the functionality of modules, and we explain each in the following subsections.

4.2.4.1 Serialize/Deserialize From Owner

Modules may need to serialize and deserialize their data for persistence, within the node's properties. For example, the *Memory* can serialize itself as in a JSON format, and upon saving the game, it would write this JSON string value as a property with a unique key, for instance "$$memory-module". Then, upon loading the game again, the memory module would try to deserialize from the node's properties to restore it’s previous state.

4.2.4.2 OnStimulusUpdate

The *OnStimulusUpdate* function may seem unconventional at first glance, so let's explore its intricacies. It's essential to grasp the concept that within a node's memory, there exists a simulated approximation of the world state, with other nodes and their respective modules.

To further illustrate this, consider a *Transformations* module containing the position and rotation of a Node in 3D space. Let's also introduce Node A, Node B, and Node B', where Node B' represents a simulated version of Node B within Node A's *Memory* module. Now, suppose Node B is in motion within 3D space. Its own Transformations module is configured to override the *OnUpdate* function to continuously update its transformations. However, if Node B happens to be out of sight from Node A, the Transformations module of Node B' should not undergo updates. This is precisely where *OnStimulusUpdate* comes into play.

For modules whose owner possesses a *simDepth* greater than zero, their update function is invoked solely when their corresponding stimulus calls for it. Furthermore, we can handle various subcases based on stimulus type. For instance, an Auditory stimulus might update Node B's position within Node B' with less precision, whereas a Visual stimulus would accurately reflect the update. Alternatively, if Node A is deaf, the Visual stimulus is disregarded entirely. This nuanced approach ensures that module updates are aligned with the prevailing stimulus conditions, enabling precise and context-aware processing.

4.2.4.3 TimeDisplace

The *TimeDisplace* function serves a critical purpose by enabling a module to advance its state based on a designated time variable. To illustrate, let's consider the *Needs* module, which is discussed in the Unity Integration. This module influences how an NPC plans and schedules its day. When planning what activities to pursue after work, the NPC's level of fatigue and energy, reflected in the Energy value, can impact the priority of goals, such as the Sleep Goal.

It's important to provide modules with the capability to shift their temporal state, particularly in scenarios where we are affecting only a clone of the node or module. This temporal displacement is essential for accurately simulating a variety of situations and their associated impacts on node behaviour.

4.2.5 Memory Module

The memory module is a simple, yet pivotal in Autonomia. It contains a relations variable of type IGraph for semantic data, and an IEventIndex for occurrences of

Events that were perceived by the NPC.

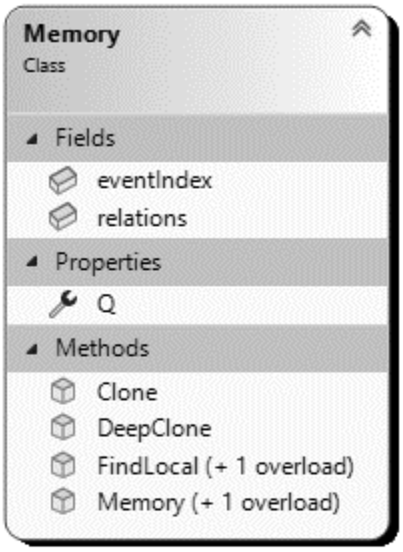


Figure 12*: Class diagram of Memory*

The main takeaway here is the *FindLocal* method. It is meant for other module to use it, in order to get references to the memory-local instance of a node. For example, let's assume an *EatFood* behaviour running with Node A as an actor, and Node B of type *Food* as its owner. In the authoring process of the behaviour, we would like to make Node A move to Node B's memory-local instance, since this represents the world in Node A's Theory of Mind. If Node B has been moved from its previous location, Node A should not be aware of it.

4.2.6 Perception Module

In the theoretical section dedicated to perception, we elaborate on why we deem this component to be of utmost importance, serving as a crucial link between an agent's memory and the external world. The quality and realism of our perception implementation directly impact the overall believability and effectiveness of our system. This module, is tightly coupled with Stimulus as it serves as an intermediate holder for sensed nodes and events. It is noteworthy, that the framework does not exclude any stimulus types from being implemented. The perception system would work seamlessly even with imaginary "six sense" stimulus. The main functionality is described as follows. During each update cycle, the perception module scans its sensed nodes and events from its stimuli. It follows a structured sequence of operations to ensure accurate and up-to-date information processing. To elaborate, for each stimulus the following process takes place.

Firstly, there is the need to update the modules of the owner within its own memory. This reflects the difference between the way we actually appear and the way we self-reflect, which is something also stored in our theory of mind. So, the *OnStimulusUpdate* method is invoked for every module of the owner's memory-local copy.

Following this, for every sensed node, the perception module locates the memory-local replica of the node and invokes its *OnStimulusUpdate* methods. This ensures that any perception-based information or modules associated with the sensed nodes are promptly updated.

Lastly, the perception module processes each sensed event. Again, it transforms the event into a memory-local event, interprets it using the *EventInterpreter* module, and attempts to match it with any subscribed event. In the occurrence of a successful match, the subscribed callback function is invoked with the memory-local event as its parameter. This mechanism enables the dissemination of relevant information to subscribed modules or components and ensuring the memory-local copy is always used to ensure consistency and believability for an agent's actions.

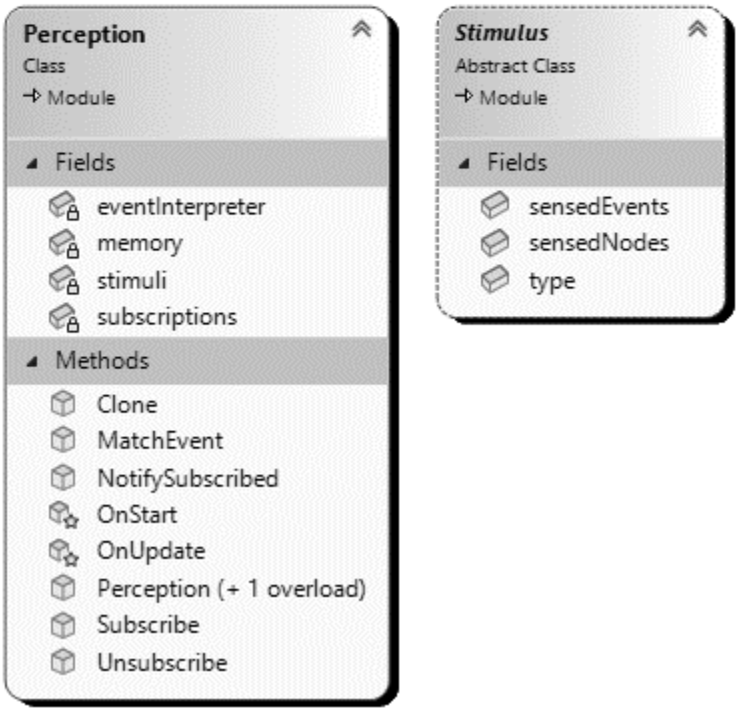


Figure 13*: Class diagram of the Perception and stimulus modules*

This subscription mechanism allows interested parties to receive notifications whenever a particular event occurs. To illustrate this functionality, consider a scenario involving a *TavernWaiter* behaviour.

In this scenario, the *TavernWaiter* behaviour has the actual waiter (an NPC) as its actor, and the tavern as its owner. The behaviour subscribes to a perception event with the description "<<Abstraction of Person>> <<enters>> <<owner>>." In simpler terms, this event signifies "If someone enters the tavern." By subscribing to this event, the behaviour can respond appropriately whenever a person enters the tavern, demonstrating the framework's flexibility in handling dynamic in-game scenarios.

4.2.7 Other Modules

In this section, we will provide brief insights into several modules within Autonomia that primarily serve as data containers or are in the process of being implemented and refined. These modules, while important, may not require in-depth technical discussion in this context, as their primary function is to store data or are subject to ongoing development and improvement.

- **GoalPlans:** This module is designed to store calculated plans and associate them with their respective goals, essentially serving as a method for "baking" behaviours. It can potentially exclude previously failed plans to prevent the agent from repeating them in the future, either due to experience or caution.
- **ActiveEvents:** The ActiveEvents module contains the list of all events currently affecting its owner. This allows the module to expose these events to other nodes' perception systems.
- **IntendedUse:** This module serves as an optimization mechanism for the A* search algorithm, contributing to enhanced search efficiency. Its role and significance are further discussed in the A* Search Algorithm section and the Unity Integration, including its extension in the designer tools.
- **EventInterpreter:** Currently, this module's implementation is relatively simple, primarily focused on type matching to interpret events. However, in the future, it will be refactored to use a hierarchical model for event matching and interpretation. Additionally, it will address the challenge of reinterpreting events as the agent learns more about them, offering intriguing possibilities for future implementations.

4.2.8 Events, Abstractions and Wildcards

4.2.8.1 Event

As discussed on Chapter 3, events are used to represent “something that happens". We model an event as having a string type and two nodes; an actor and a subject. Furthermore, each event can carry any additional value using its "parameters"; this variable is a custom *DynamicParameters* class which can be seen as a dictionary of string keys and object values, but with extra logic for comparisons. Lastly, each event has a list of strings named "sensedFrom". This represents the stimuli types this event can be - as the name suggests - sensed from.

One crucial aspect of the *Event* class is its capacity to use abstractions and wildcards for event matching. This feature enables the framework to establish relationships between events that share common patterns. To showcase the usefulness of this feature, which will also be used later on with expressions, we give the following examples:

- **E1:** John (actor), grab (type), abstraction of Item (subject)
- **E2:** John (actor), grab (type), abstraction of Sword (subject)
- **E3:** John (actor), grab (type), Moonlight Sword (subject)
- **E4:** John (actor), grab (type), Turtle Shield (subject)

In the above examples, E1 can be matched with E3 and E4, because both the sword and the shield are items, but E2 can be matched only to E3, since it reduces the abstraction to only swords.

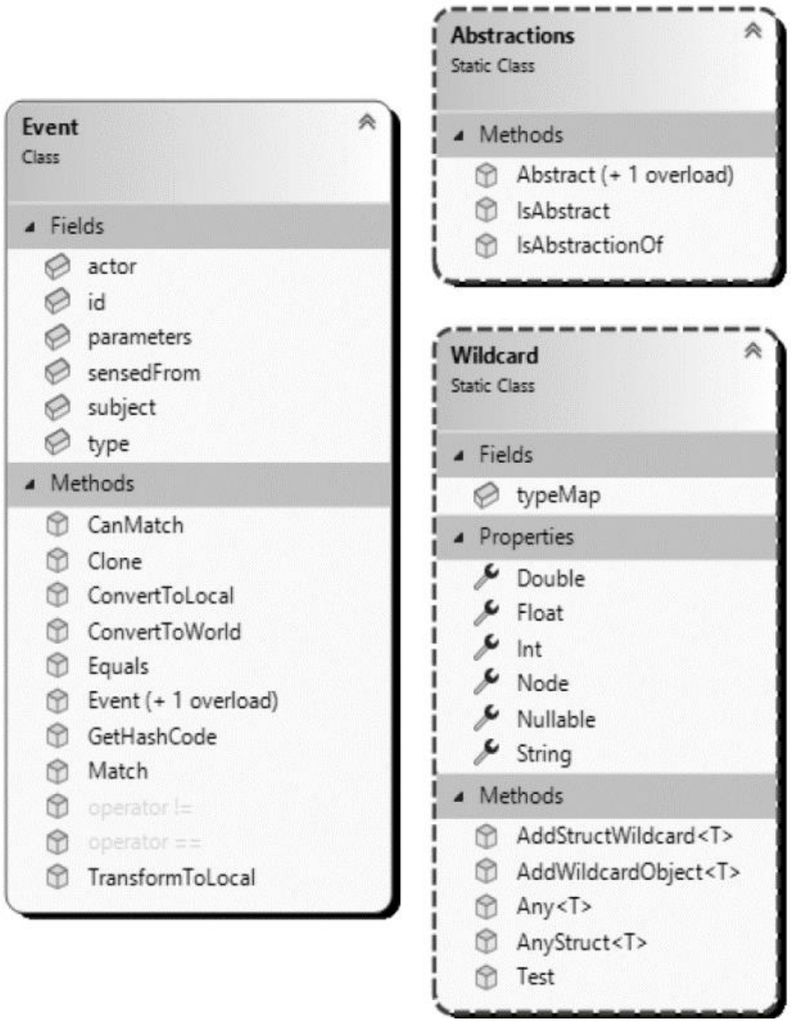


Figure 14*: Class diagram for Event, Abstractions and Wildcard*

4.2.8.2 Abstractions

The *Abstractions* static class has three main methods:

- **Abstract:** which receives an array of labels as parameters and returns a new node containing those labels. Abstract nodes have their own unique, randomly assigned id and carry the extra “Abstract” label.
- **IsAbstract:** Returns true if the given node is abstract.
- **IsAbstractOf:** Receives two parameters, Node A and Node B. It returns true if Node A is an abstraction of Node B, which in effect means if Node A’s labels are an inclusive subset of Node B’s labels.

4.2.8.3 Wildcard

Wildcard is another static class which contains definitions for unlike values of specific types to be represented that this value can be anything. For example, a *Wildcard.Int* returns the minimum possible value an integer can have.

4.2.9 AutonomyDB

Within our framework, we introduce the *AutonomyDB* singleton class, a fundamental component responsible for the storage and retrieval of our world state from a persistent graph database. The *AutonomyDB* leverages the *IDatabaseClient* interface, providing the flexibility for users to employ custom databases of their choice, should an alternative be preferred.

In our specific implementation, we have utilized the *IDatabaseClient* interface with the Neo4jClient, enabling seamless communication with our Neo4J database. A visual representation of the interface's class diagram can be found in Figure 15, thoughtfully organized to encompass fundamental database operations.

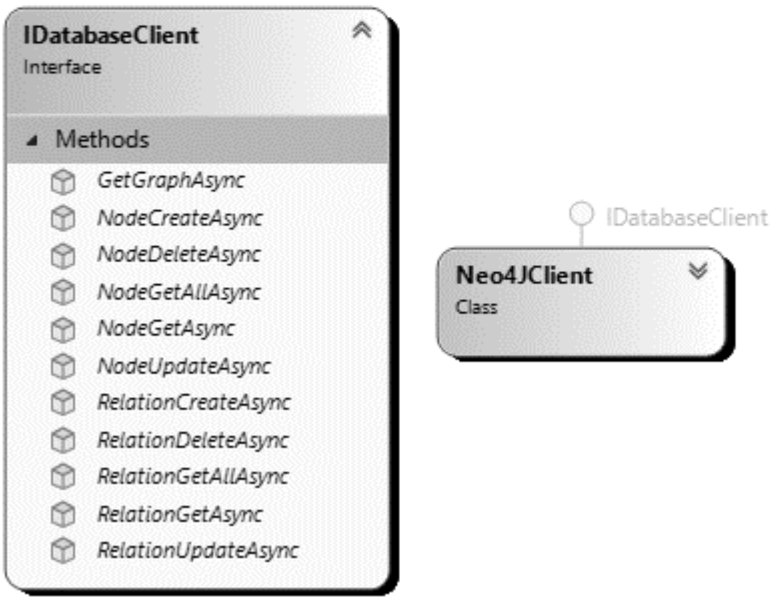


Figure 15: IDatabaseClient interface diagram with our Neo4JClient.

When *AutonomyDB* is asked to undertake the task of loading data from the database, it executes several critical operations. Initially, it parses the query response and populates the world state using the *NodeFactory* to produce nodes. It is noteworthy that the world is represented as a literal Memory module (refer to Section 3.2.3). Next, *AutonomyDB* leverages the Injector class (4.2.2) to inject each injectable item to its corresponding target. Upon the conclusion of this step, each Node has been assigned its injectable modules, behaviours, and defined goals.

It's important to note that, at this point, every node type possesses the option to override an *Initialize* method, allowing for specialized initialization procedures. At this moment, we can also begin initializing custom NPC memory. In our testing scenarios, we have employed a cloning strategy by replicating the world state and assigning a clone to the memory of each NPC. It's worth mentioning that we have the capability to implement recursive Memories, enabling a more comprehensive Theory of Mind. If desired, the described process can be repeated for each layer of *Memory* modules.

Furthermore, various filtering mechanisms can be applied to simulate more realistic scenarios. For example, instead of creating a perfect clone of the world state, we can introduce an elimination step for 'unrelated' nodes or even incorporate randomization of metadata within the *Memory* of each agent, adding an element of variability to their cognitive processes.

4.2.10 Engine

To simplify future implementations with Autonomia, we have introduced an *Engine* class that serves as the central driver for all framework functionalities. Figure 16 provides an overview of the primary functions it offers.

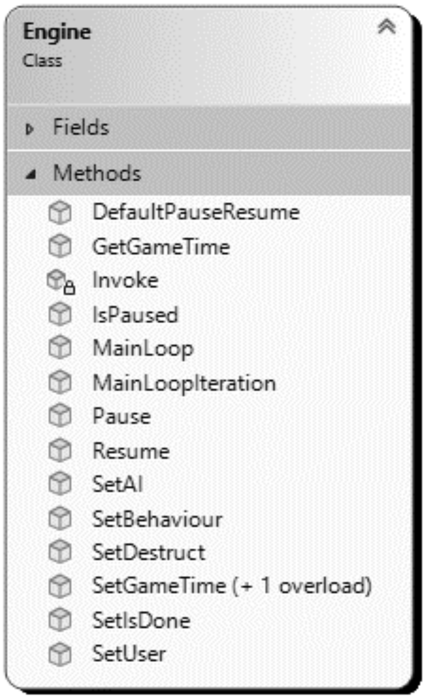


Figure 16: Engine class diagram

One of the key functions is the *MainLoopIteration*, acting as the heartbeat of Autonomia, regulating its processes. It's important to note that while the Engine class is optional for utilizing Autonomia's capabilities, we have designed it to facilitate the learning and adoption of the system, offering a more accessible entry point for developers.

4.2.11 System Clock

The *SystemClock* class is a fundamental component of the Autonomia framework, responsible for managing time-related operations. It serves as a centralized timekeeping mechanism for coordinating actions and behaviours of autonomous agents within the framework's simulated environment. This class measures time in milliseconds using a value of type long. This would mean, we could have a maximum of around 10^{11} days or 273.790.926 years in a 64bit system.

An advantage of using milliseconds as a time unit in our framework is the fact that integrations of it can implement any custom time system they prefer, simply by creating a casting mechanism between *SystemClock* and their time system. This can be seen in our Unity Integration, where the time system inherits a 24 hour per day, 365 days per year counter.

4.2.12 Expressions

The *Expression* class as of today has been formed through many iterations while implementing its respective designer tools through the Unity Integration. The finalized class diagram is shown in Figure 17.

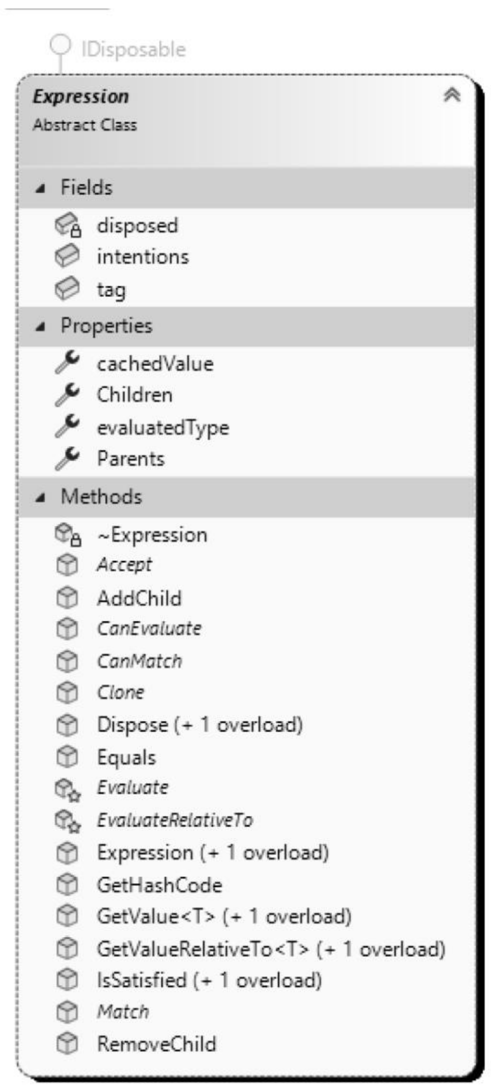


Figure 17*: Class diagram of Expression*

Expressions within the Autonomia framework serve as crucial tools for building and managing complex chains of logic and evaluations. These expressions are primarily used as preconditions and effects for behaviours and goals, providing a versatile mechanism for specifying agent actions and conditions. Each expression needs to implement a set of methods to ensure its proper functionality:

Table 1: Expression base class methods

Method	**Description**
CanEvaluate	Determines if the expression is syntactically ready for evaluation
Evaluate	Computes the value of the expression and caches it for future reference
EvaluateRelativeTo	Similar to Evaluate but performs computations relative to an observer
CanMatch	Checks if the expression can match a given parameter expression
Match	Matches the current expression to a parameter expression, assuming compatibility
IsSatisfied	Determines if the expression is currently in effect
IsSatisfiedRelativeTo	Similar to IsSatisfied but considers an observer

The framework provides a uniform *GetValue* method that invokes *Evaluate* and returns the cached value if applicable. Directly accessing the cached value is possible but recommended only if *Evaluate* was called within the same scope.

A full set of implemented expressions are given on the Unity Integration chapter, serving as a practical guide to showcase how these expressions can be effectively utilized within the Unity game engine, allowing developers to harness their power to design complex and dynamic NPC interactions and behaviours. On top of that, on the Unity Integration we implemented a node-graph tool for creating expression graphs.

It is worth noting that using C#'s hashsets is a common practice for storing expressions rather than lists in our framework. Hashsets provide faster operations at the cost of increased storage requirements. However, to ensure compatibility with our matching mechanism, two specific requirements must be met: a) a custom

ExpressionComparer, b) the *GetHashCode* method must be overridden to return a hash code based on the evaluated type. This is a logical step as we will be performing matching operations on expressions of different classes that share the same evaluated type. This will be shown in chapter 5.

4.2.13 Behaviours

The way behaviours are used in the system has been discussed in Chapter 3. In this chapter we will elaborate on the implementation of behaviours that allow them to be easily authorable and how we recommend them to be structured.

4.2.13.1 Behaviour States and Events

Behaviours contain two simple enum classes which are essential to their functionality. First, we have the behaviour's states which can represent if a behaviour is currently running, is paused, stopped or if it is in a persistent state. The persistent state is a special case for behaviours marked with the ability to be persistent. When those events mark their completion, they do not stop running, but as the name suggests, persist while other potential behaviours follow.

Next, we have five possible events a behaviour can signal. Either when a behaviour starts, updates, stops, completes, or fails two things will occur: a) an notify function will invoke subscribed callback methods of the specific event, essentially working as listeners, and b) a TaskCompletionSource value dedicated to that event will have its result set.

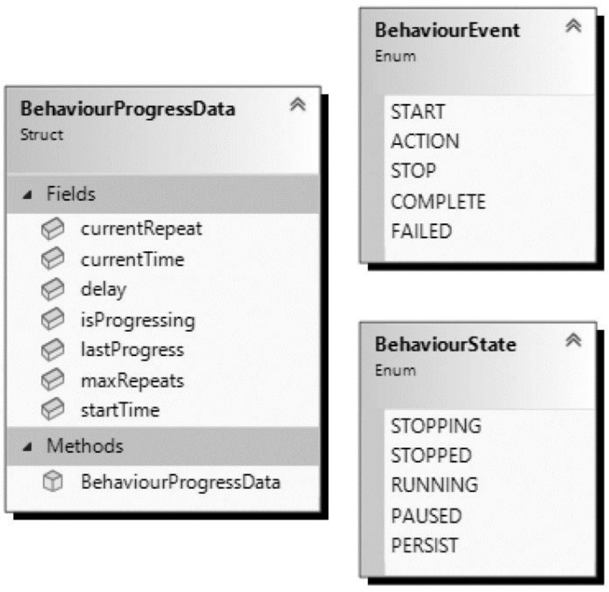
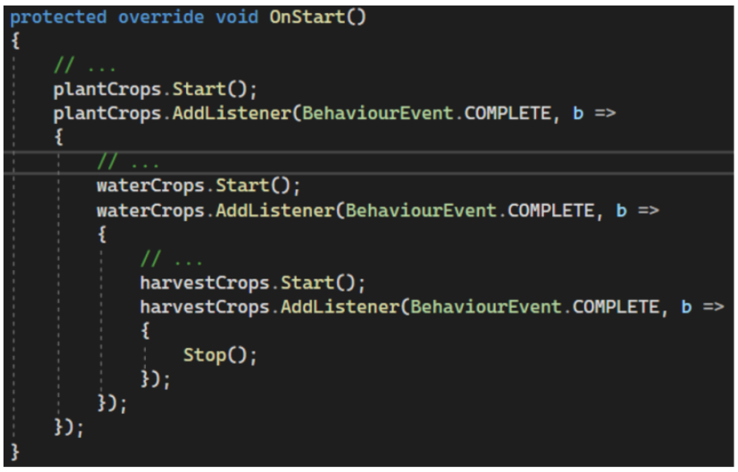
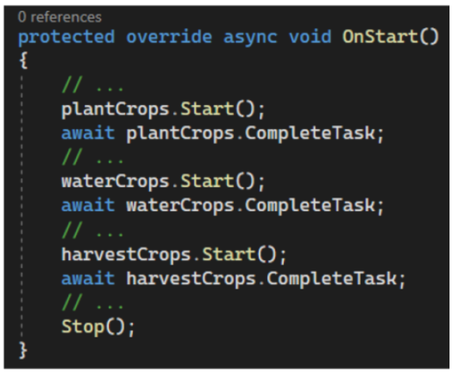


Figure 18: Class diagrams of behaviour's metadata

In the context of asynchronous programming in C#, a *TaskCompletionSource* is a valuable mechanism. It represents a task that can be manually completed, typically when you have asynchronous operations that aren't directly represented by a *Task*. This way, we allow a more readable form of interactions between behaviours. In Figure 19, you can see a side-by-side comparison of the two methods using as an example a FarmWorkBehaviour, where the objective is to plant, water and then harvest your crops, assuming each task is its own behaviour. In our experience, we found the asynchronous programming method to be significantly more readable when writing more complicated behaviours since the AddListener technique will tend to create a nested chaos of callbacks.



```
protected override void OnStart()
{
    // ...
    plantCrops.Start();
    plantCrops.AddListener(BehaviourEvent.COMPLETE, b =>
    {
        // ...
        waterCrops.Start();
        waterCrops.AddListener(BehaviourEvent.COMPLETE, b =>
        {
            // ...
            harvestCrops.Start();
            harvestCrops.AddListener(BehaviourEvent.COMPLETE, b =>
            {
                Stop();
            });
        });
    });
}
```



```
0 references
protected override async void OnStart()
{
    // ...
    plantCrops.Start();
    await plantCrops.CompleteTask;
    // ...
    waterCrops.Start();
    await waterCrops.CompleteTask;
    // ...
    harvestCrops.Start();
    await harvestCrops.CompleteTask;
    // ...
    Stop();
}
```

Figure 19: Side-by-side comparison of recursive listener events versus using asynchronous programming.

4.2.13.2 Behaviour Authoring

Behaviour has a plethora of methods that can be overridden to alter and control their functionality. We name five essential methods of overriding:

Table 2: Behaviour base class core methods

Method	**Description**
OnInitialize	This method is called when the behaviour is injected to a *BehaviourNode* and is assigned its owner. It can be used to cache values for later use and is as it name suggests, used for initialization purposes.
OnStart	Computes the value of the expression and caches it for future reference
OnAction	This method is called right when the behaviour is asked to begin its functionality. At this point, we assert that the actor of the behaviour has been assigned.
OnStop	This is the ticking mechanism of a behaviour. For each behaviour, action is called every “data.delay” milliseconds.

	Shorter delays are meant for more interactive, real-time behaviours like fighting and combat, where we can use larger delays for more passive behaviours like thinking, sleeping or eating. The delay can also be used to simulate reflexes of an agent. Possibly some future integrations could allow more dexterous agents to use shorter delays and be dominant in a fight thanks to that.
OnGetCost	This method is called when the behaviour is asked to stop. In our previous *GrabBehaviour* example, we use the *OnStop* method to release the respective item from the agent's grasp
OnGetTime	When overriding this method, a behaviour can return its representative value, meaning how effective this behaviour is for the agent at the current moment. For example, an *EatFoodBehaviour* would return a higher value if the agent is currently hungry
OnSetActor	This method should try and return a time estimate, the behaviour will take. For example, a *TavernWaiter* behaviour could return a time estimate of eight hours (converted in milliseconds), to allow the schedule module to plan an agent's day
OnSetOwner	These optional overrides allow custom user code to run when the assignment of an actor or an owner occurs. For example, we can override the *OnSetActor* method, to iterate and replace the previous actor for every expression in preconditions and effects with the current actor

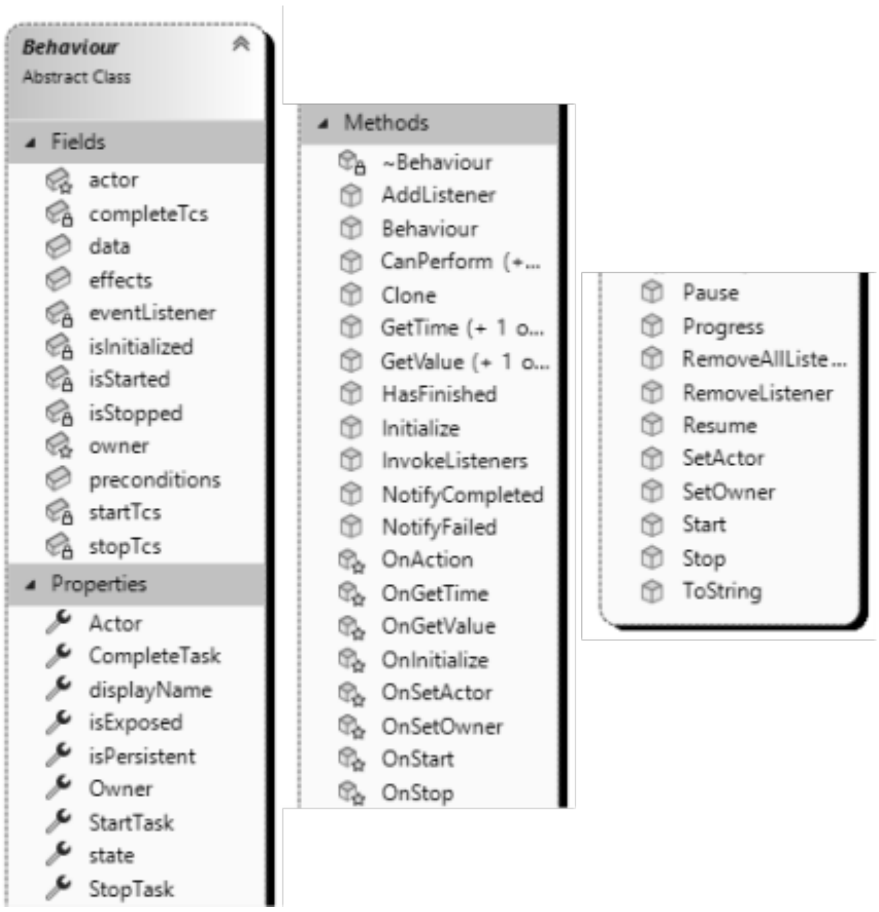


Figure 20: Class diagram of Behaviour

4.2.14 Goals and Planning Algorithms

4.2.14.1 Goals and Plans

The implementation of goals as mentioned in Chapter 3 is straightforward and simple. Its class diagram is shown in Figure *21*. It also proved helpful to create a *Plan* class which can encompass a sequence of behaviours in the context of a specific goal, as the name suggests. The *Plan* class also enables us to define the value of a plan, being a combination of the goal's value and the sum of the behaviour's values.

This is conceptually reasonable, and programmers can find their own balance between the importance of valuable behaviours and goal. For instance, let us assume the goal of "Climb Mountain" and a plan that can satisfy that goal. It may not be effective to walk to the top of a mountain because it is either dangerous, or far away, but the final value is analogous to how important it is for the agent to climb the mountain and see the view. This is further expanded by getting the time cost of a plan, enabling us to plan accordingly.

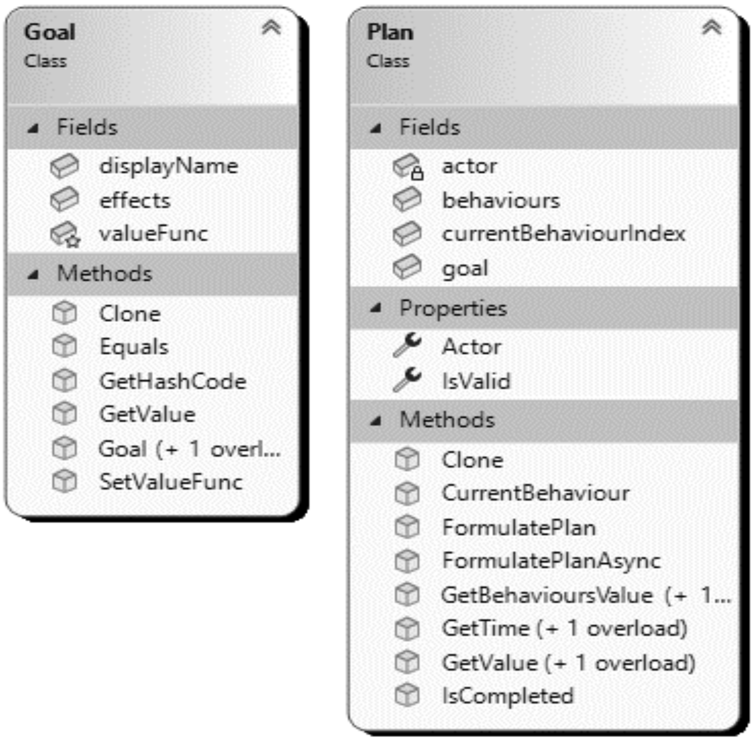


Figure 21*: Class diagram of Goal and Plan.*

4.2.14.2 A* Search and the IntendedUses optimization

In the context of Goal-Oriented Action Planning (GOAP), A* search is an essential algorithm for finding the most optimal sequence of actions to achieve a specific goal. A* search begins with an initial state representing the current state of the world, including information about objects, conditions, and resources. It explores the space of possible actions by applying each action to the current state, generating a new state as a result. A* uses a heuristic function (typically denoted as "h") to estimate the cost from the current state to the goal state. This cost evaluation is often denoted as "g." The algorithm combines "h" and "g" to calculate an "F" value for each state, where F = g + h.

States are organized in a priority queue based on their F values. A* selects the state with the lowest F value for expansion. It checks at each step whether the current state satisfies the goal conditions. If so, the search terminates, and the sequence of actions leading to this state is the optimal solution. A* continues expanding and backtracking through states, generating new states by applying actions until either the goal state is reached, or the queue is exhausted.

A* guarantees optimality when it has a consistent heuristic (h) and no infinite costs are present. It is also considered complete, meaning it finds a solution if one exists within a finite state space. In the context of Autonomia, A* search helps identify the most efficient sequence of behaviours to achieve a specific goal by considering both the values of behaviours and the goal effects.

A* is known for its exhaustive nature, and there exist numerous optimization strategies to enhance its performance. One fundamental optimization involves shifting the search approach from progression to regression. Instead of starting from the initial point, the algorithm commences from the goal point and backtracks. This modification significantly improves performance; however, it may lead to plans that feel mechanical and lack creative elements. Moreover, the time and space complexity of A* are directly tied to the size of the search space. As the number of available behaviours increases, the algorithm's execution becomes exponentially slower and demands more computational resources.

To further optimize the algorithm, we propose a novel strategy we termed "GOAP with Intended Uses" within our framework. In our context, behaviours are linked to nodes, and nodes can conceptually represent a wide range of entities or concepts. Our optimization capitalizes on the observation that most things have an intended range of uses. For instance, a fork is primarily designed for eating, rather than for aggressive purposes, although it could theoretically be used in such a way. To implement this approach, we introduce the "IntendedUse" module, which attributes intentions coupled with values to nodes. Subsequently, each behaviour associated with a node inherits these intentions.

Now, when we initialize our A* search, each agent searches its memory to identify behaviours that align with the intention of the goal, significantly reducing the available search space for the algorithm. Additionally, we employ the matching intention value as our heuristic value, streamlining the search process and further optimizing the efficiency of the A* algorithm within our context. Additionally, now game designers have control over behaviours that can be prioritized over others. If a game designer wants to attribute a higher value to cooking, rather than eating a fruit, he can tweak the intention values to allow the cooking behaviour to overpower the rest, but still have other behaviours available to fallback, if the cooking behaviour for any reason cannot occur.

Lastly, we deem noteworthy to highlight some key elements of Autonomia’s A* GOAP. In this implementation A* nodes represent sets of expressions. This means that both our precondition and effects fall into the procedural side of GOAP. This contradicts common approaches which use string-boolean or string-object pairs, known as blackboards. Expressions can take any form; it would even be possible for an expression to query the Memory module for more information during the search, compromising our algorithm’s performance. This is always configurable, and each developer can use the system for their own needs. We just want to highlight the freeform power of expressions.

By integrating the planning methodology described above with our dynamically evolving environments, we believe that our agents can exhibit a remarkable degree of autonomy and controlled unpredictability, enriching the overall user experience in a positive manner.

4.2.14.3 The ScheduleDay algorithm

The schedule module utilizes the GOAP algorithm to schedule the day of an NPC, based on its understanding of the world and expected state of things at given times. To achieve this, the schedule can be thought of as a calendar. It implements a *SortedDictionary* of long type keys representing days within the calendar and lists of *ScheduledSlot* as values.

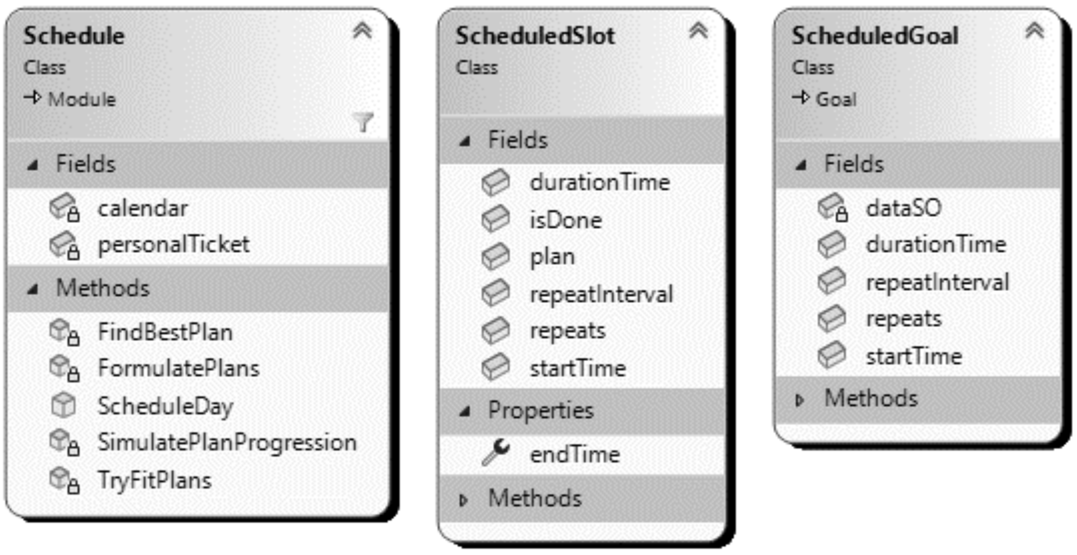


Figure 22: Class diagrams of Schedule, ScheduledSlot, and ScheduledGoal

4.2.14.4 Scheduled Behaviours vs. Other Behaviours

First, let us assume we have our daily scheduled and contained in scheduled slots within our calendar. How do we begin the execution of those behaviours and how do we solve conflicts with other demanded behaviours? As discussed in the BehaviourController module section, every module attempting to take control needs to compete with others, measuring out their importance values. So, when the schedule module detects that the NPC should execute a planned behaviour, it tries its personal ticket through the behaviour controller module to assert control.

4.2.14.5 ScheduleDay Algorithm

With what we have already established, scheduling an agent's day is not such a complicated task. We can simplify the algorithm into three main steps: a) formulate plans, b) add *ScheduledGoal* plans to calendar, and c) try to add other plans on calendar respecting what has already been planned.

In the first step, we use our GOAP planner to formulate plans for all our goals. Depending on that stage our game is, we can even replan older instances to make sure they are still consistent and reasonable. Then, we iterate through our plans and keep a reference to all plans referring to a *ScheduledGoal*. Then we create a *ScheduledSlot* for this plan based on the estimated time the Plan returns from its behaviours.

Finally, we may have other goals that do not need to be met at a specific time slot, but it would be appropriate to try to schedule them on our agent's day. For example, in the evening our agent may expect that he will be hungry, so he may have already planned to grab lunch by then. This is the most sophisticated part of the *ScheduleDay* algorithm, so let us provide simplified pseudo code for its algorithm.

Chapter 5

5 Unity Integration

The decision to embed Autonomia within a game engine was not taken lightly but was, in fact, a strategic choice that has proven to be pivotal. It has allowed us to uncover subtle intricacies and requirements that only real-world scenarios can reveal. Through this practical approach, Autonomia has evolved into a framework that not only meets but tries to anticipate the demands of modern software development.

5.1 Overview

Unity is a game engine that allows developers to create interactive and immersive games for various platforms and devices. Unity is widely used by both indie and AAA game studios, and some of the popular games made with Unity [87] include Among Us [88], Fall Guys [89], Hearthstone [90], Ori and the Blind Forest [91], and more. In this section we explain why we have chosen Unity and why we believe it to be a powerful, versatile, and user-friendly game engine that offers many advantages for framework implementation.

For starters, Unity is a game engine that supports high-quality graphics, physics, audio, and animation for creating realistic and engaging games. Unity enables developers to achieve impressive performance and optimization across platforms, as it uses a low-level rendering API called Scriptable Render Pipeline (SRP) that allows developers to customize the rendering process according to their needs. Unity also offers a variety of features and tools that enhance the game development process, such as scripting, asset management, debugging, testing, etc. In addition, Unity provides an integrated marketplace that enables access to thousands of ready-made assets, such as models, textures, sounds, scripts, etc. that can be used for free or purchased.

Another advantage of using Unity is that it is a versatile game engine that supports cross-platform development and deployment, allowing developers to create games that can run on various devices and platforms, such as PC, mobile, console, web, AR, VR, etc. Unity also offers a range of features and tools that facilitate the integration of various technologies and services, such as cloud, analytics, monetization, multiplayer, etc.

Last but not least, Unity is a user-friendly game engine that is easy to learn and use for both beginners and experts. Unity has an intuitive and customizable interface that allows developers to work efficiently and comfortably. Unity also has a rich documentation and tutorial system that provides comprehensive and clear guidance on how to use the engine and its features. Also important for us was the fact that Unity has the capabilities to freely extend and program its Editor, allowing developers to extend the functionality of the engine by creating custom tools or plugins. This plays a pivotal role in the following sections.

5.2 Autonomia’s Designer Tools

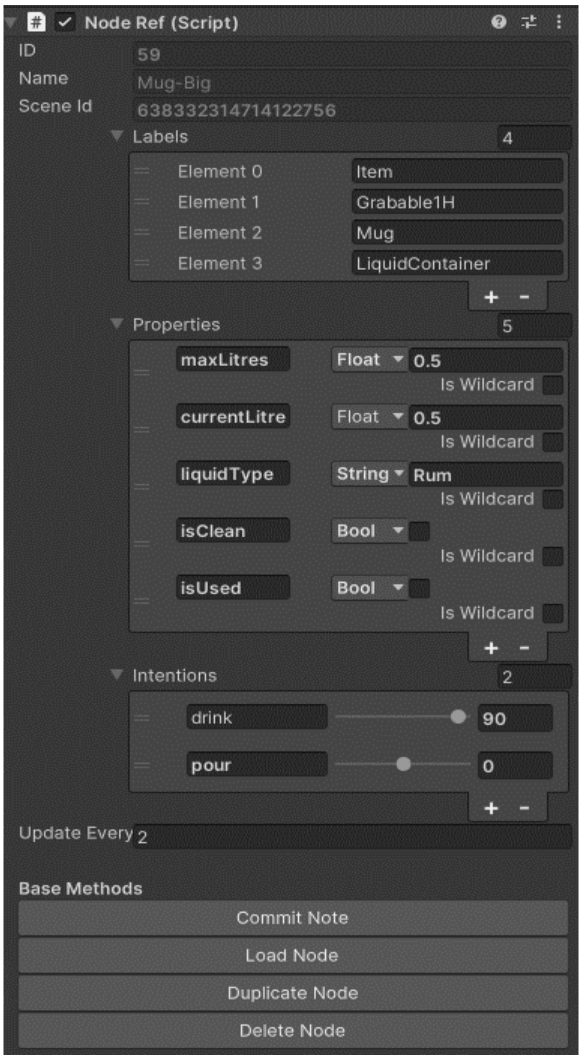
During the design process of Autonomia, a clear distinction was made between two primary working stack layers: the designer layer and the developer layer. Designers play a crucial role in envisioning and crafting the desired AI behaviours, while programmers are tasked with the responsibility of translating these envisioned behaviours into functional implementations.

Designers require the ability to swiftly construct and modify game environments and establish the connections between various behaviours and goals. The main tool at their disposal in this endeavor is the expression component of the EBG model.

5.2.1 NodeRef Script

In Unity's game scene, every element needs to inherit from the *MonoBehaviour* class. This requirement sets the foundation for integrating game objects seamlessly into Unity's framework and its main game thread. To bridge the gap between Unity's game objects and our graph database nodes, we introduced a *NodeRef* class. Early in the development process, it became evident that designers required control over the world-state, as well as the ability to expand it within Unity. Without the proper tools, every mundane item in a level would have to be manually created in the respective graph database and be associated with a node's ID, a cumbersome and unnecessary task.

To address this issue, we utilized Unity’s editor capabilities. By implementing a custom inspector for the NodeRef class, we enable a wide range of functionalities. These include the creation of nodes, loading from existing IDs, and the ability to modify labels, properties, and even custom module properties. As illustrated in Figure 23, this includes features such as 'intended uses'.



Node Ref (Script)

ID 59
Name Mug-Big
Scene Id 638332314714122756

Labels 4

Element 0	Item
Element 1	Grabable1H
Element 2	Mug
Element 3	LiquidContainer

Properties 5

maxLitres	Float	0.5	Is Wildcard
currentLitre	Float	0.5	Is Wildcard
liquidType	String	Rum	Is Wildcard
isClean	Bool		Is Wildcard
isUsed	Bool		Is Wildcard

Intentions 2

drink	90
pour	0

Update Every 2

Base Methods

Figure 23: NodeRef drawer example

The addition of this feature significantly streamlines the level design process. In our recommended pipeline, we've converted a library of assets into unity prefabs, each of which contains a *NodeRef* script with a) unsigned ID, b) predetermined labels, and c) properties that describe this object class. For instance, in our case study all tables contain the labels “Item”, “Table”, “DropArea”. After placing all the objects as per the environment design, with a single button click we can create and commit all the nodes in the database. Then, each *NodeRef* can be further customized to include additional labels, properties, or different intended use values.

5.2.2 DesignerValue

In the Unity integration of Autonomia, it is often required to assign key-value pairs directly within the Unity editor, a feature not readily available out of the box. To address this, we introduced the "DesignerValue" class, which allows the assignment of common data types such as int, long, float, and string to designated keys. Additionally, it includes an "IsWildcard" checkbox that, when activated, assigns a predefined Wildcard value of the respective data type upon initialization. This approach simplifies the configuration of properties and settings, seamlessly integrating with Unity's editor and providing an efficient means for developers and designers to customize values without manual code adjustments.

5.2.3 Intended Uses Drawer

During the development of our use case, we discovered that the "Intended Uses" property played a pivotal role in achieving flexibility and automation. Rather than hardcoding specific variations of behaviours, we leveraged this property to enable automatic planning based on "Intended Uses" values. To facilitate its easy customization through the Unity editor, a custom property drawer was created, streamlining the process of fine-tuning agent behaviours and enhancing the adaptability of our system.

As an example, consider the "Drink" and "Pour" behaviours, both of which are injected to *LiquidContainer* labeled nodes. Now, envision the following hypothetical scenario: "Person A wishes to fill Cup B with water. In close proximity to Person A are two LiquidContainers, one being another Cup C and the other a Pitcher D." In this scenario, our system should intelligently guide Person A to use the Pitcher to fill Cup B, rather than Cup C, aligning with the intended use. Furthermore, Person A should also be directed to drink water from Cup A and not from the Pitcher. By manipulating the intended use values of “Drink” and “Pour” for each Node, the above situation is easily realized. Furthermore, in a scenario where Person A is in great need of drinking water, and only the Pitcher D is available, it will use it because it still exposes the behaviour of Drink.

5.2.4 Node Debugger

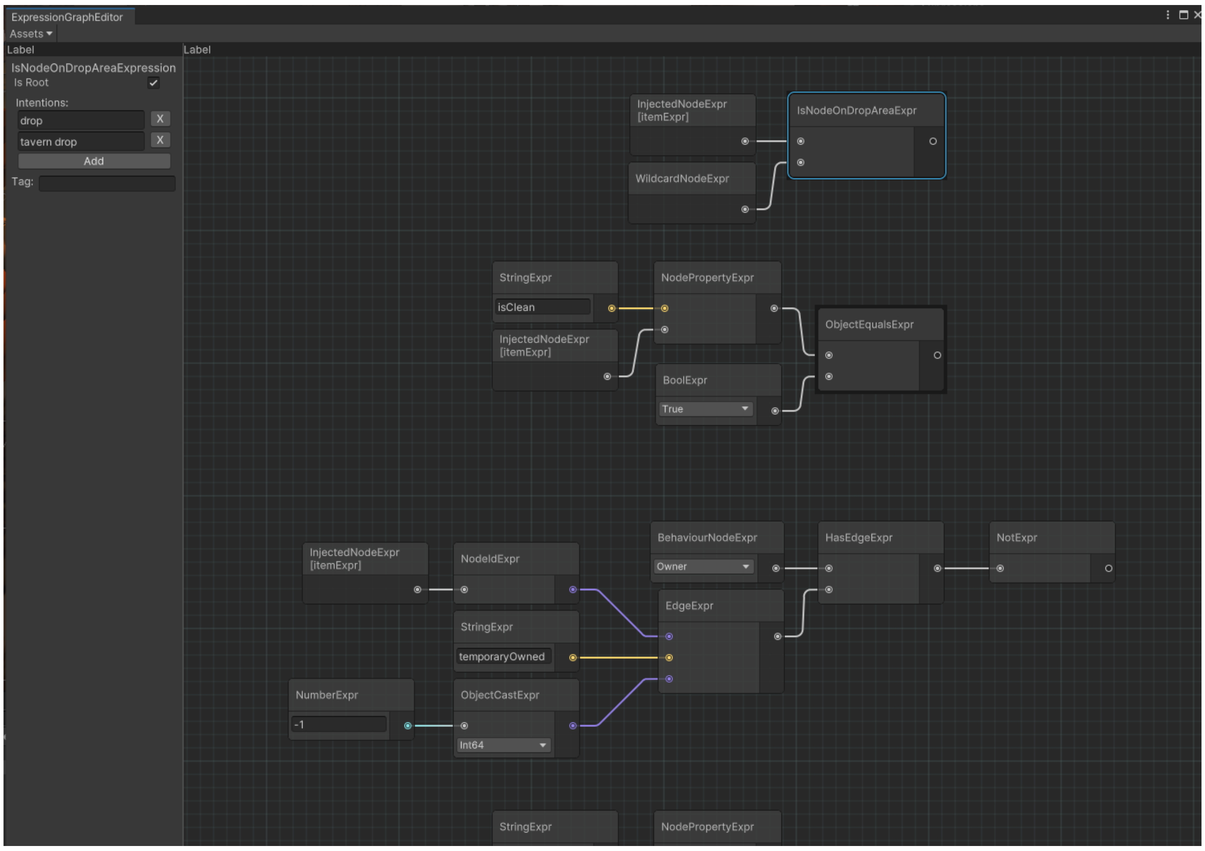
Understanding the internal state of NPCs is pivotal for any implementation. To tackle this challenge, the *Node Debugger* was introduced, a simple tool that can be extended to provide insights into any node's "brain" at any given moment. Throughout the implementation of the use case, two modules were of great importance while debugging; the *ActiveEvents* associated with a node and the real-time status of complex behaviour queues. These were critical pieces of information when tracking and comprehending an NPC's decision-making processes, allowing for a more thorough and effective implementation.

5.2.5 Expression Graph Editor

The *Expression Graph Editor* offers a user-friendly graphical interface that simplifies the creation of complex expression graphs. Within this editor, each node corresponds to an expression, and the connections between nodes symbolize the flow of information. Attempting to define intricate expression graphs solely through code would prove exceedingly challenging, unreadable and in general a bad practice. Fortunately, Unity provides the flexibility to extend its built-in node graph system with custom functionality. Consequently, a straightforward methodology was devised for authoring expressions, further enhancing the efficiency of expression design within Autonomia.

This streamlined pipeline mandates that expressions "declare" their evaluation type and children during script creation. For instance, consider a *MathExpression* node, which functions as a processor expression and evaluates to a float value. It expects three specific children: an arithmetic operation (addition, subtraction, etc.) at index 0 and two floats at indices 1 and 2. This structure is exemplified in Figure 24 below.

Additionally, we have implemented an inspector for each node to facilitate the modification of multiple properties by designers. For instance, expressions also hold string values representing intentions for optimizing Goal-Oriented Action Planning (GOAP). With our system, these intentions can be easily altered without the need for recompilation or prior knowledge of the underlying codebase.



ExpressionGraphEditor

IsNodeOnDropAreaExpression

Is Root ✓

Intentions:

drop

tavern drop

Tag:

Figure 24: Expression Graph of DrinkTemporaryOwnedDrink goal

By integrating this methodology and the associated features into our *Expression Graph Editor*, we have significantly enhanced the accessibility and adaptability of our expression system. Designers and developers can now intuitively create and fine-tune complex agent behaviours without the constraints of coding intricacies.

5.2.6 Expression Library

A library of expressions has been created for the purposes of implementing the case study scenario, while also serving as a pre-made, tested and usable package for future uses of the Unity integration. All the expressions were created having in mind their reusability and use even out of the context of the Prometheus Tavern case study. Below are listed the most common categories of nodes and their common use in the caste study. It is also noteworthy, that most of the source code for the expression nodes is not coupled to Unity. It can be reused for other engines and platforms, although this will come at the loss of the expression graph editor tool.

5.2.6.1 Primitive Expressions

The primitive category of expressions includes objects such as strings, numbers and boolean values and their wildcard representations. All of those expression nodes are producers, meaning they are used as leaves within the expression tree. The use of such values is mandatory and common through every implementation.

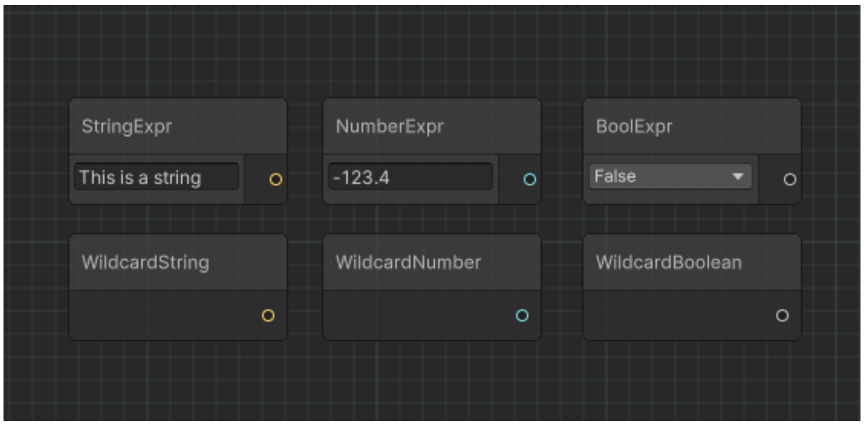


Figure 25: Primitive expressions

5.2.6.2 Logic and Math Expressions

Another important tool in the base-set of the expression library are math and logic nodes, offering an unlimited number of conditionality to be applied. We have implemented the general arithmetic *MathExpression* and *CompareExpression*, as well as boolean algebra expressions such as *AndExpression* and *OrExpression*. In addition, expressions can also expose any C# functionality. In this example with have also wrapped the C# “Equals” method in a custom *ObjectEqualsExpression*. This enables us to compare nodes, events and more.

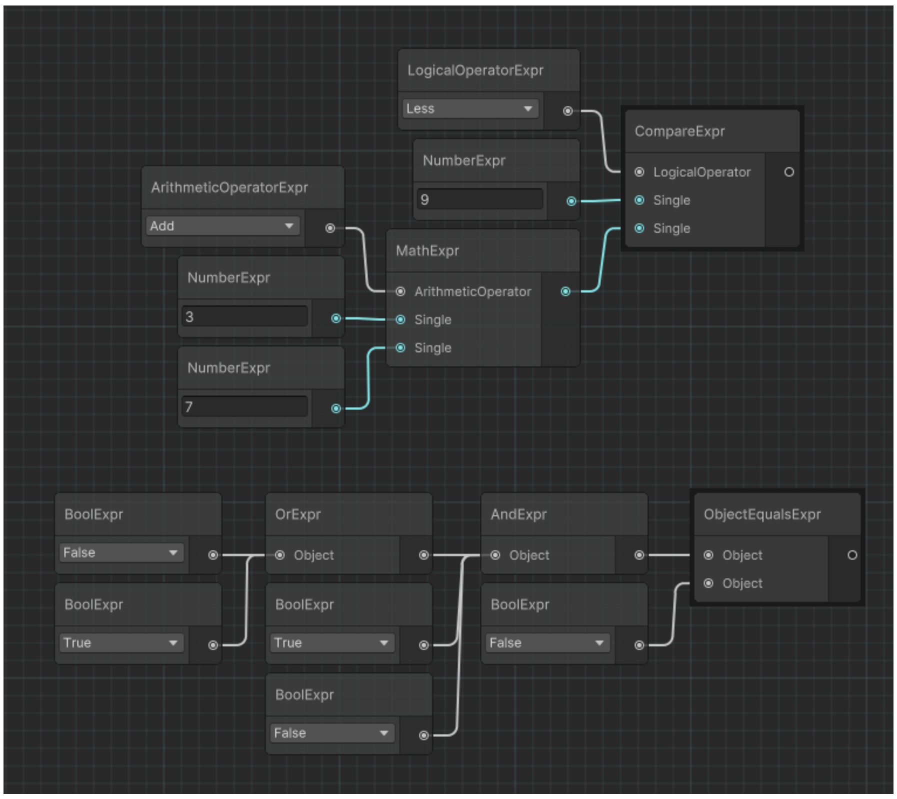


Figure 26: Logic and Math expression examples

5.2.6.3 Node Expressions

Autonomia Node-related expressions played a central role in crafting the majority of expression graphs, necessitating the creation of numerous such expressions. Among these, the most frequently used were the wildcard and abstract nodes, each returning their respective node types. Additionally, a custom expression, known as *BehaviorNodeExpression*, was introduced. This expression serves the purpose of injecting the corresponding owner or actor into each behavior through the visitor pattern, a pattern thoughtfully implemented for all expressions.

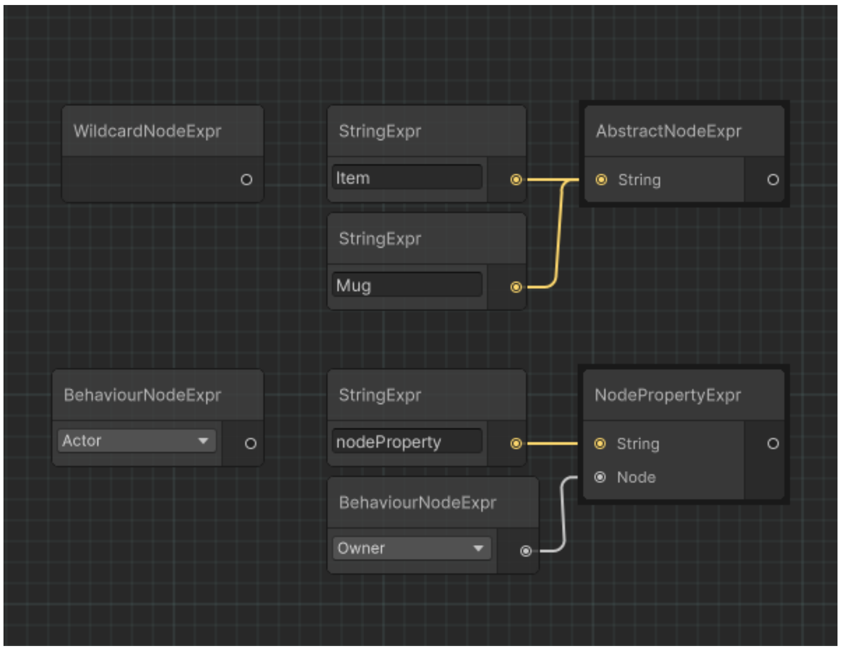


Figure 27: Node expressions

5.2.6.4 Event Expressions

The greater the number of advanced expressions at our disposal, the higher the level of complexity we can achieve in our expressive capabilities. In this section, we introduce the *Event* expression, which is coupled with *DictionaryExpression*, *DynamicValueExpression*, and *StimulusTypesListExpression*, enabling the creation of a wide range of events through the graph. In *Figure 28*, an illustrative example of such an event is presented. To elaborate further, the following expressions can be interpreted as follows:

*“The expression evaluates to true, when the following **event is active**; the **actor of the behaviour grabs** an **abstraction of a sword item**, with **any** of his two **hands**, and the event can be **sensed** through **visual** means.”*

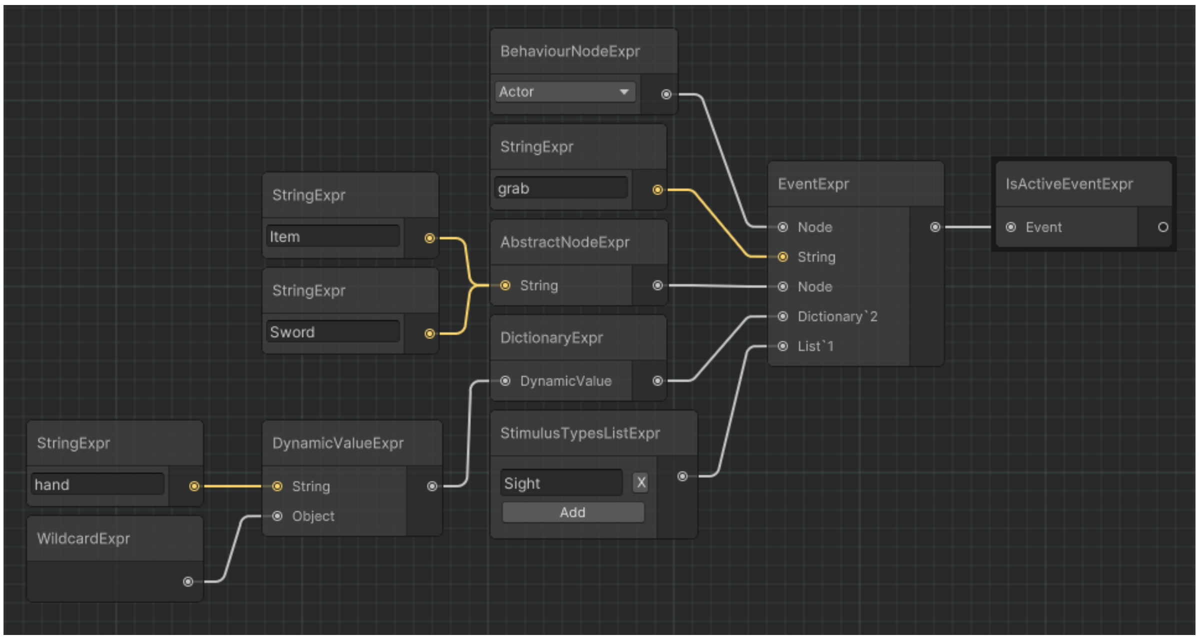


Figure 28: A grab event created through the expression graph editor

5.2.6.5 Utility Expressions

Utility expressions represent a general category that will be more properly categorized in the future. The expression graph is "strongly typed", meaning you cannot assign the output of some expression to the input of another if their declared types do not match. For that reason, there exists an expression node called *ObjectCastExpression* which supports type casting for all primitive C# types. When the need for a user specific cast arises, the developer can simply script a new expression node as shown in Figure 29 with the *NodeCastExpression*, which casts its child to be Node type.

Furthermore, we introduce another crucial part of the unity integration, the *InjectedObjectExpression* which can be accessed through code by the developer to manually inject a specific value prior to the evaluation of the expression graph.

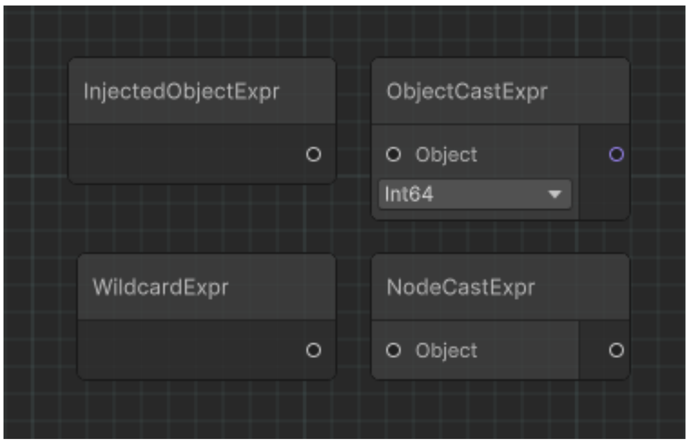


Figure 29: Commonly used utility expressions

5.2.6.6 Honorable Mentions

With the expression system an infinite number of user created nodes can exist. Out of the most unexpected node combinations we came upon while creating our case study, was the general category of *SuperExpressions*, specifically the *MatchNodePropertiesExpression*. This expression node will upon evaluation try to expand itself, based on its Node children to create new expressions, that dictate that all their properties should match.

Another surprising expression was the *IfExpression* node. In our implementation, this expression node can control the direction of its base functions based on the evaluated value of the condition. This provides great flexibility to generalize and reuse a plethora of expression graphs.

Lastly, we created some expression nodes for memory queries. For instance, the *HasEdgeExpression* which uses the Memory module of the given node to validate if this node has the knowledge of the given edge to be true.

The previous expression nodes are shown in the following figure. The former can be read as

“The root expression evaluates to true, when there exists any node that all its properties can match with the first node”, whereas the latter can be read as *“If the actor node has knowledge of an edge of type ‘owns’ between him and the owner of the behaviour, the root node will evaluate to true or false otherwise”*.

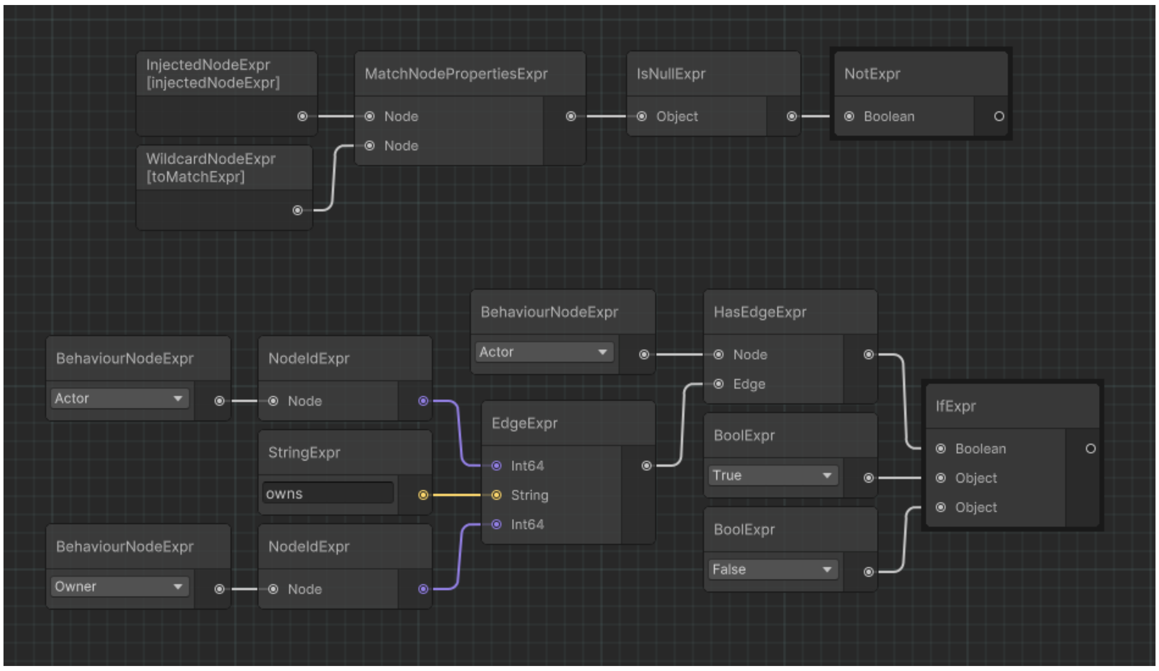


Figure 30: MatchNodePropertiesExpression and IfExpression example

5.3 Use Case: Prometheus Tavern

To evaluate the Autonomia Framework, it’s Unity integration and to offer an open-source foundational scene, this thesis introduces the "Prometheus Tavern" case study. This scenario serves as a comprehensive test bed for the majority of Autonomia's features, by simulating multiple agents concurrently engaging in real-time behaviour and planning. Notably, the "Prometheus Tavern" is not a traditional game scene, as it lacks a player character, but it can be extended in the future to support one. It is better described as a simulation environment. In the case study exist two types of NPCs: a) a waiter tasked with serving customers and keeping the tavern organized, and b) customers who enter the tavern, order drinks, drink them and leave when satisfied.

In this case study, we mainly rely on real-time planning for behaviors instead of using simplified behavior models, which might have made the system faster. This approach helps us evaluate how well Autonomia's agents can adapt and potentially appear realistic. It's worth noting that this study is limited based on our own knowledge, skills, and time constraints. Unlike typical games, we're not prioritizing lifelike graphics or animations; our focus is primarily on the behaviors and memory functions themselves.

In the following sections we describe how the game scene was structured, how the goals and behaviours were modeled and lastly our experience on using

Autonomia Framework for the first time.

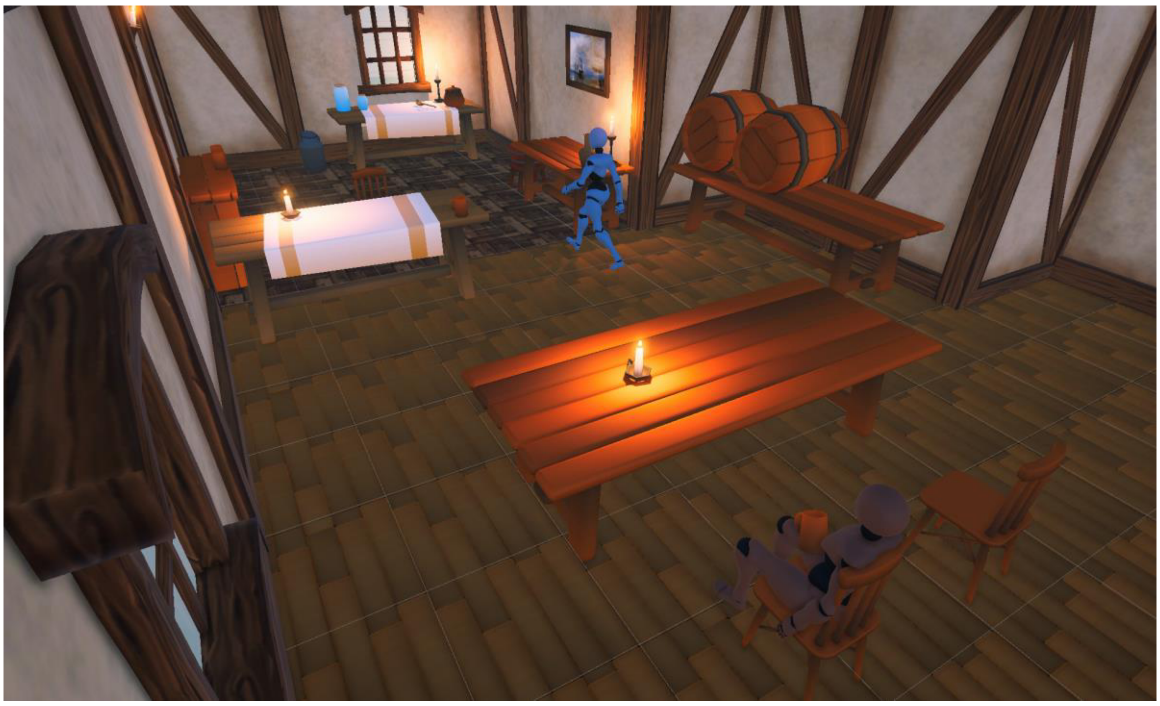


Figure 31: Prometheus Tavern case study scene

5.3.1 Prometheus Tavern Scene

In the Prometheus Tavern scene exist various objects common to a mediaeval tavern. Among those some are wooden barrels with different drinks, tables, chairs, mugs, pitchers. Below, we list the main behaviours injected to items based on their corresponding labels.

Table 3: Main exposed behaviours in the Prometheus Tavern

Behaviour Name	Node Labels	Preconditions	Effects
GrabItem	Grabable1H	- Actor is close to owner - Actor is not grabbing anything - Owner is not grabbed by someone else.	- Actor grabs owner
DropItem	Grabable1H	- Actor is grabbing owner - Actor is close to drop	- Actor drops owner.

		position	
DropItemOnArea	DropArea	- Actor is close to owner - Actor is grabbing	- Actor drops item on drop area
CleanItem	Cleaner	- Actor is grabbing item - Actor is close to owner	- Actor cleans item
LiquidContainerDrink	LiquidContainer	- Actor is grabbing a liquid container item - Liquid Container Item's current litres are greater than zero	- Actor drinks item
LiquidContainerPour	LiquidContainer	- Actor is grabbing a liquid container item - Actor is close to owner - Owner's current litres are greater than zero	- Actor pours liquid from owner liquid container to an item liquid container
Sit	Sittable	- Actor is near owner - No-one else is sitting on owner - Actor is not sitting anywhere else	- Actor sits on owner

5.3.2 NPC Goals

A common approach on designing game scenes that will use the GOAP algorithm, involves establishing specific goals and subsequently designing the environment to facilitate the achievement of these goals through one or more viable ways. Each goal is further explained in the following table. These goals do not necessarily represent how a waiter should act in a real-life scenario. They have been designed around testing the NPC's ability to adapt to their changing environment and conditions.

Table 4: Waiter goals table

Waiter

Goal Name	**Effects**	**Value**
Organize and Clean Mug	- Actor cleans an abstraction of a mug - That mug must be dirty - That mug must not be used - Actor drops that mug on any drop area	Number of mugs that fall into this category multiplied by 15
Take Order	- Actor engages in dialogue with customer	Number of pending customers calling multiplied by 85
Prepare Order	- Actor "considers" [2] an item that matches an injected item.	Number of pending orders to be created multiplied by 80
Serve Order	- Actor serves order to customer	Number of readied orders to be served multiplied by 75
Store Order	- Actor drops a readied order on any drop area	Number of pending orders to be created multiplied by 90
Rest	- Actor sits	Analogous to the Energy Need

[2] "Consider" is non-physical behaviour that is exposed by every item. It allows the planner the consider item nodes even when their behaviours do not directly expose effects that match the current's step preconditions

Table 5: Customer goals table

Customer		
Goal Name	**Effects**	**Value**
Find Sit	- Actor sits - Actor should not be sitting already	Static value of 90
Order	- Actor calls any waiter node	Static value of 50
Drink	- Actor drinks a liquid container - That liquid container must be temporarily owned by him	Static value of 80
Leave	- Actor sits	Static value of 100

5.3.3 Prometheus Tavern Challenges

To evaluate the agents' capacity to adapt to a dynamic environment, we designed a set of challenges that artificially modified the scene and its characteristics. These challenges were intended to make it more demanding for NPCs to accomplish their goals while simultaneously validating their perception and memory capabilities. In the subsequent sections, we outline the challenges that the NPCs encountered and describe how they successfully addressed these changing conditions.

5.3.3.1 Moving Items Around in Runtime

When NPCs attempted to reach an object, they would dynamically track the item if it moved within their field of vision. However, as expected, if the item left their field of vision, they would halt and reassess their plans and goals, marking the item as missing until it was noticed again. For instance, if a waiter had planned to sit in a specific chair, and that chair suddenly vanished, he would pause, reevaluate, and, if suitable, select another chair to sit in.

It's important to note that NPCs would remain unaware of an item changing position until they either observed the item in its new location or reached the spot where they believed the item should be, only to discover it was missing. For instance, if a waiter started outside of the tavern, planned to sit in his chair, and the chair vanished before he entered the tavern, he would continue walking toward the chair since he hadn't yet realized that the chair was no longer there.

5.3.3.2 Different Perception Sensors

In our case study, each NPC was equipped with two types of stimuli: one for sight and one for sound. The sight sensor took the form of a cone-shaped mesh that extended from their eyes, while the sound sensor was represented as a spherical shape surrounding the agent.

The behavior of an agent was influenced by these sensors. For instance, an agent would not plan to clean a dirty mug if they hadn't visually perceived it yet. However, they would respond to a customer's call, even if they were not currently looking in that direction at that particular moment.

5.3.3.3 Item Preferences using *IntendedUses*

NPCs consistently exhibit a sense of preference in their decision-making. For example, a waiter would avoid sitting in a chair on the "customer side" of the tavern unless there were no other options. Similarly, customers would typically choose a seat that: a) is not the waiter's chair, or b) in most cases, the seat that looks the most appealing.

Moreover, the waiter has a predefined spot for placing clean mugs, but in cases where that area is unavailable or obstructed, he would reevaluate his plan and select the next best available option for placement.

5.3.3.4 Depleted Barrel

When preparing an order, the waiter would usually pour rum from the barrel item which contains rum. In this challenge, the barrel would be emptied in runtime while the actor would be taking an order and the agent should plan accordingly, using another liquid container containing rum to fill up the order’s mug. And when the barrel would be artificially filled again, if the NPC noticed, he would go back to using the barrel again.

5.3.3.5 Dynamic Goal Values

The behavior of the waiter NPC is dynamic and influenced by his knowledge. For instance, if there were numerous dirty mugs to attend to, he would prioritize cleaning most of them, giving the appearance of being busy with that task before moving on to other activities, such as taking new orders. Additionally, when faced with multiple orders to prepare, he efficiently readies both orders before serving them, rather than dealing with them individually, which optimizes his workflow.

5.3.4 Happy Surprises during Development

Throughout the development and testing process, there were instances where the NPCs pleasantly surprised the developers. For instance, sometimes the waiter would place an order out of the customer's reach. Instead of requiring manual programming to handle this situation, the customer autonomously stood up, fetched the drink, and returned to their seat, all executed by the system without explicit instructions. Furthermore, if the customer's seat was moved farther from the table, when they finished their drink, the system automatically planned for them to stand up, place the empty mug on the table, and then return to their seat.

Chapter 6

6 Expert-Based Evaluation

In this chapter the evaluation process of the Autonomia Framework will be presented and discussed. A system's evaluation can take multiple forms and it is crucial to repeat this step multiple times throughout the design and development process.

As far as this thesis is concerned, Autonomia is a newborn system, and the current version is but a prototype. Even by the time of writing this thesis, it is an undeniable fact that there exist a plethora of problems and limitations that need to be stated and classified. This is a normal phenomenon in the field of R&D and should be wholeheartedly accepted and acknowledged.

Bellow, we iterate on Autonomia's self-reported, interview-based [92] and expert-based heuristic evaluation [54], [93] taken place by the time of writing this thesis, whose questions can be found in Appendix A and B. We conceptually consider two parts in the evaluation; part one aiming to measure the behaviour realism of Autonomia's agents and part two aiming to judge the features and architectural decisions of the framework. Both were executed on the *Prometheus Tavern* case study.

6.1 Evaluation Part I

6.1.1 The Process

The evaluation experiment can be divided into four different phases: a) the game scene preparation, b) the introduction narrative, c) the execution of the scenarios with the user, d) user-filled questionnaire, and e) an ending discussion.

To streamline the evaluation of different challenges affecting the NPCs, the game scene was duplicated into multiple instances, with each instance having a slightly different setup to test a different challenge.

Upon arriving, users were individually informed about the system's experimental stage and were explicitly asked to assess the realism of the NPC's behaviours from a logical standpoint rather than an aesthetic one.

The scenarios showcased in this evaluation, are effectively the same challenges from section 5.3.3, in the context of the *Prometheus Tavern*. While the experiments were running, the evaluator would at sometimes narrate parts of the scenarios due to certain aspects not being fully implemented. For instance, visual cues for a "dirty" mug were not present, so relevant information had to be provided via Unity's inspector. Furthermore, the evaluator would be open to suggestions from the user, to further challenge the NPCs to adapt. Of course, only suggestions that would be applicable were considered.

In the evaluation process followed a user-filled questionnaire containing seven questions that could be rated on a scale from 1 to 5, where 1 signified "Strongly Disagree" and 5 signified "Strongly Agree". Six out of seven questions were focused on the realism of the behaviours of the NPCs, were the last one focused on the reactions' speeds of the agents. Lastly, an open discussion session was conducted to collect the user's feedback and insights on the agents' behaviours, perception, and responsiveness.

6.1.2 Results

The evaluation results revealed several positive aspects of Autonomia's agent behavior adaptation in dynamic environments. The users generally agreed that Autonomia's agents were capable of adapting to changes made during runtime. Specifically:

- Users almost unanimously agreed that the agents adapted well to their changing environment, even when modifications were introduced during gameplay.
- Users strongly agreed that NPCs adjusted their behavior based on the specific objects and their available preferences within the tavern.
- Users noticed that NPCs changed their goal prioritization depending on dynamic scene conditions. For example, they would prioritize cleaning dirty mugs when many were present before taking new orders.

However, there were also some dissenting opinions:

- Some users disagreed that the agents had different reactions to various stimuli. It was mentioned that the lack of animations and in-game sound might have contributed to this opinion.
- Users generally disagreed when asked whether the agents responded and planned quickly. This outcome was expected due to the nature of the scene, which required agents to make real-time plans for almost every action. The scene had a total of 40 nodes, each exposing 2-4 behaviours and the average NPC plan had a length of 5.

The results indicated that Autonomia's agents demonstrated adaptability to changing environments, but there were areas where further improvement and user feedback could be beneficial.

6.2 Evaluation Part II

The second part of the evaluation involved the professional expertise of four Unity developers, with three of them having experience specifically in game development and R&D gamification. These individuals were considered ideal candidates for the framework's intended user group, given their expertise and background in the field.

6.2.1 The Process

In this evaluation, the users were explained the system in its entirety, delving into a technical analysis and demonstration of the available features and architectural decisions of Autonomia. Each section of the system was accompanied by examples from the existing codebase. In cases where users displayed a heightened interest, the evaluator even extended the codebase with new examples based on the user's suggestions, fostering an interactive and engaging process.

The primary areas of focus during this technical demonstration included: a) the *Memory*, *Perception* and *EventInterpreter* modules, which constitute the theory of mind approach of Autonomia, b) the module and injection mechanisms, c) the EBG system, and lastly d) the editor tools, focusing mostly on the Expression Graph Editor. In many points of the demonstration the user was encouraged to ask questions regarding the system and why each decision was taken.

At the conclusion of the technical demonstration, an open-ended interview was conducted with users, consisting of a total of 18 questions that spanned functionality, features, system usability, performance, scalability, customization, extendibility potential, and future improvements.

6.2.2 Results

The second part of the expert-based evaluation identified several issues, including some inherent to Autonomia and others that are relatively easier to address. However, it also emphasized the significant strengths of the system, which serve to mitigate the identified weaknesses. Table 6 summarizes the collective findings for each category of questions.

Table 6: Extracted results from the expert-based evaluation.

Question Categories	Heuristic
Overall Impressions	All experts highlighted that the system is exceptionally complex and vast. For the system to actually be applied, it would need careful planning and good practices. Upon initial inspection, it would be impossible to use by someone inexperienced, but with comprehensive documentation and enough time to familiarize oneself, it would be as manageable as any other system learned over time. All experts agreed that the Autonomia Framework has the potential to significantly enhance AI agent behaviours. Implementing Autonomia in a future game would introduce a fresh and innovative approach, breathing more life into NPC agents compared to contemporary industry games.
Functionality and Features	Many experts identified the treatment of knowledge as their favorite feature within Autonomia. They appreciated how the system allows for generalized, substantial communication of knowledge. The way Memory interacted with Perception and Interpretations was particularly commended and considered one of the system's greatest and unique strengths. Additionally, some experts viewed the system's freeform extensibility as a valuable feature. Regarding missing functionalities or features, all experts underscored the significance of comprehensive documentation and paradigms that would facilitate the onboarding of new users to Autonomia. One expert also mentioned the absence of emotional aspects in the current agents as an area for potential improvement.
Usability	In response to more specific questions about usability, most experts agreed that individuals with prior experience using similar tools would have an easier time comprehending the system. However, new users would likely encounter challenges, especially when dealing with specific abstract concepts.

	Among the three experts with experience in game development, they found the terminology in the system to be well-defined and structured. However, the latter expert did identify some instances where terms were used inappropriately. All experts unanimously described the learning curve of the system as exponential initially but becoming linear or logarithmic as users gained more hands-on experience.
Performance	Experts expressed that the NPCs in Prometheus Tavern exhibited slow performance. However, after gaining a technical understanding of the system's inner workings, they recognized the reasons behind this performance issue and suggested that performance should be a more carefully considered aspect in future games, even if it meant compromising the planning capacity of NPCs. There were no crashes during the evaluation. However, some unexpected behaviors were observed. For example, an NPC might serve a dirty mug to a customer. Such occurrences were attributed to implementation issues rather than problems with the framework itself.
Integration and Compatibility	All experts strongly agreed that the system is highly capable of accommodating other subsystems, emphasizing the framework's generalized and modular nature.
Scalability	The general consensus regarding the scalability of the system was that it heavily depends on the specific implementation. Autonomia is a versatile tool that can support a wide range of unique and complex functions. However, without careful planning and the use of specialized algorithms, it will most likely not scale well. Experts could not pinpoint a core mechanism within Autonomia that would inherently limit the scalability of subsequent systems.

Customization and Extensibility	All experts provided positive responses in this specific category. As previously mentioned in other areas, they believe the system is highly customizable and extendable, whether through generalized expressions or the module system.
Future Improvements	Among the most common responses were suggestions for creating a manual complete with examples, paradigms, and even video tutorials to facilitate user understanding. Additionally, experts recommended the development of more visual tools for various other functionalities of the system, such as visual programming of behaviours.
Comparison with existing techniques	When compared to other techniques, experts noted that the framework, despite its high complexity, provides users with a well-structured pipeline to model and integrate various AI techniques. It offers both the theoretical and practical foundation for future implementations to accomplish tasks that would have been more challenging or resource-intensive without it. Additionally, some experts pointed out that the value of using Autonomia depends on the specific project at hand. For simple, one-dimensional projects, using this system may not be worthwhile due to its steep learning curve and the presence of various subsystems, which could be cumbersome. However, for those aiming to create an open-ended, sandbox-style world with dynamic agents, Autonomia would be a valuable addition to the team's toolkit.

Chapter 7

7 Conclusions and Future Work

7.1 Conclusions

Over the past decade, a significant volume of research has been dedicated to AI agents, with each endeavor addressing specific challenges related to enhancing the believability of such agents. Despite successful advancements in various aspects of this issue, the underlying problem remains unresolved. This is primarily due to the multifaceted and highly complex nature of the challenge, necessitating a unified approach that combines a diverse range of artificial intelligence techniques and solutions. However, no previous work has provided a tool capable of fulfilling this role.

This thesis has introduced Autonomia, an extendable and customizable framework designed to enhance the believability of Non-Player Characters (NPCs). Autonomia's approach centers on the modeling of a memory system that can realistically evolve through an NPC's perception and understanding of the world, leveraging the Theory of Mind as a foundational concept. From an engineering standpoint, Autonomia's open-source nature aims to provide a shared foundation for future research and development, with its modular architecture enabling different teams of developers and researchers to contribute their unique expertise while sharing their work for everyone to capitalize from, in an accessible manner.

Autonomia does not position itself as a complete and ready-made solution for all problems, as this approach would not align with its open-source philosophy. Its aim is to serve as an initial seed, a starting point for potential future advancements in research to create more sophisticated systems. This thesis represents the first step toward building a complex framework, with the hope that it may one day grow into a thriving ecosystem of innovation and progress.

7.2 Future Work

Autonomia's future is filled with unlimited potential and possibilities. The framework has demonstrated its promise through its modular nature, but, like any other innovative system, it will benefit from the establishment of common standards and protocols to guide its development and ensure coherence. This is a pivotal step towards fulfilling the potential of Autonomia and creating an ecosystem of research and innovation.

7.2.1 Documentation and Examples

As of this point, the expert-based evaluation showed the system to be extremely complex and hard to use for someone unfamiliar with its already large codebase. The immediate next step would be to document all the components contained in the *Autonomia.Core* namespace as well as provide complete and comprehensive examples of creating *Modules*, *Behaviours*, and *Goals*, *Expressions*, *Interpretations,* and new *Injection* attributes. In addition, extra documentation should be provided for the Unity Integration as it extends *Autonomia.Core* in various ways, for example creating new expression nodes for the expression graph editor.

7.2.2 Standardized Protocols and Design Principles

In the future development of Autonomia, there is a need to establish a framework for evaluating protocols and interfaces for various aspects of agent believability. For example, defining the characteristics of an emotionally driven AI could lead to the creation of a generalized, community-accepted abstract module. This abstract module can serve as an interface within Autonomia, allowing for the integration of different implementations of emotional AI. Such an approach enables seamless comparison and swapping of implementations in the same execution environment, fostering a powerful comparison tool.

Additionally, the framework should include a comprehensive expression model for common behaviors exposed by most nodes. For instance, behaviors like "grab" or "walk" should have standardized preconditions and effects, providing a baseline for modeling more complex behaviors in future iterations of Autonomia. This approach ensures that the framework can accommodate a wide range of use cases and encourages the development of a consistent and adaptable system.

7.2.3 Refactored User Interface

The existing user interface, as integrated within Unity, was primarily designed for functionality testing and served as a prototype. In future developments, there is a need to create a more user-friendly and intuitive interface that goes beyond the limitations of Unity Integration. This improved interface should be designed without focusing on Unity, with the intention of being user-centric and accommodate working with different engines or systems. Maintaining consistency and coherency in the interface design will be crucial for ensuring a seamless user experience across various environments and platforms.

UIs mentioned in this section include the Expression Graph Editor, the NodeRef script, a more sophisticated debugger and more.

7.2.4 Advanced Debugging Tools

In the context of this thesis, a basic node debugger was implemented. However, for Autonomia to align with its modular and extensible goals, a more sophisticated debugging system is required. Each module should incorporate support for custom debugging information, extending beyond plain text. For example, the current system lacks the capability to visually inspect an NPC's memory. An even more valuable feature would be the ability for framework users to make real-time changes to the system during application execution, facilitating faster testing iterations and debugging.

7.2.5 Player, Dialogues and Emotion

The way Autonomia currently is, it would be simple addition to extend it with a dialogue system that utilizes the NPC's memory to allow them to freely converse. Already, simply queries in a cypher-like syntax can be executed. It would not be far stretched to allow potential players to converse with the NPCs through machine learning assisted prompts. It would be a spectacle to observe emotional agents freely conversing, behaving, and planning their goals while exchanging information regarding their experiences and daily lives.

7.2.6 More Case Studies

To prove Autonomia's efficiency as a tool, more cases studies need to be performed. This would allow for a better understanding of the tool itself, and even birth more design principles and guidelines for future generations.

Bibliography

[1] H. Warpefelt, M. Johansson, and H. Verhagen, *Analyzing the believability of game character behavior using the Game Agent Matrix*. 2013.

[2] H. Warpefelt and H. Verhagen, “A model of non-player character believability,” *J. Gaming Virtual Worlds*, vol. 9, pp. 39–53, Mar. 2017, doi: 10.1386/jgvw.9.1.39_1.

[3] H. Warpefelt, “The Non-Player Character : Exploring the believability of NPC presentation and behavior,” 2016, Accessed: Oct. 09, 2023. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-128079

[4] R. Rogers, J. Woolley, B. Sherrick, N. D. Bowman, and M. B. Oliver, “Fun Versus Meaningful Video Game Experiences: A Qualitative Analysis of User Responses,” *Comput. Games J.*, vol. 6, no. 1, pp. 63–79, Jun. 2017, doi: 10.1007/s40869-016-0029-9.

[5] C. A. Oswald, C. Prorock, and S. M. Murphy, “The perceived meaning of the video game experience: An exploratory study,” *Psychol. Pop. Media Cult.*, vol. 3, no. 2, pp. 110–126, 2014, doi: 10.1037/a0033828.

[6] M. S. Lee and C. Heeter, “Cognitive Intervention and Reconciliation - NPC Believability in Single-Player RPGs,” *Int. J. Role-Play.*, no. 5, Art. no. 5, Jan. 2015, doi: 10.33063/ijrp.vi5.236.

[7] F. D. Schönbrodt and J. B. Asendorpf, “The Challenge of Constructing Psychologically Believable Agents,” *J. Media Psychol.*, vol. 23, no. 2, pp. 100–107, Jan. 2011, doi: 10.1027/1864-1105/a000040.

[8] D. Brown, *The suspension of Disbelief in Videogames*. 2016. doi: 10.13140/RG.2.1.3175.8968.

[9] M. S. Lee and C. Heeter, “What do you mean by believable characters?: The effect of character rating and hostility on the perception of character believability,” *J. Gaming Virtual Worlds*, vol. 4, no. 1, pp. 81–97, Mar. 2012, doi: 10.1386/jgvw.4.1.81_1.

[10] E. Arts, “The Sims Video Games - Official EA Site,” Electronic Arts Inc. Accessed: Sep. 18, 2023. [Online]. Available: https://www.ea.com/games/the-sims

[11] “Sims AI (Autonomy) Issues :: The Sims™ 4 General Discussions.” Accessed: Oct. 28, 2023. [Online]. Available: https://steamcommunity.com/app/1222670/discussions/0/3172198151262339776/

[12] Professional-Ad-7032, “Sim AI is a piece of shit,” r/Sims4. Accessed: Oct. 28, 2023. [Online]. Available: www.reddit.com/r/Sims4/comments/qrpf88/sim_ai_is_a_piece_of_shit/

[13] “Mass Effect™ Legendary Edition - EA Official Site.” Accessed: Sep. 18, 2023. [Online]. Available: https://www.ea.com/games/mass-effect/mass-effect-legendary-edition

[14] M. Goldberg, “The Mass Effect 3 Ending Wasn’t the Game’s Biggest Problem,” Collider. Accessed: Oct. 28, 2023. [Online]. Available: https://collider.com/mass-effect-3-ending-problems/

[15] “Mass Effect: Andromeda Review,” Giant Bomb. Accessed: Oct. 28, 2023. [Online]. Available: https://www.giantbomb.com/reviews/mass-effect-

andromeda-review/1900-762/

[16] “Detroit: Become Human I Official Site I Quantic Dream.” Accessed: Sep. 18, 2023. [Online]. Available: https://www.quanticdream.com/en/detroit-become-human

[17] K. Orland, “Detroit: Become Human review: Robotic in all of the wrong ways,” Ars Technica. Accessed: Oct. 28, 2023. [Online]. Available: https://arstechnica.com/gaming/2018/05/detroit-become-human-review-a-lack-of-humanity/

[18] M. H. published, “Detroit: Become Human review,” pcgamer. Accessed: Oct. 28, 2023. [Online]. Available: https://www.pcgamer.com/detroit-become-human-review/

[19] I. Granic, A. Lobel, and R. C. M. E. Engels, “The benefits of playing video games,” *Am. Psychol.*, vol. 69, no. 1, pp. 66–78, 2014, doi: 10.1037/a0034857.

[20] I. Spence and J. Feng, “Video Games and Spatial Cognition,” *Rev. Gen. Psychol.*, vol. 14, no. 2, pp. 92–104, Jun. 2010, doi: 10.1037/a0019491.

[21] D. Bavelier, R. L. Achtman, M. Mani, and J. Föcker, “Neural bases of selective attention in action video game players,” *Vision Res.*, vol. 61, pp. 132–143, May 2012, doi: 10.1016/j.visres.2011.08.007.

[22] C. Dweck and D. Molden, “Self Theories: Their Impact on Competence Motivation and Acquisition,” 2005, pp. 122–140.

[23] J. L. Sherry, “Flow and Media Enjoyment,” *Commun. Theory*, vol. 14, no. 4, pp. 328–347, Nov. 2004, doi: 10.1111/j.1468-2885.2004.tb00318.x.

[24] J. McGonigal, *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin, 2011.

[25] A. Suh, C. Wagner, and L. Liu, “The Effects of Game Dynamics on User Engagement in Gamified Systems,” in *2015 48th Hawaii International Conference on System Sciences*, HI, USA: IEEE, Jan. 2015, pp. 672–681. doi: 10.1109/HICSS.2015.87.

[26] “Enhancing User Engagement through Gamification.” Accessed: Oct. 10, 2023. [Online]. Available: https://www.tandfonline.com/doi/epdf/10.1080/08874417.2016.1229143?needAccess=true

[27] X. Li, A. Sigov, L. Ratkin, L. A. Ivanov, and L. Li, “Artificial intelligence applications in finance: a survey,” *J. Manag. Anal.*, vol. 0, no. 0, pp. 1–17, 2023, doi: 10.1080/23270012.2023.2244503.

[28] “Artificial Intelligence in Education: A Review I IEEE Journals & Magazine I IEEE Xplore.” Accessed: Oct. 14, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9069875

[29] P. Maes, “Artificial life meets entertainment: lifelike autonomous agents,” *Commun. ACM*, vol. 38, no. 11, pp. 108–114, Nov. 1995, doi: 10.1145/219717.219808.

[30] M. Choudhury, S. Prabhu, A. K. Sabri, and H. A. Marhoon, “Impact of artificial intelligence (AI) in the media and entertainment industry,” *AIP Conf. Proc.*, vol. 2736, no. 1, p. 060012, Sep. 2023, doi: 10.1063/5.0171147.

[31] Y. Mintz and R. Brodie, “Introduction to artificial intelligence in medicine,” *Minim. Invasive Ther. Allied Technol.*, vol. 28, no. 2, pp. 73–81, Mar. 2019,

doi: 10.1080/13645706.2019.1575882.
[32] M. Raj and R. Seamans, “Primer on artificial intelligence and robotics,” *J. Organ. Des.*, vol. 8, no. 1, p. 11, May 2019, doi: 10.1186/s41469-019-0050-0.
[33] O. Zawacki-Richter, V. I. Marín, M. Bond, and F. Gouverneur, “Systematic review of research on artificial intelligence applications in higher education – where are the educators?,” *Int. J. Educ. Technol. High. Educ.*, vol. 16, no. 1, p. 39, Oct. 2019, doi: 10.1186/s41239-019-0171-0.
[34] K. Shaukat Dar *et al.*, “The Impact of Artificial intelligence and Robotics on the Future Employment Opportunities,” *Trends Comput. Sci. Inf. Technol.*, Sep. 2020, doi: 10.17352/tcsit.000022.
[35] R. Bogue, “The role of robots in entertainment,” *Ind. Robot Int. J. Robot. Res. Appl.*, vol. 49, no. 4, pp. 667–671, Jan. 2022, doi: 10.1108/IR-02-2022-0054.
[36] G. N. Yannakakis, “Game AI revisited,” in *Proceedings of the 9th conference on Computing Frontiers*, in CF ’12. New York, NY, USA: Association for Computing Machinery, May 2012, pp. 285–292. doi: 10.1145/2212908.2212954.
[37] B. Suits, “What is a Game?,” *Philos. Sci.*, vol. 34, no. 2, pp. 148–156, Jun. 1967, doi: 10.1086/288138.
[38] J. Stenros, “The Game Definition Game: A Review,” *Games Cult.*, vol. 12, no. 6, pp. 499–520, Sep. 2017, doi: 10.1177/1555412016655679.
[39] K. Claypool and M. Claypool, “On frame rate and player performance in first person shooter games,” *Multimed. Syst*, vol. 13, pp. 3–17, Jul. 2007, doi: 10.1007/s00530-007-0081-1.
[40] V.-M. Karhulahti, “Suspending Virtual Disbelief: A Perspective on Narrative Coherence,” Nov. 2012, doi: 10.1007/978-3-642-34851-8_1.
[41] E. Adams, *Fundamentals of Game Design*. New Riders, 2013.
[42] “Current AI in games. A Review.” Accessed: Oct. 09, 2023. [Online]. Available: https://eprints.qut.edu.au/45741/1/AJIIPS_paper.pdf
[43] M. Fernández, “Automata and Turing Machines,” in *Models of Computation: An Introduction to Computability Theory*, M. Fernández, Ed., in Undergraduate Topics in Computer Science. , London: Springer, 2009, pp. 11–32. doi: 10.1007/978-1-84882-434-8_2.
[44] “Nimrod (computer),” *Wikipedia*. Sep. 02, 2023. Accessed: Oct. 09, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Nimrod_(computer)&oldid=1173418001
[45] “*Spacewar!*,” *Wikipedia*. Aug. 24, 2023. Accessed: Oct. 09, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Spacewar!&oldid=1171970547
[46] D. Jagdale, “Finite State Machine in Game Development,” pp. 384–390, Oct. 2021, doi: 10.48175/IJARSCT-2062.
[47] M. Pirovano, “The use of Fuzzy Logic for Artificial Intelligence in Games”.
[48] K. Dill, E. R. Pursel, P. Garrity, and G. Fragomeni, “Design Patterns for the Configuration of Utility-Based AI,” 2012.
[49] C.-U. Lim, R. Baumgarten, and S. Colton, “Evolving Behaviour Trees for the Commercial Game DEFCON,” Apr. 2010, pp. 100–110. doi: 10.1007/978-3-642-12239-2_11.

[50] D. I. B. March 11 and 2005, “GDC 2005 Proceeding: Handling Complexity in the *Halo 2* AI,” Game Developer. Accessed: Oct. 09, 2023. [Online]. Available: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai

[51] T. Bylander, “The computational complexity of propositional STRIPS planning,” *Artif. Intell.*, vol. 69, no. 1, pp. 165–204, Sep. 1994, doi: 10.1016/0004-3702(94)90081-7.

[52] R. E. Fikes and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving,” *Artif. Intell.*, vol. 2, no. 3, pp. 189–208, Dec. 1971, doi: 10.1016/0004-3702(71)90010-5.

[53] H. Kautz, B. Selman, and J. Hoffmann, “SatPlan: Planning as Satisfiability”.

[54] B. Bonet and H. Geffner, “Planning as Heuristic Search: New Results,” in *Recent Advances in AI Planning*, S. Biundo and M. Fox, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 360–372. doi: 10.1007/10720246_28.

[55] “STRIPS, a retrospective.” Accessed: Oct. 09, 2023. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=076ae14bfc68acdbaf2ab24913e152d49540e988

[56] J. Orkin, “Three States and a Plan: The A.I. of F.E.A.R.,” 2006.

[57] “Goal-Oriented Action Planning (GOAP).” Accessed: Oct. 10, 2023. [Online]. Available: https://alumni.media.mit.edu/~jorkin/goap.html

[58] “*Transformers: War for Cybertron*,” *Wikipedia*. Jul. 11, 2023. Accessed: Oct. 10, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transformers:_War_for_Cybertron&oldid=1164766388

[59] “AI Action Planning on Assassin’s Creed Odyssey and Immortals Fenyx Rising.” Accessed: Oct. 09, 2023. [Online]. Available: https://www.gdcvault.com/play/1027004/AI-Action-Planning-on-Assassin

[60] I. Georgievski and M. Aiello, “HTN planning: Overview, comparison, and beyond,” *Artif. Intell.*, vol. 222, pp. 124–156, May 2015, doi: 10.1016/j.artint.2015.02.002.

[61] K. Erol, J. Hendler, and D. Nau, “Complexity Results for HTN Planning,” *Ann. Math. Artif. Intell.*, vol. 18, Apr. 2003, doi: 10.1007/BF02136175.

[62] P. Zhao, “Probabilistic contingent planning based on HTN for high-quality plans.” arXiv, Sep. 28, 2023. doi: 10.48550/arXiv.2308.06922.

[63] D. S. Nau *et al.*, “SHOP2: An HTN Planning System,” *J. Artif. Intell. Res.*, vol. 20, pp. 379–404, Dec. 2003, doi: 10.1613/jair.1141.

[64] D. Nau, Y. Cao, A. Lotem, and Hž. Mu, “SHOP: Simple Hierarchical Ordered Planner”.

[65] D. Nau *et al.*, “Applications of SHOP and SHOP2,” *IEEE Intell. Syst.*, vol. 20, no. 2, pp. 34–41, Mar. 2005, doi: 10.1109/MIS.2005.20.

[66] G. Skinner and T. Walmsley, “Artificial Intelligence and Deep Learning in Video Games A Brief Review,” in *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, Feb. 2019, pp. 404–408. doi: 10.1109/CCOMS.2019.8821783.

[67] OpenAI *et al.*, “Dota 2 with Large Scale Deep Reinforcement Learning.” arXiv, Dec. 13, 2019. doi: 10.48550/arXiv.1912.06680.

[68] K. Kunanusont, S. M. Lucas, and D. Pérez-Liébana, “General Video Game AI: Learning from screen capture,” in *2017 IEEE Congress on Evolutionary Computation (CEC)*, Jun. 2017, pp. 2078–2085. doi: 10.1109/CEC.2017.7969556.

[69] J. S. Park, J. C. O’Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, “Generative Agents: Interactive Simulacra of Human Behavior.” arXiv, Aug. 05, 2023. doi: 10.48550/arXiv.2304.03442.

[70] J. McCoy, M. Treanor, B. Samuel, B. Tearse, M. Mateas, and N. Wardrip-Fruin, “Authoring Game-based Interactive Narrative using Social Games and Comme il Faut”.

[71] M. Guimaraes, P. Santos, and A. Jhala, “CiF-CK: An architecture for social NPCS in commercial games,” in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug. 2017, pp. 126–133. doi: 10.1109/CIG.2017.8080425.

[72] J. Dias, S. Mascarenhas, and A. Paiva, “FAtiMA Modular: Towards an Agent Architecture with a Generic Appraisal Framework,” in *Emotion Modeling: Towards Pragmatic Computational Models of Affective Processes*, T. Bosse, J. Broekens, J. Dias, and J. van der Zwaan, Eds., in Lecture Notes in Computer Science. , Cham: Springer International Publishing, 2014, pp. 44–56. doi: 10.1007/978-3-319-12973-0_3.

[73] S. Mascarenhas, M. Guimarães, P. A. Santos, J. Dias, R. Prada, and A. Paiva, “FAtiMA Toolkit -- Toward an effective and accessible tool for the development of intelligent virtual agents and social robots.” arXiv, Mar. 04, 2021. Accessed: Sep. 14, 2023. [Online]. Available: http://arxiv.org/abs/2103.03020

[74] J. Orkin, “Symbolic Representation of Game World State: Toward Real-Time Planning in Games”.

[75] J. Kessing, T. Tutenel, and R. Bidarra, *Designing Semantic Game Worlds*. 2012. doi: 10.1145/2538528.2538530.

[76] N. Afonso and R. Prada, “Agents That Relate: Improving the Social Believability of Non-Player Characters in Role-Playing Games,” in *Entertainment Computing - ICEC 2008*, S. M. Stevens and S. J. Saldamarco, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 34–45. doi: 10.1007/978-3-540-89222-9_5.

[77] A. M. Leslie, O. Friedman, and T. P. German, “Core mechanisms in ‘theory of mind,’” *Trends Cogn. Sci.*, vol. 8, no. 12, pp. 528–533, Dec. 2004, doi: 10.1016/j.tics.2004.10.001.

[78] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, “An Integrated Theory of the Mind.,” *Psychol. Rev.*, vol. 111, no. 4, pp. 1036–1060, 2004, doi: 10.1037/0033-295X.111.4.1036.

[79] H. de Weerd, R. Verbrugge, and B. Verheij, “How much does it help to know what she knows you know? An agent-based simulation study,” *Artif. Intell.*, vol. 199–200, pp. 67–92, Jun. 2013, doi: 10.1016/j.artint.2013.05.004.

[80] R. Lämmel, “Foundations of Tree- and Graph-Based Abstract Syntax,” in *Software Languages: Syntax, Semantics, and Metaprogramming*, R. Lämmel, Ed., Cham: Springer International Publishing, 2018, pp. 87–108. doi: 10.1007/978-3-319-90800-7_3.

[81] Rockstar Games, “Red Dead Redemption 2.”

[82] “Projects · GitLab,” GitLab. Accessed: Oct. 28, 2023. [Online]. Available: https://gitlab.com/

[83] “Zacharias Pervolarakis / Autonomia Framework · GitLab,” GitLab. Accessed: Oct. 28, 2023. [Online]. Available: https://gitlab.com/zackper/Autonomia-Framework

[84] BillWagner, “C# docs - get started, tutorials, reference.” Accessed: Oct. 28, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/

[85] “Neo4j Desktop.” Accessed: Sep. 16, 2023. [Online]. Available: https://neo4j.com/download/

[86] “Neo4j Graph Database & Analytics – The Leader in Graph Databases,” Graph Database & Analytics. Accessed: Sep. 16, 2023. [Online]. Available: https://neo4j.com/

[87] “Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine,” Unity. Accessed: Sep. 16, 2023. [Online]. Available: https://unity.com

[88] “Among Us - Εφαρμογές στο Google Play.” Accessed: Oct. 27, 2023. [Online]. Available: https://play.google.com/store/apps/details?id=com.innersloth.spacemafia&hl=el

[89] “Play Fall Guys and stumble towards greatness!” Accessed: Oct. 27, 2023. [Online]. Available: https://www.fallguys.com/en-US/

[90] “Hearthstone.” Accessed: Oct. 27, 2023. [Online]. Available: https://hearthstone.blizzard.com/en-us

[91] “Blind Forest - Ori.” Accessed: Oct. 27, 2023. [Online]. Available: https://www.orithegame.com/blind-forest/

[92] “Using psychophysiological techniques to measure user experience with entertainment technologies.” Accessed: Oct. 28, 2023. [Online]. Available: https://www.tandfonline.com/doi/epdf/10.1080/01449290500331156?needAccess=true

[93] A. Karoulis, S. Demetriadis, and A. Pombortsis, “Comparison of expert-based and empirical evaluation methodologies in the case of a CBL environment: the ‘Orestis’ experience,” *Comput. Educ.*, vol. 47, no. 2, pp. 172–185, Sep. 2006, doi: 10.1016/j.compedu.2004.09.002.

Appendix A

Expert-Based Evaluation Part I Questionnaire

1. The agents were capable of adapting to moving objects in the scene.
2. NPCs responded in the same way either for audio or visual cues.
3. The NPCs would alter their behaviour based on the objects of the environment.
4. The NPCs were unable to adapt to the changes in the environment while the game was running.
5. The NPCs seemed to behave based on their preferences.
6. The NPCs would not alter their behaviours based on priorities or goals.
7. I would say the agents responded quickly.

Questions were rated from 1 to 5, where 1 represents “Strongly Disagree” where 5 means “Strongly Agree”.

Appendix B

Expert-Based Evaluation Part II Questionnaire

Overall Impressions

1. What are your initial impressions of the programming framework?
2. On a scale of 1 to 10, how would you rate the framework's overall usability.
3. Does Autonomia have the potential to enhance adaptability and believability of AI agents?

Functionality and Features

1. Which specific features or functionalities did you find most valuable or innovative?
2. Were there any missing features or functionalities that you expected to be present?
3. Can you describe any difficulties or challenges you encountered while using specific features?

Usability

1. Were you able to easily understand and navigate through the framework's user interface?
2. Did you encounter any confusing terminology?
3. How would you rate the learning curve for someone new to the framework?

Performance

1. How did the framework perform in terms of speed and responsiveness for your specific use cases?
2. Did you encounter any crashes or unexpected behavior?

Integration and Compatibility

1. Would you believe it would be possible to integrate other system's within Autonomia?

Scalability

1. Based on your personal experience, do you believe the system would be scalable?

Customization and Extensibility

1. Do you believe you would be able to customize or extend the framework to meet your specific needs?
2. Did you encounter any limitations when trying to modify or extend the framework's functionality?

Future Improvements

1. What improvements or additional features would you like to see in future versions of the framework?

Comparison

1. What unique advantages or disadvantages do you see in this framework?

Recommendation

1. Would you recommend this framework to your colleagues or peers in the industry? Why or why not?