DIPLOMA THESIS

A Psychoanalyst Artificial Intelligence Model in a Computer Game

Submitted at the
Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfilment of the requirements for the degree of
Electrical Engineering

under supervision of

Prof. Dr. Dietmar Dietrich
Institute number: 384
Institute of Computer Technology
Vienna University of Technology

and

Dipl. Ing. Alexander Wendt
Institute number: 384
Institute of Computer Technology
Vienna University of Technology

by

Enrique Muñoz Fernández
Matr.Nr. 1129167
Rautenstrauchgasse 5/212/2
A-1110 Vienna

September 2012

Abstract

Artificial intelligence is a promising topic, which is being developed to become helpful for humanity in usages such as robot control, house automation or medical diagnosis among others. However, simulating intelligence is a highly complex issue and many researchers are working on it in many different ways. Whereas most researchers are working on statistical methods or on other cognitive psychological models, in the Technical University of Vienna there is a pioneering implementation of a psychoanalytical model with a top-down approach based on the Id-Ego-Superego theory of the Austrian psychoanalyst Sigmund Freud. This model is called Artificial Recognition System (ARS from now on). Although this model is almost finished it has only been tested in a simple simulator made for this purpose.

The intention and the subject of this thesis are testing the ARS model in a more complex world which is not specifically designed for ARS. The result of this thesis is having an artificial agent handled by the ARS decision unit running in the computer game Unreal Tournament 2004 (UT2004). In order to achieve it, a connection interface between UT2004 and the ARS decision unit is done. This thesis talks about the whole interface but it is specialized in the body perception inputs and only a part of the environment perception inputs. The other part of the environment perception and the outputs are explained in more detail in [Ort12] because the work was divided. The results of the thesis demonstrate that the ARS decision unit is capable of working in an external environment which is more complex and that it behaves well. Each of the functionalities of the ARS will be verified with test cases in Unreal Tournament 2004. This thesis is only a starting point from which it would be possible to dig deeper into the adaptation or extraction of the ARS model in a wide variety of different environments or to do a more accurate implementation of it in Unreal Tournament 2004.

Acknowledgements

First of all, I would like to thank the Technical University of Catalonia (UPC) for giving me the opportunity to carry out my thesis abroad as well as the project leader, Dietmar Bruckner, for offering me to work inside the ARS project in the Technical University of Vienna (TU Wien).

I would especially like to thank my supervisor Alexander Wendt and also Clemens Muchitsch for their guidance and helpful support during the realization of this thesis. Without the help of both this thesis would not have been possible.

I would also like to express my gratitude to my family, especially to my parents, and my friends for their support in both the good and bad moments while I was doing my degree and for helping me to relax in periods of too much work load. I would also like to thank especially my cousin Rosa Martínez for reading the whole thesis and help me with the last corrections.

I should not forget to mention Ernest Ortuño, co-worker during the realization of this thesis. We have invested a lot of time on it but it was a worthy experience.

Agradecimientos

Primeramente me gustaría dar las gracias a la Universidad Politécnica de Cataluña (UPC) por concederme la oportunidad de realizar el proyecto final de carrera en el extranjero así como también a Dietmar Bruckner por ofrecerme trabajar dentro del proyecto ARS de la Universidad Técnica de Viena (TU Wien).

También quiero agradecer especialmente a mi tutor Alexander Wendt y a Clemens Muchitsch por guiarme y ayudarme durante la realización de la tesis. Sin su ayuda la realización de esta tesis no habría sido posible.

Me gustaría también expresar mi gratitud a mi familia, en especial a mis padres, y a mis amigos por su apoyo incondicional en los malos y buenos momentos a lo largo de toda la carrera y por ayudarme a desconectar en los momentos en que ha sido necesario. También quiero dar las gracias de manera especial a mi prima Rosa Martínez por leer toda la tesis y ayudarme con las últimas correcciones.

No debo olvidarme de Ernest Ortuño, compañero de trabajo durante la realización de esta tesis. Invertimos mucho tiempo en ello pero ha sido una grata experiencia

Table of contents

Abbreviations

AFSM	Augmented Finite State Machine
AI	Artificial Intelligence
ALMA	A Layered Model of Affect
ARS	Artificial Recognition System
BDI	Belief-Desire-Intention
CBR	Case Based Reasoning
DU	Decision Unit
EEC	Emotion Eliciting Conditions
EIS	Environment Interface Standard
FPS	First Person Shooters
GUI	Graphical User Interface
GB2004	Gamebots 2004
GWT	Global Workspace Theory
IDE	Integrated Development Environment
IRMA	Intelligent Resource-Bounded Machine Architecture
IVA	Intelligent Virtual Agent
MIT	Massachusetts Institute of Technology
MAS	Multi-Agent System
OCC	Ortony, Clore and Collins
PRS	Procedural Reasoning System
SOAR	State, Operator and Result
TP	Thing Presentation
TPM	Thing Presentation Mesh
UE2	Unreal Engine 2 Runtime
UT2004	Unreal Tournament 2004
WP	Word Presentation

1. Introduction

Artificial Intelligence is a promising topic that wants to become very helpful for the humanity in usages such as robot control, house automation or medical diagnosis among others. To get to this point, a lot of people are working on implementing new AI architectures in order to find the most humanistic one. The architecture, which we are going to work with in this thesis, is called ARS (Artificial Recognition System). It was developed by the Institute of Computer Technology in the Technical University of Vienna and is based on the psychoanalytic theory of Id-Ego-Superego from Sigmund Freud[1].

In a few words, the ID represents the needs of the body, the Superego contains the social rules and the Ego has to manage with the two previous for trying to satisfy the necessities in a socially correct way. The ARS model [Deu11, pp.67-107] seems to be the first model in the research area of Artificial Intelligence based on a psychoanalysis theory, the second topographical model of Sigmund Freud, and also using a top-down approach. Other models are commonly based in statistical methods or in cognitive psychology instead of psychoanalysis. The ARS implementation in Unreal Tournament 2004 is pioneer, but there are already implementations of models such as SOAR or BDI (e.g. Goal Agent).

SOAR [Lai87] stands for State, Operator and Result, and it is a symbolic cognitive architecture. In order to make a decision when a problem occurs, a SOAR architecture searches through a problem space (the list of different possible states that the system can reach at a specific time) for a goal state (which corresponds to the solution of the problem).Thus, in general words, it looks for a solution in the previously saved acknowledgements that it has. BDI [Bra87] stands for Belief, Desire, Intention where Belief represents the informational state of the agent, Desire the motivational state of the agent and Intention the deliberative state of the agent.

Before implementing these decision unit architectures in a robot or any other type of gadget, they have to be deeply tested with simulators in different kind of environments and fix the problems that may appear during the simulation or try to improve the behaviour in it. This is precisely the aim of this thesis: to implement the ARS model into a new different environment. It is important to realise that in this working field two different kinds of professionals have to work together: psychoanalysts and engineers, in order to make both subjects become one thing.

[1] Sigmund Freud (1856-1939) was an Austrian neurologist who founded the discipline of psychoanalysis. He developed theories about the unconscious mind and the mechanism of repression, postulated the existence of libido (an energy with which mental process and structures are invested), developed therapeutic techniques and proposed that dreams help to preserve sleep by representing as fulfilled wishes that would otherwise awake the dreamer. [15]

1.1 Motivation

At the moment, the ARS decision unit has only been tested in a simple simulator created just for this aim called ARSIN [Deu11, p.116]. For this reason, the motivation of working into the implementation of such decision unit into a more complex world appeared, in this case the computer game Unreal Tournament 2004. This will give the opportunity to check the ARS model more deeply and detect if it contains any kind of conceptual errors and therefore solve them. Moreover, inserting the ARS decision unit in another environment will provide new requirements from possible applications for ARS. These new requirements will have to be fulfilled during the realization of this thesis. Another motivation for doing this thesis is that at the end of the thesis some psychoanalytical use cases can be implemented and therefore a psychological evaluation of the ARS bot in the UT2004 can be done. Due to time limitations the ARS bot is not going to be finished but later, when a complete bot is done, it can participate in the Botprize competition [4] (see also Chapter 6.5) where the human similarity will be measured and a comparison among ARS and other AI architectures will be done. After the participation of the ARS bot in the Botprize competition [4] it will be possible to say whether the ARS architecture is better than the others or not. Until then it would be risky to affirm anything about this topic.

After this thesis, the ARS will be more extractable and it will be easier to implement it to other simulation worlds just doing a few changes. The ARS research group will be able then to test the model in different environments. After the main goal of this thesis is achieved, new working lines will appear. For this reason at the end of this thesis (see Chapter 7.3) new proposals are made in order to continue improving the ARS system.

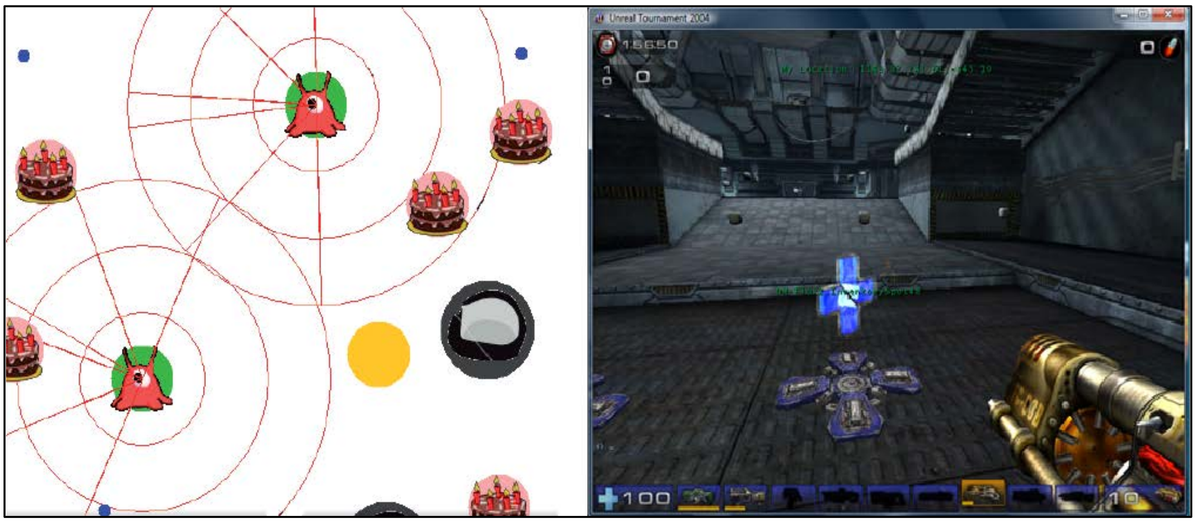


Figure 1.1: ARS world vs. UT2004 world [10]

1.2 Problem statement

The ARS research group [3] has an implementation of the ARS decision unit which they are continuously improving. To improve and test the existing decision unit they created the simulator ARSIN in order to detect if something was not working properly and correct it. At this point, they decided that it would be good to implement the existing ARS framework into another environment that can be more similar to the reality than the ARSIN. The final goal of any AI system is to be the more human-like as possible and this can be a good way to improve the ARS in order to achieve the goal. To do it, a bot in the computer game Unreal Tournament 2004, which is chosen as the new environment, is going to be done. By analysing the behaviour of the ARS bot in UT2004 it is possible to decide how human are the decisions taken by the ARS. Moreover, when an ARS bot with all the functionalities is done, it can take part in the Botprize competition (see Chapter 6.5) where the humanity of the ARS bot will be tested by an external jury and also a comparison among other bots will be done. To create a bot in UT2004 which is commanded by the ARS decision unit it is necessary to create an interface between both sides and connect them. To connect them means to receive the outputs from one side and insert them in the inputs of the other side and the same in the other way. A brief analysis of the work was previously done and the following possible difficulties were identified:

- The ARS had not been used in an external world before and it is not prepared to it yet. For this reason some difficulties may appear in order to obtain the outputs and to add the new inputs to the decision unit.
- Unreal Tournament 2004 world is totally different to the existing world, so ARS actions and items have to be modified to allow the ARS agent to live in the new world. The interface will have to process the UT2004 sensors information and adapt them in such a way that ARS can understand it.
- The way of navigating of the ARS in his own simulator is completely different to the computer game and the ARS resolution is much lower than the UT2004 resolution. It will be necessary to think about how to implement the navigation in Unreal Tournament 2004.

These three issues above need to be solved in order to achieve the goal of this thesis and achieve a successful connection between the ARS and the UT2004.

1.3 Task setting

The objective of this thesis is to implement the ARS Decision Unit in an external simulation world that is more complex than the existing one (specifically in the Unreal Tournament 2004 computer game). To achieve it, the following steps are considered to do:

- Extract the ARS inputs and outputs.
- Make the ARS independent from the current simulator.
- Adapt the outputs of the UT2004/Pogamut engine to the demanded inputs of the ARS bot.
- Adapt the outputs of the ARS bot to the UT2004 engine.
- Once the information is adapted, match the UT2004 outputs to the ARS inputs and the ARS outputs to the UT2004 inputs.
- Extract requirements for ARS based on usage of the agent in UT2004.

All the above mentioned steps have to be followed in a precise way to define independent interfaces that can be reused in other environments different to the UT2004 with only little modifications.

Once the connection between UT2004 and ARS is successful, some use case will be defined in order to check that the ARS decision unit is working properly inside the UT2004 environment.

1.4 Methodology

This thesis can be divided into two parts: the theoretical part and the practical part. Chapter 2 is related to the background of the thesis and Chapter 3 provides a description of all the used frameworks in this thesis and the requirements on the ARS model. Therefore, these two chapters constitute the theoretical part, whereas the rest of the chapters are related to the practical work made in this thesis.

Focusing first on the theoretical part: a large number of different Artificial Intelligence architectures already exist. For this reason, in this thesis an explanation of the architectures which are considered to be more important is done in Chapter 2 in order to make the reader to have a general knowledge of AI architectures. The architectures here described are: the Subsumption architecture, the Belief, Desire and Intention architecture (BDI), the State, Operator and Result architecture (SOAR).

A description of similar projects that also consist of the creation of an Unreal Tournament 2004 bot is also done in order to know how other UT2004 bots are working, increase the general knowledge of the subject of this thesis and also get some ideas of how the bots are created and how do they connect their decision unit with the UT2004.

In Chapter 3 there is an explanation of why the UT2004 is used for this thesis and not any other computer game as well as the reasons for using the Pogamut platform. The used frameworks in this thesis: UT2004, Pogamut and ARS are also explained in this chapter and a set of requirements on the ARS model are given as well.

Before doing the practical work of the thesis the next steps were followed:

- Read about other artificial intelligence architectures in order to have a general knowledge about it and be easier to understand all what is necessary for this work.
- Read carefully the documentation about the ARS and learn how it works.
- Read also about other implemented bots in UT2004 and get some ideas of how to implement the ARS bot in UT2004.
- Do a research on how to access the inputs and outputs of the UT2004 from the Eclipse IDE to create a connection between them.

The real aim of this thesis, which is to implement the ARS decision unit in the UT2004, is done in the practical part whose structure is next described: In Chapter 4, there is a conceptual description of how the implementation of the ARS in UT2004 is done. In Chapter 5 the structure of the written code is given using some UML diagrams for this purpose. Afterwards, some use cases are defined in Chapter 6. The behaviour of the ARS bot in each of the use cases is analysed in the same chapter. The Botprize competition [4] is presented in this chapter too because it could be useful to validate the ARS system in the future. Afterwards, the achievements of this thesis are written and new working lines for the ARS research group are proposed in Chapter 7. In this chapter, the project Emohawk is also presented because it can be useful for a future implementation of the ARS on it.

1.5 Limitations

Due to time limitations at the end of the thesis the achieved bot was not complete, lacking some of the necessary functionalities of the bot to be able to navigate individually through an UT2004 map. For this reason, at the end of this thesis the ARS bot is not ready to take part in the Botprize competition. However, the ARS research group can now continue the work started by this thesis and participate in Botprize later. For helping them a serial of proposals are done in Chapter 7.3 in order to continue with the implementation of the ARS bot and other working lines to improve the ARS as well.

2. State of the Art

It is important to provide general knowledge about Artificial Intelligence to the reader before explaining the work that was indeed realized during this thesis. For this reason, this chapter gives a general overview on some of the most well-known AI architectures followed by the description of three different Unreal Tournament 2004 bots. Thus, the reader will be able to roughly compare the ARS bot, which is going to be implemented in this thesis, with the other bots here described.

2.1 Decision Making Architectures

The most important and well-known AI architectures, basically because of their amount of related applications are: the subsumption architecture, Belief, Desire and Intention (BDI) and State, Operator and Result (SOAR). These three architectures are next described:

2.1.1 Subsumption Architecture

Subsumption architecture is an AI concept first described in [Bro86] by Rodney Brooks in the year 1986. Brooks was a professor of the M.I.T working in the Artificial Intelligence Laboratory whose goal was to develop artificial and complete creatures capable of inhabiting the real world and not a simplified one. He defined his goal after seeing the results of the robot “Shakey”. Although “Shakey” was using a big amount of computational power, it was only able to operate well within a laboratory environment that was designed on purpose. If one element of the environment was changed, “Shakey” was forced to sit around wondering on the change before it could take any action.

The “divide and conquer” approach (Figure 2.1), which was the traditional robot control system, lacks robustness according to Brooks because each sub-system is required for the robot to function. For this reason Brooks defined the new architecture based in the following ideas [11]:

1. Complex behaviour can emerge from a complex environment, even if the system itself is not incredibly complex.
2. Things should be simple: First, if a system starts to become unnecessarily complex, it is probably wrong. Second, components should not be designed merely to fix a single problem within the system.

3. Robots should be able to make useful three dimensional maps of the world that would allow it to wander around independently of humans.
4. Robots should be able to operate in the real world and not only in a specific environment.
5. Robots must be robust.
6. Robots must be self-sustaining.

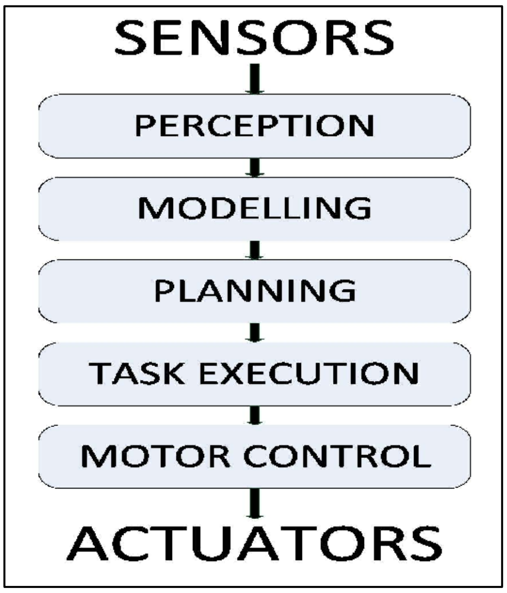


Figure 2.1: The traditional model for robot control system ("divide and conquer" approach)

Brooks also believed that for creating true intelligence it was necessary that the intelligences were situated and embodied. To be a situated intelligence means that it has to respond in real time to events in the real world causing the robot to react in a dynamic way. Embodied refers to an intelligence that have some kind of physical form. Brook's reasoning behind this idea is that it forces the developers to implement the ideas and not only theorize about what is possible.

The subsumption architecture builds control layers stacked one over the other rather than splitting the decision process into a set of serial tasks as before. Each of the layers can obtain information from the layers below but not from the layers above. Instead of using a central control that decides which layer to use in every case, all layers operate in an asynchronous mode. Brooks said: "complex capabilities are built on top of simpler capabilities".

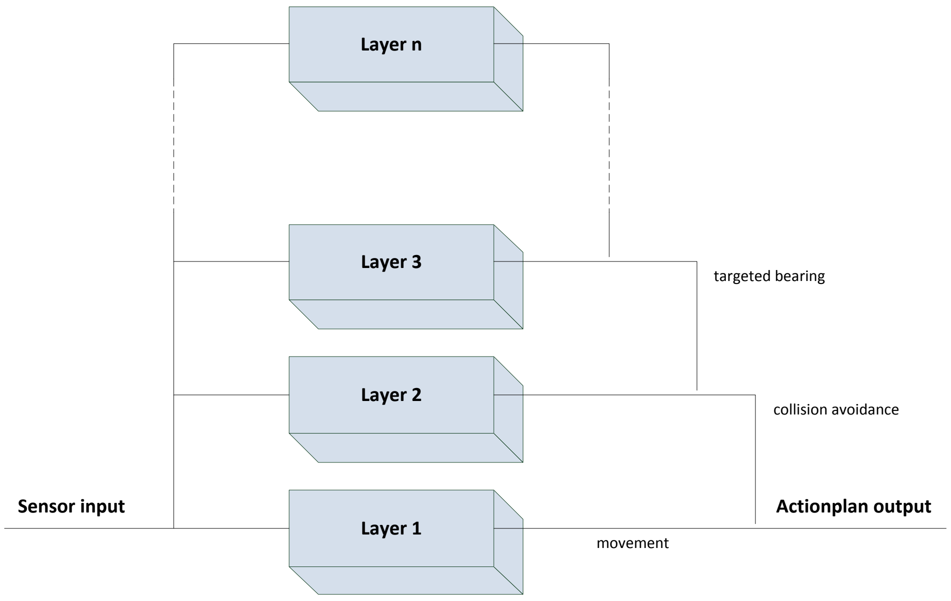


Figure 2.2: Subsumption architecture

Each of the layers that are visible in Figure 2.2 is implemented as an Augmented Finite State Machine (AFSM) that has the following four possible states [11]:

1. Output: An output message is created and sent and a new state is entered.
2. Side effect: One of the module's variables is set to a new value and a new state is entered.
3. Conditional dispatch: Using its variables, the module makes a calculation on the input which determines what the next state will be.
4. Event dispatch: The module monitors the incoming messages. When a sequence of events becomes true, the module branches to one of a series of states.

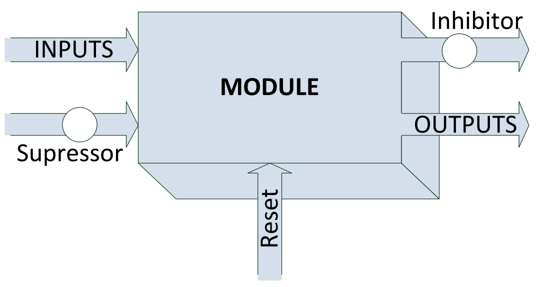


Figure 2.3: A single module of the subsumption architecture

The most important thing of this architecture is that there is no common goal for every module. Thus, each module has its own goal and there is no central control within all the layers. In this way, a robot can pursue multiple goals at the same time. Brooks thinks each layer as a "level of competence", defining it as an informal specification of a desired class of behaviours for a robot over all environments it will encounter. Therefore, it is possible to increase the complexity of a robot by adding more layers on the top when the layers above are working. For this reason this is clearly a bottom-up design.

The specific robot designed by Brooks with this architecture had the structure in Figure 2.4.

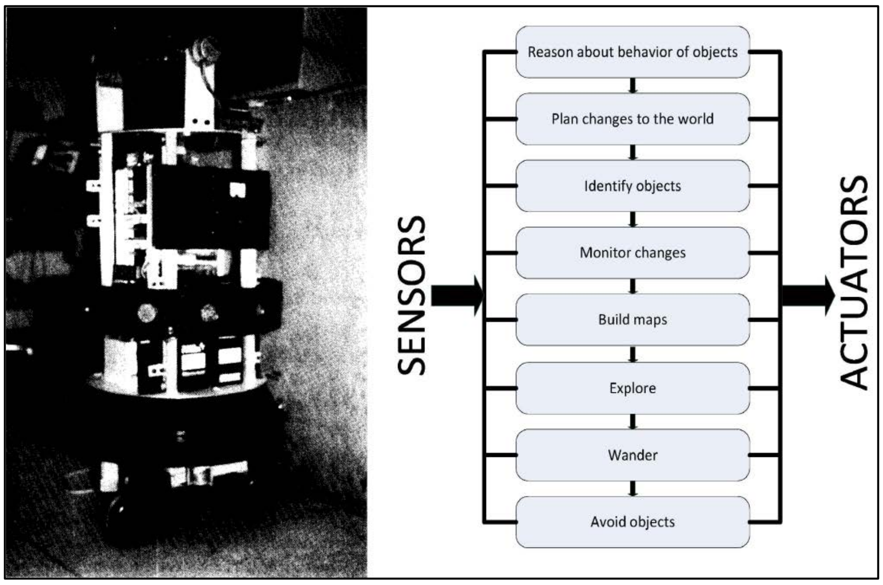


Figure 2.4: First subsumption robot (Allen) and its model for the robot control system [Bro86, p.5]

The subsumption architecture is commonly used in behaviour-based robotics. In [Hoo09] there is an example of an UT2004 bot whose controller uses a subsumption architecture evolution which is directly based on [Tog04].

2.1.2 BDI

BDI states for Belief, Desire and Intention. Among several approaches that propose different types of mental attitudes, BDI is the most adopted model. There are several reasons for its success, but perhaps the most important are that the BDI model combines a respectable philosophical model of human practical reasoning, (originally developed by Michael E. Bratman in [Bra87]), a number of implementations (in the IRMA architecture and the various PRS-like systems currently available), several successful applications (including the fault diagnosis system for the space shuttle, as well as factory process control systems and business process management), and finally, an elegant abstract logical semantics, which have been taken up and elaborated upon widely within the agent research community.

BDI is described as a philosophical theory of the practical reasoning in [Bra87], where the human reasoning is explained using the following concepts: beliefs, desires and intentions. The BDI model assumes that actions are deduced from a process which is called practical reasoning. Two steps compose this process: in the first step, a set of desires are selected to be achieved according to the current situation of the agent's beliefs; the second step is responsible for finding out how to achieve the specific goals created as a result of the previous step [Woo00, pp.21-46].

The three mental concepts that are part of the BDI model [Bra87] are described as follows:

- **Beliefs:** They symbolize the knowledge of the world: they store all the sensors data and combine it with the agent’s view of the world. Thus, it can be seen as the informative component of the system. Beliefs can be obtained through perception, communication or contemplation. They have to represent the reality and have to be obtained from valid arguments and evidence. The beliefs cannot be determined in one only sensing action because the world is changing constantly (is dynamic) so the information is needed to be updated appropriately after the perception of each action.

- **Desires:** They can also be called Goals. They save information about the objectives to be achieved and also about the priorities for each of them. Thus, they can be seen as the representation of the motivational state of the system.

- **Intentions:** They symbolize the current selected action plan. They can be seen as the deliberative component of the system. Therefore the agent chooses an intended action that will satisfy its desires given the current beliefs.

In the year 1995 (7 years later than Bratman’s BDI description) Rao & Georgeff used the BDI model for software agents and introduced a formal theory [Rao95] and an abstract BDI interpreter. Most of the actual and past BDI systems are founded on this interpreter. The interpreter works using the beliefs, desires and intentions of the agent, which correspond to the concepts of the mental notions with only few changes. The most significant change is that goals are a set of consistent specific desires that can be achieved all together, avoiding the need of a complex phase of goal deliberation. The principal task of the interpreter is to realize the means-end process by choosing and executing the right plans for a certain goal.

The process of practical reasoning in a BDI agent is presented in Figure 2.5:

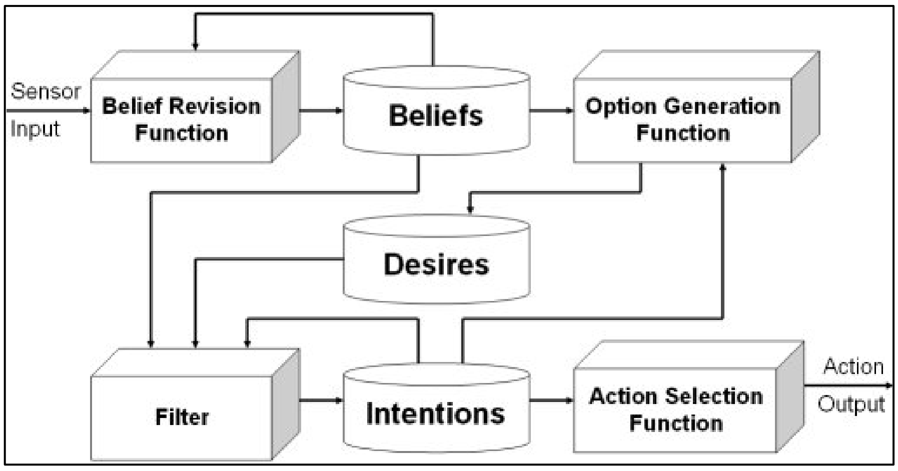


Figure 2.5: Process of practical reasoning in a BDI agent [Woo99, p.58]

As shown in this figure, there are seven main components in a BDI agent:

1. **Beliefs:** represent the information about the current environment of the agent.
2. **Belief revision function**: it creates a new set of beliefs based on a perceptual input and the current beliefs.
3. **Option generation function**: it determines the desires of the agents based on the current beliefs and the current intentions.
4. **Desires:** they represent possible courses of actions available to the agent.
5. **Filter:** represents the agent's deliberation process and determines the agent's intentions on the basis of its current beliefs, desires, and intentions.
6. **Intentions:** they symbolize the agent's current focus (those states of affairs that it has tried to bring about).
7. **Action selection function**: determines an action to perform on the basis of current intentions.

Afterwards the BDI framework was enhanced with emotions [Jia07]. To explain the new framework Figure 2.6 is going to be used.

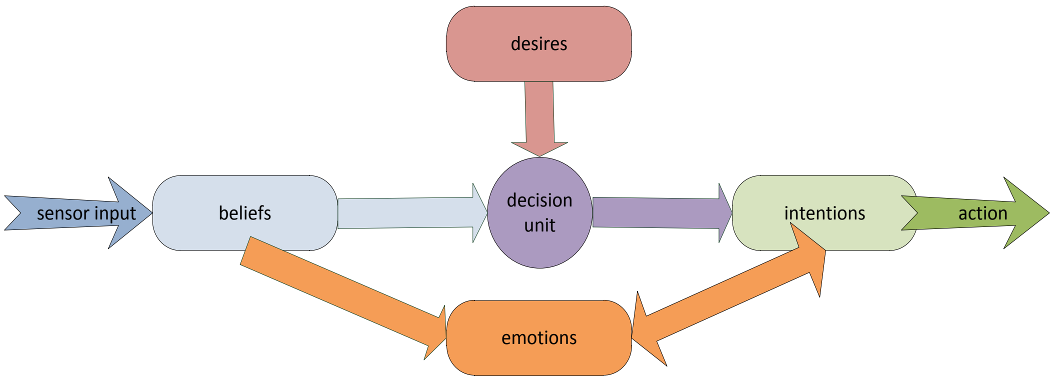


Figure 2.6: eBDI architecture

As shown in Figure 2.6, emotions take a role twice during the cycle belief→desire→intention. Primary emotions are generated immediately after updating the beliefs as a response to the changes in the view of the world. These emotions could be fear as a response to threats, sadness as response to the death of another agent, etc. These primary emotions are then used to adapt the intentions, which in turn generate secondary emotions. The process of adapting the intentions is carried out in a loop for an undetermined number of cycles and can produce changing emotions. For example, from anger to sadness, that triggers other intentions, which again produce emotions and so on.

Another adaptation to the concept has been made by [Ho03], [Ho04] through adding autobiographic memory to the agent. Both perceptions and actions are stored together as episodes in a way which allows to recapitulate past events both for adapting the agent’s own behaviour as well as to communicate these episodes to other agents for their profit.

In Chapter 2.2 there is an explanation of the UnrealGoal bot ([Dig11, pp. 1-18], [Hin10] and [Hin12]) which is a bot based on the BDI architecture.

2.1.3 SOAR

SOAR is a symbolic cognitive[2] architecture created by John E.Laird, Allen Newell and Paul S. Rosenbloom from the University of Michigan, Carnegie-Mellon and Stanford respectively and presented in [Lai87]. Its name states for State, Operator and Result.

A cognitive architecture is a fixed infrastructure that supports the acquisition and use of knowledge and it consists of [Lai08, p.2]:

- Memories for storing knowledge
- Processing units that extract, select, combine and store knowledge
- Languages for representing the knowledge that is stored and processed

To reproduce accurately the human cognitive mechanisms, short-term and long-term memories are included in the SOAR's model.

SOAR is based on operators, which are similar to reactive plans, and states (which include its highest-level goals and beliefs about its environment). Operators are rated by preconditions which help selecting operators for execution basing the decision on the agent's current state. Selecting high-level operators for execution leads to sub-goals and thus a hierarchical expansion of operators ensues. [Lai08]

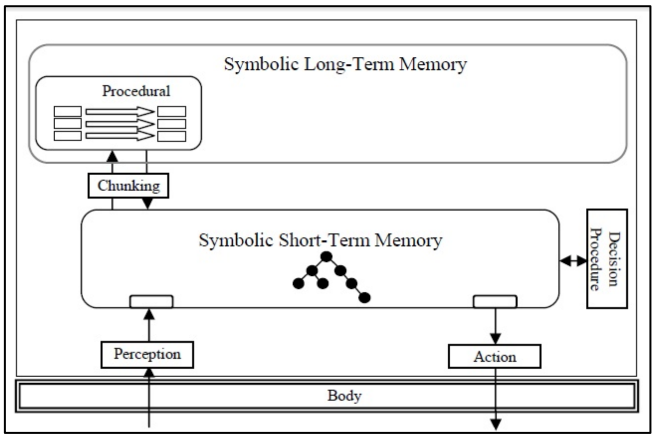


Figure 2.7: SOAR architecture [Lai08, p.3]

In Figure 2.7 the memory structure is showed and it works as follows: The perception sensor system receives information about the environmental state and forwards these information elements to the short-term memory that holds the current goal state. In SOAR it is possible to hold several goal states in parallel. The short-term memory is temporary and contains data elements that are necessary in order to compare perceived data with productions. As the system receives elements from a dynamically changing environment, the short-term memory serves as a buffer in order to stabilize the information load. In addition it is a bus that interconnects all memory modules. The long-term

[2] Cognitive: of, relating to, or being conscious intellectual activity (as thinking, reasoning, remembering, imagining, or learning words)

memory is a production system that compares productions with elements that are held in the short-term memory. If these elements match with defined productions, they are written in the short-term memory or removed from it. In contrast, the action system translates control information to the actuator. Goal-based problems are converted to productions and stored in the long-term memory. This conversion is called chunking which is the process of learning from experiences as new productions are formed and added to the long-term memory in case a problem is solved. [Lai87][Lai08]

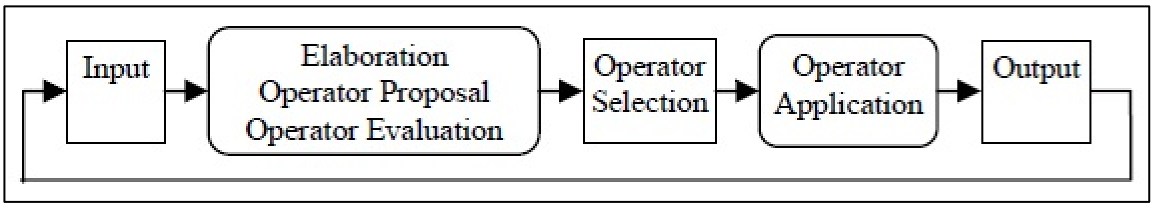


Figure 2.8: SOAR's processing cycle [Lai08, p.4]

As explained in [Lai08, pp.3-4] the SOAR has the processing cycle shown in Figure 2.8:

1. **Input:** Changes in the perception sensor are processed and sent to short-term memory.
2. **Elaboration:** Rules process entailments of short-term memory. For example, a rule might check if the goal is to catch an object and the object's distance, and then create a structure signalling if the object is in a reachable area or not.
3. **Operator Proposal:** Rules suggest operators which are suitable to the present situation basing on the characteristics of the situation, which is tested under the conditions of the rules.
4. **Operator Evaluation:** Rules create the order of priority of the suggested operators basing on the current situation and goal. These priorities can be represented either in a symbolic way (A is better than B) or in a numeric way (the estimated usefulness of A is 82).
5. **Operator Selection:** According to the previously generated preferences, the current operator is selected. If the preferences are insufficient for making a decision, an impasse arises and SOAR automatically creates a sub-state in which the goal is to resolve that impasse. In the sub-state, SOAR recursively uses the same processing cycle to select and apply operators, leading to automatic, reactive meta-reasoning [Lai08, p.4]. The impasses and resulting sub-states provide a mechanism for SOAR to deliberately perform any of the functions (elaboration, proposal, evaluation, application) that are performed automatically/reactively with rules [Lai08, p.4].
6. **Operator Application:** Rules which match the present situation and the present operator structure execute the actions of an operator. As multiple rules are able to fire either in parallel or in sequence, a flexible and expressive means of encoding operator actions is provided. Unless there is enough application knowledge, an impasse comes up with a sub-state leading to dynamic task decomposition. It is here where a complex operator is carried out recursively by simpler operators.
7. **Output:** The output commands are sent to the actuator.

An implementation of SOAR into the computer games Unreal Tournament and Quake is given in [Lai02] which uses the interface called SOAR General Input/Output (SGIO).

2.1.4 BDI vs. SOAR

The SOAR and BDI models have a lot in common. Indeed, the SOAR model seems to be fully compatible with the BDI architectures. It is possible to match the BDI concepts with SOAR as follows [Atal98, p.8]:

1. I***ntentions*** are selected operators in SOAR.

2. ***Beliefs*** are contained in the actual state in SOAR.

3. ***Desires*** are goals (including those generated from sub-goaled operators).

4. C***ommitment*** *strategies* are strategies for defining operator termination conditions.

For instance, operators may be terminated only if they are achievable, unachievable or irrelevant.

Bratman's insights [Bra87] about the use of commitments in plans are applicable in SOAR as well.

In SOAR, a selected operator (commitment) restricts the new operators (options) which the agent wants to consider. Specifically, the operator restricts the problem-space that is picked out in its sub-goal. This problem-space in turn constrains the choice of new operators that are considered in the sub-goal (unless a new situation causes the higher-level operator itself to be reconsidered). Either SOAR or BDI architectures have been used in several large-scale applications up to now. Thus, they share concerns of efficiency, real-time, and scalability to large-scale applications.

For instance, PRS and dMARS have been applied in air-combat simulation [Gos96], which is also one of the large-scale applications for SOAR. Despite such commonality, there are some key differences in SOAR and BDI models. Interestingly, looking at these differences, the two models seem to complement perfectly between each other. For example, SOAR research has typically fallen back on to cognitive psychology and practical implementations for rationalizing design decisions whereas BDI architectures have appealed to logic and philosophy [Atal98, p.9]. Furthermore, SOAR has often taken an empirical approach to architecture design, where systems are first constructed and some of the underlying principles are understood via such constructed systems. Thus, SOAR includes modules like the chunking one, which is a form of explanation-based learning and a proper system for maintaining state consistency. In contrast, the approach in BDI systems is first to understand clearly the logical or philosophical underpinnings and then build systems.

BDI theories could potentially inform and enrich the SOAR model, while BDI theorists and system builders may gain some new insights from SOAR's experiments with chunking and truth maintenance systems. The danger here is that both could end up reinventing each other's work in different disguises. [Atal98, p.9]

2.2 UnrealGoal Bot

The UnrealGoal bot is a bot for the computer game Unreal Tournament 2004 which is programmed in a specific language called Goal which is just made for programming agents. It was developed in the Delft University of Technology (The Netherlands). Both the language and the bot are next described in this chapter.

2.2.1 GOAL

Description extracted from [2]:

"GOAL is a programming language specifically created for programming rational agents. GOAL agents derive their choice of action from their beliefs and goals. The language provides the basic building blocks to design and implement rational agents. The language elements and features of GOAL allow and facilitate the manipulation of an agent's beliefs and goals and to structure its decision-making. The language provides an intuitive programming framework based on common sense notions and basic practical reasoning."

The main features of GOAL [2] are:

- **Declarative beliefs**: The information that the agents have and their beliefs or their knowledge about the environment where they move is represented by a symbolical, logical language in order to achieve their goals.

- **Declarative goals**: The agents have a variety of goals specifying their desires at any moment either in a near or a distant future. Declarative goals determine a state of the environment that the agent is seeking. However, they do not indicate actions or procedures of how to reach these states.

- **Blind commitment strategy**: Agents stick to their goals and they only drop them when they have been accomplished. This strategy (blind commitment strategy) is used by default by GOAL agents. This behaviour was built into GOAL agents because rational agents may not have goals that they are already achieved.

- **Rule-based action selection**: The process of selecting actions is done by the agents using the so-called action rules once their beliefs and goals are given. These rules have to specify the selection of action in a way that multiple actions can be performed at any time once the agent's rules are given. In such case, an arbitrary action will be selected for execution by the agent.

- **Policy-based intention modules**: To achieve a subset of their goals, agents may use a subset of their action and only relevant knowledge to the achievement of those goals. GOAL provides modules to structure action rules and knowledge dedicated to achieving specific goals. Modules can be seen as policy-based intentions according to Michael Bratman in [Bra87].

- **Communication at the knowledge level**: Communication among different agents is needed to exchange information, and to coordinate their actions. This communication is established by using the knowledge representation language used to represent their beliefs and goals too.

2.2.2 UT2004-GOAL Interface

After giving a brief description of the GOAL programming language we can describe the interface between the GOAL agents and the UT2004 bots basing the contents on [Hin10] and [Hin12].

The UnrealGoal Bot provides behaviour with three routines: Goto, Pursue and Halt. Goto deals with navigating the map, Pursue attempts to attack an opponent and follow him through the map and Halt makes the bot stand on the spot. The three routines are linked in a simple state machine (see Figure 2.9).

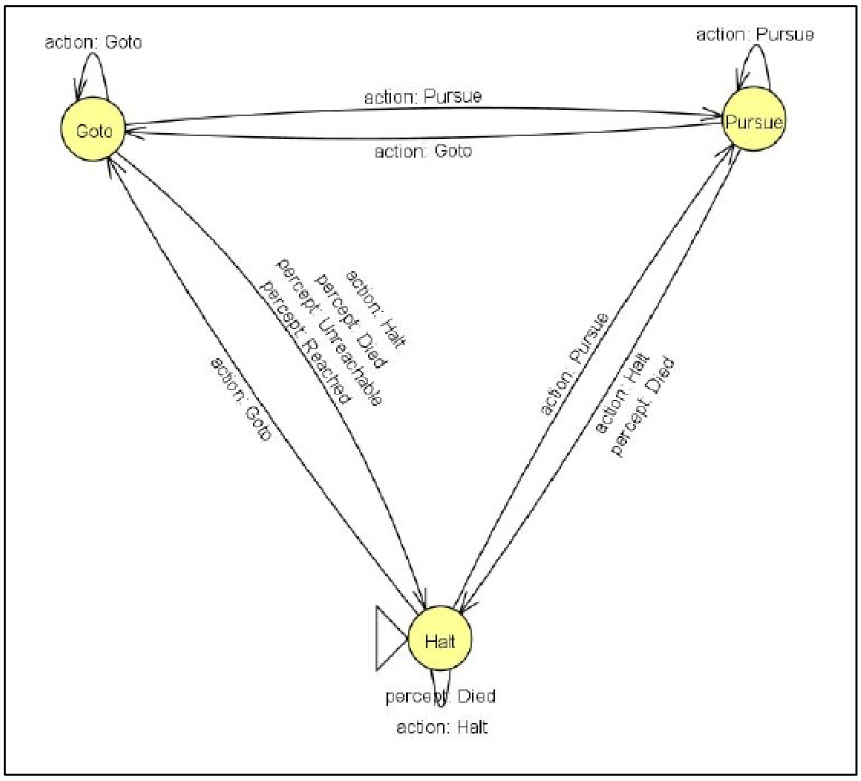


Figure 2.9: UnrealGoal Bot Behaviour State Machine [Hin12, p.11]

The bot can change the state in two different ways:

1. **Voluntary change:** by performing the Goto, Pursue or Halt action
2. **Forced change:** by reaching a location, being unable to reach a location or death.

The developers of the UnrealGoal Bot have chosen to use EIS[3] because it offers several benefits. First of all, it increases the reusability of environments. EIS makes complex multi-agent environments, for example gaming environments, more accessible. It provides support for event and notification handling and for launching agents and connecting to bots. EIS is based on several principles: the first one is portability which means in this context that the easy exchange of environments is facilitated. Environments are distributed via jar-files that can easily be plugged in by platforms that adhere to EIS. Secondly, it imposes only minimal restrictions on the platform or environment. For example there are no restrictions on the use of different technical options for establishing a connection to the environment. TCP/IP, RMI, JNI or wrapping of existing Java-code among others can be used. Another principle is the separation of concerns. Implementation issues related to the agent platform are separated from those related to the environment. EIS facilitates acting, active sensing (actions that yield perceptions), passive sensing (retrieving all perceptions), and perceptions-as-notifications (perceptions sent by the environment). Another principle is a standard for actions and perceptions.

The connection between Goal-agents (executed by the Goal interpreter) and UT2004 is established using EIS (see Figure 2.10). The connection environment consists of several varied components. The first component is Goal's support for EIS. Basically this becomes into a sophisticated MAS-loading-mechanism that creates agents and establishes the connection between them and entities, together with a mapping between Goal-perceptions/actions and EIS ones. Connection to EIS is facilitated by Java-reflection[4]. Entities, from the environment-interface-perspective, are instances of the UnrealGoal Bot. Pogamut is internally connected to GameBots, which is a plugin that gives the possibility of connecting UT2004 to external controllers using the TCP/IP protocol.

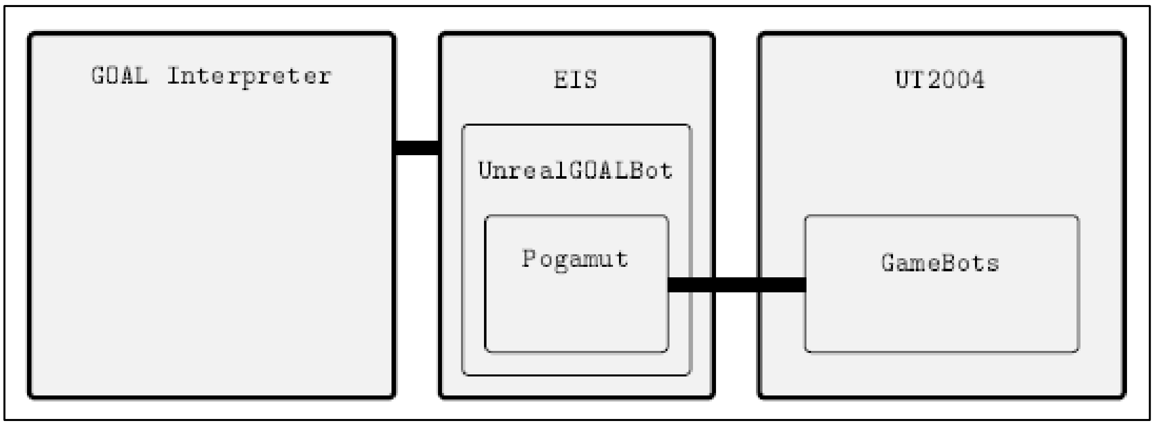


Figure 2.10: UnrealGoal Boat Interface [Hin10, p.10]

[3] The Environment Interface Standard (EIS) is a proposed standard implemented in Java for interfaces between platforms and environments. See [Beh09].

[4] Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it's possible for a Java class to obtain the names of all its members and display them.

Entities consist of three components: (1) an instance of the UnrealGoal Bot that allows access to UT, (2) a so called ‘action performer’ which evaluates EIS-actions and executes them through the UnrealGoal Bot, and (3) a ‘perception processor’ that queries the memory of the UnrealGoal Bot and yields EIS-perceptions.

The instantiation of EIS for connecting Goal to UT2004 distinguishes three classes of perceptions:

1. **Map-perceptions:** are sent only once to the agent and contain static information about the current map. That is navigation-points (there is a graph overlaying the map topology), positions of all items (weapons, health, armour, power-ups etcetera), and information about the flags (the own and the one of the enemy).

2. **See-perceptions:** consist of what the bot currently sees such as visible items, flags and other bots.

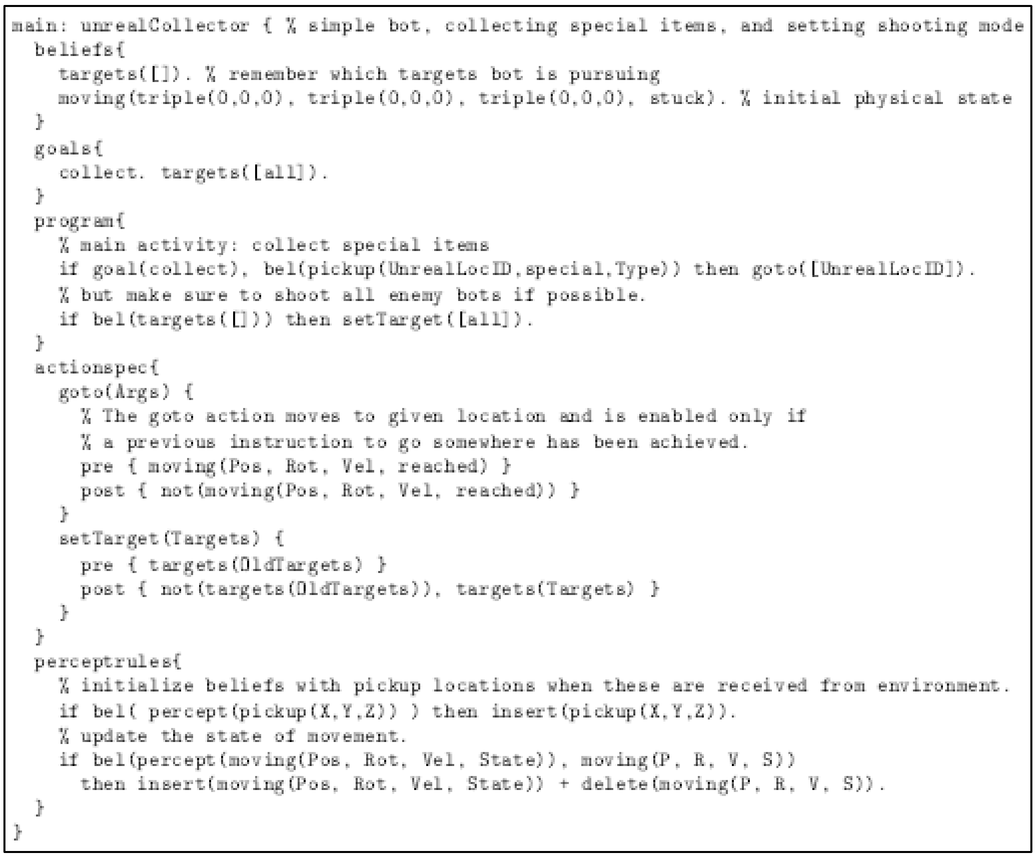
3. **Self-perceptions:** consist of information about the bot itself. That is physical data (position, orientation and speed), status (health, armour, ammo and adrenaline), all carried weapons and the current weapon.

We have identified several layers of abstraction, ranging from (1) really low level interaction with the environment, which is what the bot sees only neighbouring waypoints and can use ray-tracing to find out details of the environment, over (2) making all waypoints available and allowing the bot to follow paths and avoid for example dodging attacks on its way, to (3) very high-level actions like winning the game. Whereas the high level allows for longer reaction times but requires more implementation effort, the low level makes a very small reaction-time a requirement and is very easy to implement. We have identified the appropriate balance between reaction-time and implementation effort to be an abstraction layer in which we provide these actions: ‘goto’ navigates the bot to a specific navigation-point or item, ‘pursue’ pursues a target, ‘halt’ halts the bot, ‘setTarget’ sets the target, ‘setWeapon’ sets the current weapon, ‘setLookat’ makes the bot look at a specific object, ‘dropweapon’ drops the current weapon, ‘respawn’ respawns the bot, ‘usepowerup’ uses a power-up, ‘getgameinfo’ gets the current score, the game-type and the identifier of the bot's team.

2.2.3 An Example: The Unreal-Pill-Collector

A Goal-agent program consists of various sections: the belief base, the goal base, the program section, the action specification and a set of the perception rules. The belief base is a set of beliefs representing the current state of affairs. The goal base is a set of goals representing in what state the agent wants to be. The program section is a set of action rules that define a strategy or policy for action selection. The action specification is a specification of the conditions for each action available for the agent when an action can be performed (precondition) and the effects of performing an action (post-condition). Finally, a set of the perception rules specify how perceptions received from the environment modify the agent's mental state. Figure 2.11 shows the agent-code of a simple Goal-agent that performs two tasks: (1) collecting pills and (2) setting a target for attack. [Hin10, p.11]

Though this agent is simple it does show that it is relatively simple to write an agent program using the interface that does something useful like collecting pills.



```
main: unrealCollector { % simple bot, collecting special items, and setting shooting mode
  beliefs{
    targets([]). % remember which targets bot is pursuing
    moving(triple(0,0,0), triple(0,0,0), triple(0,0,0), stuck). % initial physical state
  }
  goals{
    collect. targets([all]).
  }
  program{
    % main activity: collect special items
    if goal(collect), bel(pickup(UnrealLocID,special,Type)) then goto([UnrealLocID]).
    % but make sure to shoot all enemy bots if possible.
    if bel(targets([])) then setTarget([all]).
  }
  actionspec{
    goto(Args) {
      % The goto action moves to given location and is enabled only if
      % a previous instruction to go somewhere has been achieved.
      pre { moving(Pos, Rot, Vel, reached) }
      post { not(moving(Pos, Rot, Vel, reached)) }
    }
    setTarget(Targets) {
      pre { targets(OldTargets) }
      post { not(targets(OldTargets)), targets(Targets) }
    }
  }
  perceptrules{
    % initialize beliefs with pickup locations when these are received from environment.
    if bel( percept(pickup(X,Y,Z)) ) then insert(pickup(X,Y,Z)).
    % update the state of movement.
    if bel(percept(moving(Pos, Rot, Vel, State)), moving(P, R, V, S))
      then insert(moving(Pos, Rot, Vel, State)) + delete(moving(P, R, V, S)).
  }
}
```

Figure 2.11: Example of an UnrealGoal Agent (Pill collector) [Hin10, p.12]

Looking at the code in Figure 2.18, it is visible that the initial beliefs state that the agent has no target and also states the physical state of the bot. The physical state is composed by the position, the rotation, the velocity and the state (the state could be stuck, moving or reached). The agent's goal base contains the single goal of collecting special items. The first rule in the program specification makes the bot go to the specific location of a special item if the agent knows its position and has the goal of collecting those. The second one sets the targets from none to all bots. The ‘goto’ action in the action-specification makes the bot move in the environment. The ‘setTarget’ action sets the target. The first percept rule stores all pickup-positions in the belief base whereas the second one stores the movement state. The code also illustrates that some of the tasks may be delegated to the behavioural layer. For example, the agent does not compute a route itself but delegates determining a route to a pickup-navigation point. [Hin10, p.11]

2.3 SomaticMarker Bot

The SomaticMarker bot was developed in the University of Zaragoza (Spain). It is based on the Somatic Marker hypothesis which is implemented using the technique Case Based Reasoning. Both are first described in this chapter in order to understand later how the SomaticMarker bot is implemented.

2.3.1 The Somatic Marker hypothesis

In [Dam94] Antonio Damasio[5], whose theory is also one of the bases of the ARS, tries to demonstrate that the emotions are needed for reasoning. He works with some patients that have damage in the central lobe of the brain. These patients do not perceive any kind of emotion feeling in front of sensible or dramatic situations and although they keep the cognitive faculties they are not able to make decisions. Thus, Damasio concludes that the decision making absence is due to the lack of emotions. Damasio describes that the before mentioned patients need an enormous amount of time to do trivial tasks and that they follow in most of the cases a bad decision making.

Damasio develops the Somatic Marker model to explain why their patients behave in this way. He says that is not possible to make a decision totally rationally because a lot of time should be required to evaluate all the different existing possibilities. Damasio says that the number of possibilities is highly decreased if the emotions are taken into account for the decision making process. Emotions help to do a first selection of the whole different possibilities to make a decision. In order to apply the Somatic Marker hypothesis different techniques exist. The SomaticMarker Bot uses one called Case Based Reasoning (CBR) which is described in the following section.

2.3.2 Case Based Reasoning (CBR)

Janet L. Kolodner[6] described in [Kol93] a new reasoning model called CBR which uses a specific knowledge from similar past problems to solve the new current problems and not only uses a general knowledge to solve them. A typical CBR system consists in a four steps cycle (see Figure 2.12):

- **Retrieve:** Given a target problem, it retrieves from memory the cases that are similar or relevant to solve it.
- **Reuse:** It links the solution from a previous case to the existing target problem. An adaptation of the solution to the new situation may be needed.

[5] Antonio Damasio is an internationally recognized leader in neuroscience. His research has helped to elucidate the neural basis for the emotions and has shown that emotions play a central role in social cognition and decision-making. [13]

[6] Janet L. Kolodner is an American cognitive scientist and Regents' Professor in the School of Interactive Computing, College of Computing at the Georgia Institute of Technology. She pioneered the computer reasoning method called case-based reasoning, a way of solving problems based on analogies to past experiences, and her lab emphasized case-based reasoning for situations of real-world complexity. [14]

• **Revise:** Having mapped the previous solution to the target situation, it tests the new solution in the real world (or a simulation) and if necessary revises it.

• **Retain:** After the solution has been successfully adapted to the target problem, it stores the resulting experience as a new case in the memory.

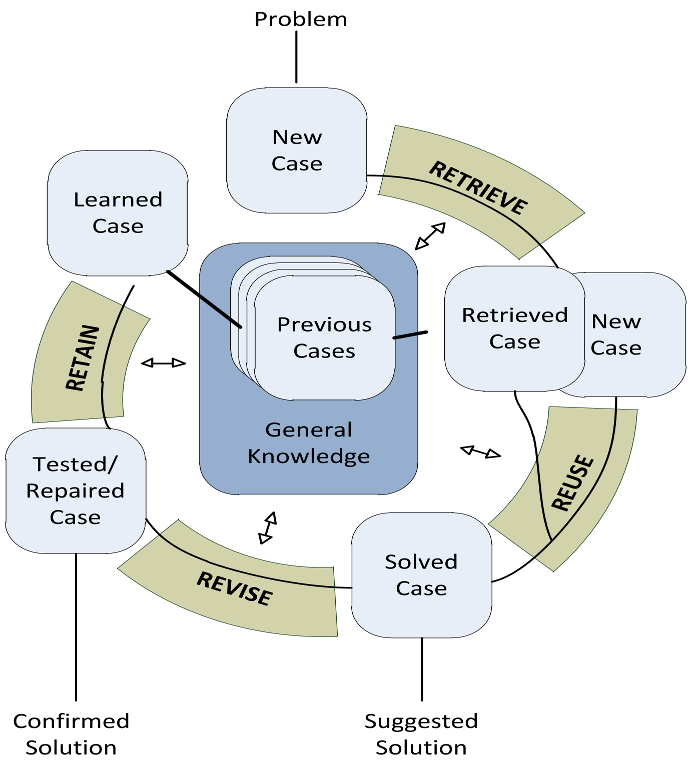


Figure 2.12: The CBR cycle

2.3.3 SomaticMarker Bot implementation

To use the CBR technique in a UT2004 Bot with the Somatic Marker hypothesis the CBR model has to be modified. The main difference with the classical CBR is that the cases are introduced at the beginning, before starting the execution of the bot, because it is not possible to introduce them during the execution of the Bot. Each of the cases introduced in the memory of the bot at the beginning consists of:

- **Environment variables:** They inform about the game state in a specific moment.
- **Behaviour:** The bot's behaviour of previous executions is stored. It is not possible to modify it in real-time.
- **Adaptability:** It is a value that indicates how well the environment variables and the previous stored behaviour in a specific case match. With this value the emotion concept of Damasio can be simulated in order to choose one of the available options to solve the problem. The pairs 'Environment variables – Behaviour' which have been negative for the Bot will decrease the adaptability value for future situations and the ones which have been positive will increase the adaptability value.

The behaviours have been created using binary trees. Depending on the conditions of the match (environment variables), one path or another in the tree is followed and one simple action or another is executed. The environment variables that are considered for the implementation of this Bot are the following:

- **DeathsToWin:** It gives the left number of deaths to win the match. It is useful to know if the behaviour of the Bot is being good or not.
- **People:** Specify if there are enemies in the vision.
- **EndGame:** Time left to end the match.
- **Armor:** Bot's instantaneous armor level.
- **Ammo:** Bot's instantaneous ammo level.
- **Health:** Bot's instantaneous health level.
- **MyDeaths:** Amount of times that the Bot has died.
- **Sensor:** It tells if the Bot was recently dead or if it is seeing a projectile in that moment. If it was recently killed it will probably not have weapons and is not appropriate to fight.
- **NWeapon:** Amount of weapons in the inventory.
- **NAmmo:** Amount of ammo in the inventory.

The simple actions considered for the SomaticMarker Bot are:

- **walk:** Walk from one Navigation Point to another.
- **protector:** Pick up the available health or armor objects.
- **pickHealth:** Pick up the available health objects.
- **pickArmor:** Pick up the available armor objects.
- **army:** Pick up any available weapon and any armor associated to any of the weapons in the inventory.
- **runAwayFromPlayer:** Flee from an enemy in the opposite direction if it is possible.
- **attackNearestVisiblePlayerWithSpeed:** Attack the nearest visible player using the fastest weapon.
- **attackNearestVisiblePlayerWithAmmo:** Attack the nearest visible player using the weapon that has more ammo.
- **attackNearestVisiblePlayerWithDamage:** Attack the nearest visible player using the weapon that damages the most.
- **attackNearestPlayerWithSpeed:** Attack the nearest player (visible or not) using the fastest weapon.
- **attackNearestPlayerWithAmmo:** Attack the nearest player (visible or not) using the weapon that has more ammo.
- **attackNearestPlayerWithDamage:** Attack the nearest player (visible or not) using the weapon that damages the most.
- **attackRandomVisiblePlayerWithAccuracy:** Attack a random visible player with the most accurate weapon.
- **attackRandomVisiblePlayerWithDamage:** Attack a random visible player with the weapon that damages the most.
- **attackFurthestVisiblePlayerWithAccuracy:** Attack the furthest visible player with the most accurate weapon.

- **attackFewerDeathsPlayerWithSpeed:** Attack the player with the least amount of deaths using the fastest weapon.
- **attackFewerDeathsPlayerWithAmmo:** Attack the player with the least amount of deaths using the weapon that has more ammo.
- **attackFewerDeathsPlayerWithDamage:** Attack the player with the least amount of deaths using the weapon that damages the most.
- **attackMostDeathsPlayerWithSpeed:** Attack the player with the highest amount of deaths using the fastest weapon.
- **attackMostDeathsPlayerWithAccuracy:** Attack the player with the highest amount of deaths using the most accurate weapon.
- **attackMostDeathPlayerWithDamage:** Attack the player with the highest amount of deaths using the weapon that damages the most.

We considered good to put the whole list of simple actions because it can help to get ideas in order to define the requirements of the ARS to implement the ARS bot in UT2004.

In the implementation of this specific SomaticMarker Bot here described there are sixteen different behaviour trees, so sixteen cases where the Bot has to find the appropriate one to decide how to behave in the actual situation. Figure 2.13 shows an example of one out of the sixteen behaviour trees.

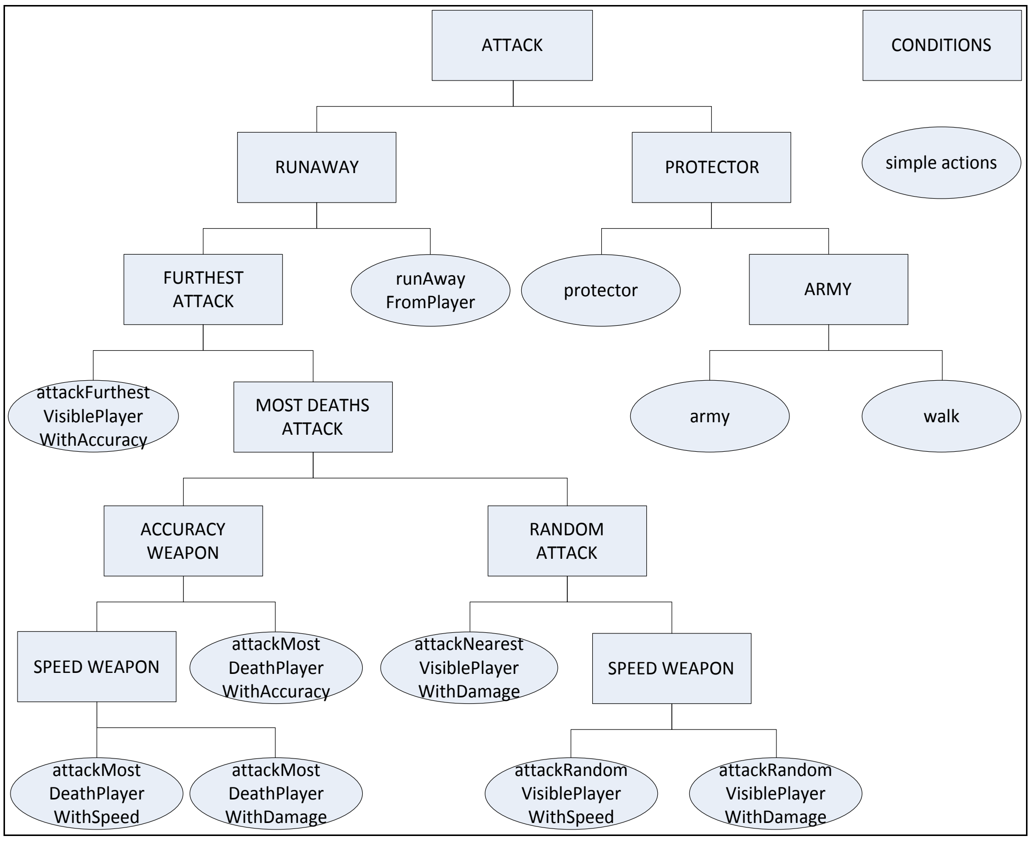

ATTACK
CONDITIONS
RUNAWAY
PROTECTOR
simple actions
FURTHEST ATTACK
runAway FromPlayer
protector
ARMY
attackFurthest VisiblePlayer WithAccuracy
MOST DEATHS ATTACK
army
walk
ACCURACY WEAPON
RANDOM ATTACK
SPEED WEAPON
attackMost DeathPlayer WithAccuracy
attackNearest VisiblePlayer WithDamage
SPEED WEAPON
attackMost DeathPlayer WithSpeed
attackMost DeathPlayer WithDamage
attackRandom VisiblePlayer WithSpeed
attackRandom VisiblePlayer WithDamage

Figure 2.13: Example of one of the sixteen behaviour trees

2.4 UT^2 Bot

The UT^2 Bot ([Sch11] and [Sch12]) was developed by the **U**niversity of **T**exas at Austin for use in the game **U**nreal **T**ournament 2004, hence the exponent of 2 after UT in the name. The architecture controlling UT^2 is a behaviour-based approach. The bot has a list of behaviour modules and each one has its own triggers. On every time step the bot iterates through the list checking triggers for each module until one of them becomes true. The module associated with the chosen trigger takes control of the bot for the current time step. Each module can potentially have its own set of internal triggers that further subdivide the range of available behavioural modes. The full architecture is shown in Figure 2.14.

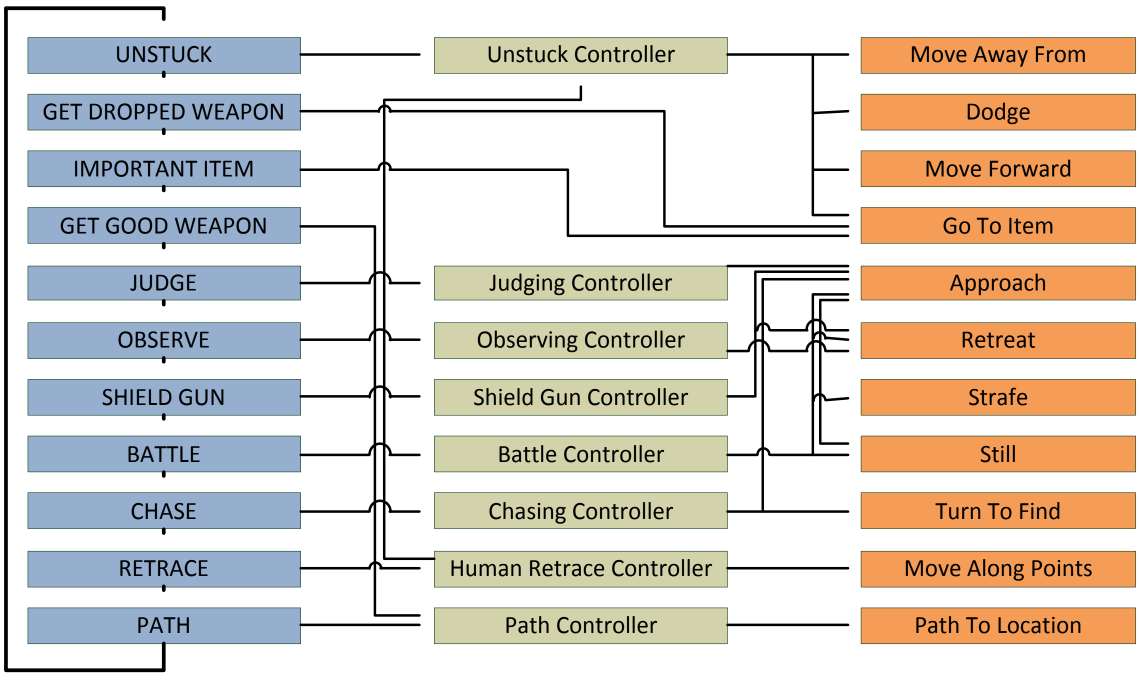


Figure 2.14: UT^2 architecture

The controller cycles through modules listed (on the left) once every logic cycle. If a module's trigger fires, then that module will define the bot's next action. Most modules have an associated controller (middle column) that further arbitrates between several available actions, or otherwise aggregates and makes use of information relevant to the actions performed by that module. All actions available to the controllers are in the right column. One of these actions is executed each logic cycle. Some control modules are too simple to need controllers: They carry out a specific action directly. Most of the control in this diagram flows from left to right, but note that the Unstuck Controller can actually make use of the Human Retrace Controller to define its action. This behaviour-based architecture modularizes bot behaviours, making the overall behaviour easier to understand, and making programming and troubleshooting easier.

The control modules, from highest to lowest execution priority are:

1. **UNSTUCK:** Getting unstuck has highest priority since being stuck is a very bot-like behaviour that prevents the execution of other actions.

2. **GET DROPPED WEAPON:** Causes the bot to rush and pick up weapons that enemies drop upon dying.

3. **IMPORTANT ITEM:** Makes the bot pursue items such as UDamage (which doubles the damage dealt by the bot for 30 seconds) or Health whenever they are nearby, even if it means breaking off from combat. Obtaining these items is considered more important than fighting.

4. **GET GOOD WEAPON:** Whenever the bot has only the starting weapons, it is not able to put up much of a fight. Therefore, running to get a good weapon is more important than fighting.

5. **JUDGE:** The judging gun has infinite ammo, so it is possible to judge at any time. This module decides when to judge a given opponent, and whether the opponent should be judged as a human or bot.

6. **OBSERVE:** In the judging game variant of UT2004 used for Botprize (see Chapter 6.5), it is very common for humans to stand back and watch other players, particularly groups of opponents. This module tries to emulate such observation behaviour.

7. **SHIELD GUN:** The shield gun is a melee weapon with regenerating ammo that is very hard to use. It also allows players to shield themselves from incoming projectiles. However, despite its versatility, it is harder to use than the standard projectile weapons available. Therefore, the bot only uses the shield gun if it is out of ammo for all other weapons and has no reason to judge or observe.

8. **BATTLE:** If the bot does have ammo when encountering an enemy that it neither wants to judge nor observe, then it enters combat.

9. **CHASE:** If an opponent is lost during combat, the bot will chase after it.

10. **RETRACE:** If there are no interesting items or opponents to interact with, the bot simply explores the level. This module uses replay of human traces to explore the map, when the traces are available.

11. **PATH:** As a failsafe for when human trace data is not available, the bot can explore using the level's built-in navigation graph.

Most of the UT^2's combat behaviour can be attributed to the battle controller (see Figure 2.14). Furthermore, the majority of the interactions between the bot and the other players occur during combat, so the bot's capacity to appear human depends a lot on the battle controller, which is briefly described next.

The battle controller used by UT^2 processes inputs based on the bot's sensors every time that the battle controller is in use and produces several outputs for each set of inputs. At the same time the outputs are used to produce an action for the UT^2 bot. The inputs are such as: enemy, ray-tracing geometry, damage, movement, shooting and ledge sensors as well as sensors for different type of items that appear in the map. Eight outputs are defined, where five of them are related to the movement (ADVANCE towards opponent, RETREAT from opponent, STRAFE left/right, GOTO Item, and STAND STILL) and the other three outputs determine whether the bot shoots, which firing mode to use and whether or not to jump. These outputs will produce one of the actions that are available (see right column in Figure 2.14).

A limitation on how UT2's behaviour was evolved is that only the battle controller was evolving within a bot that had many other components. Such an evolved battle controller could be integrated with other evolved sub-controllers to build up a hierarchical controller as was done in [Hoo09] whose approach was directly based on [Tog04]. The method used in these works involves evolving the components of a subsumption architecture [Bro86] within several separate subtasks leading up to the full task. [Sch12, p.26]

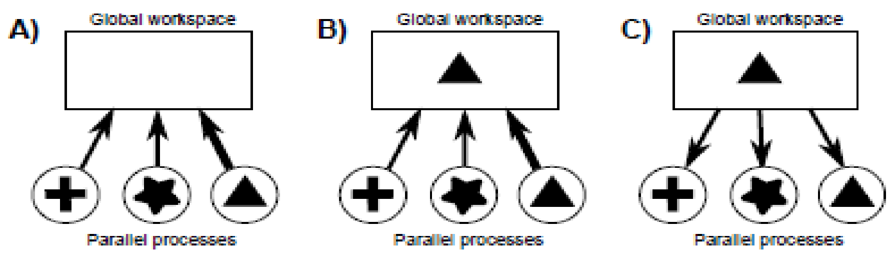
2.5 NeuroBot

The NeuroBot is another bot implemented in the computer game Unreal Tournament 2004. However, it is quite different to the previous bots because it is based on a neural network which uses a different cognitive architecture which is called global workspace theory. The global workspace theory is first explained in general in order to understand later better the NeuroBot's architecture. All the information of this chapter is extracted from [Fou11a] and [Fou11b].

2.5.1 Global Workspace Theory

The global workspace theory (GWT) is a simple and widely used cognitive architecture developed by Bernard J. Baars in [Baa88, pp. 27-51] as a model of consciousness. The basic idea is that a number of separate parallel processes compete to place their information in the global workspace. The winning information becomes conscious and therefore becomes globally accessible to the rest parallel unconscious cognitive processes (see Figure 2.15). To support this idea, GWT believes that a big part of the brain consists of highly specialized areas. The global workspace theory was later described using a theatre metaphor by Baars in [Baa97, pp. 292-309]. This metaphor makes the GWT more comprehensible. In the theatre there is a stage, which is compared to the working memory, where actors are moving in and out making speeches and competing (or sometimes cooperating) in order to be in the spotlight. The players are compared to a number of processors in working memory while the spotlight is a mechanism for attention which reveals the contents of the consciousness. In the theatre it exists also the audience, which is watching the play and it is compared to all other processors of the system that have access to what becomes visible by the spotlight and they are only activated when something interesting happens or someone from the cast calls them. The last element of the metaphor is the backstage, which corresponds to the contextual systems where each context is a non-conscious coalition of processors that shape conscious contents without ever becoming conscious themselves.

The NeuroBot is a neural system considered as a distributed parallel system where a number of different specialized processes coexist. Therefore the above described architecture is suitable for it.



A) Three parallel processes compete to place their information in the global workspace
B) The process on the right wins and its information is copied into the global workspace
C) Information in the global workspace is broadcast back to parallel processes and the cycle of competition and broadcast begins again

Figure 2.15: Operation of the global workspace architecture

2.5.2 NeuroBot Architecture

The NeuroBot is a system that uses a global workspace architecture implemented in spiking neurons[7] to control an agent within the Unreal Tournament 2004 computer game. The architecture uses a biologically-inspired approach and it is based on theories about the high level control circuits in the brain being the first neural implementation of a global workspace that is embodied in a dynamic real time environment. The neural network consists of approximately 20.000 neurons which are implemented using the Izhikevich model (see Appendix D). The network is divided into the following specialized modules:

Workspace modules

The global workspace provides the system with a working memory where all the different kinds of active modules can broadcast information that all other modules can access at any time. Also, it facilitates cooperation and competition between the behavioural modules and ensures that only a single module controls the motor output at any point in time. In this design, the global workspace consists of four identical layers, which are used as a kind of working memory and are named workspace modules, which are connected to each other in a chain manner (see Figure 2.16). The first area concerns the desired state of the body of the agent while the second area represents the proprioceptive sensory data, i.e. the incoming information about the real state of the agent's body. The third type represents the exterioceptive sensory data, i.e. the information about the visible part of the environment. Finally, the fourth type represents the internal state of the agent and in this case includes only the level of its health condition.

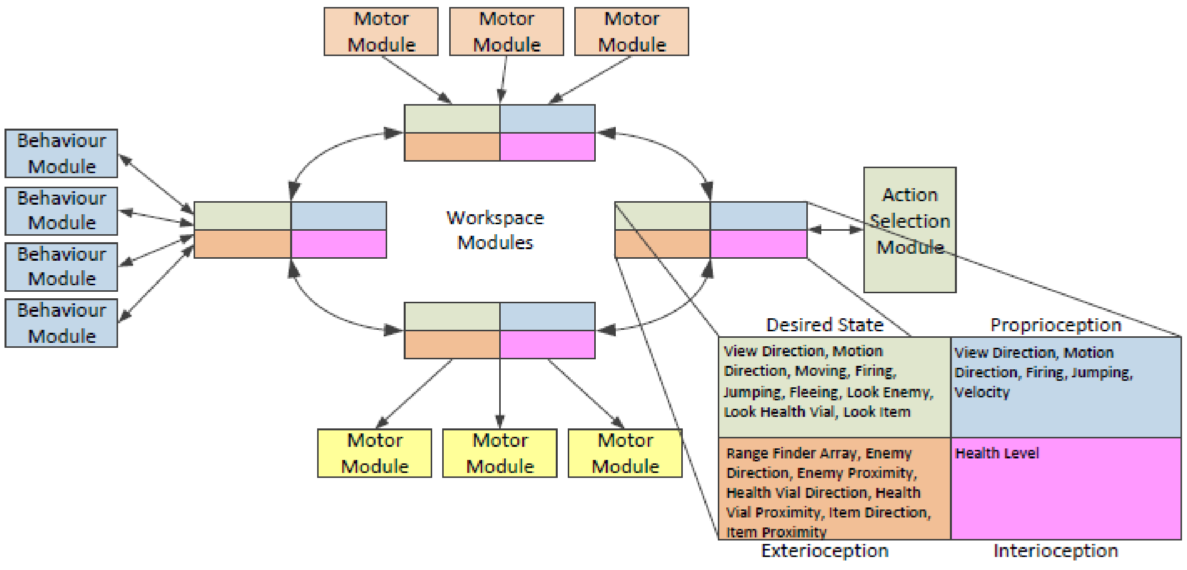

Motor Module
Motor Module
Motor Module
Behaviour Module
Behaviour Module
Behaviour Module
Behaviour Module
Workspace Modules
Action Selection Module
Desired State
Proprioception
View Direction, Motion Direction, Moving, Firing, Jumping, Fleeing, Look Enemy, Look Health Vial, Look Item
View Direction, Motion Direction, Firing, Jumping, Velocity
Range Finder Array, Enemy Direction, Enemy Proximity, Health Vial Direction, Health Vial Proximity, Item Direction, Item Proximity
Health Level
Exterioception
Interioception
Motor Module
Motor Module
Motor Module

Figure 2.16: Global neural workspace architecture

[7] Neurons are electrically excitable specialized cells that constitute the brain. A single neuron can be connected to up to 10,000 other neurons. Neurons have the ability to propagate electrical signals very fast and over very large distances. They achieve this by rapidly changing the difference in voltage between the interior and the exterior parts, i.e. their membrane potential, which results the generation of electrical pulses that are called potential or spikes.

Each datum in the workspace is represented by a population of neurons. Data representing locations in the bot's environment are represented using an egocentric one-dimensional neuron population vector in which each neuron represents a direction in 360º view around the bot, with the directions expressed relative to the current view direction, and the salience indicated by the firing rate. Figure 2.17 shows a possibly encoding for wall proximity for an agent with a wall on its left side. Other workspace data represent scalar values (for example, whether firing is taking place or the current health level) using a rate code, in which the overall activity of a neuron encodes the value.

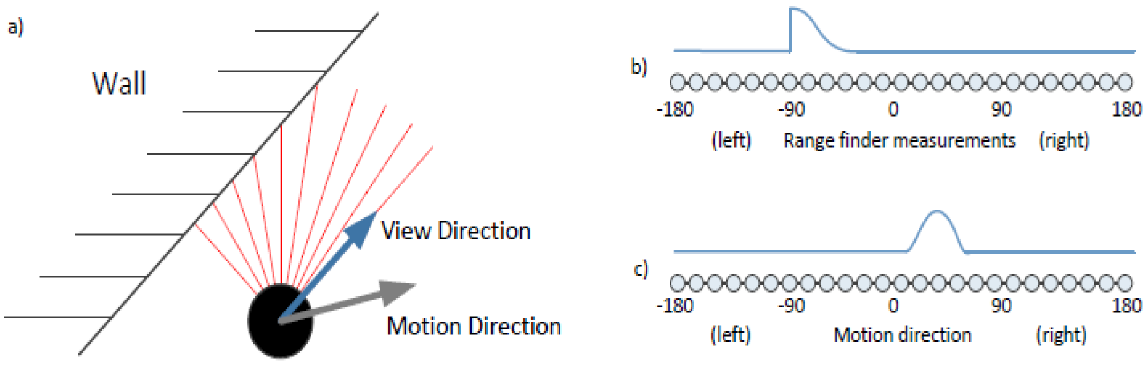


Figure 2.17: a) Representation of the agent and the surrounding environment. b) Firing activity of the input layer of the "range finder" sensory module. c) Firing activity of the output layer of the "motion direction" motor module.

Sensory modules

The bots access information about the game in a somewhat abstract form. The bot has direct access to its location, direction and velocity, as well as access to the location of all currently visible items, players and navigation points. Second, the spatial structure of the immediate environment of the bot is available as a range-finder data. The bot can decide where to cast rays, and it gets back a distance measurement of the point of intersection with a wall or other object, which enables it to build up a map of its environment at an appropriate spatial resolution. Third types of data are the scalars that describe different states of the agent, such as crouching, standing, jumping and shooting. The bot can also access interioceptive data as scalar measures of overall health, adrenaline and armour.

To use this data within this system it is necessary to convert the scalar values containing information about the UT2004 environment into patterns of spikes that are fed into the neural network. To begin with, four pre-processing steps are applied to the input data. First, information about an enemy player or other objects is converted into a direction and distance measurement that is relative to the bot's location. Second, information about pain is processed as the rate of decrease of health plus the number of hit points. Third, the velocity of the bot is calculated, and finally, a series of predefined cast rays are converted into a simulated range-finder sensor. After the previously explained conversions, each piece of data is converted into a number ranged between 0 and 1 which is multiplied by a weight. The result λ is a mean firing rate (spikes/millisecond), which is used to deliver spikes governed by a Poisson distribution with rate λ to the neuron layers representing the corresponding sensory data.

Motor modules

In the present architecture, each motor module is governed by a single desired state and corresponds to a different command or short algorithm. The desired states “motion direction”, “view direction”, “firing” and “jumping” correspond respectively to the commands move, rotate, shoot and jump. The four remaining modules are only used to set the value of a corresponding flag, in order to print the desires of the agent in the graphical interface and indeed in real time. These modules follow a general and very simple structure with a single output layer and connections to a workspace node.

Behaviour modules

It exist 10 different modules, corresponding each of them to a specific behaviour:

1. **Moving module:** it determines whether the agent will move or not. It cannot be used alone because it could only generate a desire to move without providing any direction for the movement.
2. **Navigation module:** it enables the agent to walk alongside the walls and avoid collisions. In particular, it controls the system’s desired direction of motion when the module has access to the workspace. Together with the moving module it forms a coalition of processes that are able to win access and broadcast information to the workspace at the same time.
3. **Exploration module:** it is similar to the navigation module but when the module is active it takes control of the view direction of the agent, rather than controlling the direction of motion.
4. **Firing module:** it enables the agent to defend itself when threatened by the presence of another bot or a human player. If an enemy is close and the agent has a sufficient health level, it will fire against this enemy.
5. **Jumping module:** the system will send the JUMP command in two different cases. First, if the system exhibits a desire to move but it senses that it does not have horizontal velocity (there is an obstacle in front or the agent is stuck for any reason). The second case is that the enemy has been observed and is approaching the agent. Then, it will start jumping in order to avoid any possible firing.
6. **Chasing module:** it controls the desired direction of view and also activates the “look enemy” desire. Then, the desired direction of view is aligned with the direction of this enemy. When this module is cooperating with the moving module, the result is to start chasing the enemy.
7. **Fleeing module:** if a visible enemy is close to the agent and the health level of the agent has decreased, the agent will start moving in the opposite direction of the enemy.
8. **Recuperating module:** if the health level is low and there is a visible health pack, this module will activate and will change the direction of view, making the agent look at this health pack. Cooperating with the moving or flee modules it will result in picking up this health pack. If there are more than one visible health packs, only the closes one will be considered.
9. **Look item module:** it works quite similar to the recuperating module. When a visible item is noticed, the system sets the view direction to be aligned to the direction of the item. If this module is active simultaneously with the moving and navigator modules it will result that the agent will approach and get the item-target.

10. **U-turn module:** this module is aimed to prevent situations where the agent controlled by the system is surrounded by walls and cannot decide which direction is the best to turn. This usually means that the agent is trapped and an obvious approach is to reverse the direction of motion.

3. Project environment and requirements

Before explaining how the ARS bot is created, it is good to know more about the three frameworks that take part in the realization of this thesis: the computer game Unreal Tournament 2004, the Pogamut platform and the Artificial Recognition System. First of all, it is explained why the UT2004 and Pogamut are chosen for this thesis instead of other similar environments. Second, an explanation of the UT2004 and the Pogamut is given. The ARS is also described in order to know how it works the decision unit of the implemented bot. To end this chapter, the requirements on the ARS model in order to create the ARS bot in UT2004 are explained.

3.1 Why using Unreal Tournament 2004 and Pogamut

The reason for choosing the computer game Unreal Tournament 2004 is that it is a 3D complex world with an unpredictable and rapidly changing environment where the ARS can be deeply proved and also because the UT2004 code is partly opened (only the graphical part is not accessible) so it is possible to modify it. Another reason for choosing UT2004 is that the other available open-source games such as Quake 2 or other versions of Unreal Tournament are older than UT2004 and provide a worse graphical experience.

A tool that provides the connection between the Eclipse or Netbeans IDE and the UT2004 was needed. For this purpose two tools were considered: JavaBot[9] and Pogamut [1].It was decided to use the Pogamut platform because the JavaBot is merely a communication protocol whereas the Pogamut provides an IDE and has a large library with predefined methods that are useful for developing agents in UT2004. Moreover, the Pogamut's staff provides continuous support and development of the platform.

3.2 Unreal Tournament 2004

Unreal Tournament 2004 (UT2004 from now on) [8] is a futuristic videogame co-developed by Epic Games and Digital Extremes. It belongs to a category of videogames known as first-person shooters (FPS), where all real time players exist in a 3D virtual world (see Figure 3.1) with simulated physics and a variety of tools that give the players additional abilities. As implied by the term first person, the senses of every player are limited by their location, bearings and occlusion within the virtual world.

In the game there are ten different game modes, from which the most famous and important ones are "Deathmatch", "Capture the flag" and "Domination". This thesis was developed using only the "Deathmatch" mode in which the bot has to kill every other bot or player that appears in the world. However, it would be good in the future to try to adapt the ARS Bot to play in the "Capture the flag" mode, which would imply to the bot different types of behaviour such as “team-working” or deciding who is enemy and who is not, and in that way testing more deeply the ARS behaviour.

Figure 3.1: UT2004 examples [10]

UT2004 is developed in its own scripting language, UnrealScript[8] and almost all its code, except the graphical part, is “open-code”, so it is possible to modify it. Through this integrated scripting environment, developers are provided with a variety of ways to adjust physics parameters, developing new game types and world objects or extending existed ones. Employing the above features gives artificial intelligence (AI) researchers a wealth of robust environments to test their agents. Thus, because of the above reasons and also because UT2004 provides a real-time, continuous, dynamic multi-agent environment, makes it the most suitable game for testing the ARS behaviour in a more complex world. [Gem09]

Moreover, a behavioural control layer called Pogamut extending Gamebots is available for UT2004, which facilitates the connection between the UT2004 videogame and the ARS Decision Unit. Pogamut is going to be explained in detail in the next chapter as well as Gamebots which is, in fact, part of it. [Gem09]

[8] UnrealScript is the language used in the Unreal Engine. UT2004 is written using this language. It is based on Java and C++ and as them it is object-oriented.

3.3 The Pogamut platform

In order to interact between the computer game Unreal Tournament 2004 and the ARS framework, a tool that allows the connection between both sides is needed. It would be interesting that this tool provides an abstract layer so that anybody can focus only in programming the bot. The tool that fulfils these requirements is called Pogamut. In Appendix A there is a little guide for installing the environment with Pogamut and UT2004 in the computer. A description of the Pogamut platform based on [1] and [Gem09] is next given:

Pogamut is a platform that was developed by a team in the faculty of Maths and Physics of the Charles University of Prague. They are continuously working on it and the actual Pogamut's version is 3.3.1. The two most important characteristics of Pogamut are: on the one hand it provides connection between Eclipse[9] or Netbeans[10] and on the other it allows programmers to develop in a simple way different behaviours that a bot can adopt in the UT2004 world. This is possible because, as we said in the previous chapter, the UT2004 is "open-code" and uses its own language (Unrealscript). Pogamut uses Unrealscript but provides a plugin for Eclipse or Netbeans that allows the bots being programmed in Java language. Thanks to this plugin we can use a big library provided by Pogamut which offers plenty of different valid methods to do different things in UT2004. In Appendix B, the useful classes and methods available in the Pogamut's library are shown.

Pogamut's architecture

As seen in Figure 3.2 the Pogamut architecture can be seen as TCP/IP client-server architecture. Gamebots 2004 (GB2004) is acting as the server and the agent platform as the client. GB2004 implements a text protocol that sends messages from UT2004 to the client or commands from the client to the UT2004. The messages can be synchronous, about periodic information of the game, or asynchronous when some kind of event is detected. The commands are used to control the bots that are being executed in the UT2004 environment.

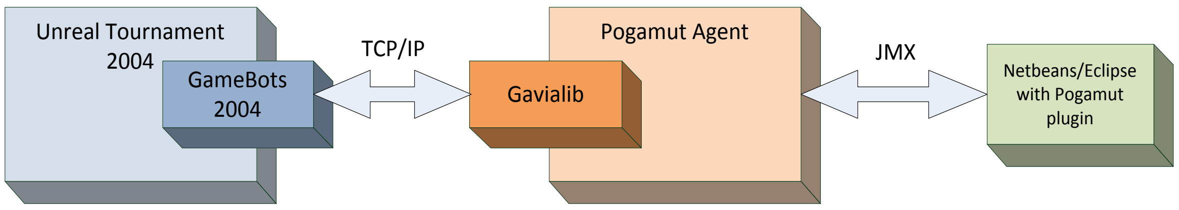


Figure 3.2: Pogamut architecture

Going more into details, the initialization of the connection is done as follows: when the client is connected with GB2004 a "HELLO" message is sent to the client. The client needs to answer with a "READY" message and then GB2004 will send to the client information about the game and the

[9] Eclipse is a multi-language software development environment comprising an integrated development environment (IDE) and an extensible plug-in system. It is mostly written in Java and it can be used to develop applications in Java.

[10] Netbeans is another IDE whose functionality is the same as Eclipse.

map that is being executed such as the Navigation Points or other Items of the map. After this interaction, GB2004 waits until the client sends an "INIT" message. When it receives "INIT", the game generates the bots and enables the communication between GB2004 and the client in order to send the messages and receive the commands.

The Gavialib block is in charge of translating the messages that it receives from the server and also the commands that sends the client throughout the TCP/IP connection. Roughly speaking, Gavialib is the translator between the Agent and UT2004, which is the translator between Java and Unrealscript. The next image shows the Gavialib architecture in more detail:

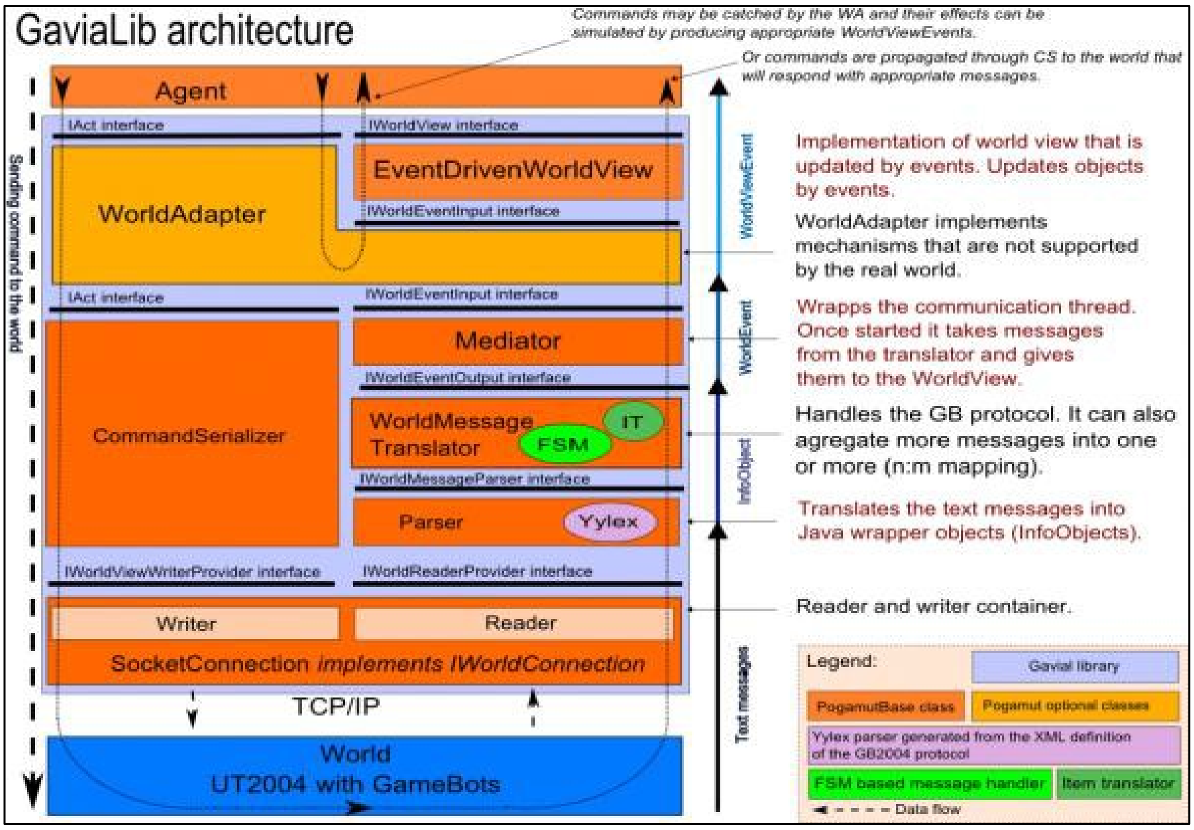

GaviaLib architecture
Commands may be catched by the WA and their effects can be simulated by producing appropriate WorldViewEvents.
Or commands are propagated through CS to the world that will respond with appropriate messages.
Agent
IAct interface
IWorldView interface
EventDrivenWorldView
WorldAdapter
IWorldEventInput interface
IAct interface
IWorldEventInput interface
Mediator
IWorldEventOutput interface
CommandSerializer
WorldMessage Translator
IT
FSM
IWorldMessageParser interface
Parser
Yylex
IWorldViewWriterProvider interface
IWorldReaderProvider interface
Writer
Reader
SocketConnection implements IWorldConnection
TCP/IP
World
UT2004 with GameBots
Sending command to the world
WorldViewEvent
WorldEvent
InfoObject
Text messages
Implementation of world view that is updated by events. Updates objects by events.
WorldAdapter implements mechanisms that are not supported by the real world.
Wrapps the communication thread. Once started it takes messages from the translator and gives them to the WorldView.
Handles the GB protocol. It can also agregate more messages into one or more (n:m mapping).
Translates the text messages into Java wrapper objects (InfoObjects).
Reader and writer container.
Legend:
Gavial library
PogamutBase class
Pogamut optional classes
Yylex parser generated from the XML definition of the GB2004 protocol
FSM based message handler
Item translator
Data flow

Figure 3.3: GaviaLib architecture [1]

Finally, the JMX[11] protocol makes possible to program the bot either in an Eclipse or Netbeans IDE and gives also the possibility of debugging the bot's code while it is being executed in UT2004 and thus being able to watch step by step what the bot is doing in the UT2004 environment.

<u>Pogamut Bot code's structure</u>

Every bot programmed using Pogamut needs to have 6 basic methods in its code. The first four methods are called only once when the bot is being executed for the first time and all of them are for

[11] Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (e. g. printers) and service oriented networks.

settings of the bot. The logic() method is called periodically four times per second while the bot is running and the last method is only called when the bot dies.

These methods are explained in more detail below:

1) prepareBot(UT2004Bot bot)

This method is called before the bot connects to the UT2004 environment. Here is the right place to create the ARS agent and also the interfaces to import/export the inputs/outputs of the ARS Decision Unit.

2) getInitializeCommand()

This method is called immediately after the previous one. It is used for setting initial properties of the agent such as the name, the starting location of the bot, etc.

3) botInitialized(GameInfo gameInfo, ConfigChange currentConfig, InitedMessage init)

This method is called when the server is initialized but the bot is not spawned in the game yet. To be more specific, it is called when the server sends an INITED message to the bot meaning that the previous INIT message from the bot was received correctly and then the connection can be established. When this method is called the three parameters on it get information. GameInfo gets information about the game, such as the type of game or the map, currentConfig receives the actual information about the bot (also the one configured in the previous method) such as the movement speed or the maximum level of health. Other settings of the bot can be configured here as well, for example if they need the connection bot-server to be done. If not it is better to do it in the getInitializeCommand() method.

4) botFirstSpawn(GameInfo gameInfo, ConfigChange config, InitedMessage init, Self self)

This method is called when the bot is spawned in the game for the first time. Here is the last place to do some preparations to the bot. The parameter ‘self’ receives information about the location and state of the bot.

5) logic()

This is the most important method because here is where the intelligence of the bot has to be implemented. This method is called four times per second, so then the decision unit of the bot can change the actions to do depending of the inputs that is receiving from the game.

6) botKilled(BotKilled event)

This method is called only when the bot has died. It is used mostly to reset some variables of the bot for being able to respawn it later. Here it is also possible, for example, to print some messages on the screen when the bot is dead

3.4 ARS

The Artificial Recognition System (ARS) is an AI model developed by the Institute of Computer Technology in the Technical University of Vienna. It is the first psychoanalytic AI model based on the theory of Id-Ego-Superego developed by the Austrian psychoanalyst Sigmund Freud. To create the ARS model a top-down modelling approach was applied. A description in detail of the ARS is given next with a previous chapter of required definitions to understand later how the ARS works.

3.4.1 Required terms definitions

The following definitions (summarized from [Lan10, pp.56-64] and [Zei10, pp.48-53]) are necessary to explain widely the ARS decision unit:

Drive

A drive is the first psychic representative of the demands of the bodily organs and signals a bodily need, as defined in [Fre15b, p.111]. According to [Die09] organic processes are transferred into psychic processes and are represented by a structure called drive. The drive representation consists of two components: the thing presentation and the affect. Whereas the thing presentation of a drive contains quality and therefore the information of its content, the affect holds the information of the amplitude of the tension. The higher the tension within a bodily organ is, the greater the affect that quantifies the corresponding thing presentation. A drive consists of the following drive-contents:

- **Source of the drive:** represents the organ that is generating the bodily tension and is no longer accessible in the psyche.
- **Aim of the drive:** is always to reduce the tension of the bodily organ but can be accomplished in different ways.
- **Object of the drive:** is the necessary object that reduces the tension of the organ
- **Affect:** is the driving force. The quota of affect defines how urgent the satisfaction of the drive is.

The drive object and the aim of the drive can be changed during the psychic processes [Fre72, p.70].

Freud classifies drives according to their drive contents. Drive contents can be classified as constructive or destructive. To nourish, repress, sleep, breath, relax or reproduce are typical constructive drive contents and are representatives of the so called life instinct or libido. To bite, excrete, kill, regress, disintegrate, halt and retreat are examples of destructive drive contents and are representative of the so called death instinct. Each of the drive representatives can be associated with its oral, anal, phallic and genital component of the sexual instinct. Additionally, each drive which is represented within the life instinct has a counterpart within the death instinct forming together a pair of opposites [Fre15a, p.127].

Memory Trace

According to [Fre33, p.75] the memory traces are a psycho-physiological concept for the representation of memories in the mind. This means that each datum that can be remembered, whether it originates from external sensory data or from psychic deliberation, becomes manifest in a memory trace.

When a generated perceptual symbol or group of symbols are compared with existing memory traces there are two possible options. If the incoming symbol resembles a memory trace that already exists, a thing presentation (later described) is generated. On the other hand, if the incoming symbol is not entirely known and therefore does not have a representative memory trace, an adapted new memory trace is created and the corresponding thing presentation is generated.

The memory trace is the system responsible for storing and retrieving previously perceived or processed object information, and associations can be established between related memory traces. The result of an activated memory trace is a thing presentation or word presentation and an attached affect. These three components originated in the memory trace system represent the data structure that is processed by the psychic apparatus. These psychic data structures are connected by associations (later described). The memory traces clearly represent the border between neurological and psychoanalytical findings.

Thing presentation

The thing presentation is the psychic representative of a stimulated memory trace and is generated by activated memory traces. They are psychic representatives of three different types of information that are delivered to the previously discussed memory traces: tension of organs, information regarding the body, and environmental information. Thing Presentations are directly associated with the perceived information produced by the sensed objects through visual, tactile or acoustic impressions. The thing presentations can only be manipulated by primary processes. They normally have a corresponding assigned affect and can be related to other thing presentations through associations within their corresponding memory traces.

Regarding the technical model, the term *thing presentation mesh* (TPM) is introduced and describes connected Thing Presentations. The term TPM is only introduced in the technical model and not in the psychoanalytic theory.

Word presentation

A word presentation is the psychic concept or description for the perceivable object and not a set of sensor-stimuli. It includes the characteristics of sound, scripts or motion among others that can be applied to every object that fits into the defined concept.

Whereas thing presentations are the main content of the primary process, word presentations are the main content of the secondary process. All information represented in a thing presentation that has to be processed in the secondary process needs an attached word presentation.

In addition the thing presentation is the direct link between the sensor data in the form of perceptual symbols and the psyche whereas the word presentation is the interface object between psychic contents and the reasoning process that considers the available information.

A word presentation is assigned to a specific thing presentation (see Figure 3.4).

Affect

The term affect is defined as a quantifiable psychic representative of a drive. It can be said that the quantification is the level of displeasure caused by the corresponding bodily urges. In particular, it describes the drive demand's intensity.

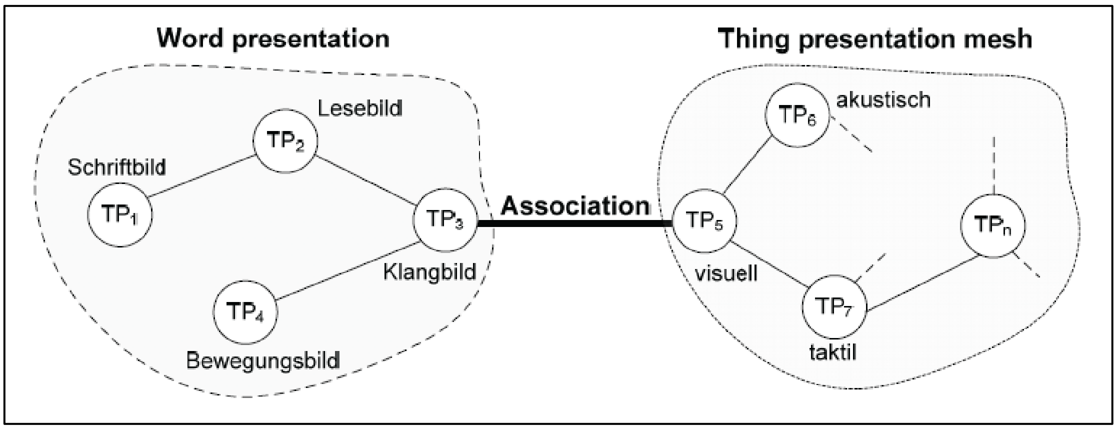


Figure 3.4: Association between word representation and TPs [Fre91, p.121] – original definitions are in German and translated below

TP: *Thing presentation;* Schriftbild: *Visual image for script;* Lesebild: *visual image for print;* Bewegungsbild: *Kinesthetic Image;* Klangbild: *Sound image;* akustisch: *acoustic;* visuell: *visual;* taktil: *tactile*

Wish

When a bodily need is not covered enough, the corresponding psychic representative emerges in the form of a drive as described above, consisting of a thing presentation and an affect. After passing unconscious and filtering mechanisms operating according to the primary process the drive is attached to a corresponding word presentation and converted into information of the secondary process. This preconscious structure that is capable of becoming conscious and is involved in the conscious process of deliberation is called wish within this work.

Since the wish is the representation of a previously experienced, satisfactory situation, it also has to include at least one plan of actions or perceptions that led to the fulfilment of the wish in the past. The utilized data structures are again accessible by the memory traces.

A wish can be seen as a combination of a drive (which is a combination of a thing presentation and an affect) and a word presentation.

Association

Associations are not defined as psychoanalytic data structures. However, for the technical realization they are seen as a component in an information mesh as it is impossible to construct any kind of non-atomic data structure without their use. Associations are formed based on the content of perceived information. The memory content is categorized due to temporal contiguity, similarity and accessibility to secondary processes. This classification is realized by associations that are differentiated by the following characteristics:

- The accordance of attributes between perceived objects
- Temporal concurrence
- Their description by word presentations

In addition, the interaction with the drive object results in the change of homeostatic stimuli. Thus, the resulting affects are connected by their drive source to the drive object. This connection is also formed by an association.

Primary process

The primary process manipulates Thing Presentations and Affects and follows the pleasure principle without taking into account the apparent limitations in the environment and time. The content is aligned to actual drives of the individual instead of formal logic. This means that the priority by which information is processed depends on the drive demand without considering the current situation.

In this mode of information processing, unpleasant excitement is avoided and contents are aligned and organized with respect to the demands of the drives. The current conditions of reality and the possibility of a delay are not considered.

Freud describes the primary process: "the governing rules of logic carry no weight in the unconscious" and therefore "contraries are not kept apart from each other but are treated as thought they were identical" [Fre40, p.48].

Secondary process

In the secondary process, word presentations constitute the majority of the processed content. The secondary process follows the reality principle where constraints from the surroundings in the form of known logical rules can be considered. Environmental, cultural, and situational conditions are considered before demands are discharged. The satisfaction of demands does not aim towards a maximum pleasure but implies given restrictions. The secondary process is able to inhibit the free discharge of drives. Secondary processes cover functionalities like goal decision making, planning and the thinking itself.

The organization of the secondary process includes data processing that allows cross-relations concerning the logic of time, location, and especially learned social rules to become applicable within this type of organization.

3.4.2 Architecture of the ARS model

First of all, a brief description of the second topographical model of Sigmund Freud (Id, Ego, and Superego) is given because it is the base of the ARS model:

The id, ego and super-ego are functions of the mind rather than parts of the brain. The ARS Decision Unit follows this model. A description of each function in more detail is given below:

Id

The id comprises the unorganized part of the personality structure that contains the basic drives. The id works following the "pleasure principle", pursing to avoid pain or displeasure provoked by increments in instinctual tension. The id is unconscious by definition. [Fre33, pp.105-106]

Ego

The Ego acts according to the reality principle; i.e. it seeks to please the id's drive in realistic ways that will benefit it in the long term rather than bringing grief. [Noa84, pp.189-194]

The Ego includes the organized part of the personality structure that includes perceptual, defensive, executive, and intellectual-cognitive functions. Conscious awareness resides in the ego, although not all of the operations of the ego are conscious. The ego separates what is real. It helps us to organize our thoughts and make sense of them and the world around us. [Sno06, pp.105-107]

Super-ego

The Super-ego aims for perfection. It comprises that organized part of the personality structure, mainly but not entirely unconscious, that includes the individual's ego ideals, spiritual goals, and the psychic agency (commonly called "conscience") that criticizes and prohibits his or her drives, fantasies, feelings, and actions. [Mey07]

In Figure 3.5 appears an image with the relation between the two topographical models of Freud even though the relation between both is not clearly defined. Whereas the id is unconscious by definition, the ego and the super-ego comprise both conscious and unconscious parts. In Figure 3.5 there is a clarifying example of the second topographical model.

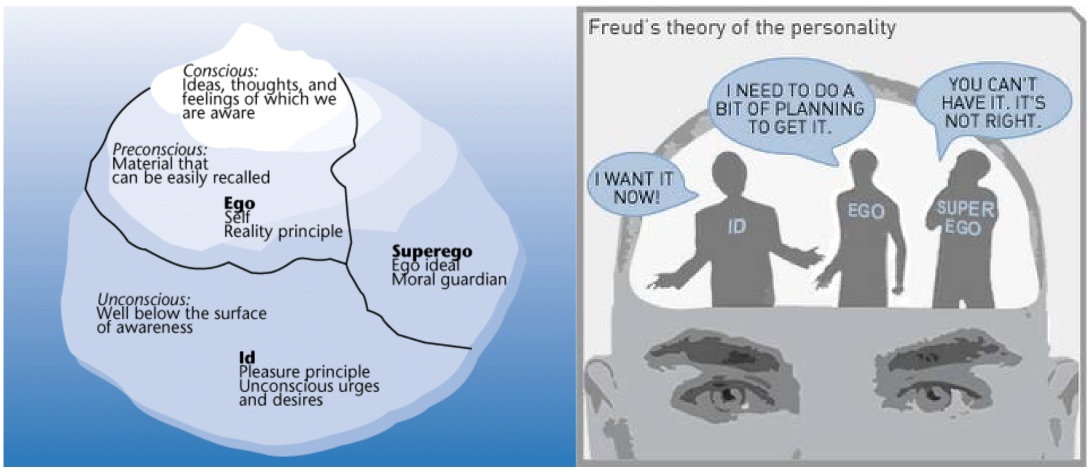

Conscious:
Ideas, thoughts, and feelings of which we are aware
Preconscious:
Material that can be easily recalled
Ego
Self
Reality principle
Superego
Ego ideal
Moral guardian
Unconscious:
Well below the surface of awareness
Id
Pleasure principle
Unconscious urges and desires
Freud's theory of the personality
I NEED TO DO A BIT OF PLANNING TO GET IT.
YOU CAN'T HAVE IT. IT'S NOT RIGHT.
I WANT IT NOW!
ID
EGO
SUPER EGO

Figure 3.5: Id-Ego-Superego structure and example [7]

The actual model of the ARS (see Figure 3.10) was reached after four steps of applying top-down design. The first view on the model was composed only of two layers and was built using shallow definitions for Id, Ego and Super-ego (see Figure 3.6).

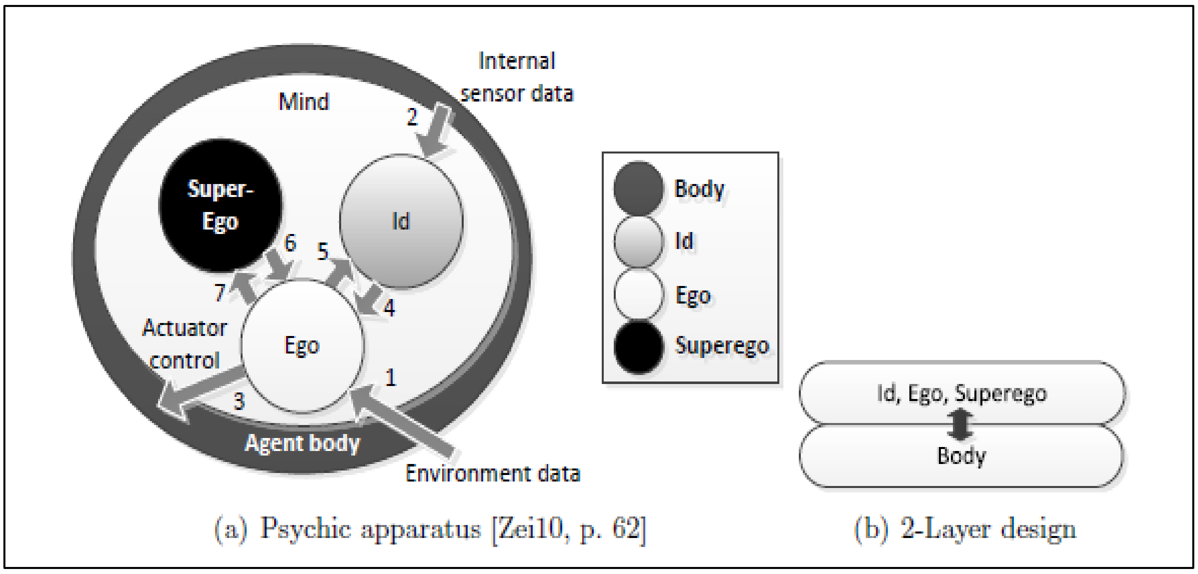


Figure 3.6: ARS model. Top-down level 1 [Deu11, p.72]

However, this model was not sufficient because the separation among Id, Ego and Super-ego and the interaction between them is not very clear in order to implement it in a model. For this reason a second model with three layers was built. In the second model the differentiation was done between primary and secondary processes. Primary processes use only thing presentations and quotas of affects while secondary processes use word presentations. The advantage is that reasoning can be done using word presentations; therefore the reasoning process belongs only to the secondary process. The Id and the Ego have to be split because they cannot be assigned to one of the processing types exclusively.

The second ARS model architecture is shown in Figure 3.7. A brief description of the second top-down level model is given. The body is split into three modules. Modules A and B provide sensor information while module M processes actuator commands. Module *Homeostatic state* (A)[12] checks if any of the internal body parameters (e.g. blood sugar level) is out of balance and sends the difference to the next module, *Drive generator* (C). There the values are converted into drive structures (see drive definition in Section 3.4.1).

[12] Homeostasis: The ability of the body or a cell to seek and maintain a condition of equilibrium or stability within its internal environment when dealing with external changes [12].

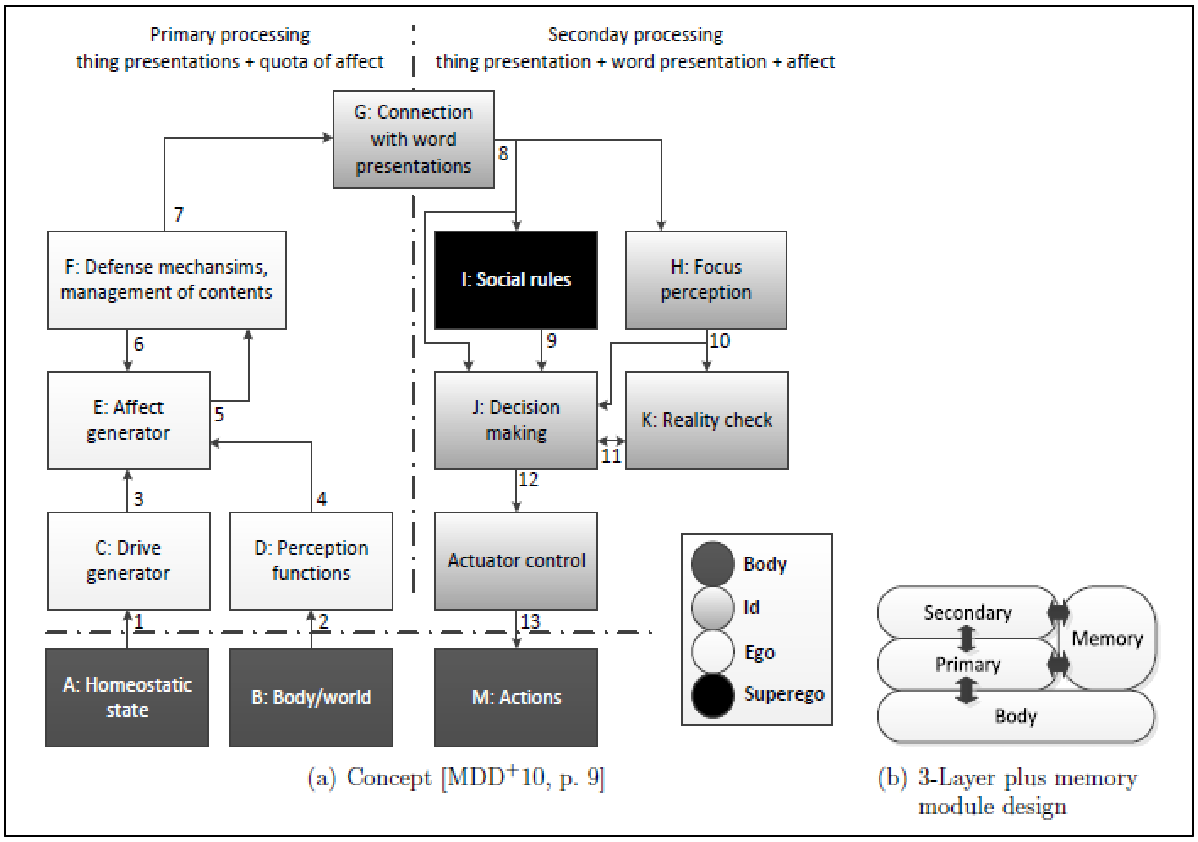
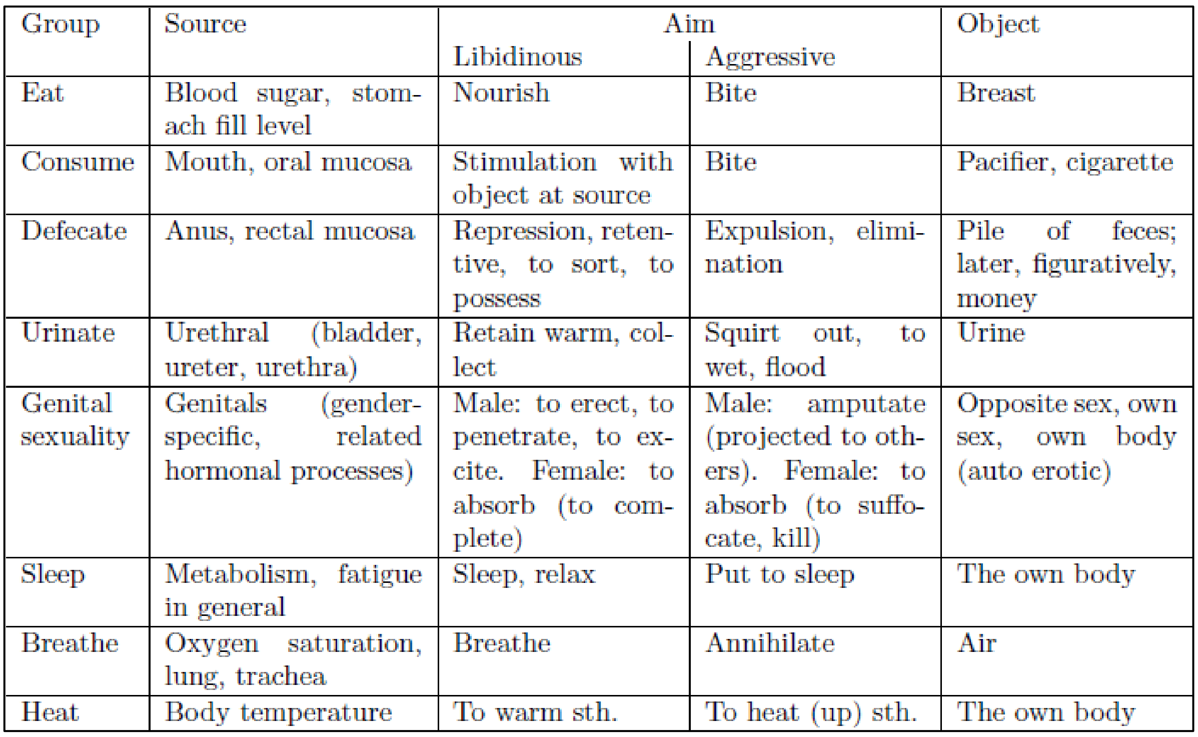


Figure 3.7: ARS model. Top-down level 2 [Deu11, p.78]

A bodily need is always represented by a pair of opposites: a constructive and a destructive drive [Fre15a, p.127]. An example is given in [LKZD10, p.717]: in case of hunger the pair of opposite drives are nourish and bite. Nourish is the constructive drive with the libidinous aim of eating to satisfy hunger. Bite is the destructive drive with an aggressive aim that is to destruct food. The first one is necessary in order to want something to eat and the second one enables the process of eating.

The list of drives in ARS can be seen in Figure 3.8.

Body and world information are provided by sensors to the module *Body/World* (B). This data is forwarded to module E where the perceived data is compared with already known patterns and completes missing information from the memory. The module *Affect Generator* (E) receives drive structures from module C and thing presentations from module D. Both structures are merged there into a mental image which represents the current situation as perceived by the system. The resulting mental image is sent to the module *Defense mechanisms, Management of contents* (F). The incoming data is analysed there by the Super-ego rules. If the Super-ego rules are totally satisfied or partially satisfied but need some changes the data is forwarded to the next module, if not, they are repressed and sent back to the previous modules.



Group	Source	Aim		Object
		Libidinous	Aggressive	
Eat	Blood sugar, stomach fill level	Nourish	Bite	Breast
Consume	Mouth, oral mucosa	Stimulation with object at source	Bite	Pacifier, cigarette
Defecate	Anus, rectal mucosa	Repression, retentive, to sort, to possess	Expulsion, elimination	Pile of feces; later, figuratively, money
Urinate	Urethral (bladder, ureter, urethra)	Retain warm, collect	Squirt out, to wet, flood	Urine
Genital sexuality	Genitals (gender-specific, related hormonal processes)	Male: to erect, to penetrate, to excite. Female: to absorb (to complete)	Male: amputate (projected to others). Female: to absorb (to suffocate, kill)	Opposite sex, own sex, own body (auto erotic)
Sleep	Metabolism, fatigue in general	Sleep, relax	Put to sleep	The own body
Breathe	Oxygen saturation, lung, trachea	Breathe	Annihilate	Air
Heat	Body temperature	To warm sth.	To heat (up) sth.	The own body

Figure 3.8: List of drives in ARS [Deu11, p.81]

In the module *Connection with word presentations* (G) the thing presentations and quota of affects are linked with word presentations. Now the decision making can be done using word presentations. The output of module G is distributed to modules H, I and J. The module *Focus perception* (H) generates a prioritized list of the secondary information whose order is determined by homeostatic demands and environmental information. The outgoing list is forwarded to modules J and K. The module *Social rules* (I) compares the current situation with the existing Super-ego rules, selects the situations that fit the rules and sends them to module J where the decision making will be done. The module *Decision making* (J) receives the whole list of current word presentations from module G, the prioritized list from module H, the fitting social rules from module I and an assessment of the fulfilment of goals from module K. The result is an action plan in form of word presentations. The module *Reality check* (K) checks the fulfilment of goals. To do it the prioritized list of demands and the list of current goals are received and evaluated. Finally, module *Actuator control* (L) disassembles the action plan provided by the decision making module into action commands that the last module *Actions* (M) can process.

After the latter described model questions appeared like: where are the affects generated or how repressed contents are attached. To answer these questions a third iteration of the top-down modelling process was done and later followed by a fourth iteration. In the new model a new layer called Neuro-symbolic is created (see Figure 3.9). The reason is that the body is now split into two layers. The body layer contains modules like sensors and actuators and the new layer does the conversion of the sensors data into neuro-symbols.

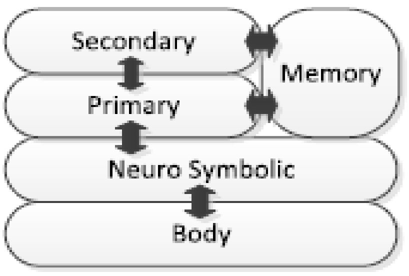


Figure 3.9: 4-layers plus memory module design [Deu11, p.88]

For this thesis is not important to show the last model (level 4) because only the input/output modules and interfaces are used. For this reason the description of the ARS is going to be done following the ARS model level 3 (see Figure 3.10), which is simpler than the last one and there are only few changes between them. Therefore, it would be easier to understand how the ARS work.

In order to show how the ARS model works, a loop of the model is going to be explained. To follow the explanation it is good to take a look at Figure 2.15:

The ARS collects information from the environment and from the body in module *Body/World/Physics.* This information can be either about the libido source, the internal parameters of the body, the environment sensors or the body sensors. Each different type of information is processed following a different path: the information about the libido is forwarded to the module *Sexual Drives*, the internal parameters of the body are sent to the module *Self-preservation Drives* and the environment and body sensors information are sent respectively to the modules *Environment and Body*.

In modules *Sexual Drives* and *Libido Accumulation*, the flow of libido received is added to the existing one in the system and the sexual drives are created. In module *Self-preservation Drives* the internal parameters of the body are converted to memory traces that represent the corresponding drives. At this point, a memory trace contains the source, aim and object of a drive. Both, the sexual drives and the self-preservation drives are forwarded to modules *Affects and Memory Traces* where the quota of affect of both drives is attached to the memory traces that contain the drive contents. In the module *Emersion*, the repressed contents from previous loops of the model are changed to be more likely to pass the defense mechanisms and are added to the new contents of the actual loop.

In modules *Environment* and *Body*, the environment and body sensors information are converted into thing presentations. The thing presentations previously created are linked with previously experienced and stored memory traces in module *Memory Traces*. Therefore, more information is added to the current perception. For example: if only a part of an object is visible but the object is known because the agent has seen it before, the not visible parts are added from the memory. The module *Emersion,* as well as in the other *Emersion* module, changes the repressed contents from previous loops of the model to be more likely to pass the defense mechanisms and are added to the new contents of the actual loop. In module *Libido discharge*, the incoming perception information is compared with memory to determine if they are good for libido discharge and for pleasure gain. If they are good, the value of the libido buffer is reduced and the pleasure gain is sent to module

Affects. In module *Affects*, the quota of affects for perception is calculated by looking up all the associated displeasure and pleasure values retrieved from memory.

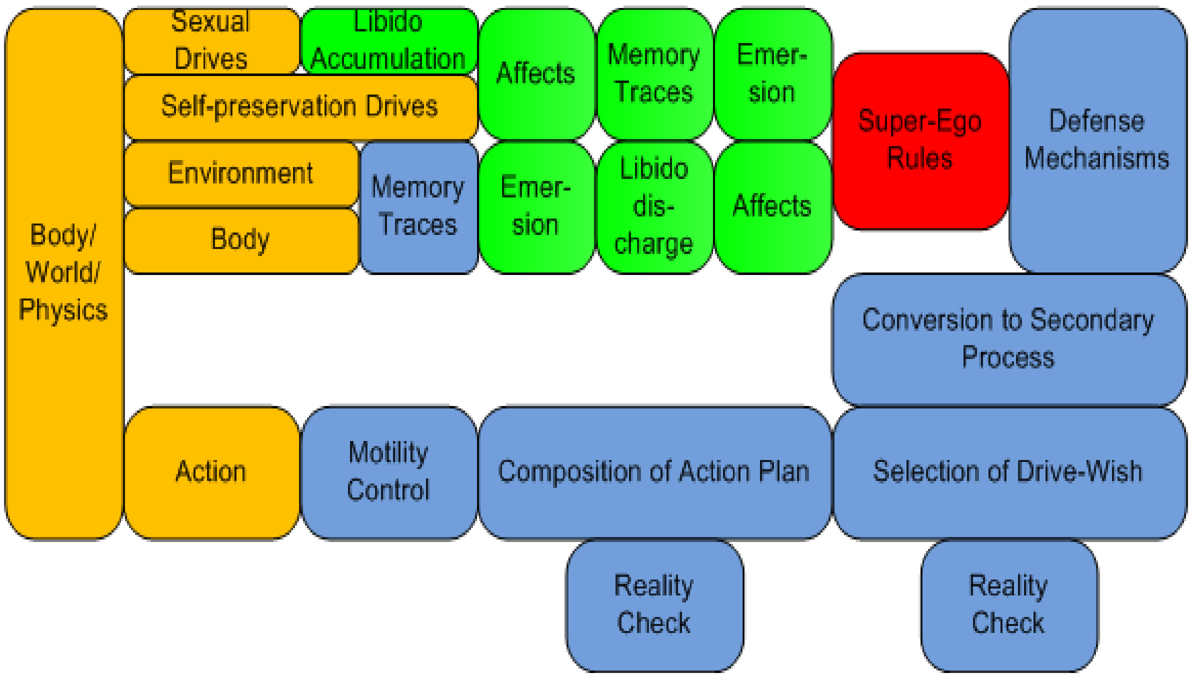


Figure 3.10: ARS functional model level 3 [3]

Afterwards, all the drive-contents are forwarded to module *Super-ego Rules* where the social rules are applied neutralising or reducing the energy of some of the drives. The module *Defense Mechanisms* decides which drive representations are allowed to become conscious and if they are not allowed it decides which defense mechanism is applied. Examples of defense mechanisms are repression, intellectualization and sublimation.

At this point all the information is still presented as thing presentations. In module *Conversion to Secondary Process*, the thing presentations and quota of affects are associated to the most fitting word presentations found in memory. The drive contents are converted into drive wishes.

The module *Selection of Drive-Wish* is responsible for selecting the drive wish that is considered to be the most important putting it in the first position of the list of drives. The available possibilities for drive-wishes satisfaction and its requirements are given in module *Reality Check*. The module *Composition of Action Plan* generates the needed actions in order to satisfy the previously selected drive wish. The module *Motility Control* evaluates how the received action plan can be best realized and provides the resulting action commands which are forwarded to the module *Action*. There the action is sent to the output in the correct format in order to execute it.

3.4.3 ARS vs. BDI Architecture

The ARS decision unit is going to be compared with a widely used architecture such as BDI. As explained in Chapter 2.1.2 the BDI architecture was developed for reasoning systems being able to work properly in dynamic environments (that change continuously). We are going to make a comparison between the two models in terms of the three main blocks of the BDI architecture: Believe, Desire and Intention [Lan08, pp.2-3].

Believe

In the BDI architecture, this layer contains the world representation. Within the ARS architecture, the (internal and external) world of the agent is perceived via sensors through the perception module. Predefined templates are compared to the actual situation and scenarios are recognized. Globally valid rules are stored in the semantic memory, past experienced situations and actions are stored in the episodic memory, following the concept of E. Tulving [Tul83].

Desire

It is the BDI's layer that defines the behaviour of the decision unit and its purpose. In ARS architecture, drives and desires vastly influence the behaviour of the system and its purpose: Staying within the optimum levels of the internal states. But also long term goals have been considered partly within the desire module and partly in the SUPER-EGO structure, which holds social rules and goals.

Intention

In BDI, different processing paths can cause different actions at the same time that have to be prioritized. In the ARS architecture, the action unit is supplied with actions from three different paths:

1. Reactive action: is comparable to human reflex actions.
2. Routine unit: processes periodical action sequences that do not need extra processing.
3. Reflective unit: provides action based on reflected strategies, plans and 'acting as if'-simulations.

The ARS decision unit has been implemented to a simulated autonomous agent where internal values, like drives, emotions and desires and the behaviour between agents are evaluated.

3.5 Requirements on the ARS model

At the beginning of this thesis, the ARS was not ready to be implemented in the UT2004 because the ARS was first created for an environment which is very different from the UT2004. For this reason, some requirements were established in order to adapt the ARS framework to the new environment. The changes to be done to satisfy the requirements are basically the implementation of new actions and new objects in the ARS. The list of the proposed requirements is this:

1. Different types of health packs, weapons, ammo and armor have to be defined as a new object in the ARS system. Also the enemies and walls need to be defined as an entity in the ARS. When these elements are created, a drive or emotion has to be associated to them in order to make the bot picking up an object (health packs, weapons, ammo or armor), avoid colliding against the walls, flee from the enemy or shoot the enemy.
2. The ammo and armor levels need to be defined as new parameters of the ARS agent. The health level is already defined but it is necessary to adapt it. This is done in the body interface (explained later in this chapter).
3. New actions need to be implemented to make the bot able to survive itself along the maps of the UT2004. For this purpose the following actions are proposed to be implemented in ARS:
    a. SEARCH action: this action is already implemented in the ARS but it needs a better search algorithm because for the moment the search algorithm is very simple consisting only of going forward and turn right twenty degrees repeatedly. When seeing the bot doing the actual search action it seems to be very silly. Therefore, the search action needs to be improved.
    b. UNREAL_MOVE_TO action: this action needs to be implemented in the ARS with the two following parameters: type of item and location (in polar coordinates). With this action, the bot will be able to go to a specific location and pick up the desired item in this location.
    c. FLEE action: when an enemy is near, the health level of the bot has to be checked and if it is low the bot has to flee from the enemy.
    d. SHOOT action: like the previous action, the health level of the bot has to be checked when the bot detects the presence of an enemy. If, in this case, the health level is high the bot has to shoot the enemy.
    e. CHANGE WEAPON action: when the bot has run out of ammo of the actual weapon, it needs to change the weapon to another available weapon.

At the end of this thesis just a few of the requirements were implemented: the health pack is defined as a new object, the enemies are not defined as a new object but are associated to an old entity called BODO that has the same functionality as an enemy, the UNREAL_MOVE_TO action is implemented and the FLEE action too but the last is not checking the health level and in any case of detection of an enemy the bot flees from the enemy.

These rest of the requirements for the ARS need to be implemented to be able to achieve a complete UT2004 bot and therefore being able to participate in the Botprize competition.

4. Model and Concepts

In order to make a successful connection between UT2004 and the ARS engine, which is the main goal of this thesis, we get rid of the Pogamut tool (see Chapter 3.3) which provides the connection between UT2004 and the Eclipse IDE. Once this connection is done a new interface is needed to connect the Pogamut to the ARS engine while the data from UT2004/Pogamut is adapted to an understandable format by the ARS framework. A general description of the model is first done in Section 4.1 and later a detailed description of every part of the interface will be provided in Section 4.2.

4.1 Preview of the model

First of all, it is important to explain which information has to be exchanged from UT2004 to ARS or vice versa. From the UT2004 we have to extract information on perceived objects by the bot and the bot’s internal parameters. From the ARS side only the output actions have to be sent to the UT2004. If we have a look at the ARS model (Figure 4.1) we can see that there are five inputs interfaces (I0.1, I0.2, I0.3, I0.4 and I0.5) and one output interface (I0.6). The first two (I0.1 and I0.2) are not going to be used because they are for the sexual drives which have no role in UT2004. Interfaces I0.3 and I0.5 are both for body parameters but here interface I0.3 is going to be used because through it we are going to insert internal parameters of the current state of the body such as health, ammo or armor and not body parameters based on perception that should enter through I0.5. Thus, interface I0.3 is the one that is going to be used to put the bot’s internal parameters into the ARS system. The perceived objects are considered as environment perception so they are going to be inserted into the ARS through interface I0.4. Finally, the output uses the only available output interface, I0.6, throughout the actions are forwarded.

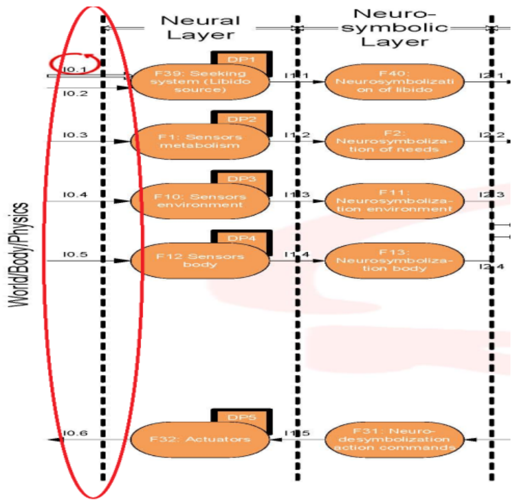

Neural Layer
Neuro-symbolic Layer
World/Body/Physics
I0.1
I0.2
I0.3
I0.4
I0.5
I0.6
DP1
DP2
DP3
DP4
DP5
F39: Seeking system (Libido source)
F1: Sensors metabolism
F10: Sensors environment
F12 Sensors body
F32: Actuators
I1.1
I1.2
I1.3
I1.4
I1.5
F40: Neurosymbolization of libido
F2: Neurosymbolization of needs
F11: Neurosymbolization environment
F13: Neurosymbolization body
F31: Neuro-desymbolization action commands
I2.1
I2.2
I2.3
I2.4

Figure 4.1: ARS model focused on the interfaces

An interface for each ARS input and another for the ARS output were created. The design process was as follows:

The output interface, called itfOutputActionProcessor, was first created. The reason to start with this interface was because it seemed the least complex and therefore we can gain experience about how to handle the interfaces for later work with the other interfaces which are more deeply connected with the simulator. Another reason for starting with the output interface was that after doing the output interface it was possible to fake the actions and see if some kind of action was done in the UT2004 computer game. Afterwards, the interface for the environment perception, itfEnvironmentProcessor, was done. The last interface done was the one for the bot's internal parameter which is called itfBodyProcessor. A detailed description of the three interfaces is done in the next subchapter.

To only need to import one entity when the ARS is extracted to other environments we decided to include the three previous defined interfaces into another one that is called itfInterfacesManager. This last interface is the one that creates the ARS entity in the UT2004 computer game.

The general structure detailed above is shown in Figure 4.2:

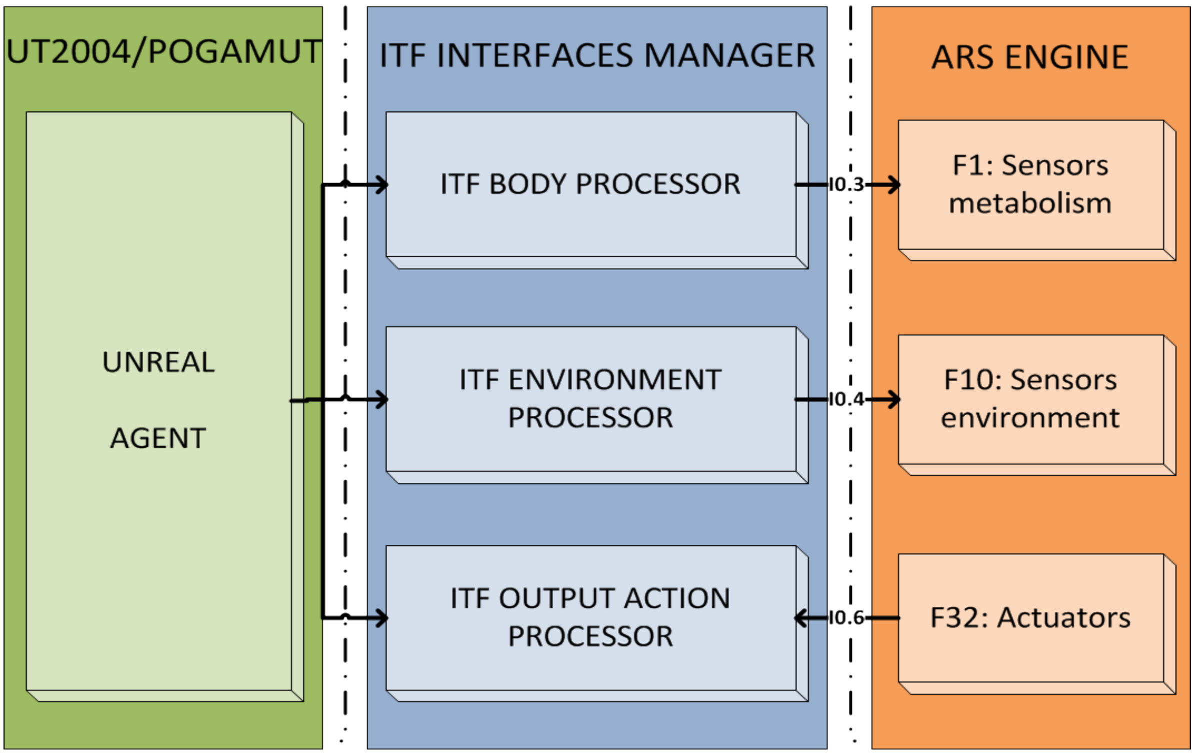


Figure 4.2: UT2004-ARS interface architecture

4.2 Description of the UT2004-ARS Interface

After a preview of the model, where a general view of the model is introduced, the structure of the UT2004-ARS interface is going to be described. For each interface described in the previous section, a detailed account will be given, commenting every decision which has been taken and mentioning all the methods that are implemented in each interface and their functionality.

4.2.1 Body interface "itfBodyProcessor"

The first thing done here was to check all the possible parameters that can be extracted from UT2004 through the Pogamut and decide which one of them we are going to insert into the ARS system. In (Appendix B) there are some lists including the Pogamut's methods. Just looking at the name of the methods we can see the big amount of parameters that are possible to extract from UT2004. Afterwards, we decided to use only three of them which are: health, ammo and armor. We considered that these three are the most important parameters in the game to be inserted in ARS. At the end we will see that the three of them are inserted in ARS but only the health one is useful because the ARS system is not prepared yet to take decisions concerning ammo or armor (see Requirements in Section 3.4).

To insert these parameters a new body is created in the ARS which is represented by the class clsUnrealBody that extends the class clsComplexBody. In the class clsUnrealBody these new methods are created:

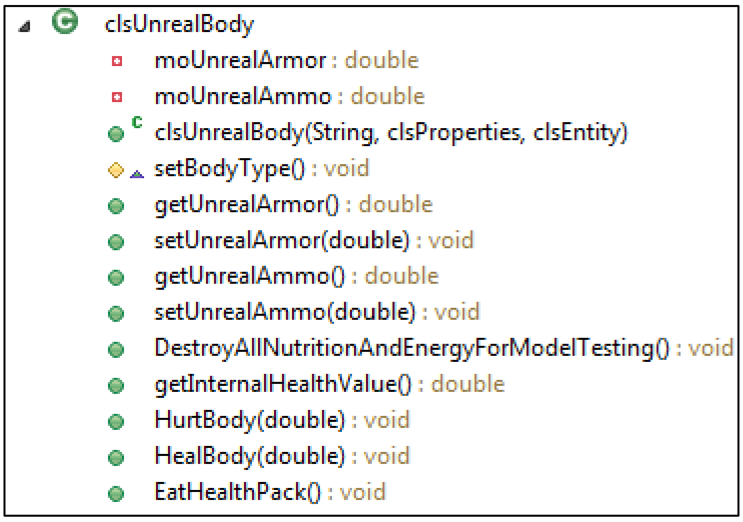

```
clsUnrealBody
    moUnrealArmor : double
    moUnrealAmmo : double
    clsUnrealBody(String, clsProperties, clsEntity)
    setBodyType() : void
    getUnrealArmor() : double
    setUnrealArmor(double) : void
    getUnrealAmmo() : double
    setUnrealAmmo(double) : void
    DestroyAllNutritionAndEnergyForModelTesting() : void
    getInternalHealthValue() : double
    HurtBody(double) : void
    HealBody(double) : void
    EatHealthPack() : void
```

Figure 4.3: clsUnrealBody attributes and methods

After knowing what we receive from the UT2004/Pogamut side and what available methods we have in the ARS side an explanation of how the itfBodyProcessor works is going to be given. The itfBodyProcessor proceeds following these steps:

1. The interface receives the current values of health, ammo and armor from the Pogamut.

2. The interface also receives the internal health value of the ARS body. The idea is to equal both health values:
    a. If the health of the UT2004 bot is higher than the health of the ARS body we call the HealBody(double amount) method from the clsUnrealBody passing as a parameter the difference between both health values.
    b. If the health of the UT2004 bot is lower than the health of the ARS body we call the HurtBody(double amount) method from the clsUnrealBody passing as a parameter the difference between both health values.
3. If the current health level is below a certain predefined threshold and a health pack exists in the visible area, we activate the nourish drive[13] in the ARS by using the method from the clsUnrealBody called DestroyAllNutritionAndEnergyforModelTesting(). This method literally empties the stomach of the ARS's body causing it to be hungry and therefore the bot wants to pick up a visible health pack.
4. The ammo and armor are set in the ARS by using the methods setUnrealAmmo() and setUnrealArmor() but as said before these two parameters are not used yet in the ARS model. So at the moment this step is useless but is considered to be useful in the future.
5. When the UT2004 bot eats a health pack the ARS's stomach has to be refilled in order to put the nourish drive off. This is done by using the method EatHealthPack() from the clsUnrealBody.

In Figure 4.4 there is a diagram that shows in a simple way all the above explained.

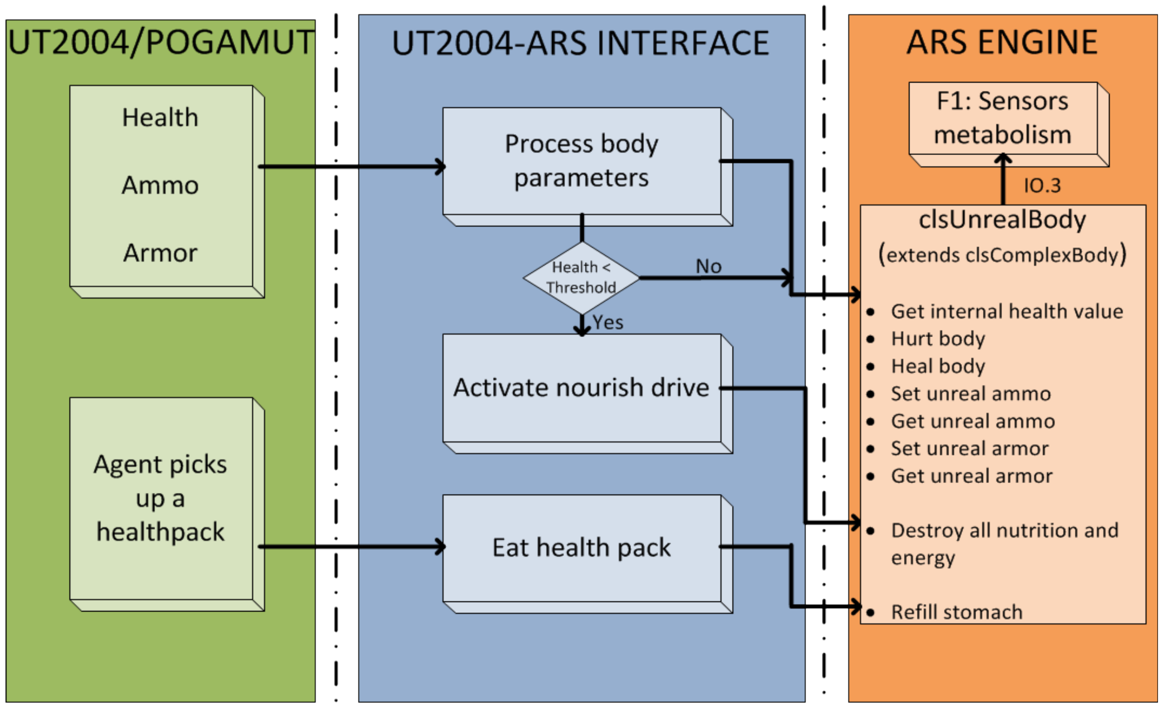


Figure 4.4: Body Processor interface

[13] The nourish drive is associated to the health packs in UT2004. Therefore, a health pack is the drive object of the drive nourish. This is done to reuse the current drives of ARS because nourish have some kind of similarity with healing.

4.2.2 Environment interface "itfEnvironmentProcessor"

This interface is done to process all the perceivable things by the bot and adapt them to the ARS model in order to let the ARS to take its own decisions like for example flee when an enemy is seen. We are going to describe this interface helped with the diagram in Figure 4.5. As we can see in the figure the interface receives items, enemies and walls from the UT2004/Pogamut. The processing of each of them is described one by one below.

Items

The interface receives a Java Collection with all the kind of objects that are currently visible. The most important items are different kind of weapons, ammo for each kind of weapon and health packs. As seen in Figure 4.5, each item contains parameters like: UnrealID, NavPointID, if the item is visible or not, in case of ammo or health the amount of it that increases, its location, the type of item and if it was dropped by other agents or not. From all of them only the UnrealID, Location and Type (printed in red at Figure 4.5) are later used in the interface. The rest of parameters are not considered useful, at least for the moment at this point in the ARS implementation. It is important to note here that for the moment the received collection with all kind of items is filtered keeping only the health packs because the rest of items are not yet implemented in the ARS model. Therefore, in case of inserting them, it would not be possible to do anything with them.

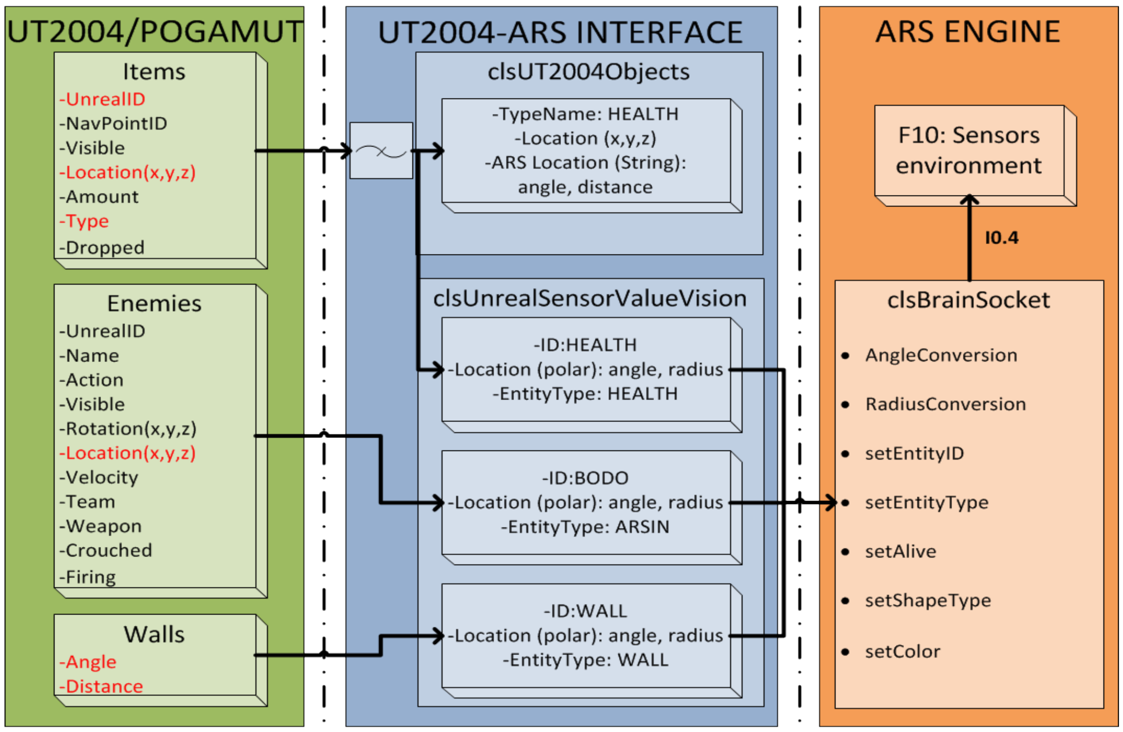


Figure 4.5: Environment Processor interface

Then, at this point the interface receives only the health packs visible by the bot. As we said before the health pack (like all the items) contains information on the UnrealID, the location in Cartesian coordinates and the type of item. The adaption of these three parameters into the ARS format is done in two steps. The first step is done in the interface using the class clsUnrealSensorValueVision made for this purpose and the second step is done using the class clsBrainSocket in the ARS engine.

To explain the first step of the adaption the clsUnrealSensorValueVision attributes and methods are shown in Figure 4.6:

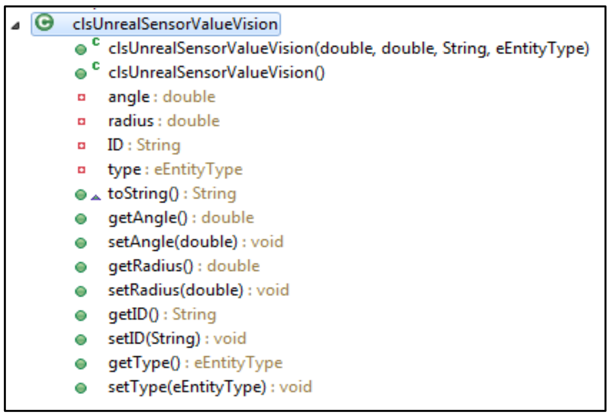

clsUnrealSensorValueVision
clsUnrealSensorValueVision(double, double, String, eEntityType)
clsUnrealSensorValueVision()
angle : double
radius : double
ID : String
type : eEntityType
toString() : String
getAngle() : double
setAngle(double) : void
getRadius() : double
setRadius(double) : void
getID() : String
setID(String) : void
getType() : eEntityType
setType(eEntityType) : void

Figure 4.6: clsUnrealSensorValueVision attributes and methods

In the environment interface the information of the items is parsed into the ARS format using the clsUnrealSensorValueVision. In case of the health pack, which is the only one that is used at the moment, the UnrealID is transformed into HEALTH using the method setID(String) , the location is converted from Cartesian coordinates to polar coordinates (using the bot's location as reference) using a method created for this purpose. These coordinates are later on set in clsUnrealSensorValueVision using the methods setAngle(double) and setRadius(double). To end the first adaption the type of item is converted into one of the ARS types from the enum class eEntityType using the method setType(eEntityType). In case of the health pack the eEntityType is just HEALTH.

Afterwards, the parameters are put into a java vector and sent from the interface to the clsBrainSocket in the ARS engine. In the clsBrainSocket the second step of adaption has to be done. The parameters ID and Type are just set into the clsBrainSocket without any modification. However, the location has to be converted into the ARS vision format and the object has to be defined with a unique shape and colour. For each type of object the following attributes have to be defined: ID, Type, if it is alive or not (meaning if it moves or not), shape and colour. All these attributes are set in the clsBrainSocket. To understand the location conversion a brief description of the ARS vision format is done:

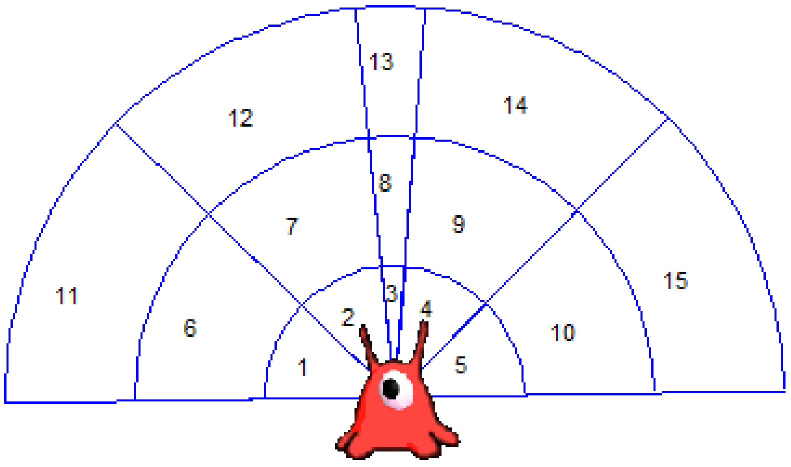


Figure 4.7: ARS Vision

Looking at Figure 4.7 it is easier to explain how the ARS vision works. The ARS vision is composed of fifteen sectors combining the angle and distance divisions. The angle division is made into five sectors which are: LEFT, MIDDLE_LEFT, CENTER, MIDDLE_RIGHT and RIGHT. The distance division is made into three sectors: NEAR, MEDIUM and FAR. Combining both divisions the ARS vision results into fifteen sectors as mentioned before. Following the numeration of Figure 4.7, the fifteen sectors are: 1) LEFT:NEAR, 2) MIDDLE_LEFT:NEAR, 3) CENTER:NEAR, 4) MIDDLE_RIGHT:NEAR, 5) RIGHT:NEAR, 6) LEFT:MEDIUM, 7) MIDDLE_LEFT:MEDIUM, 8) CENTER: MEDIUM, 9)MIDDLE_RIGHT:MEDIUM, 10) RIGHT:MEDIUM, 11) LEFT:FAR, 12) MIDDLE_LEFT:FAR, 13) CENTER:FAR, 14) MIDDLE_RIGHT:FAR, 15) RIGHT:FAR.

Therefore, the location of the perceived items has to be converted from Polar coordinates (angle and radius) into two Strings (e.g.: LEFT and MEDIUM) depending on which sector of the ARS vision is located by using the methods AngleConversion() and RadiusConversion() of the clsBrainSocket.

Now the conversion of all the items from the UT2004 to the ARS is done. However, by converting the UT2004 location to the ARS vision a lot of accuracy is lost in the location of the items. For this reason the items are saved into a new class called clsUT2004Objects. In this class the type of the item, the location in Cartesian coordinates and the location in ARS vision are stored. In this way when at the output the ARS decides to pick up an item whose location is for example LEFT, MEDIUM it will be compared with the saved objects list which will return the exact location of the item in Cartesian coordinates in order to be able to go to its location and pick up the correct item.

Enemies

In the same way as with the items, a Java Collection is received in the interface containing all the current visible enemies by the bot. As shown in Figure 4.5 a lot of parameters are provided for each of the enemies in the collection but only the location in Cartesian coordinates is used by the interface.

Like with the items a class clsUnrealSensorValueVision is created to do the first step of adaptation of the enemy's parameters. The location is converted exactly as explained above for the items but for the ID and Type there is a detail to be explained. All the enemies will have ID = BODO and Type = ARSIN due to the following: the BODO was a previously implemented entity in the ARS engine whose functionality was also to act as an enemy. Because of this, instead of defining a new entity for the enemy in the ARS engine, it was decided to reuse the entity BODO. Of course, the entity BODO had previously defined its own shape and colour and it is necessary to use it without modifying the properties of the entity.

Walls

Until now, the items and enemies were obtained from the UT2004 by methods especially created for it available in the Pogamut library. However, there are no created methods in the Pogamut's library to obtain the walls. For this reason a decision was made to use an available tool in Pogamut which is called ray casting. The ray casting allows the definition of a set of rays that depart from the bot's body with a predefined angle and length. A Boolean variable is defined for each of the rays and it is set to true in case that the ray detects a wall. In order to detect the walls and set its location into the ARS vision a ray was defined for every sector of the ARS vision (15 sectors = 15 rays). In other words, for each angle three rays were defined (one for each distance). The angle and distance of the rays was decided to be positioned in the centre of each of the sectors of the ARS vision. Moreover, we considered appropriate to define two more rays: one ninety degrees left and another ninety degrees right both with length NEAR. In Figure 4.8 the resulting "ray casting bot" is shown. The rays that are red painted in the Figure mean that are detecting a wall and the green ones are not detecting anything.

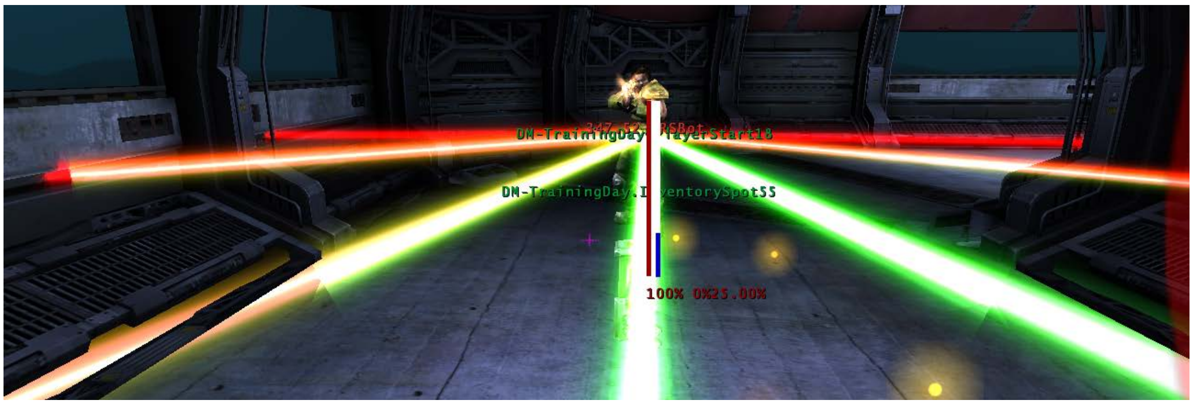


Figure 4.8: "Raycasting Bot" [10]

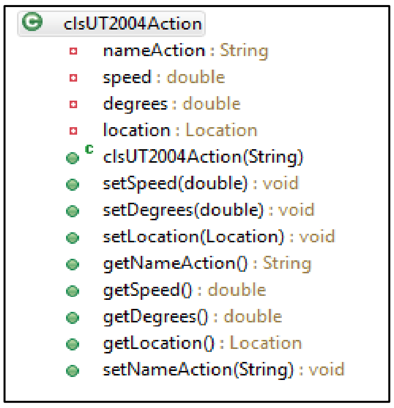
Using the ray casting sensors as explained above, the walls and their location in Polar coordinates are obtained. Then, they are sent to the interface, which this time does not need to transform the coordinates because they are already in Polar form. In the interface the whole collection of walls are added together into the same Java Vector where the items and enemies are added and everything is sent to the ARS engine. Note that the walls are inserted in the ARS engine but it does not do anything with them because it is not prepared for it yet. A more detailed explanation of the walls is given in [Ort12, pp.52-53].

4.2.3 Output interface “itfOutputActionProcessor”

The ARS framework is providing its output with actions to do. What exactly arrives to the output of the ARS is a Java ArrayList of the type clsCommandAction. Each position of the ArrayList contains an XML String with a simple action and the possible parameters of the action. The format of the received XML String is like this:

<ActionName>ActionName@Parameter1:Parameter2:...:ParameterN</ActionName>.

To separate the name of the action and its parameters a new class is created called clsUT2004Action. In Figure 4.9 the attributes and methods of this class can be seen:



clsUT2004Action
nameAction : String
speed : double
degrees : double
location : Location
clsUT2004Action(String)
setSpeed(double) : void
setDegrees(double) : void
setLocation(Location) : void
getNameAction() : String
getSpeed() : double
getDegrees() : double
getLocation() : Location
setNameAction(String) : void

Figure 4.9: clsUT2004Action attributes and methods

The name of the action is set for every received action with the method setNameAction(String) and the attributes speed, degrees or location are also set with the specific set method for each parameter depending on the type of action received. At the end of this thesis only three final actions are possible to receive in the output of the ARS because they are the only ones implemented in ARS that are appropriate for the UT2004 game. These three final actions are: SEARCH, UNREAL_MOVE_TO and FLEE. The search action is in fact a sequence action that provides the interface with the simple actions MOVE_FORWARD (with a speed parameter), TURN_LEFT or TURN_RIGHT (with the degrees to turn as a parameter).

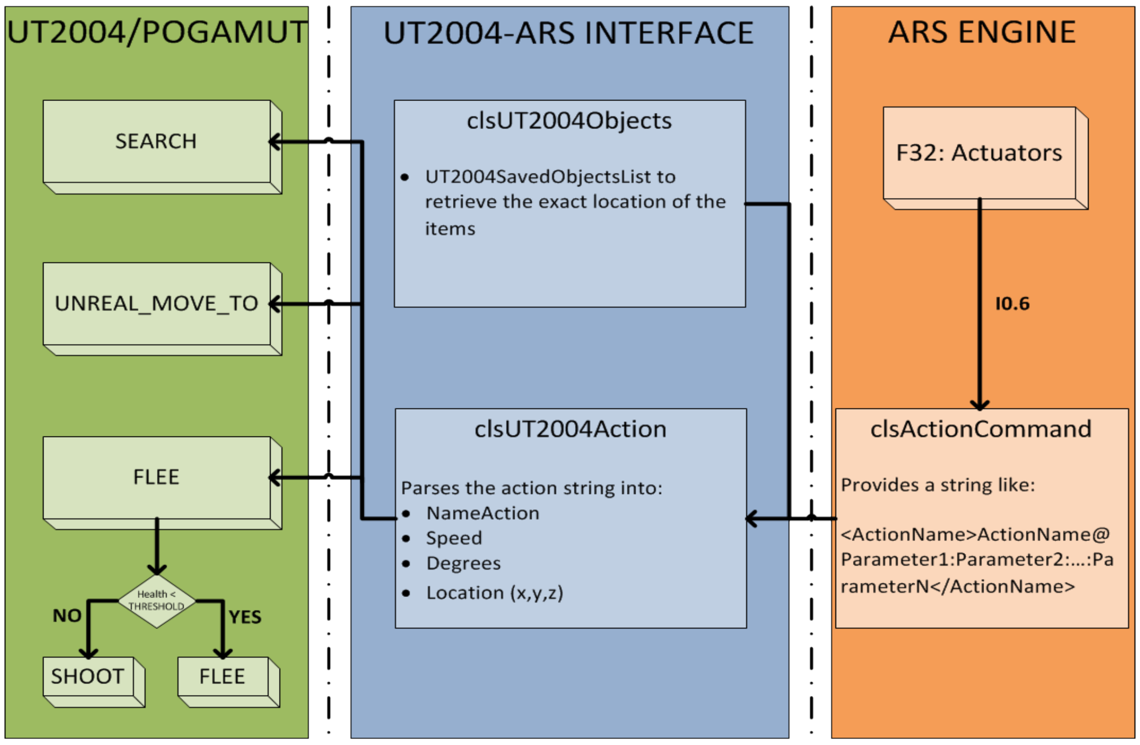


Figure 4.10: Output Action Processor interface

In case of receiving the UNREAL_MOVE_TO action, the type of item to pick up and its location are given as a parameter. The location of the item is given in the ARS format (e.g.: LEFT, FAR) but it is of very low resolution. For this reason, as it was explained before in the environment interface section, the exact location of the items was stored in a class clsUT2004Objects. When receiving the UNREAL_MOVE_TO action, it is the moment to compare the location received in the action String with the ones that are saved to retrieve the exact location where the bot has to move to pick up the item.

In case the FLEE action is received, the health level is checked and the bot decides whether flee from the enemy or shoot it. If the bot's health level is over a certain threshold the bot would be encouraged to shoot the enemy. If not, it will flee away from the enemy. These two sub-cases were done because there was not a shoot action implemented in ARS yet and we considered that it would be good to show the bot shooting because it is the main action in the UT2004 game. More information about the output environment is given in [Ort12, pp.54-55].

5. Implementation

Whereas in the previous chapter the implementation of the ARS in the computer game UT2004 was explained in a conceptual way, here it is going to be described in a more technical way. Some UML diagrams are done to help with the explanation of the code and to see the structure of the realized classes and how they are connected between them. To start with the description of the implementation, a general view of the whole structure is going to be done. Afterwards, a description of each of the three more specific interfaces (body, environment and output) is going to be done.

5.1 Description of the ARSBot

As shown in Figure 5.1, the ARSBot class is the one that contains the main() and therefore it is the executable class. The ARSBot is part of the Pogamut and it is connected to the UT2004. The general structure of a Pogamut class and also a general description of the Pogamut platform are given in Chapter 3.3. The ARSBot class contains an instance of the class clsInterfacesManager which implements the interface itfInterfacesManager (see Figure 5.1). The class clsInterfacesManager provides the interface between Pogamut and ARS. Therefore, it is responsible for connecting the Pogamut and the ARS and to adapt the transferred information from one side to the other in order to be understandable in both sides. In order to insert the ARS agent into the UT2004, the clsInterfacesManager has an instance of the class clsARSIN to be able to create new ARS agents. It also has instances of the classes clsBodyProcessor, clsEnvironmentProcessor and clsOutputActionProcessor which implement the interfaces itfBodyProcessor, itfEnvironmentProcessor and itfOutputActionProcessor respectively. This is done in this way because, as explained in Chapter 4, the ARS model has various interfaces and each different kind of information has to go through a specific interface. For this reason, the interface itfBodyProcessor is responsible for inserting the information about the internal parameters of the bot such as health, the interface itfEnvironmentProcessor inserts information about the perception of the bot and the interface itfOutputActionProcessor provides the output actions from the ARS system.

The ARSBot works as follows: the ARS agent is created in the prepareBot() method using the createARSAgent() method from clsInterfacesManager. Inside the createARSAgent() method, after creating the agent, the agent's entity is sent to all the interfaces (Body, Environment, Output) in order to have the same entity in all of them. This is done using the private method setARSAgentToInterfaces() from clsInterfacesManager. Later a new name is defined for it in the method getInitializeCommand(). The bot is already initialized in the method botInitialized() and it is in this method where the ray casting sensors are created using the method createRayCasting(). As explained in Chapter 4.2.2, a ray casting sensor is created for each of the fifteen sectors of the ARS's vision area. Once the bot is initialized and with the ray casting sensors, the logic() method of the ARSBot class is called four times per second in order to update the information. In each of the loops of the logic() method the following actions are done:

1. Receive the visible objects, enemies and walls of the bot. The objects and enemies are detected by using specific methods from the Pogamut's library whereas the walls are detected using ray casting. For detecting the walls the methods checkRayCastingSensors() and wallObjectsDetection() are created. In the method checkRayCastingSensors(), for each ray casting sensor, a Boolean variable is defined that is set to true if there is an obstacle and set to false if not. The fifteen ray casting sensors are checked and the Boolean variables are updated to the corresponding value. The method wallObjectsDetection() uses the previously described Boolean variables and converts the information into a string that contains the location of the detected walls in the ARS format (i.e. RIGHT:NEAR).
2. The objects, enemies and walls information is forwarded and adapted to the ARS by using the method setUT2004VisibleObjects() of the class clsInterfacesManager. This method is just calling the method setObjectsARS() of the environment interface that is the one responsible for inserting the perception into the ARS system. A description of how this is done is given later in Chapter 5.3.
3. The health, ammo and armor levels are received. These values are easily obtained thanks to the already implemented Pogamut library that contains specific methods for it. The methods used in this case are getHealth(), getCurrentAmmo() and getArmor() from the Pogamut's class Info (see Appendix B).
4. The health, ammo and armor levels are sent to the ARS using the setBodyParameters() method of the class clsInterfacesManager. This method is just calling the method with the same name in the body interface that is the responsible for inserting this kind of parameters in the ARS system. A better description of how this method works is given in Chapter 5.2.
5. The processing() method of clsInterfacesManager is called in order to execute one loop of the ARS model. This method calls the methods sensing(), updateEntityInternals(), updateInternalState() and processing() of clsARSIN in this order. Therefore, the external sensors of the agent are checked, the internal parameters of the ARS agent are updated with the new sensed information and a loop to the model of the ARS is done.
6. The ARS actions are received using the getActions() method of clsInterfacesManager which is calling the getActions() method of the output interface that is the one responsible for extracting the actions to do from the ARS decision unit. More details about how the getActions() method works are given in Chapter 5.4.
7. The requested action is executed using the method doAction() of the ARSBot. This method consists of a structure "if-else if" that receives the action to do and depending on what action has to do, it calls the corresponding method from the Pogamut's library (see Appendix B) in order to execute the desired action. In some cases, it is executed the private method moveRandomlyUsingRayCast() of the ARSBot class that, as the name says, it consists of making the bot move randomly avoiding the walls using the ray casting sensors and a combination of simple actions from the Pogamut's library (see Appendix B) such as moving forward with a certain speed or turning to one side a specific amount of degrees.

8. The actions are deleted from the stack because if not the actions are accumulated. This is done with the clearStack() method of clsInterfacesManager. This method is calling the clearStack() method from clsARSIN which is the one that is really deleting the actions from the stack.

These eight steps are followed at every loop until the bot is dead when the botKilled() method is called and a message telling that the bot is dead is sent.

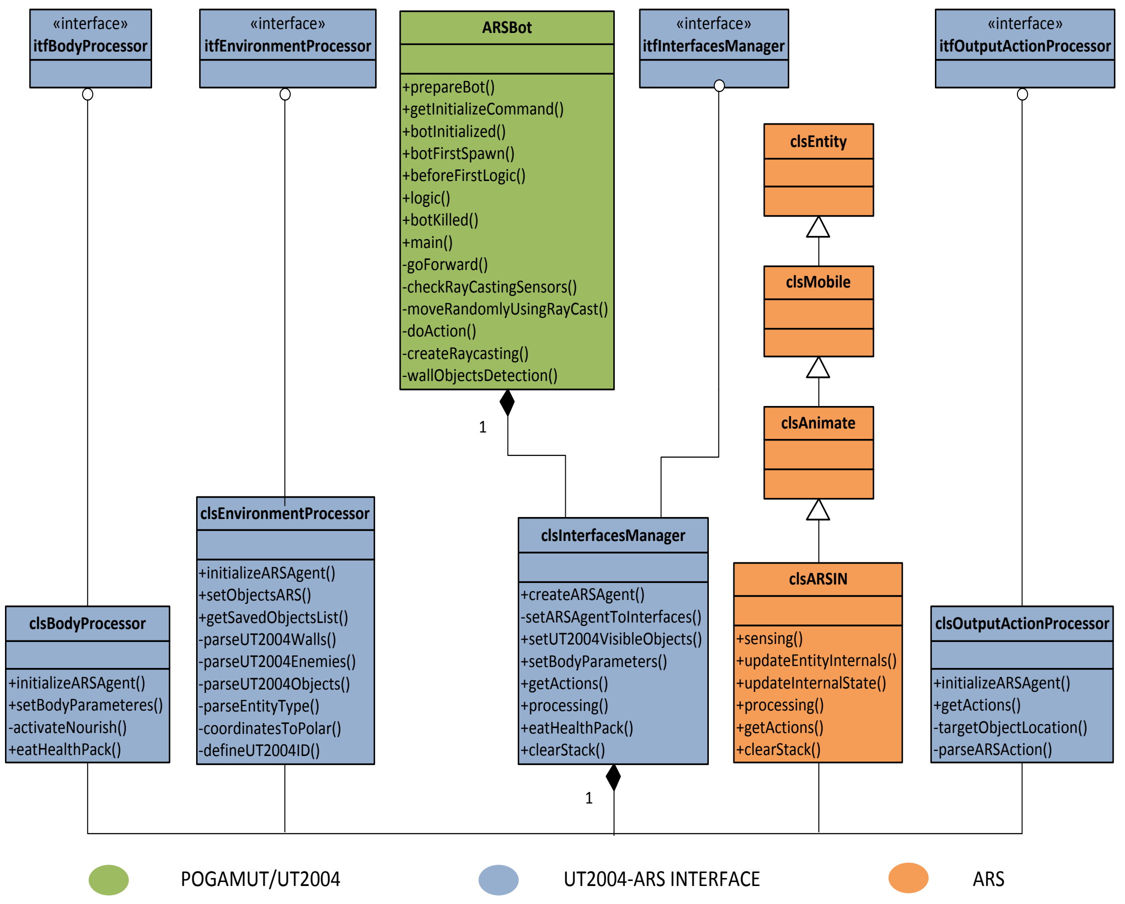


Figure 5.1: UT2004-ARS interface UML diagram

5.2 Description of the Body Interface

The body interface is responsible for doing the fourth step of the list of actions in the logic() method explained in Chapter 5.1. Thus, the goal of this interface is to insert the body parameters of the bot. A new body is created in the ARS for the aim of this thesis. The new body is called clsUnrealBody:

clsUnrealBody

During the realization of the thesis, there was the necessity of inserting parameters such as ammo and armor in the ARS body but these parameters were not defined in any of the already implemented bodies of the ARS. Also, it was necessary to insert the health level from the UT2004. There was a health parameter already defined in ARS, but it was totally different to the one that now is necessary (which is just a number which is not related to any other parameter). The already implemented health was internally calculated based on the levels of vitamins, minerals and this kind of stuff.

There was the possibility of modifying the existing body or to create a new one. Finally, it was decided to create a new one (clsUnrealBody) because it was considered to be simpler. Thus, the new body clsUnrealBody contains all the necessary attributes and methods in order to insert the ammo, armor and health level into the ARS system. The methods of the Unreal body are here explained:

- **getUnrealArmor():** it returns the agent's armor level.
- **setUnrealArmor():** it sets the agent's armor level into the ARS body.
- **getUnrealAmmo():** it returns the agent's ammo level of the current weapon.
- **setUnrealAmmo():** it sets the agent's ammo level into the ARS body.
- **DestroyAllNutritionAndEnergyForModelTesting():** it destroys all the nutrition stuff of the ARS body. This is done to make the ARS become hungry and activate the nourish drive. This will make the agent to pick up a health pack in the game.
- **getInternalHealthValue():** it return the internal health level, which is internally calculated based on the levels of vitamins, minerals and others.
- **HurtBody():** it reduces the internal health level of the ARS an specific amount.
- **HealBody():** it increases the internal health level of the ARS an specific amount.
- **EatHealthPack():** it increases specifically the amount of carbohydrate, fat, minerals, proteins, vitamins and water of the body in order to deactivate the nourish drive after eating a health pack.

As visible in Figure 5.2, the class clsUnrealBody extends from the class clsComplexBody, which at the same time extends from clsBaseBody. The class clsEntity, which is "the father" of the clsARSIN has an instance of the clsBaseBody. For this reason, the UnrealBody entity can be obtained from the ARSIN instance by using the method getBody() of clsBaseBody and doing a cast to the clsUnrealBody. This is done in the initializeARSAgent() method of the class clsBodyProcessor.

After knowing a little bit of the class clsUnrealBody, the process of how the body parameters are inserted is explained:

The health, ammo and armor levels are received as parameters in the method setBodyParameters() of clsBodyProcessor. There, the methods setUnrealArmor() and setUnrealAmmo() from clsUnrealBody, which is instantiated in clsBodyProcessor, are called and the armor and ammo levels respectively are sent directly to the Unreal body. To send the health level is a little bit more complex. The ARS agent has an internal health level which is received using the method getInternalHealthValue(). The health level from the UT2004 is also received. Therefore, it is necessary to equal both values. This is done using the methods HurtBody() or HealBody() from clsUnrealBody depending on which health value is higher than the other. Later, it is checked if the health level is below a certain threshold and in case it is true, the Nourish drive of the ARS system is activated by using the method DestroyAllNutritionAndEnergyForModelTesting(). This is done to make the agent become hungry and go to find a health pack. If a health pack is eaten, the method eatHealthPack() is called and the nourish drive is deactivated.

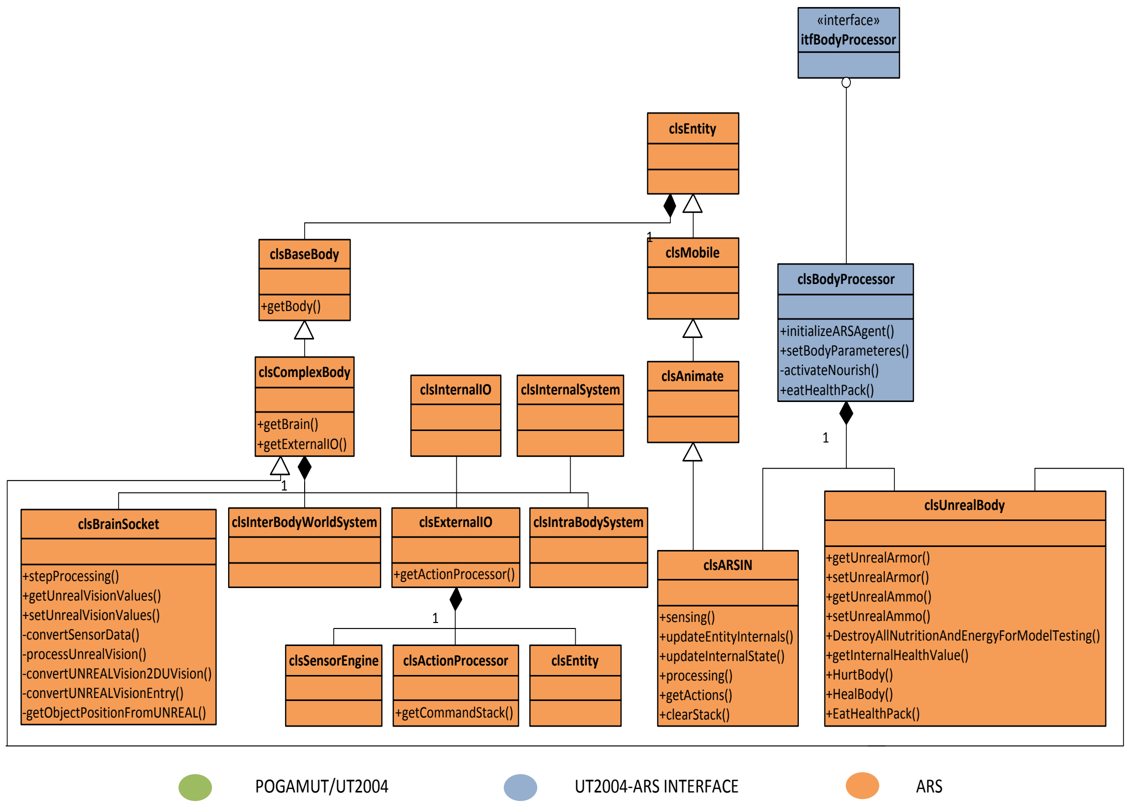


Figure 5.2: Body interface UML diagram

5.3 Description of the Environment Interface

The environment interface is responsible for doing the second step of the list of actions in the logic() method explained in Chapter 5.1. The goal of this interface is to insert and adapt to the ARS the perception of the bot, in other words, what the bot sees. Before explaining how this is done is good to know a little bit about two new defined classes: clsUnrealSensorValueVision, clsUT2004Objects and one modified class: clsBrainSocket.

clsUnrealSensorValueVision

This class is useful for storing the perceived objects by the bot in the environment. In order to do it, the attributes angle, radius, id and type are defined. Angle and radius are necessary to define the location of the object in polar coordinates. With the attributes id and type, the name and type of object is defined respectively. To work with this attributes the following methods are defined in this class:

- **getAngle():** it returns the angle from the polar coordinates of the location of the perceived object.
- **setAngle():** it sets the angle from the polar coordinates of the location of the perceived object.

- **getRadius():** it returns the radius from the polar coordinates of the location of the perceived object.
- **setRadius():** it sets the radius from the polar coordinates of the location of the perceived object.
- **getID():** it returns the name of the perceived object.
- **setID():** it sets a name to the perceived object.
- **getType():** it returns the type of object which is perceived.
- **setType():** it sets the type of object which is perceived.

clsUT2004Objects

As explained in Chapter 4.2.2, when the location of the perceived objects is converted into the ARS format (i.e. LEFT:FAR) and sent to the ARS system a lot of accuracy is lost. For this reason, the exact location of the items is saved in the clsUT2004Objects. Thus, in case of wanting to pick up one of the objects, the exact location can be retrieved from this class.

This class has the following attributes: typeName, location, radius and phi. The typeName contains the type of object, the location contains the location of the object in Cartesian coordinates and the radius and phi contain the two components of the location in the ARS format as a string. Then, when the output action of the ARS is to pick up a health pack which is situated in the vision sector LEFT:FAR the previous attribute would be: typeName = HEALTH_PACK, radius = FAR and phi = LEFT. Afterwards, it has to be checked which of the saved Cartesian locations corresponds to LEFT:FAR and retrieve it in order to make the bot go to this exact location. The following methods are available in this class:

- **setTypeName():** it sets the type of object.
- **getTypeName():** it returns the type of object.
- **setLocation():** it stores the location of the object in Cartesian coordinates.
- **getLocation():** it returns the location of the object in Cartesian coordinates.
- **setRadius():** it sets the radius location of the object into the ARS format (NEAR, MEDIUM or FAR).
- **getRadius():** it returns the radius component of the location of the object in the ARS format.
- **setAngle():** it sets the angle component of the location of the object in the ARS format (LEFT, MIDDLE_LEFT, CENTER, MIDDLE_RIGHT, RIGHT).
- **getPhi():** it returns the angle component of the location of the object in the ARS format.

clsBrainSocket

The perception of the bot (items, enemies and walls) is inserted into the ARS system through the class clsBrainSocket. The useful methods of this class for the implementation of the ARS in the UT2004 are next described:

- **stepProcessing():** it does a complete loop to the model processing the inserted information and giving an output depending on the inputs at the end of the loop.

- **setUnrealVisionValues():** it is responsible of inserting all the received information as parameter into the ARS framework.
- **getUnrealVisionValues():** it returns all the information that is in the brain socket.
- **convertSensorData():** this method adapts the inputs data into understandable data for the ARS system. In case of the data from the UT2004, it calls the method processUnrealVision() that will adapt the UT2004 information to the ARS.
- **processUnrealVision():** it transforms the whole perception into the ARS format by calling the convertUNREALVision2DUVision() method that at the same time calls other methods in order to do the transformation step by step.
- **convertUNREALVision2DUVision():** it creates a vision entry by calling the method convertUNREALVisionEntry().
- **convertUNREALVisionEntry():** for each kind of object a unique shape and colour is defined in this method. Also, an ID and a Boolean indicating if the item is alive or not are set. The location of the perceived objects is also set here by calling the method getObjectPositionFromUNREAL().
- **getObjectPositionFromUNREAL():** this method is responsible for adapting the location of the objects to the ARS format (i.e. RIGHT:NEAR).

To insert the information from the UT2004 into the ARS, the objects, enemies and walls are received as parameters in the method setObjectsARS(). In this method the information is adapted to the ARS by using the private methods parseUT2004Walls, parseUT2004Enemies and parseUT2004Objects of clsEnvironmentProcessor. The three of them adapt the received information and set it to the class clsUnrealSensorValueVision previously explained. The information of the exact location of the objects is also set to the clsUT2004Objects in Cartesian coordinates in order to save the exact location and do not lose accuracy (see Chapter 4.2.2). Once everything is parsed, it is put into a Java vector of the type clsUnrealSensorValueVision. Afterwards, this vector with all the data has to be sent to the brain socket of the ARS agent. For doing it, first it is necessary to get the body and cast it to clsComplexBody. This is done with the method getBody() of clsBaseBody. Once this is done, it is necessary to get the brain. For it, the method getBrain() of clsComplexBody is called. For finally inserting the information in the ARS, the method setUnrealVisionValues() of clsBrainSocket is called with the vector that contains the information as parameter. Afterwards, the information is processed in order to be understandable by the ARS with the methods convertSensorData(), processUnrealVision(), convertUNREALVision2DUVision(), convertUNREALVisionEntry() and getObjectPositionFromUNREAL() from the class clsBrainSocket that were previously explained. To understand better this explanation, it is better to take a look to Figure 5.3 and see how the structure of the environment interface is.

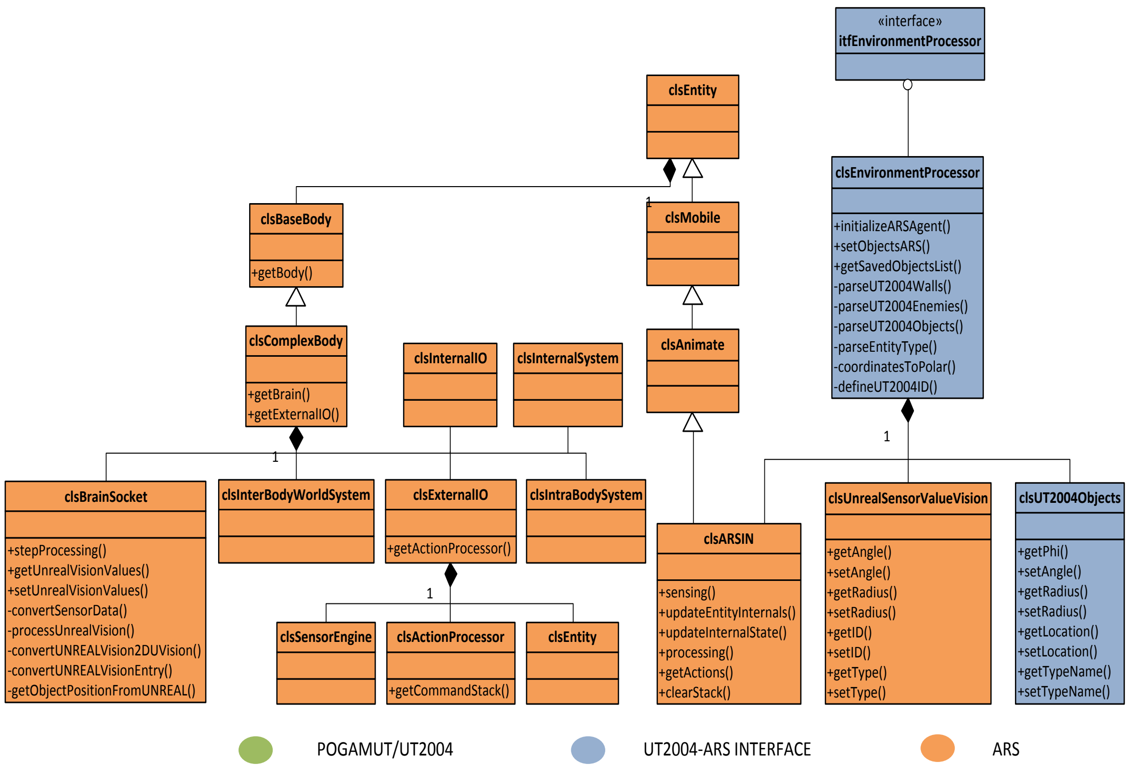


Figure 5.3: Environment interface UML diagram

5.4 Description of the Output Interface

The output interface is responsible for doing the sixth step of the list of actions in the logic() method explained in Chapter 5.1. The goal of this interface is to receive the output actions from the ARS, adapt them and send them to the ARSBot where the desired actions will be executed. Before explaining how this is done, it is good to give a brief description of the class clsUT2004Action that was created on purpose for this thesis.

<u>clsUT2004Action</u>

The output of the ARS is received in a XML String like this: <ActionName>ActionName@Parameter1:Parameter2:...:ParameterN</ActionName>.

The purpose of creating the class clsUT2004Action is to split the XML String into different attributes that contain the action name and each of the parameters separately. In this way it would be easier later to read the actions with their parameters in order to make the bot execute them. For the moment, the defined attributes in this class are: nameAction, speed, degrees, location. Note that these are the useful attributes for the actions that are being used for the moment. In case that new actions are defined, it is possible that new attributes have to be defined here. The created methods in this class are:

- **setNameAction():** it sets the name of the action that the ARS provides in the output.
- **getNameAction():**it gets the name of the action that the ARS provides in the output.
- **setSpeed():** it sets the speed with which the action has to be executed. It is only useful for actions like move in which the speed of movement is defined.
- **getSpeed():** it gets the speed with which the action has to be executed.
- **setDegrees():** it sets the degrees that the agent has to turn. It is only useful for the action turn.
- **getDegrees():** it gets the degrees that the agent has to turn.
- **setLocation():** it sets the location where the agent has to move to.
- **getLocation():** it gets the location where the agent has to move to.

In order to get the output actions, the method getActions of clsARSIN is called. In this method, first it is obtained the body clsBaseBody and it is cast to clsComplexBody. To obtain an entity clsActionProcessor, the getExternalIO() method is called and later the getActionProcessor(). Once this is done the method getCommandStack() is called and an ArrayList of the type clsActionCommand (XML String) is returned. Once the ArrayList with the actions is obtained, every position of it is parsed to the class clsUT2004Action using the method parseARSAction(). Therefore, the parameters of the actions received in a XML String are set into the different attributes of the class clsUT2004Action (see Chapter 4.2.3 for more details). In case that the exact location of an object is necessary, it is possible to obtain it with the targetObjectLocation() method that will retrieve the exact location from the class clsUT2004Objects where it was saved. To end, an ArrayList of the type clsUT2004Action is forwarded to the clsInterfacesManager which at the same time will send the ArrayList to the ARSBot. The ARSBot will receive the actions and execute them (see step 7 of Chapter 5.1).

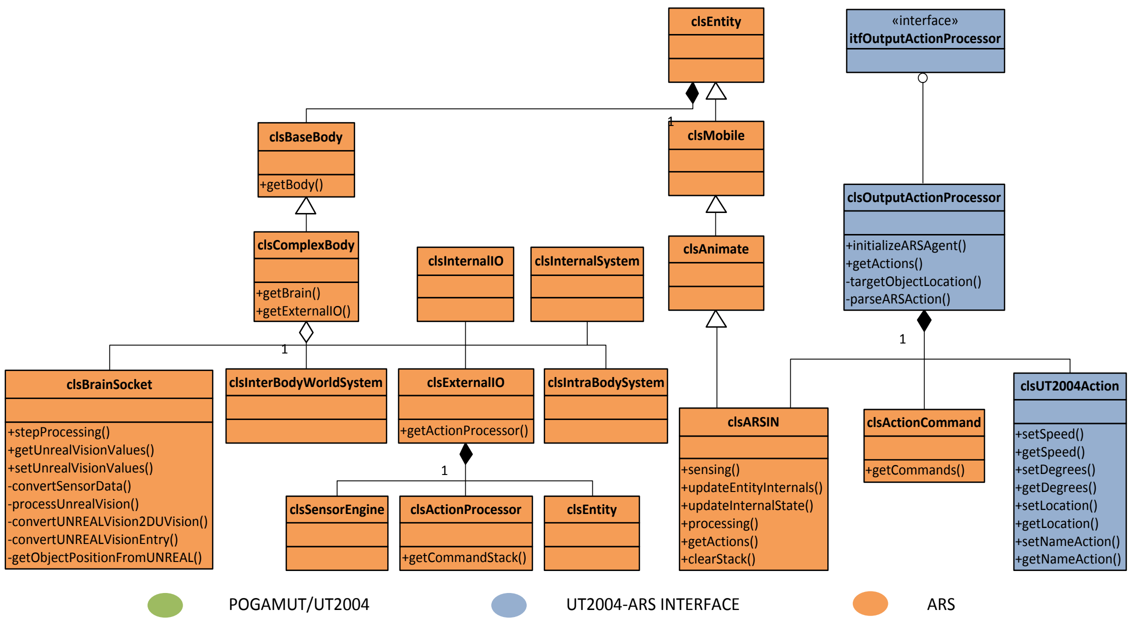


Figure 5.4: Output interface UML diagram

6. Validation

In order to check if the connectivity of the interface between the ARS and the UT2004 and the new actions defined in ARS were working properly some use cases were defined. With the actual implementation of the ARS only three different use cases were possible to be defined in this thesis because before the end of the thesis, only the output actions SEARCH, UNREAL_MOVE_TO and FLEE of the ARS were useful. The conditions for testing the bot are explained in Chapter 6.1 and later the three use cases are described in Chapters 6.2, 6.3 and 6.4. At the end, in Chapter 6.5, the Botprize competition, which is a competition where the ARS could be tested for checking its human-likeness, is presented.

6.1 Preconditions

The evaluation of the three use cases is going to be done using an UT2004 map called TrainingDay (see Figure 6.1). This map is the smallest and simplest map of the game and for this reason we consider it the most appropriate for testing our bot.

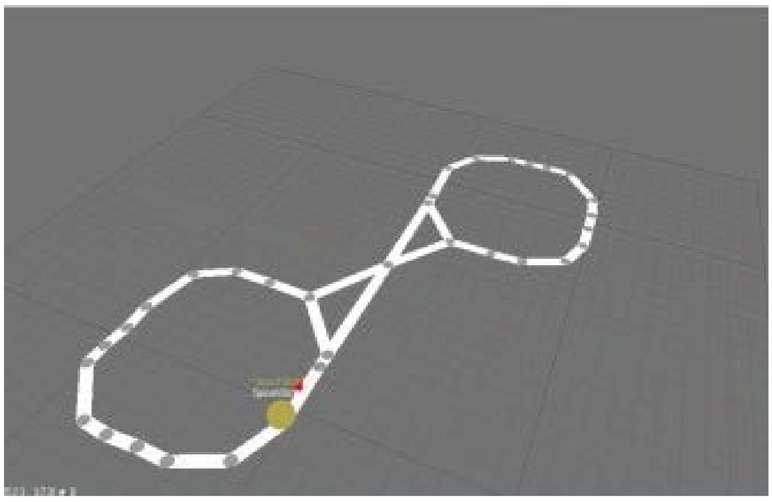


Figure 6.1: Geometry of the TrainingDay map [1]

The game mode is Deathmatch, but this is not important because we are not going to play a Deathmatch game that consists of killing as much bots as possible. Therefore, for the moment it would be possible to select any other game mode.

To evaluate the use cases we are going only to insert one bot in the game and ourselves as a spectator. When an enemy is required (third use case), we are going to act as a player for being the enemy of the bot.

6.2 First use case: Search

The search sequence was already defined in the ARS framework before starting this thesis and it can be useful for exploring the UT2004 map. Therefore, this was the first use case to be done. Although the search sequence was already implemented in ARS, the search algorithm was not good enough for an UT2004 bot, because it consisted only in going forward and turning right twenty degrees repeatedly.

In case that the search sequence was good, the search use case should work as shown in Figure 6.2: Every time that the bot has no activated drives or emotions, the search sequence is provided in the ARS output. The next step is to decompose the sequence into the simple actions which are going forward or turn to any of the sides a specific amount of degrees. Once the simple actions are obtained, the bot executes them in UT2004.

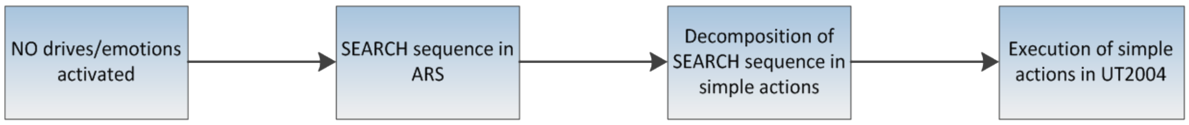

NO drives/emotions activated
SEARCH sequence in ARS
Decomposition of SEARCH sequence in simple actions
Execution of simple actions in UT2004

Figure 6.2 - Desired Search Use Case

Because of the simplicity of the ARS's search algorithm, we decided to implement our own search algorithm in the UT2004 and use it instead of the simple actions provided by the search sequence of the ARS. Therefore, we can say that the simple actions are ignored and the new search algorithm is used instead. In Figure 6.3 the implemented search use case is shown:

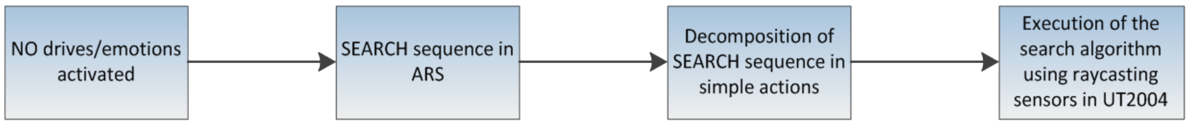

NO drives/emotions activated
SEARCH sequence in ARS
Decomposition of SEARCH sequence in simple actions
Execution of the search algorithm using raycasting sensors in UT2004

Figure 6.3: Implemented Search Use Case

In Figure 6.4, the search algorithm used is shown. This search algorithm uses ray casting(section 4.2.2). To navigate, the bot only takes into account the 'near' rays.

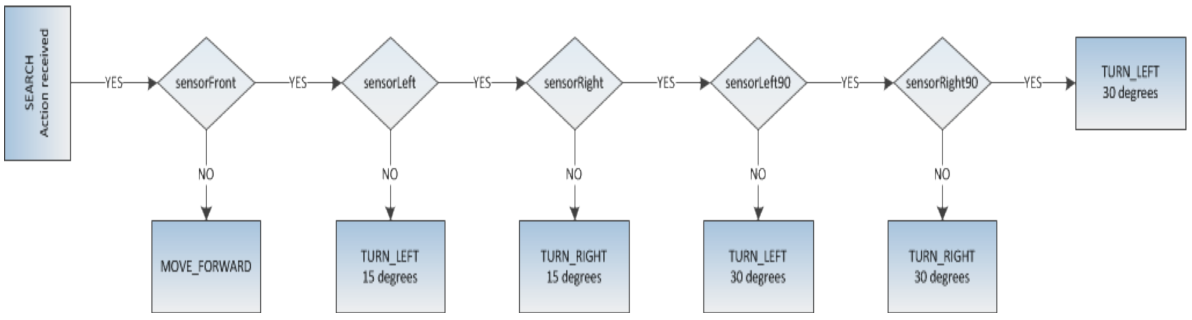


Figure 6.4: Search algorithm

It is important to say that at the end the search use case should work like Figure 6.2, because in the implemented one the ARS decision unit is not the one that decides where to go in order to search. This was only a trick in order to see the bot really searching in the UT2004 until the ARS search sequence is improved.

6.3 Second use case: Pick up a health pack

This use case works as shown in Figure 6.5:

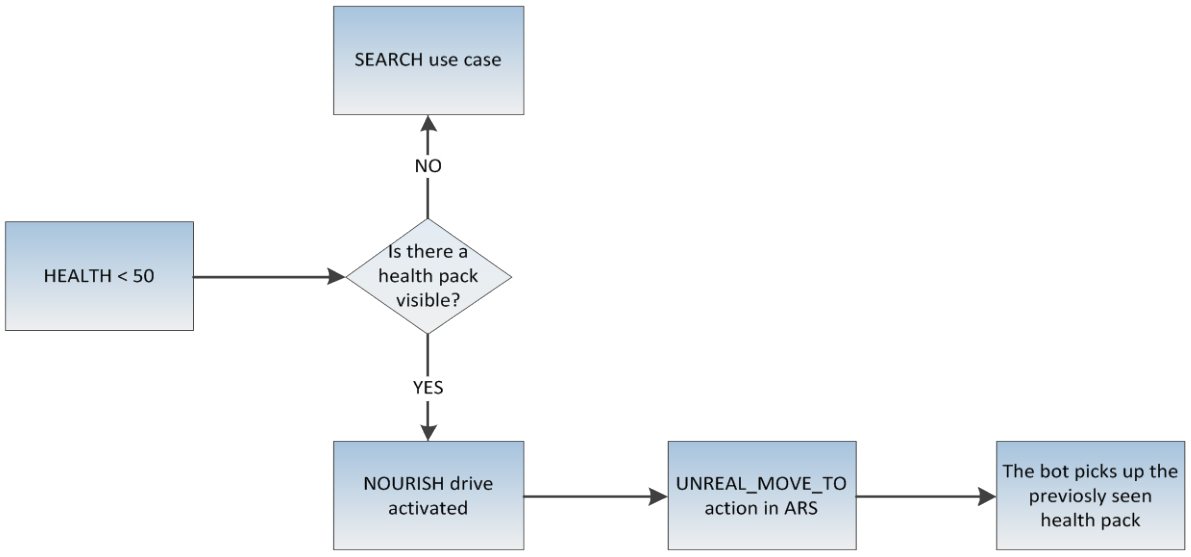

SEARCH use case
NO
HEALTH < 50
Is there a health pack visible?
YES
NOURISH drive activated
UNREAL_MOVE_TO action in ARS
The bot picks up the previosly seen health pack

Figure 6.5: Pick up a health pack use case

When the bot's health level is below fifty (out of two-hundred), the bot has to check if there is a health pack in its visible area. If there is not, the search use case will be done (see Chapter 6.2). If there is, the nourish drive of the ARS would be activated. To activate the nourish drive the stomach parameters are emptied making the bot to become hungry. Being hungry will make the bot to go to pick up a health pack because the health pack is defined as the drive object of the drive nourish. Therefore, the action UNREAL_MOVE_TO would be generated in the output of the ARS system. This action is received in the interface with a String with this format: "UNREAL_MOVE_TO|HEALTH:DIRECTION:DISTANCE". Thus, the bot knows which type of object has to pick up (in this case a health pack) and its location in order to move to it. After receiving this action, the interface compares the ARS location with the previously saved one (see Section 4.2.2) and retrieves the exact location of the item to pick up in Cartesian coordinates. Once the bot knows the exact location of the item, it has just to go there and pick it up. Whereas in the search action the bot navigates using the ray casting sensors, in this use case it navigates using the path planner of the Pogamut. Using the path planner, the navigation is done by following some navigation points that exist in the map with a well-known location. In Figure 6.6, the path grid that connects the navigation points can be seen.

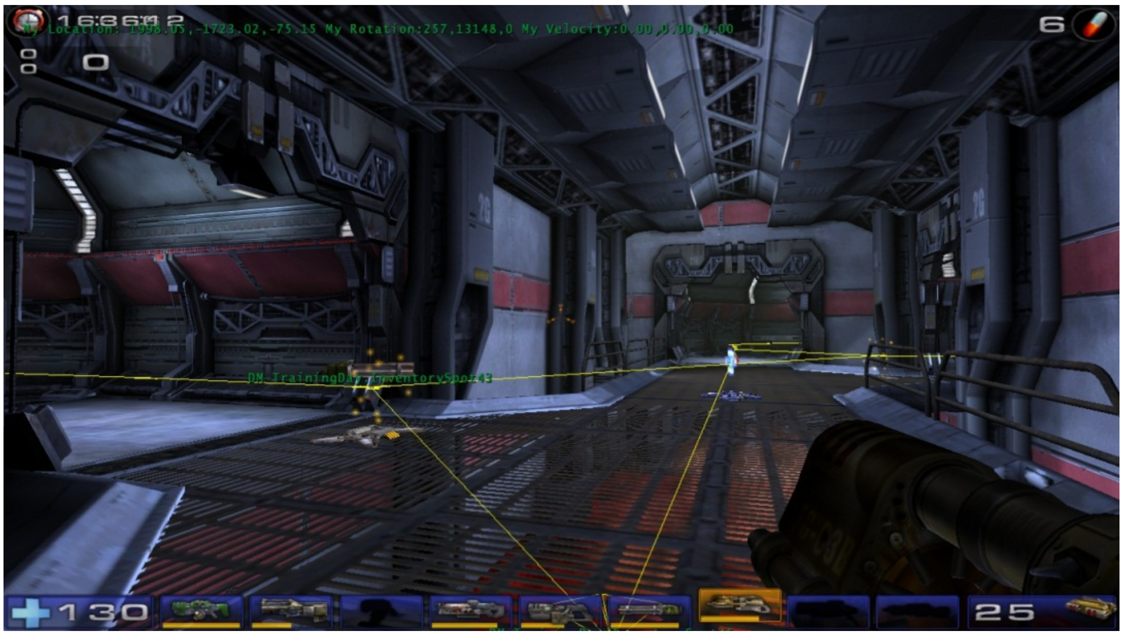


Figure 6.6: UT2004 grid [10]

6.4 Third use case: Panic

When a visible enemy is detected by the bot, the panic emotion, which has the highest priority, is triggered. This emotion provides the flee sequence in the output of the ARS. While implementing this use case we realised that the bot had never shot and we considered that we should make the Bot shoot somewhere because this is the main action in the UT2004 game. The problem was that the shoot action had not been defined in the ARS system yet. For this reason we decided to split the flee action into two depending on the Bot’s health level when it receives the flee action. Thus, when the bot receives the flee sequence, its health level is checked and acts like this:

a) If health >= 75 the bot will shoot the enemy. (Note that the UT2004 health limit is 200 but it starts with 100).
b) If health < 75 the bot will flee from the enemy. In order to flee, the bot turns 180 degrees and executes the same algorithm as in Figure 6.4.

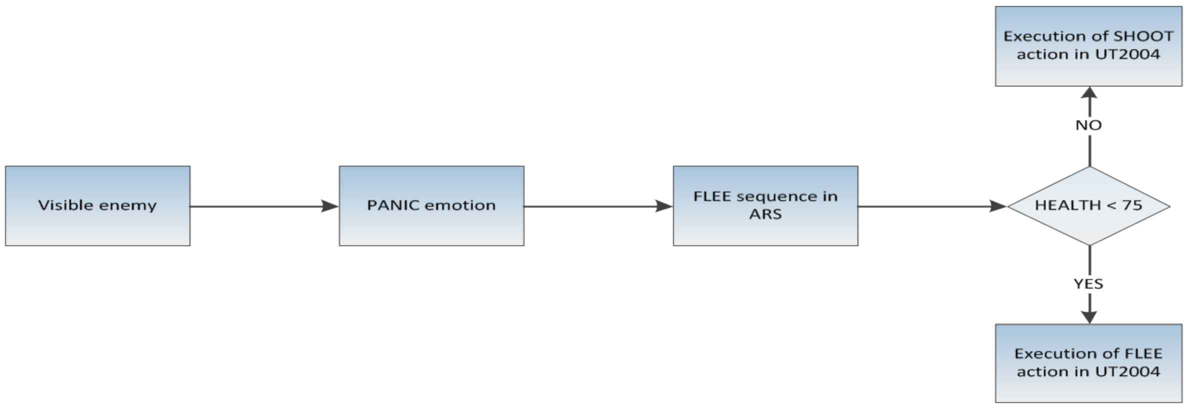

Visible enemy
PANIC emotion
FLEE sequence in ARS
HEALTH < 75
NO
Execution of SHOOT action in UT2004
YES
Execution of FLEE action in UT2004

Figure 6.7: Panic use case

We justify the insertion of the shoot action into the panic use case by saying that if the health is high and the bot sees an enemy is like the bot is encouraged to attack and when the health is low the bot prefers to flee because it is safer than attacking.

Afterwards, the shoot action should be defined in the ARS system creating thus a new shoot use case in which the ARS decision unit is the one that truly decides whether is appropriate to shoot or not. In Figure 6.8 we show how we think that the shoot and flee should work. In case of the shoot use case (when an enemy is visible and the health level is over 75) the bite drive can be reused and associated to the shoot action because both are destructive actions. Therefore, when the bite drive would be activated, the shoot action would be provided by the ARS and the bot would shoot the enemy in the UT2004. The flee use case would work like now but the flee sequence of the ARS should provide the simple actions to do as it would have to do it in the search sequence.

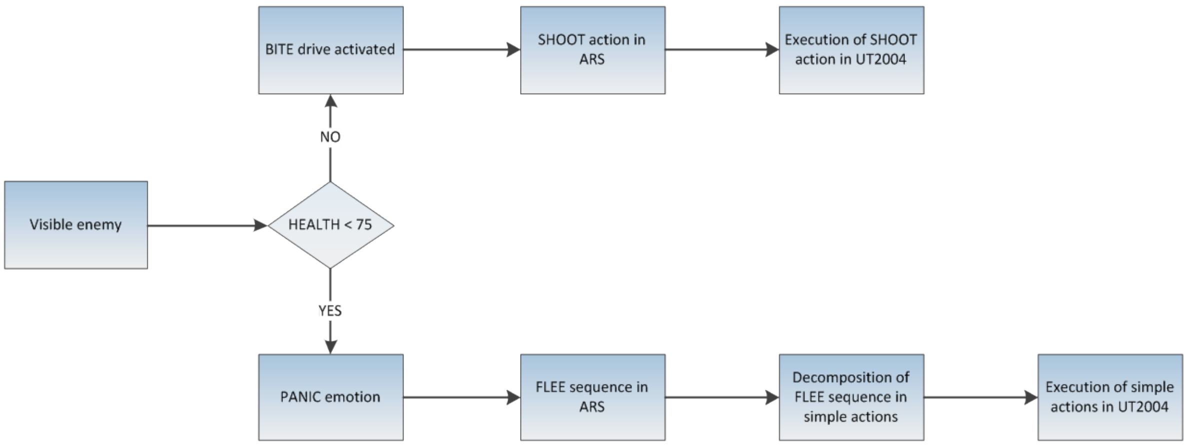

BITE drive activated
SHOOT action in ARS
Execution of SHOOT action in UT2004
NO
Visible enemy
HEALTH < 75
YES
PANIC emotion
FLEE sequence in ARS
Decomposition of FLEE sequence in simple actions
Execution of simple actions in UT2004

Figure 6.8: How SHOOT and FLEE should really work

6.5 Botprize competition

The Botprize competition is a competition where the human similarity of different UT2004 bots is measured and also compared among them. There's a prize for the bots that achieve a certain humanness rate (it varies year by year). This competition could be a good way to validate the ARS but at the end of this thesis the implementation of the ARS in the UT2004 game was not ready to participate in the competition because some necessary actions, like for example shooting, are lacking. However, when the bot is complete, meaning that it can navigate itself through an UT2004 map, it would be good to participate in the Botprize competition and check the humanness of the ARS.

In order to participate in the competition, an UT2004 game with socket-based interface that gives the possibility of controlling bots from an external program is needed. This interface is called Gamebots and it is already provided by the Pogamut platform. Thus, as this implementation of the ARS uses Pogamut, it is possible to participate in the Botprize competition. Another rule to participate is that the chat utility of the game has to be disabled.

The competition consists of a set of judges that play the game against the participant bots and also against other human players. During the game, the judges have to try to identify which of the players are human and which are bots by tagging them. Two different tags are available: HUMAN or BOT. To be able to tag the opponents, the LinkGun, which is a kind of weapon of the UT2004, is modified in order to have two different modes of shooting. The primary mode is used to tag what is considered to be a bot player and the secondary mode to tag what is considered to be a human player. Therefore, when a judge that is playing believes that the opponent is a bot, he will shoot the opponent using the primary mode of the LinkGun. Otherwise, when he believes that the opponent is a human, he will shoot the opponent using the secondary mode of the LinkGun. The judge can change his decision by shooting the opponent again before the game ends.

There is only one metric in the competition which is the humanness rate. This metric is obtained by calculating the percentage of times that a bot has been judged human over all the times that it has been judged.

To encourage the judges to judge the best as possible, some prizes are given to the best judge, the one with the highest combined score over the judging rounds, and for the judge with the highest humanness rating. [4]

7. Conclusions

Although the ARS bot is not complete, we can say that the main goal of the thesis is achieved because now a connection between the ARS and the UT2004 exists and moreover some use cases were defined due to the successful changes done in the ARS system. However, a lot of work is still necessary to do in order to achieve a complete bot and to be able to participate in the Botprize competition (see Chapter 6.5). For this reason, the explanation of the achievements and also of the possible future work is done in this chapter. Also, an explanation of another kind of project that could be the next step of the ARS development is given at the beginning of this chapter. This project is called Emohawk and it is considered to be a good environment where the ARS could be implemented in. The reason is that the Emohawk project seems to be more humanistic because it happens in a city where human-appearance players live and do more realistic actions, not like in UT2004. Moreover, after this thesis, the implementation of ARS in the Emohawk project appears to be easy because it uses the Pogamut platform and the UT2004 game too, but with a new map created for this purpose. Thus, only few changes seem to be needed to implement ARS in the Emohawk project.

7.1 The Emohawk project

The project Emohawk is built upon the Pogamut platform (see Chapter 3.3) and it uses UT2004 as the virtual world. The first goal of the Emohawk project was to develop the following story, created only for this reason, into the virtual world:

"There is a small city with about a thousand inhabitants. There lives Bruno, aged 19, in the town. Bruno has a girlfriend Anne and... well... yet another girlfriend Clementine. The girls do not know about each other. Bruno has just been with Anne in the cinema. Now, he has to walk her home quickly since Clementine is waiting for him to be taken to the very next movie. Besides, there is an emohawk roaming around. Emohawk is an alien creature that is found adorable by all IVAs in the story." [Bid09, p.10]

In order to implement this story with the Pogamut new sensors and actors had to be added to. Likewise, a new UT2004 non-violent map had to be created representing the 3D virtual city where the story has to take place as well as new character models with animations had to be created too. The new UT2004 map and the new characters were created using the Unreal Engine 2 Runtime (UE2).

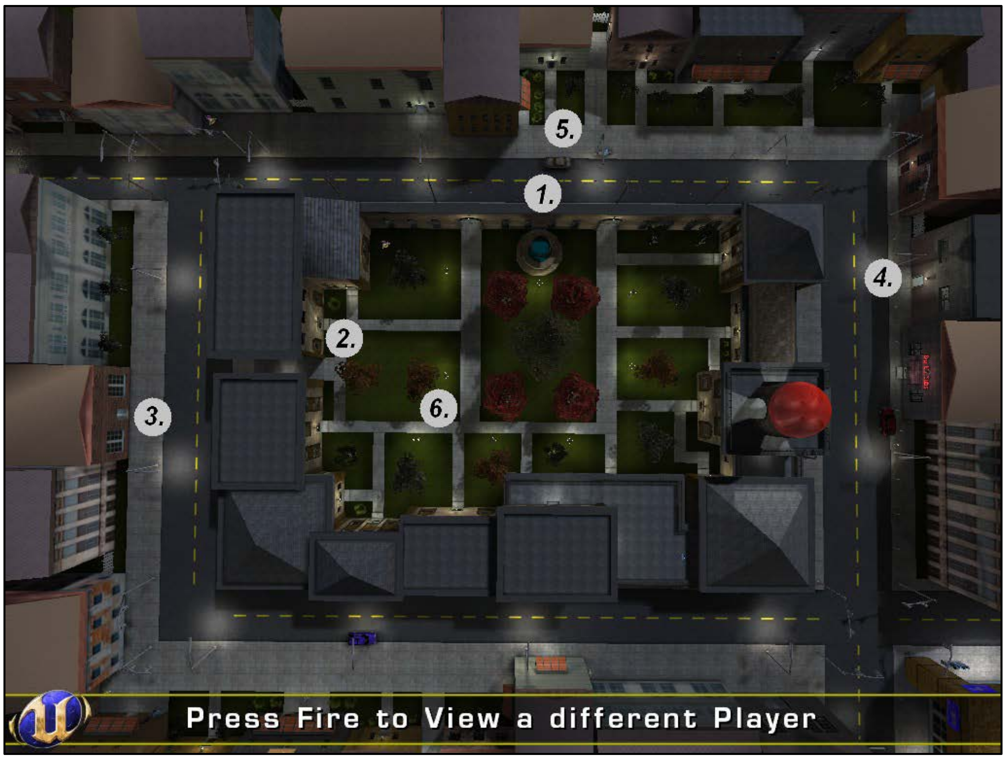


Figure 7.1: New UT2004 scenario DM-UnrealVille [Bid09, p.25]

In Figure 7.1 the new created scenario can be seen. Its name is UnrealVille and to satisfy the story requirements the following places are created in the new scenario: 1) Cinema and Bruno and Anne starting place, 2) Emohawk location, 3) Anne's home location, 4) Clementine's home location as well as her starting place, 5) Meeting place in front of the cinema and 6) Location of the park.

Apart from the virtual city the two female intelligent virtual agents (IVA) (Anne and Clementine), the male IVA (Bruno) and the alien IVA (Emohawk) were created too. Both, female and male IVAs, are controlled by affect-driven architecture. The affect-driven architecture in Emohawk is controlled by the emotional model ALMA (see Appendix C). ALMA model was chosen because it is a universal model, valid for any environment, it is written in Java like Pogamut, several affect types are featured and it is free for non-commercial purposes.

There are three ways to express the affects in the Emohawk. First, the action or proposal that the agent does when interacting with other agents are chosen depending on the affect values. Second, the affects make the agent do some kind of gesture as smiling during a conversation between agents. Third, there are some flares around the agent's head that change colour, size, speed and direction of movement depending on the affects and their intensity. Some of the ALMA emotions are mapped to colours following this table:

Emotion	Color
anger	bright red
fear	dark green
joy	yellow
distress	blue
love	pink
hate	black
liking	violet
disliking	brown
unspecified	white

Figure 7.2: Mapping of ALMA emotions to the flare's colours [Bid09, p.26]

There are also three interaction possibilities between IVAs in the Emohawk project: casual conversation, actions and proposals. Without going into detail the names of the possible actions are: compliment, kiss, sex, insult, slap, bye, leave, cuddle, kick and proposal. The different available proposals, again without going into detail, are: go to the cinema, go to the park, go home, kiss, have sex and leave. For more details check [Bid09].

The IVAs architecture can be seen in Figure 7.3. The information from the environment is processed by the Pogamut sensors and also by the agent perception module. This module generates inputs for the ALMA model and saves important data into the agent memory. The agent decision making is done evaluating the Pogamut sensory information, the agent's history from the memory (mostly about last actions and proposals), ALMA affects and the agent's current state. Thus, the Emohawk decision making module is a Mealy finite state machine because their outputs are determined both by its current state and by the values of its inputs. Later, the finite state machine will be described.

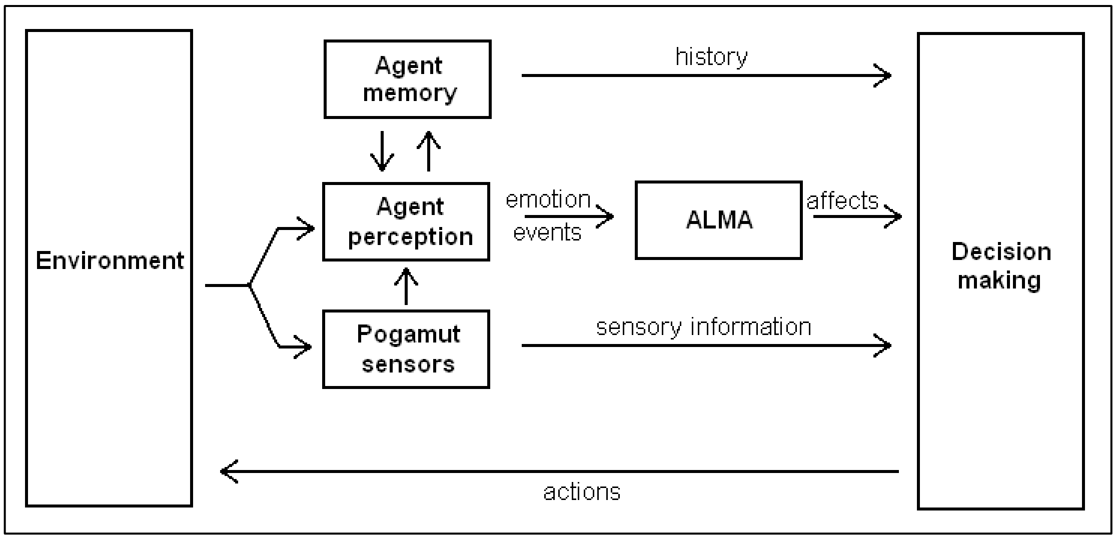


Figure 7.3: Emohawk agent affect-driven architecture [Bid09, p.28]

The agents use ALMA emotions to compute emotional attitude toward other agents defining in this way a new affect type called feeling. According to [Bid09] the feeling intensity is measured using this formula:

Feeling int. = 0.5 * gratitude + joy + liking + 1.5 * love – 2 * anger – distress – disliking – 1.5 * hate

The feeling values range from -5.5 to 4. If the value is between 1 and 2 the feeling is considered as friendship, above 2 as love. On the other hand, values below zero indicate that the agent dislikes the agent who is interacting with.

In Figure 7.4 the decision making finite state machine with its possible transitions can be seen:

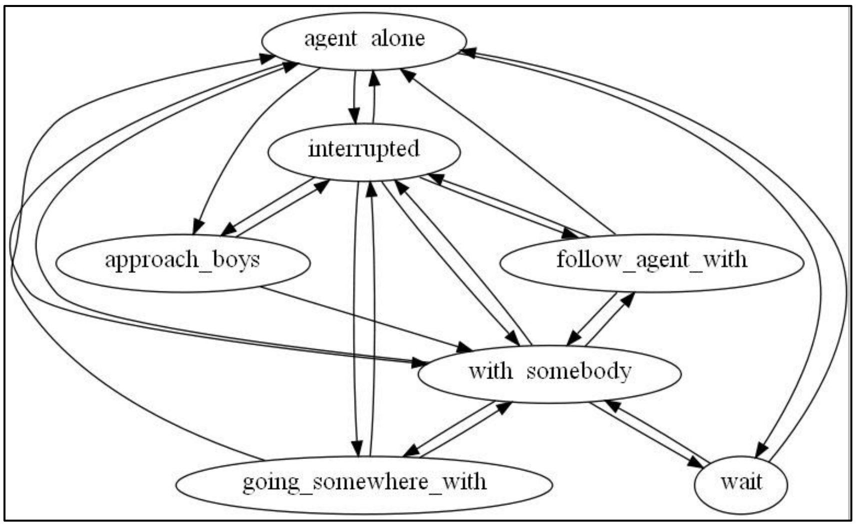


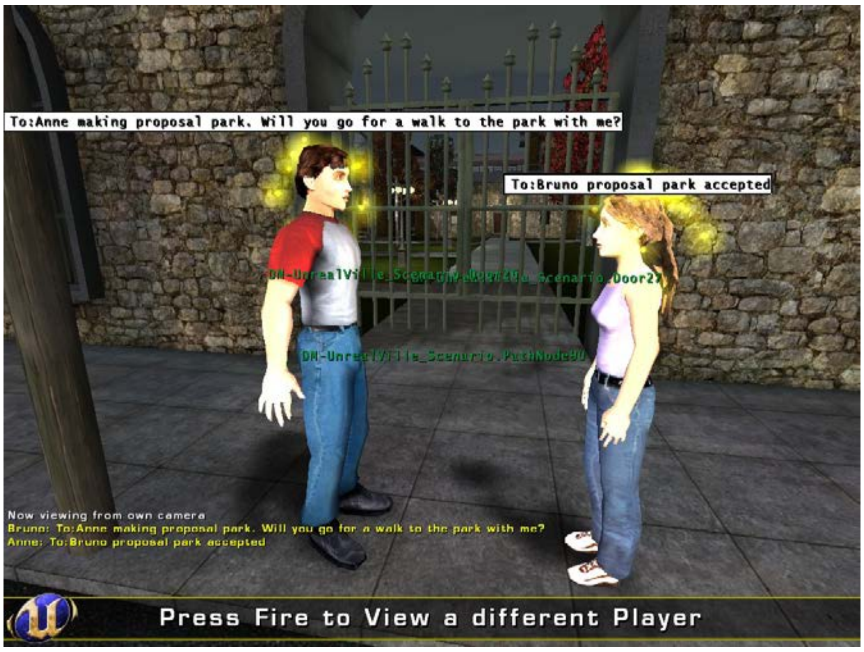
Figure 7.4: IVAs decision making FSM [Bid09, p.32]

The description of each of the states is as follows:

- **Agent_alone:** an agent is in this state when it is alone in the environment. If the agent is a boy, he will start to explore the map with the target of meeting someone. In case that the agent is a girl she will just go home
- **Follow_agent_with:** agent A proposes to agent B to go somewhere and agent B accepts. Then agent B switches to this state and follows agent A until they reach the destination
- **Going_somewhere_with:** when the same situation explained in the state above takes place, the agent A switches to this state.
- **Interrupted:** the agent switches to this state when another agent approaches it while it is doing any activity.
- **Wait:** the agent is in this state when waiting for someone. This state will be abandoned when the agent that is being waited appears or when the waiting timer runs out.
- **With_somebody:** this is the state where the agent is with somebody but still not doing any activity. Proposals are done in this state.

- **Approach_boys:** used only by girl-agents, they will switch to this state after some time waiting at home. The girl will explore the map looking for boys. There is the possibility of finding Bruno with the other girl and then the conflict among the three agents will start.

The user has two possibilities of using the Emohawk project. It is possible only to watch the story as a spectator or to join the scenario as a new character.



To:Anne making proposal park. Will you go for a walk to the park with me?

To:Bruno proposal park accepted

```
Now viewing from own camera
Bruno: To:Anne making proposal park. Will you go for a walk to the park with me?
Anne: To:Bruno proposal park accepted
```

DM-UnrealVille_Scenario.Door27
DM-UnrealVille_Scenario.PathNode90
Press Fire to View a different Player

Figure 7.5: Emohawk project view [Bid09, p.17]

7.2 Achievements

The main goal of this thesis was to implement the connection between the UT2004 computer game and the ARS system. At the end of the thesis we can say that this goal is achieved. Moreover, three use cases were done to prove that the connection is working well. During the implementation of the use cases some lacks of the ARS were discovered and we also realised that the ARS was not prepared to be implemented in UT2004 because it is a totally different world to the one that is used to live in. For this reason some changes were done in the ARS system like defining new actions with its drives or emotions.

This thesis was the first step to implement ARS in an external world and it has helped to notice what is not working well in ARS and to define new requirements for ARS. Afterwards ARS should be improved and a complete implementation in UT2004 would be done in order to participate in the Botprize competition (see Chapter 6.5).

Another of the goals was to make the interface as independent as possible so that it can become useful to adapt the ARS to any other environment different to UT2004. We tried to grant independence but it is not really independent yet because the inputs from another environment would be different and it would be different to process them. In any case we think that not a lot of work has to be done to adapt this interface to other environments. In fact, the UT2004-ARS interface here explained is a mediator between UT2004 and ARS. The ARS has a separate interface to connect to the mediator that after the work of this thesis is more known how to use it.

7.3 Future work

This thesis was just the beginning of the ARS extraction to other worlds which are at the same time more complex than the initial simulator. Thus a few proposals are listed here for further work:

1) **More use-cases:** it would be good to create new actions in the ARS framework which are useful for the UT2004 world. The first proposal is to implement the SHOOT action in the ARS and receive it in the interface in order to make the Bot shoot to an enemy in the game. Until now, the shoot was done in the third use case as an ‘extension’ of the FLEE action. The second proposal is to adapt the UNREAL_MOVE_TO action to be able to pick up any kind of item that appears in the map. At the moment the items that the interface receives from the UT2004 map are filtered leaving only the health packs and then when a bot detects a health pack (under a specific health level condition) it makes the ARS to provide the UNREAL_MOVE_TO action. The string received from the ARS is like this: UNREAL_MOVE_TO@ItemType:Angle:Distance. To improve this action, all the items should be introduced in the ARS and activate some kind of drive for each type of item under a certain condition. As an example: Pick up ammo for the actual weapon when the Bot’s ammo level is under a specific level.

2) **Improve the navigation of the Bot:** at the end of this thesis the navigation of the Bot was done using the ray casting sensors provided by Pogamut and using a simple algorithm to

avoid the Bot to be stuck. Despite the detected walls are introduced into the ARS system by using the ray casting sensors, the system does nothing with them for the moment. The ARS decision unit should in some way memorize the position of the walls and avoid them. This would not be easy because the resolution navigation of the ARS is very low if we compare it with the UT2004.

3) **Improve ARS vision resolution:** As mentioned in the previous point, the ARS vision resolution is very low compared with the UT2004. This fact will cause the improvement of the navigation very hard to implement. Therefore, it would be good to make the ARS resolution higher.

4) **Improve ARS processing speed:** the ARS system is working too slow for the UT2004 game and because of it sometimes the bot stops because the action is not yet available in the ARS output. A possible solution to make it faster it could be to divide the ARS model into two or three different threads. The first division can be between the primary process and the secondary and then the primary process can be divided between body perception and environment perception. In this way, applying concurrency, the ARS should be faster.

5) **Participate in Botprize competition:** when all the necessary actions for a UT2004 soldier are implemented in ARS and also in the interface in order to reproduce them in the game it would be good to participate in the Botprize competition and see how human is the ARS-Bot behaviour and if it is better than other bots or not.

6) **Implement ARS in other environments:** UT2004 is a good starting point to implement the ARS framework in other environments but maybe is not the most appropriate because both worlds are very different. It can be better to implement ARS in a more humanistic world such as Emohawk (see Section 7.1). In fact, Emohawk is implemented using the UT2004 and the Pogamut as well

Appendixes

A. Installing Pogamut

Note: This tutorial is made for Eclipse IDE and Windows O.S users

These are the steps to follow in order to get Pogamut correctly installed:

1- Install the Subclipse plugin:
- Start Eclipse and go to "Menu-->Help-->Install New Software"
- Click on the "Add" button on the right-upper side of the window
- Write Subclipse in the name field and http://subclipse.tigris.org/update_1.8.x in the location field. (This link is valid for Eclipse Helios Service Release 2. Check http://subclipse.tigris.org to find the appropriate link for other Eclipse versions)
- Select the new site from the combobox
- Install Subclipse plugins

2- Install M2Eclipse plugin:
- The same steps as to install Subclipse but with different update site
- The link this time is http://download.eclipse.org/technology/m2e/releases
- and the name M2Eclipse

3- Install a graphical SVN client (it is advised to use TortoiseSVN):
- Download it from: http://tortoisesvn.net/downloads.html and install it

4- Install the game Unreal Tournament 2004

5- Install the latest JDK Java Platform (if you have not installed it yet) from: http://www.oracle.com/technetwork/java/javase/downloads/index.html

6- Install Unreal Engine Runtime from: http://apacudn.epicgames.com/Three/WebHome.html

7- Install the Pogamut platform following the steps of the installator. You can find the Pogamut installator in: http://pogamut.cuni.cz/main/tiki-index.php?page=Download

<u>How to import a Pogamut Maven project?</u>

1- Start the Eclipse

2- Go to File->New-->Project

3- Select Maven Project:

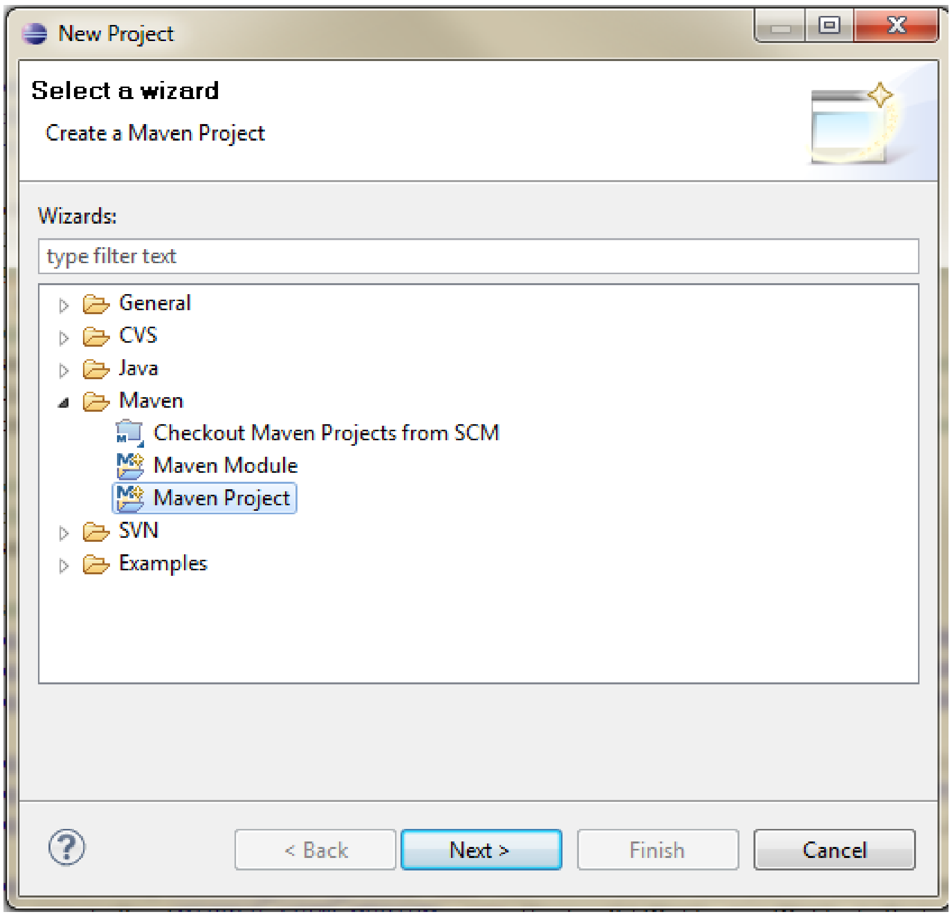
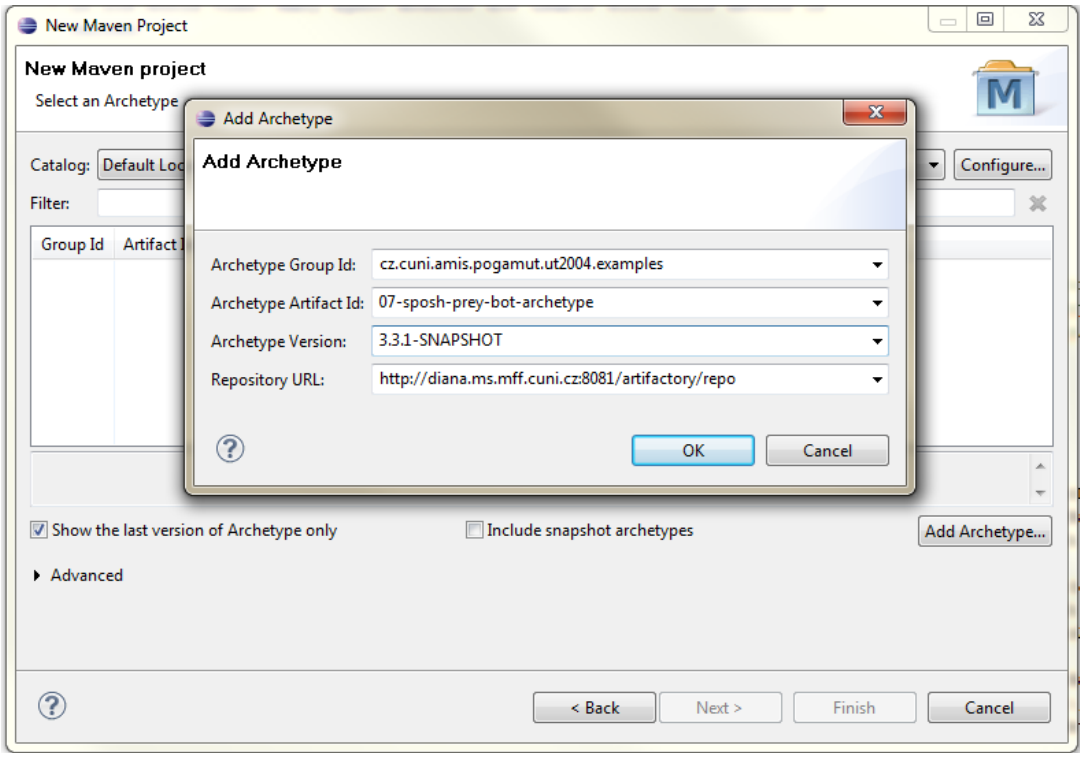


Figure A.1: Screenshot of how to create a Maven Project

4- Choose a location for the project and click Next

5- Click Add Archetype

6- Check the Pogamut's archetypes in the following link http://diana.ms.mff.cuni.cz:8081/artifactory/libs-snapshot-local/archetype-catalog.xml and fill the following window like in the image with the desired archetype:



New Maven Project

New Maven project

Select an Archetype

Catalog: Default Loc

Filter:

Add Archetype

Archetype Group Id: cz.cuni.amis.pogamut.ut2004.examples

Archetype Artifact Id: 07-sposh-prey-bot-archetype

Archetype Version: 3.3.1-SNAPSHOT

Repository URL: http://diana.ms.mff.cuni.cz:8081/artifactory/repo

☑ Show the last version of Archetype only

☐ Include snapshot archetypes

Figure A.2: Screenshot of how to add an archetype

7- Click in the "Include snapshot archetype" to see the recently added archetype in the actual window. Select it and click Next

8- Copy again the Artifact Id and click Finish

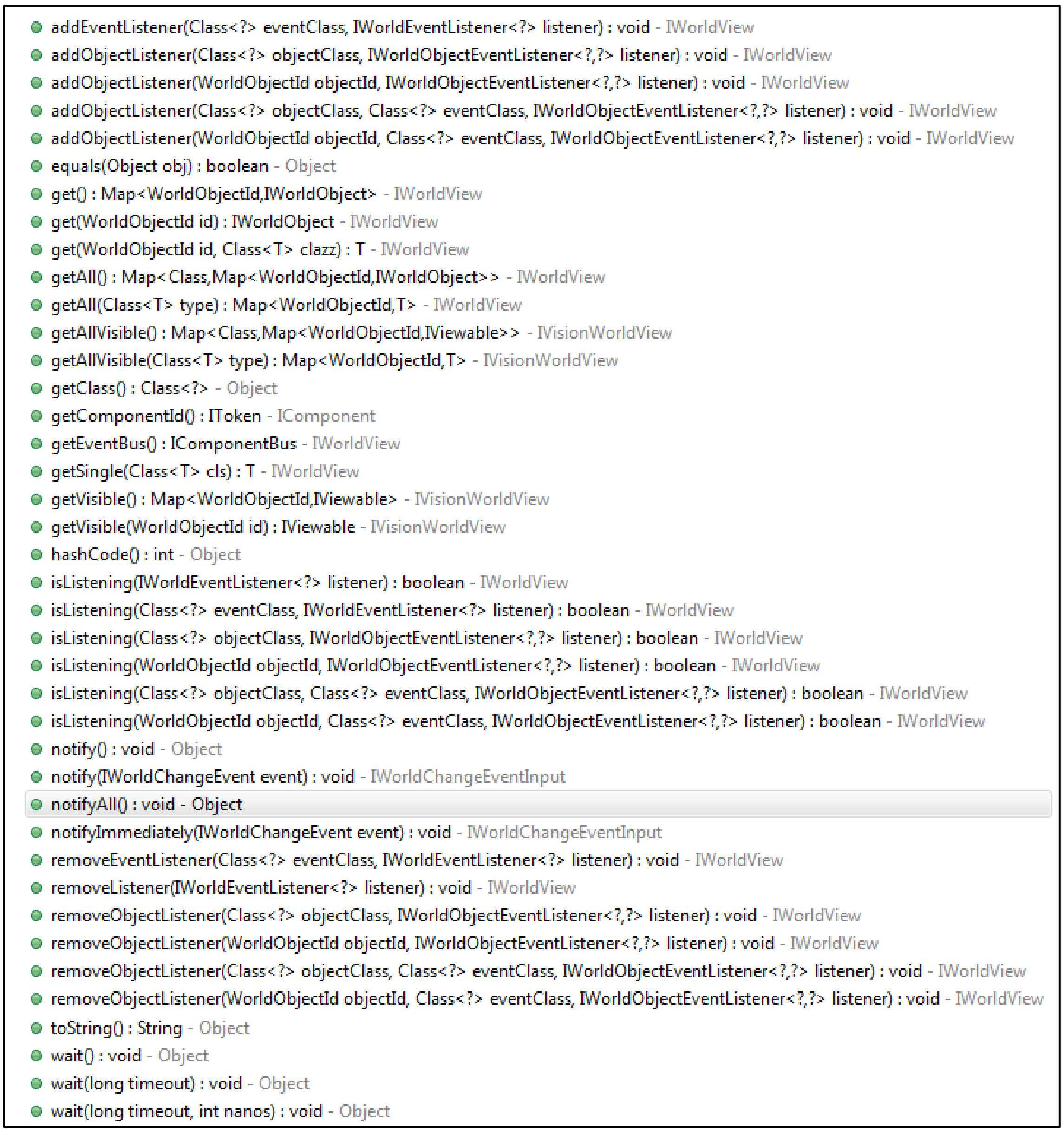
B. Pogamut's basic classes

The following classes are the most important or useful from the Pogamut's library:

- World

By using this class it is possible to obtain all the objects of the UT2004 world as well as adding or removing Listeners from the world in order to detect (or not if we remove the Listener) when something has changed in the world.

These are all the methods of this class:



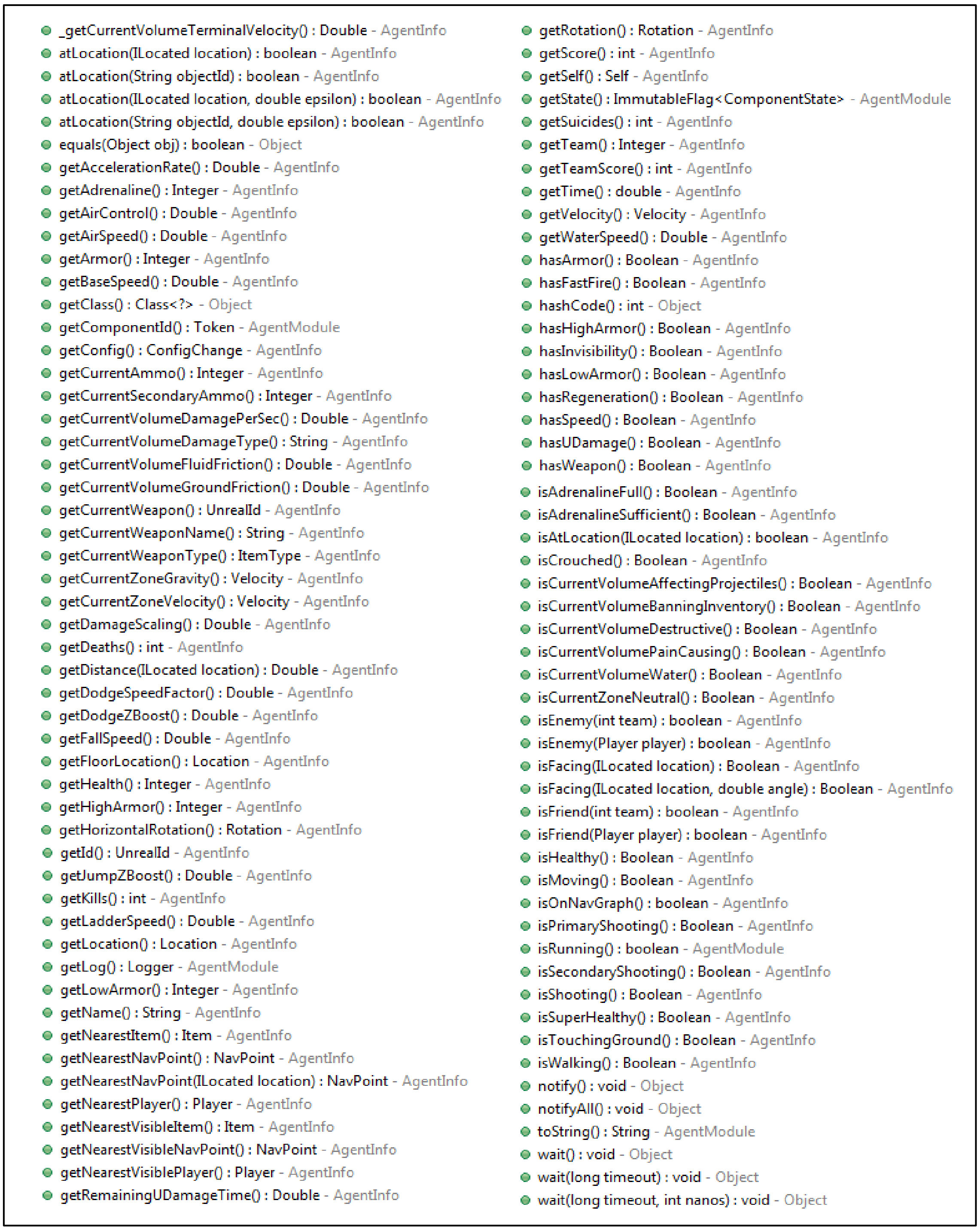
```
addEventListener(Class<?> eventClass, IWorldEventListener<?> listener) : void - IWorldView
addObjectListener(Class<?> objectClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
addObjectListener(WorldObjectId objectId, IWorldObjectEventListener<?,?> listener) : void - IWorldView
addObjectListener(Class<?> objectClass, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
addObjectListener(WorldObjectId objectId, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
equals(Object obj) : boolean - Object
get() : Map<WorldObjectId,IWorldObject> - IWorldView
get(WorldObjectId id) : IWorldObject - IWorldView
get(WorldObjectId id, Class<T> clazz) : T - IWorldView
getAll() : Map<Class,Map<WorldObjectId,IWorldObject>> - IWorldView
getAll(Class<T> type) : Map<WorldObjectId,T> - IWorldView
getAllVisible() : Map<Class,Map<WorldObjectId,IViewable>> - IVisionWorldView
getAllVisible(Class<T> type) : Map<WorldObjectId,T> - IVisionWorldView
getClass() : Class<?> - Object
getComponentId() : IToken - IComponent
getEventBus() : IComponentBus - IWorldView
getSingle(Class<T> cls) : T - IWorldView
getVisible() : Map<WorldObjectId,IViewable> - IVisionWorldView
getVisible(WorldObjectId id) : IViewable - IVisionWorldView
hashCode() : int - Object
isListening(IWorldEventListener<?> listener) : boolean - IWorldView
isListening(Class<?> eventClass, IWorldEventListener<?> listener) : boolean - IWorldView
isListening(Class<?> objectClass, IWorldObjectEventListener<?,?> listener) : boolean - IWorldView
isListening(WorldObjectId objectId, IWorldObjectEventListener<?,?> listener) : boolean - IWorldView
isListening(Class<?> objectClass, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : boolean - IWorldView
isListening(WorldObjectId objectId, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : boolean - IWorldView
notify() : void - Object
notify(IWorldChangeEvent event) : void - IWorldChangeEventInput
notifyAll() : void - Object
notifyImmediately(IWorldChangeEvent event) : void - IWorldChangeEventInput
removeEventListener(Class<?> eventClass, IWorldEventListener<?> listener) : void - IWorldView
removeListener(IWorldEventListener<?> listener) : void - IWorldView
removeObjectListener(Class<?> objectClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
removeObjectListener(WorldObjectId objectId, IWorldObjectEventListener<?,?> listener) : void - IWorldView
removeObjectListener(Class<?> objectClass, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
removeObjectListener(WorldObjectId objectId, Class<?> eventClass, IWorldObjectEventListener<?,?> listener) : void - IWorldView
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.1: World class methods

- Info

This class contains information about the internal state of the bot, such as the health level, adrenalin, movement speed, ammo for the actual weapon, etc.

These are the usable methods of this class:



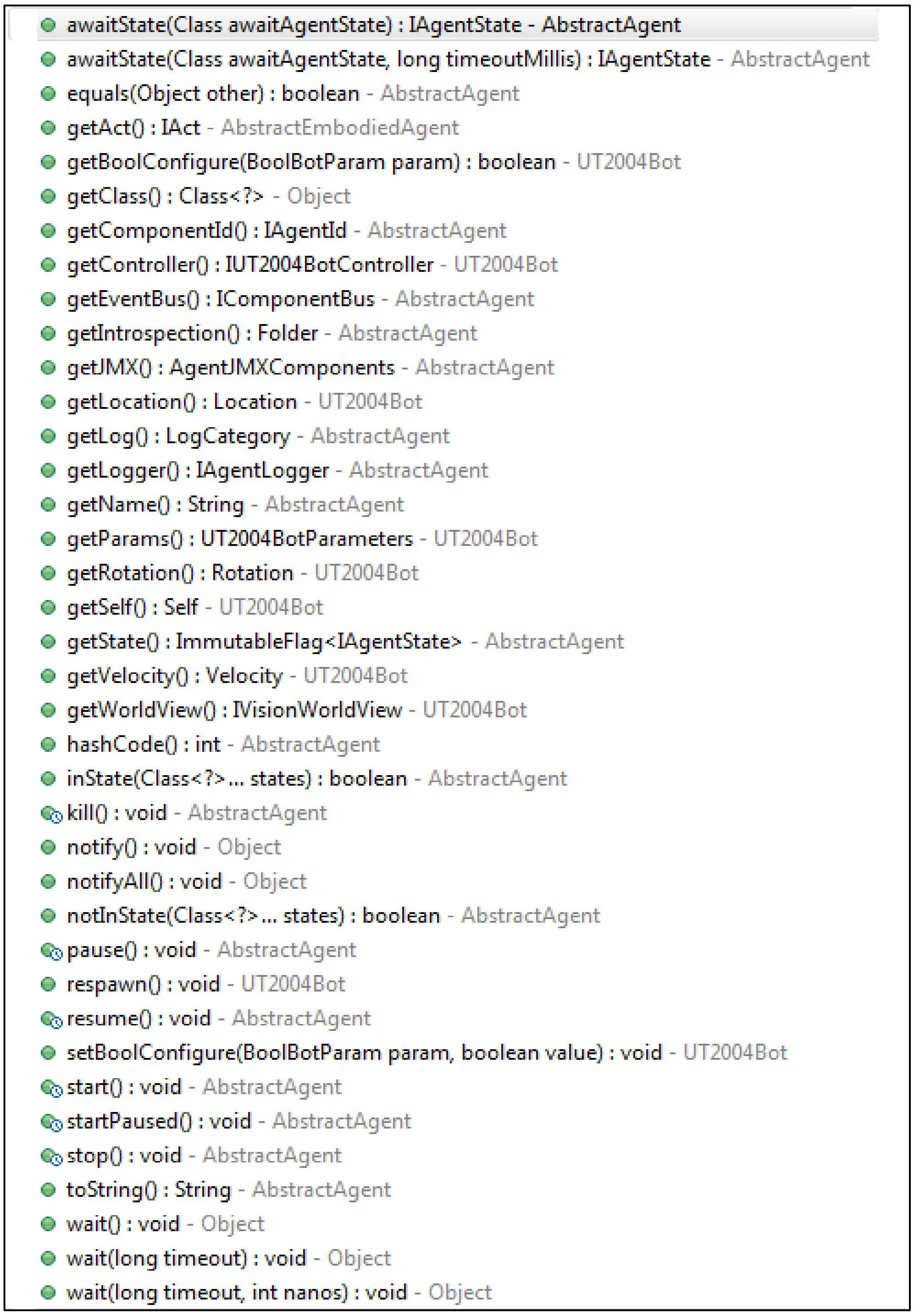
- _getCurrentVolumeTerminalVelocity() : Double - AgentInfo
- atLocation(ILocated location) : boolean - AgentInfo
- atLocation(String objectId) : boolean - AgentInfo
- atLocation(ILocated location, double epsilon) : boolean - AgentInfo
- atLocation(String objectId, double epsilon) : boolean - AgentInfo
- equals(Object obj) : boolean - Object
- getAccelerationRate() : Double - AgentInfo
- getAdrenaline() : Integer - AgentInfo
- getAirControl() : Double - AgentInfo
- getAirSpeed() : Double - AgentInfo
- getArmor() : Integer - AgentInfo
- getBaseSpeed() : Double - AgentInfo
- getClass() : Class<?> - Object
- getComponentId() : Token - AgentModule
- getConfig() : ConfigChange - AgentInfo
- getCurrentAmmo() : Integer - AgentInfo
- getCurrentSecondaryAmmo() : Integer - AgentInfo
- getCurrentVolumeDamagePerSec() : Double - AgentInfo
- getCurrentVolumeDamageType() : String - AgentInfo
- getCurrentVolumeFluidFriction() : Double - AgentInfo
- getCurrentVolumeGroundFriction() : Double - AgentInfo
- getCurrentWeapon() : UnrealId - AgentInfo
- getCurrentWeaponName() : String - AgentInfo
- getCurrentWeaponType() : ItemType - AgentInfo
- getCurrentZoneGravity() : Velocity - AgentInfo
- getCurrentZoneVelocity() : Velocity - AgentInfo
- getDamageScaling() : Double - AgentInfo
- getDeaths() : int - AgentInfo
- getDistance(ILocated location) : Double - AgentInfo
- getDodgeSpeedFactor() : Double - AgentInfo
- getDodgeZBoost() : Double - AgentInfo
- getFallSpeed() : Double - AgentInfo
- getFloorLocation() : Location - AgentInfo
- getHealth() : Integer - AgentInfo
- getHighArmor() : Integer - AgentInfo
- getHorizontalRotation() : Rotation - AgentInfo
- getId() : UnrealId - AgentInfo
- getJumpZBoost() : Double - AgentInfo
- getKills() : int - AgentInfo
- getLadderSpeed() : Double - AgentInfo
- getLocation() : Location - AgentInfo
- getLog() : Logger - AgentModule
- getLowArmor() : Integer - AgentInfo
- getName() : String - AgentInfo
- getNearestItem() : Item - AgentInfo
- getNearestNavPoint() : NavPoint - AgentInfo
- getNearestNavPoint(ILocated location) : NavPoint - AgentInfo
- getNearestPlayer() : Player - AgentInfo
- getNearestVisibleItem() : Item - AgentInfo
- getNearestVisibleNavPoint() : NavPoint - AgentInfo
- getNearestVisiblePlayer() : Player - AgentInfo
- getRemainingUDamageTime() : Double - AgentInfo
- getRotation() : Rotation - AgentInfo
- getScore() : int - AgentInfo
- getSelf() : Self - AgentInfo
- getState() : ImmutableFlag<ComponentState> - AgentModule
- getSuicides() : int - AgentInfo
- getTeam() : Integer - AgentInfo
- getTeamScore() : int - AgentInfo
- getTime() : double - AgentInfo
- getVelocity() : Velocity - AgentInfo
- getWaterSpeed() : Double - AgentInfo
- hasArmor() : Boolean - AgentInfo
- hasFastFire() : Boolean - AgentInfo
- hashCode() : int - Object
- hasHighArmor() : Boolean - AgentInfo
- hasInvisibility() : Boolean - AgentInfo
- hasLowArmor() : Boolean - AgentInfo
- hasRegeneration() : Boolean - AgentInfo
- hasSpeed() : Boolean - AgentInfo
- hasUDamage() : Boolean - AgentInfo
- hasWeapon() : Boolean - AgentInfo
- isAdrenalineFull() : Boolean - AgentInfo
- isAdrenalineSufficient() : Boolean - AgentInfo
- isAtLocation(ILocated location) : boolean - AgentInfo
- isCrouched() : Boolean - AgentInfo
- isCurrentVolumeAffectingProjectiles() : Boolean - AgentInfo
- isCurrentVolumeBanningInventory() : Boolean - AgentInfo
- isCurrentVolumeDestructive() : Boolean - AgentInfo
- isCurrentVolumePainCausing() : Boolean - AgentInfo
- isCurrentVolumeWater() : Boolean - AgentInfo
- isCurrentZoneNeutral() : Boolean - AgentInfo
- isEnemy(int team) : boolean - AgentInfo
- isEnemy(Player player) : boolean - AgentInfo
- isFacing(ILocated location) : Boolean - AgentInfo
- isFacing(ILocated location, double angle) : Boolean - AgentInfo
- isFriend(int team) : boolean - AgentInfo
- isFriend(Player player) : boolean - AgentInfo
- isHealthy() : Boolean - AgentInfo
- isMoving() : Boolean - AgentInfo
- isOnNavGraph() : boolean - AgentInfo
- isPrimaryShooting() : Boolean - AgentInfo
- isRunning() : boolean - AgentModule
- isSecondaryShooting() : Boolean - AgentInfo
- isShooting() : Boolean - AgentInfo
- isSuperHealthy() : Boolean - AgentInfo
- isTouchingGround() : Boolean - AgentInfo
- isWalking() : Boolean - AgentInfo
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - AgentModule
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Figure B.2: Info class methods

- Bot

This class represents the instance of the bot that is running at the moment and saves the information related to the bot and the game.

These are the methods of this class:



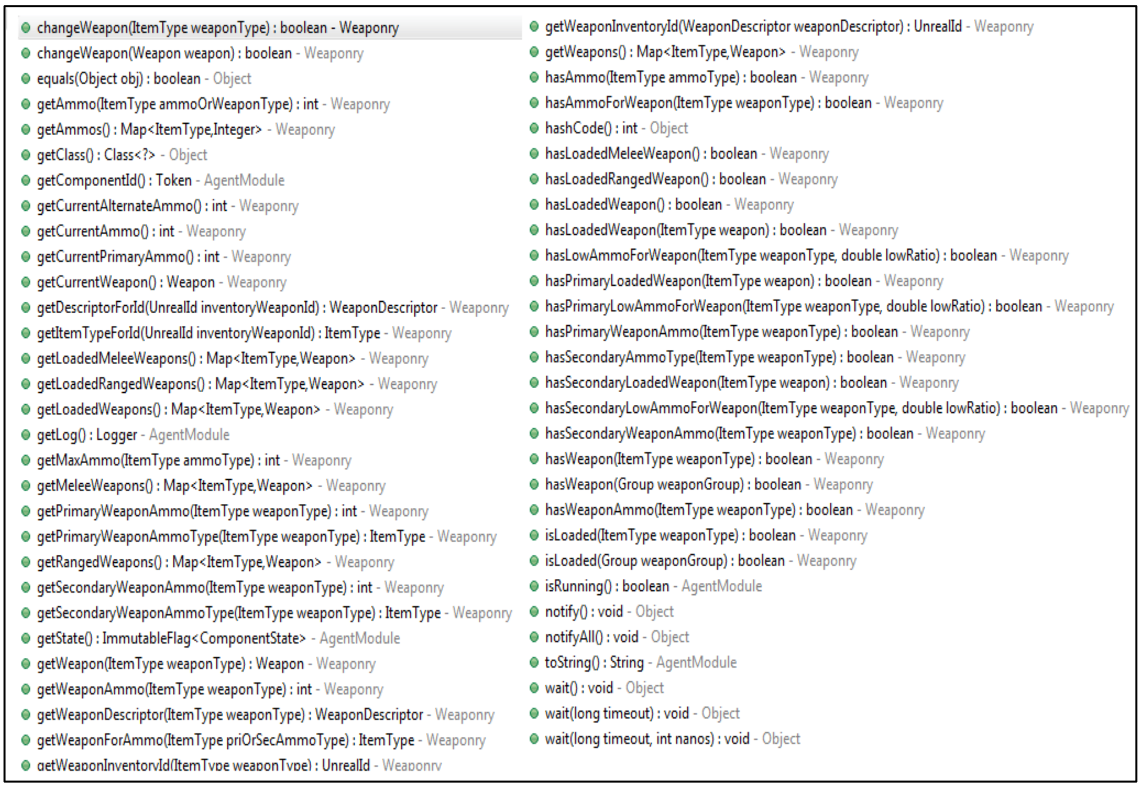
```
awaitState(Class awaitAgentState) : IAgentState - AbstractAgent
awaitState(Class awaitAgentState, long timeoutMillis) : IAgentState - AbstractAgent
equals(Object other) : boolean - AbstractAgent
getAct() : IAct - AbstractEmbodiedAgent
getBoolConfigure(BoolBotParam param) : boolean - UT2004Bot
getClass() : Class<?> - Object
getComponentId() : IAgentId - AbstractAgent
getController() : IUT2004BotController - UT2004Bot
getEventBus() : IComponentBus - AbstractAgent
getIntrospection() : Folder - AbstractAgent
getJMX() : AgentJMXComponents - AbstractAgent
getLocation() : Location - UT2004Bot
getLog() : LogCategory - AbstractAgent
getLogger() : IAgentLogger - AbstractAgent
getName() : String - AbstractAgent
getParams() : UT2004BotParameters - UT2004Bot
getRotation() : Rotation - UT2004Bot
getSelf() : Self - UT2004Bot
getState() : ImmutableFlag<IAgentState> - AbstractAgent
getVelocity() : Velocity - UT2004Bot
getWorldView() : IVisionWorldView - UT2004Bot
hashCode() : int - AbstractAgent
inState(Class<?>... states) : boolean - AbstractAgent
kill() : void - AbstractAgent
notify() : void - Object
notifyAll() : void - Object
notInState(Class<?>... states) : boolean - AbstractAgent
pause() : void - AbstractAgent
respawn() : void - UT2004Bot
resume() : void - AbstractAgent
setBoolConfigure(BoolBotParam param, boolean value) : void - UT2004Bot
start() : void - AbstractAgent
startPaused() : void - AbstractAgent
stop() : void - AbstractAgent
toString() : String - AbstractAgent
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.3:Bot class methods

- Weaponry

This class contains all the information about the weapons that the bot has at the moment.

These are the methods of this class:



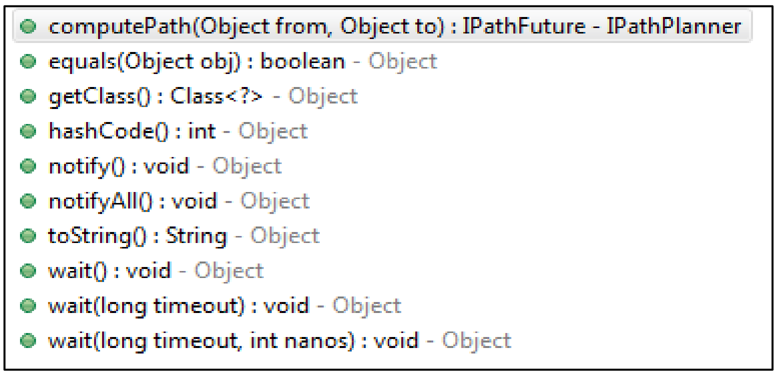
```
changeWeapon(ItemType weaponType) : boolean - Weaponry
changeWeapon(Weapon weapon) : boolean - Weaponry
equals(Object obj) : boolean - Object
getAmmo(ItemType ammoOrWeaponType) : int - Weaponry
getAmmos() : Map<ItemType,Integer> - Weaponry
getClass() : Class<?> - Object
getComponentId() : Token - AgentModule
getCurrentAlternateAmmo() : int - Weaponry
getCurrentAmmo() : int - Weaponry
getCurrentPrimaryAmmo() : int - Weaponry
getCurrentWeapon() : Weapon - Weaponry
getDescriptorForId(UnrealId inventoryWeaponId) : WeaponDescriptor - Weaponry
getItemTypeForId(UnrealId inventoryWeaponId) : ItemType - Weaponry
getLoadedMeleeWeapons() : Map<ItemType,Weapon> - Weaponry
getLoadedRangedWeapons() : Map<ItemType,Weapon> - Weaponry
getLoadedWeapons() : Map<ItemType,Weapon> - Weaponry
getLog() : Logger - AgentModule
getMaxAmmo(ItemType ammoType) : int - Weaponry
getMeleeWeapons() : Map<ItemType,Weapon> - Weaponry
getPrimaryWeaponAmmo(ItemType weaponType) : int - Weaponry
getPrimaryWeaponAmmoType(ItemType weaponType) : ItemType - Weaponry
getRangedWeapons() : Map<ItemType,Weapon> - Weaponry
getSecondaryWeaponAmmo(ItemType weaponType) : int - Weaponry
getSecondaryWeaponAmmoType(ItemType weaponType) : ItemType - Weaponry
getState() : ImmutableFlag<ComponentState> - AgentModule
getWeapon(ItemType weaponType) : Weapon - Weaponry
getWeaponAmmo(ItemType weaponType) : int - Weaponry
getWeaponDescriptor(ItemType weaponType) : WeaponDescriptor - Weaponry
getWeaponForAmmo(ItemType priOrSecAmmoType) : ItemType - Weaponry
getWeaponInventoryId(ItemType weaponType) : UnrealId - Weaponry
getWeaponInventoryId(WeaponDescriptor weaponDescriptor) : UnrealId - Weaponry
getWeapons() : Map<ItemType,Weapon> - Weaponry
hasAmmo(ItemType ammoType) : boolean - Weaponry
hasAmmoForWeapon(ItemType weaponType) : boolean - Weaponry
hashCode() : int - Object
hasLoadedMeleeWeapon() : boolean - Weaponry
hasLoadedRangedWeapon() : boolean - Weaponry
hasLoadedWeapon() : boolean - Weaponry
hasLoadedWeapon(ItemType weapon) : boolean - Weaponry
hasLowAmmoForWeapon(ItemType weaponType, double lowRatio) : boolean - Weaponry
hasPrimaryLoadedWeapon(ItemType weapon) : boolean - Weaponry
hasPrimaryLowAmmoForWeapon(ItemType weaponType, double lowRatio) : boolean - Weaponry
hasPrimaryWeaponAmmo(ItemType weaponType) : boolean - Weaponry
hasSecondaryAmmoType(ItemType weaponType) : boolean - Weaponry
hasSecondaryLoadedWeapon(ItemType weapon) : boolean - Weaponry
hasSecondaryLowAmmoForWeapon(ItemType weaponType, double lowRatio) : boolean - Weaponry
hasSecondaryWeaponAmmo(ItemType weaponType) : boolean - Weaponry
hasWeapon(ItemType weaponType) : boolean - Weaponry
hasWeapon(Group weaponGroup) : boolean - Weaponry
hasWeaponAmmo(ItemType weaponType) : boolean - Weaponry
isLoaded(ItemType weaponType) : boolean - Weaponry
isLoaded(Group weaponGroup) : boolean - Weaponry
isRunning() : boolean - AgentModule
notify() : void - Object
notifyAll() : void - Object
toString() : String - AgentModule
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.4: Weaponry class methods

- Pathplanner

Thanks to this class it is possible to compute the path between two positions in the map. It uses a path grid that connects all the navigation points in the map.

These are the methods of this class. In this case only the computePath() method is important:



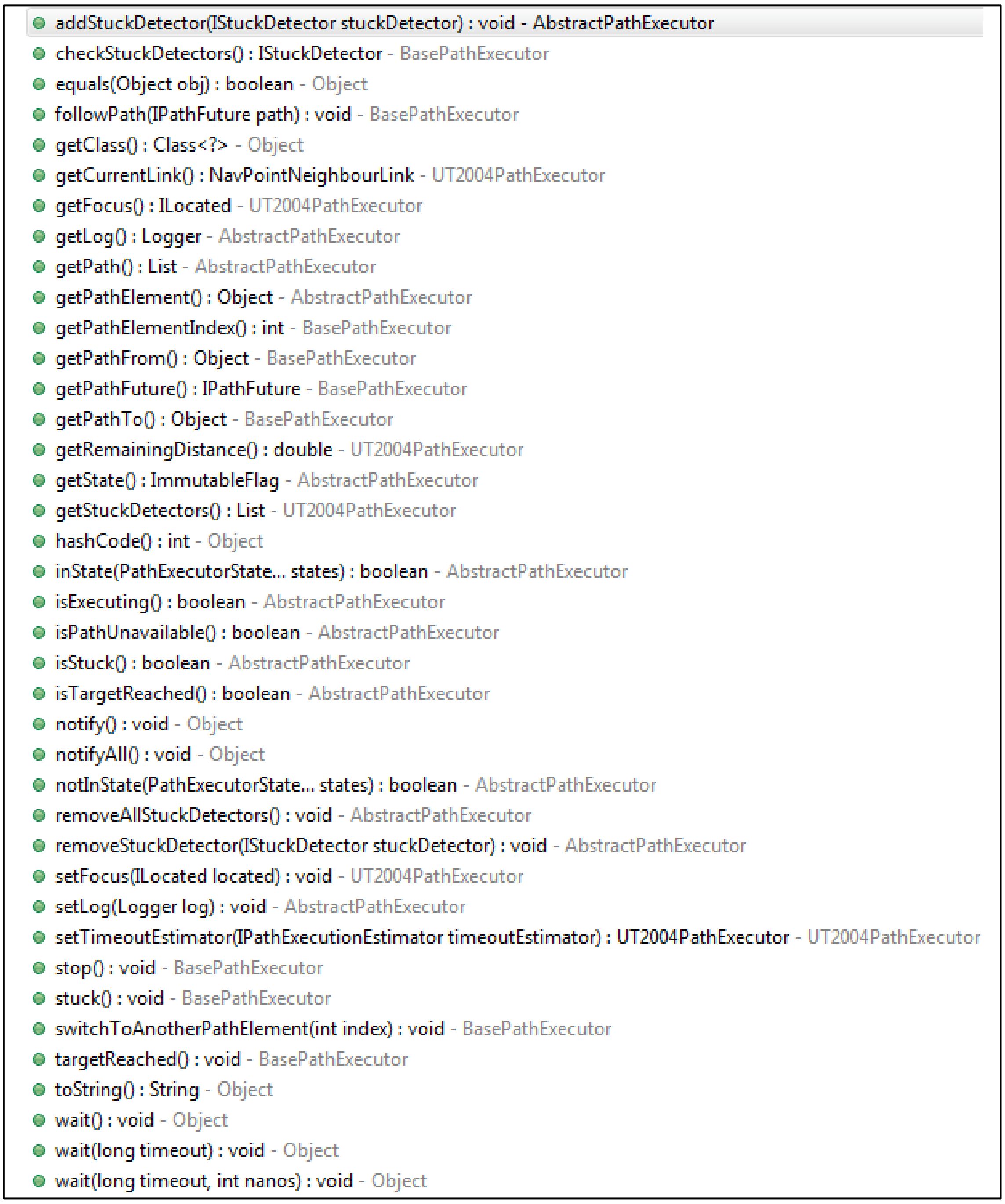
```
computePath(Object from, Object to) : IPathFuture - IPathPlanner
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.5: Pathplanner class methods

- Pathexecutor

This class is the one which has to execute the path that the pathPlanner has computed. It allows also to access nodes that are on the way of the path previously computed. It also gives the possibility of knowing if the bot has been stuck somewhere.

These are the methods for this class:



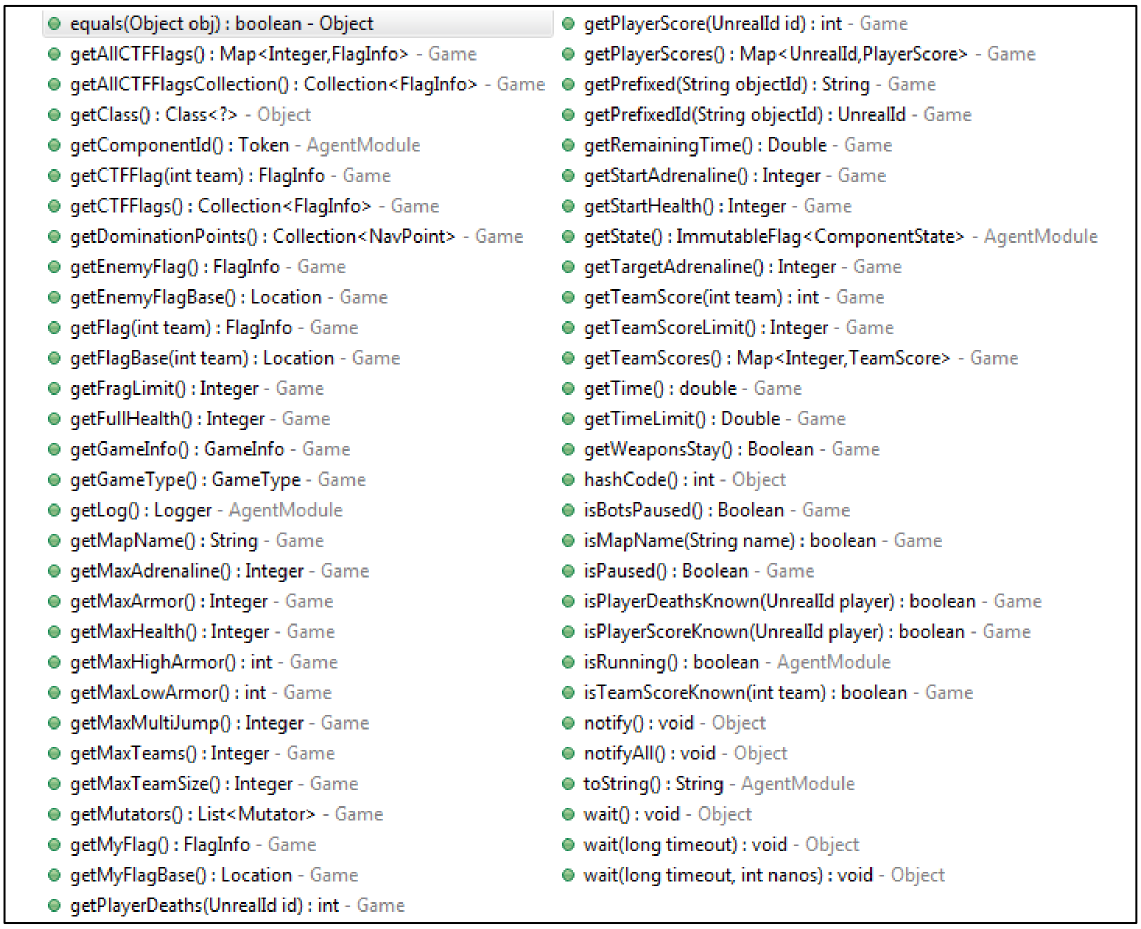
```
addStuckDetector(IStuckDetector stuckDetector) : void - AbstractPathExecutor
checkStuckDetectors() : IStuckDetector - BasePathExecutor
equals(Object obj) : boolean - Object
followPath(IPathFuture path) : void - BasePathExecutor
getClass() : Class<?> - Object
getCurrentLink() : NavPointNeighbourLink - UT2004PathExecutor
getFocus() : ILocated - UT2004PathExecutor
getLog() : Logger - AbstractPathExecutor
getPath() : List - AbstractPathExecutor
getPathElement() : Object - AbstractPathExecutor
getPathElementIndex() : int - BasePathExecutor
getPathFrom() : Object - BasePathExecutor
getPathFuture() : IPathFuture - BasePathExecutor
getPathTo() : Object - BasePathExecutor
getRemainingDistance() : double - UT2004PathExecutor
getState() : ImmutableFlag - AbstractPathExecutor
getStuckDetectors() : List - UT2004PathExecutor
hashCode() : int - Object
inState(PathExecutorState... states) : boolean - AbstractPathExecutor
isExecuting() : boolean - AbstractPathExecutor
isPathUnavailable() : boolean - AbstractPathExecutor
isStuck() : boolean - AbstractPathExecutor
isTargetReached() : boolean - AbstractPathExecutor
notify() : void - Object
notifyAll() : void - Object
notInState(PathExecutorState... states) : boolean - AbstractPathExecutor
removeAllStuckDetectors() : void - AbstractPathExecutor
removeStuckDetector(IStuckDetector stuckDetector) : void - AbstractPathExecutor
setFocus(ILocated located) : void - UT2004PathExecutor
setLog(Logger log) : void - AbstractPathExecutor
setTimeoutEstimator(IPathExecutionEstimator timeoutEstimator) : UT2004PathExecutor - UT2004PathExecutor
stop() : void - BasePathExecutor
stuck() : void - BasePathExecutor
switchToAnotherPathElement(int index) : void - BasePathExecutor
targetReached() : void - BasePathExecutor
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.6: Pathexecutor class methods

- Game

This module saves all the information about the existing game such as the name of the map, the type of game, maximum health or armour level, etc.

These are the usable methods of this class:



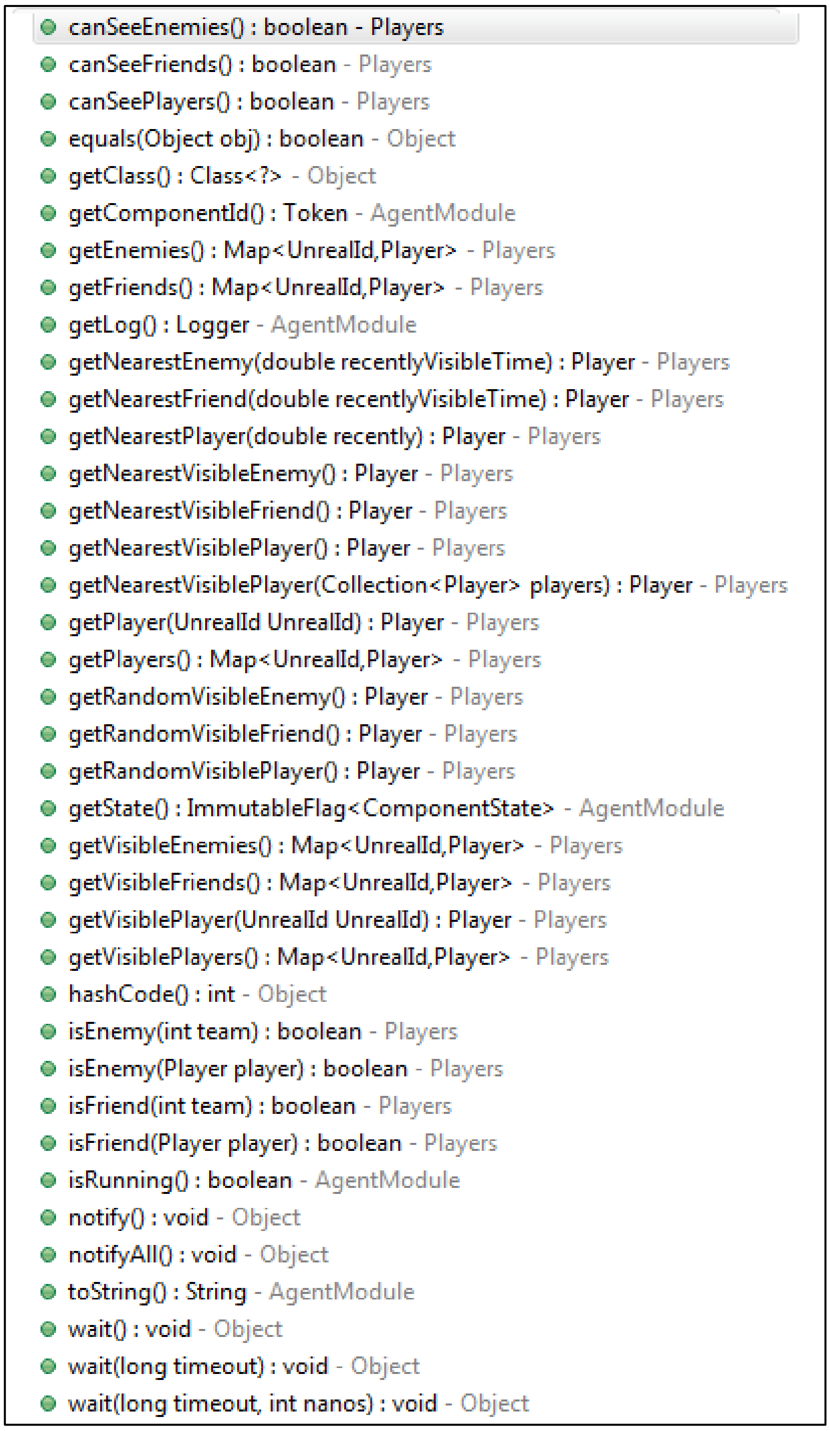
- equals(Object obj) : boolean - Object
- getAllCTFFlags() : Map<Integer,FlagInfo> - Game
- getAllCTFFlagsCollection() : Collection<FlagInfo> - Game
- getClass() : Class<?> - Object
- getComponentId() : Token - AgentModule
- getCTFFlag(int team) : FlagInfo - Game
- getCTFFlags() : Collection<FlagInfo> - Game
- getDominationPoints() : Collection<NavPoint> - Game
- getEnemyFlag() : FlagInfo - Game
- getEnemyFlagBase() : Location - Game
- getFlag(int team) : FlagInfo - Game
- getFlagBase(int team) : Location - Game
- getFragLimit() : Integer - Game
- getFullHealth() : Integer - Game
- getGameInfo() : GameInfo - Game
- getGameType() : GameType - Game
- getLog() : Logger - AgentModule
- getMapName() : String - Game
- getMaxAdrenaline() : Integer - Game
- getMaxArmor() : Integer - Game
- getMaxHealth() : Integer - Game
- getMaxHighArmor() : int - Game
- getMaxLowArmor() : int - Game
- getMaxMultiJump() : Integer - Game
- getMaxTeams() : Integer - Game
- getMaxTeamSize() : Integer - Game
- getMutators() : List<Mutator> - Game
- getMyFlag() : FlagInfo - Game
- getMyFlagBase() : Location - Game
- getPlayerDeaths(UnrealId id) : int - Game
- getPlayerScore(UnrealId id) : int - Game
- getPlayerScores() : Map<UnrealId,PlayerScore> - Game
- getPrefixed(String objectId) : String - Game
- getPrefixedId(String objectId) : UnrealId - Game
- getRemainingTime() : Double - Game
- getStartAdrenaline() : Integer - Game
- getStartHealth() : Integer - Game
- getState() : ImmutableFlag<ComponentState> - AgentModule
- getTargetAdrenaline() : Integer - Game
- getTeamScore(int team) : int - Game
- getTeamScoreLimit() : Integer - Game
- getTeamScores() : Map<Integer,TeamScore> - Game
- getTime() : double - Game
- getTimeLimit() : Double - Game
- getWeaponsStay() : Boolean - Game
- hashCode() : int - Object
- isBotsPaused() : Boolean - Game
- isMapName(String name) : boolean - Game
- isPaused() : Boolean - Game
- isPlayerDeathsKnown(UnrealId player) : boolean - Game
- isPlayerScoreKnown(UnrealId player) : boolean - Game
- isRunning() : boolean - AgentModule
- isTeamScoreKnown(int team) : boolean - Game
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - AgentModule
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Figure B.7: Game class methods

- Players

This class keeps information about the actual players of the game and gives the possibility of knowing which ones are enemies and which ones are not.

These are the methods of this class:



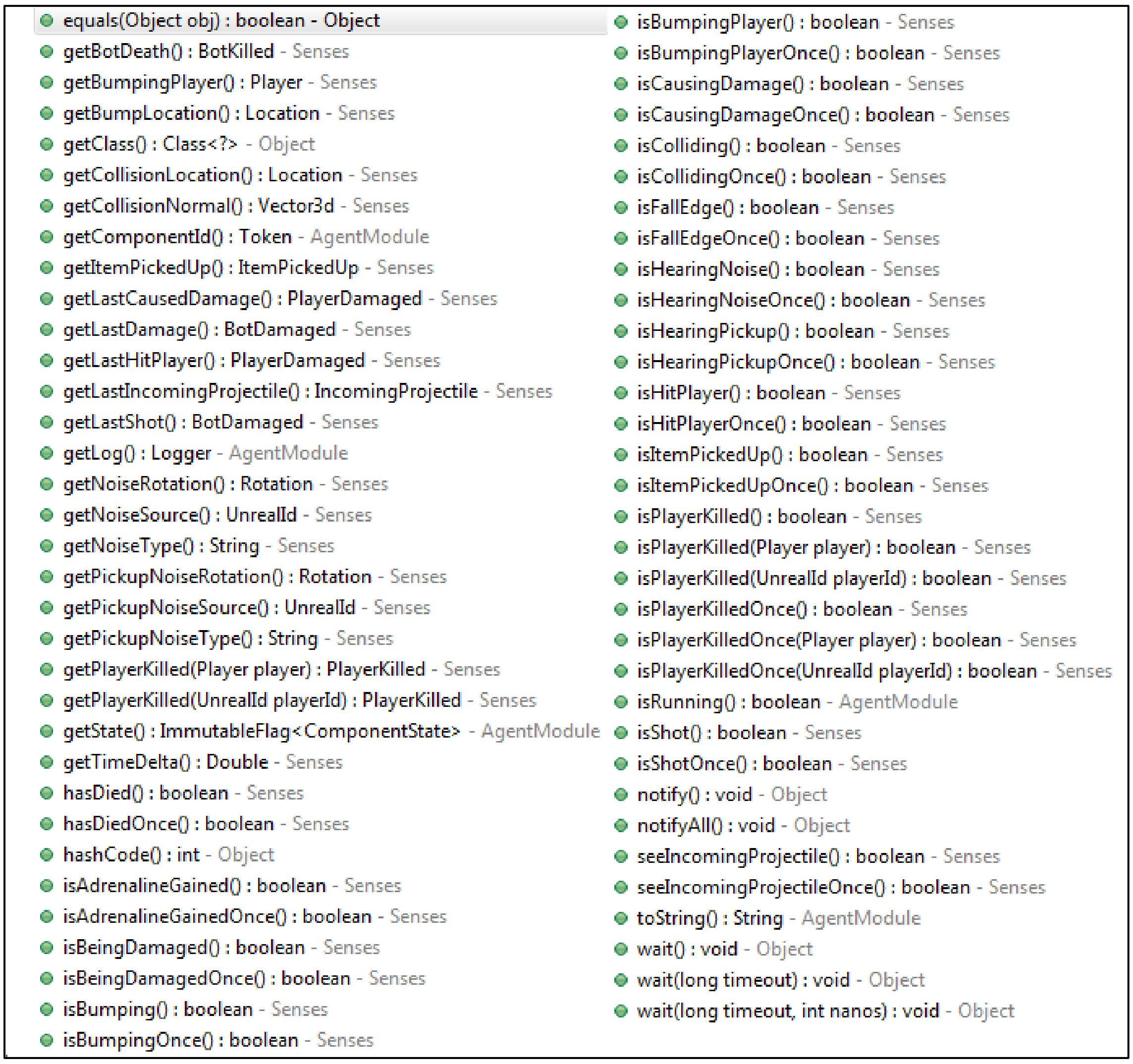
```
canSeeEnemies() : boolean - Players
canSeeFriends() : boolean - Players
canSeePlayers() : boolean - Players
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getComponentId() : Token - AgentModule
getEnemies() : Map<UnrealId,Player> - Players
getFriends() : Map<UnrealId,Player> - Players
getLog() : Logger - AgentModule
getNearestEnemy(double recentlyVisibleTime) : Player - Players
getNearestFriend(double recentlyVisibleTime) : Player - Players
getNearestPlayer(double recently) : Player - Players
getNearestVisibleEnemy() : Player - Players
getNearestVisibleFriend() : Player - Players
getNearestVisiblePlayer() : Player - Players
getNearestVisiblePlayer(Collection<Player> players) : Player - Players
getPlayer(UnrealId UnrealId) : Player - Players
getPlayers() : Map<UnrealId,Player> - Players
getRandomVisibleEnemy() : Player - Players
getRandomVisibleFriend() : Player - Players
getRandomVisiblePlayer() : Player - Players
getState() : ImmutableFlag<ComponentState> - AgentModule
getVisibleEnemies() : Map<UnrealId,Player> - Players
getVisibleFriends() : Map<UnrealId,Player> - Players
getVisiblePlayer(UnrealId UnrealId) : Player - Players
getVisiblePlayers() : Map<UnrealId,Player> - Players
hashCode() : int - Object
isEnemy(int team) : boolean - Players
isEnemy(Player player) : boolean - Players
isFriend(int team) : boolean - Players
isFriend(Player player) : boolean - Players
isRunning() : boolean - AgentModule
notify() : void - Object
notifyAll() : void - Object
toString() : String - AgentModule
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.8: Player class methods

- <u>Senses</u>

This class collects information about the sense of the bot. It is possible to detect for example if the bots has heard a noise or if it is being damaged or the way that the bot died among other possibilities.

These are the available methods of this class:



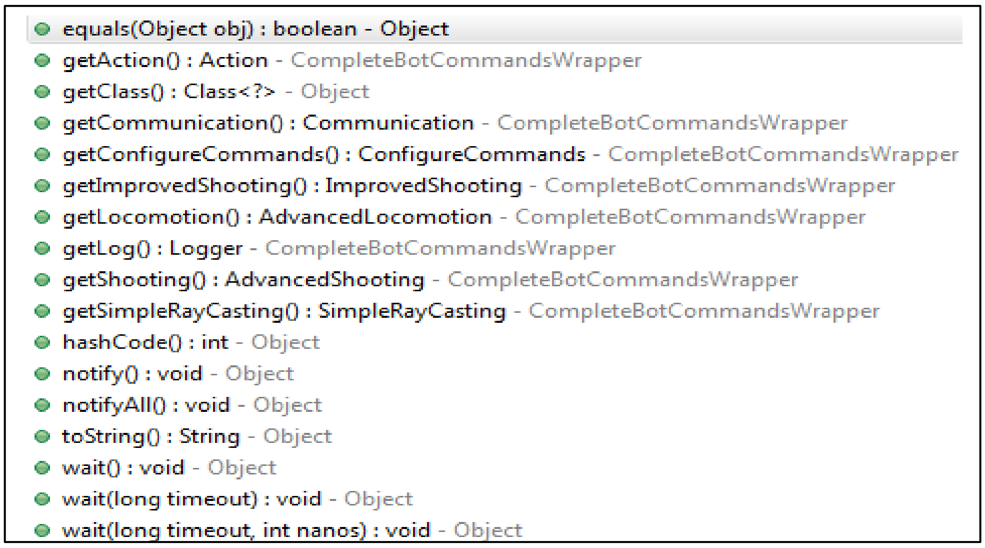
```
equals(Object obj) : boolean - Object
getBotDeath() : BotKilled - Senses
getBumpingPlayer() : Player - Senses
getBumpLocation() : Location - Senses
getClass() : Class<?> - Object
getCollisionLocation() : Location - Senses
getCollisionNormal() : Vector3d - Senses
getComponentId() : Token - AgentModule
getItemPickedUp() : ItemPickedUp - Senses
getLastCausedDamage() : PlayerDamaged - Senses
getLastDamage() : BotDamaged - Senses
getLastHitPlayer() : PlayerDamaged - Senses
getLastIncomingProjectile() : IncomingProjectile - Senses
getLastShot() : BotDamaged - Senses
getLog() : Logger - AgentModule
getNoiseRotation() : Rotation - Senses
getNoiseSource() : UnrealId - Senses
getNoiseType() : String - Senses
getPickupNoiseRotation() : Rotation - Senses
getPickupNoiseSource() : UnrealId - Senses
getPickupNoiseType() : String - Senses
getPlayerKilled(Player player) : PlayerKilled - Senses
getPlayerKilled(UnrealId playerId) : PlayerKilled - Senses
getState() : ImmutableFlag<ComponentState> - AgentModule
getTimeDelta() : Double - Senses
hasDied() : boolean - Senses
hasDiedOnce() : boolean - Senses
hashCode() : int - Object
isAdrenalineGained() : boolean - Senses
isAdrenalineGainedOnce() : boolean - Senses
isBeingDamaged() : boolean - Senses
isBeingDamagedOnce() : boolean - Senses
isBumping() : boolean - Senses
isBumpingOnce() : boolean - Senses
isBumpingPlayer() : boolean - Senses
isBumpingPlayerOnce() : boolean - Senses
isCausingDamage() : boolean - Senses
isCausingDamageOnce() : boolean - Senses
isColliding() : boolean - Senses
isCollidingOnce() : boolean - Senses
isFallEdge() : boolean - Senses
isFallEdgeOnce() : boolean - Senses
isHearingNoise() : boolean - Senses
isHearingNoiseOnce() : boolean - Senses
isHearingPickup() : boolean - Senses
isHearingPickupOnce() : boolean - Senses
isHitPlayer() : boolean - Senses
isHitPlayerOnce() : boolean - Senses
isItemPickedUp() : boolean - Senses
isItemPickedUpOnce() : boolean - Senses
isPlayerKilled() : boolean - Senses
isPlayerKilled(Player player) : boolean - Senses
isPlayerKilled(UnrealId playerId) : boolean - Senses
isPlayerKilledOnce() : boolean - Senses
isPlayerKilledOnce(Player player) : boolean - Senses
isPlayerKilledOnce(UnrealId playerId) : boolean - Senses
isRunning() : boolean - AgentModule
isShot() : boolean - Senses
isShotOnce() : boolean - Senses
notify() : void - Object
notifyAll() : void - Object
seeIncomingProjectile() : boolean - Senses
seeIncomingProjectileOnce() : boolean - Senses
toString() : String - AgentModule
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.9: Senses class methods

- <u>Body</u>

This class includes all the commands such as Action, AdvancedLocomotion, AdvancedShooting, Communication and SimpleRayCasting. So this would be the most important class in order to adapt the behaviour of the bot depending on the outputs of the ARS decision unit.

The available methods in this class are:

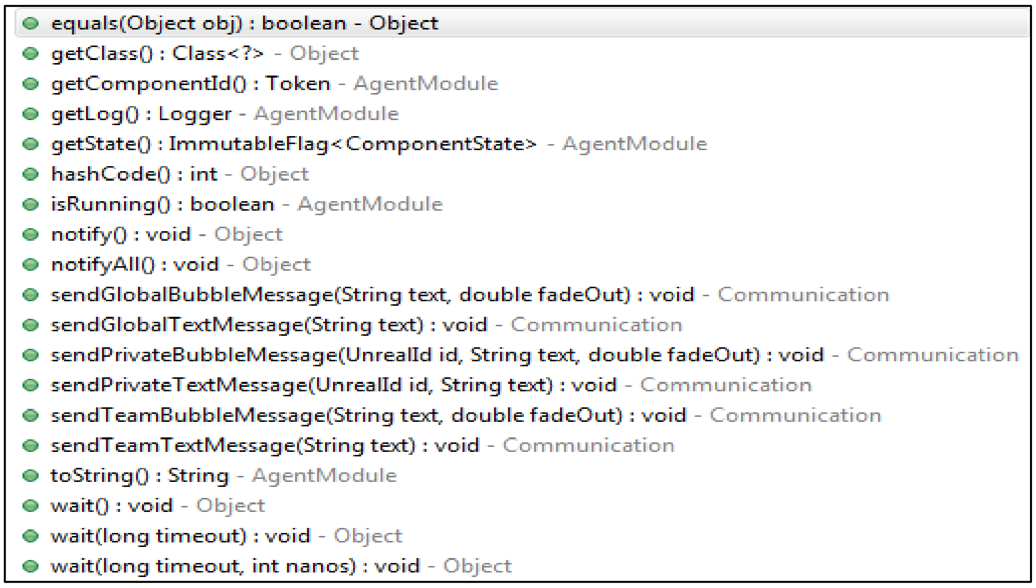


```
equals(Object obj) : boolean - Object
getAction() : Action - CompleteBotCommandsWrapper
getClass() : Class<?> - Object
getCommunication() : Communication - CompleteBotCommandsWrapper
getConfigureCommands() : ConfigureCommands - CompleteBotCommandsWrapper
getImprovedShooting() : ImprovedShooting - CompleteBotCommandsWrapper
getLocomotion() : AdvancedLocomotion - CompleteBotCommandsWrapper
getLog() : Logger - CompleteBotCommandsWrapper
getShooting() : AdvancedShooting - CompleteBotCommandsWrapper
getSimpleRayCasting() : SimpleRayCasting - CompleteBotCommandsWrapper
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - Object
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.10: Body class methods

It is interesting now to see in more detail the possibilities that offer some of these methods:

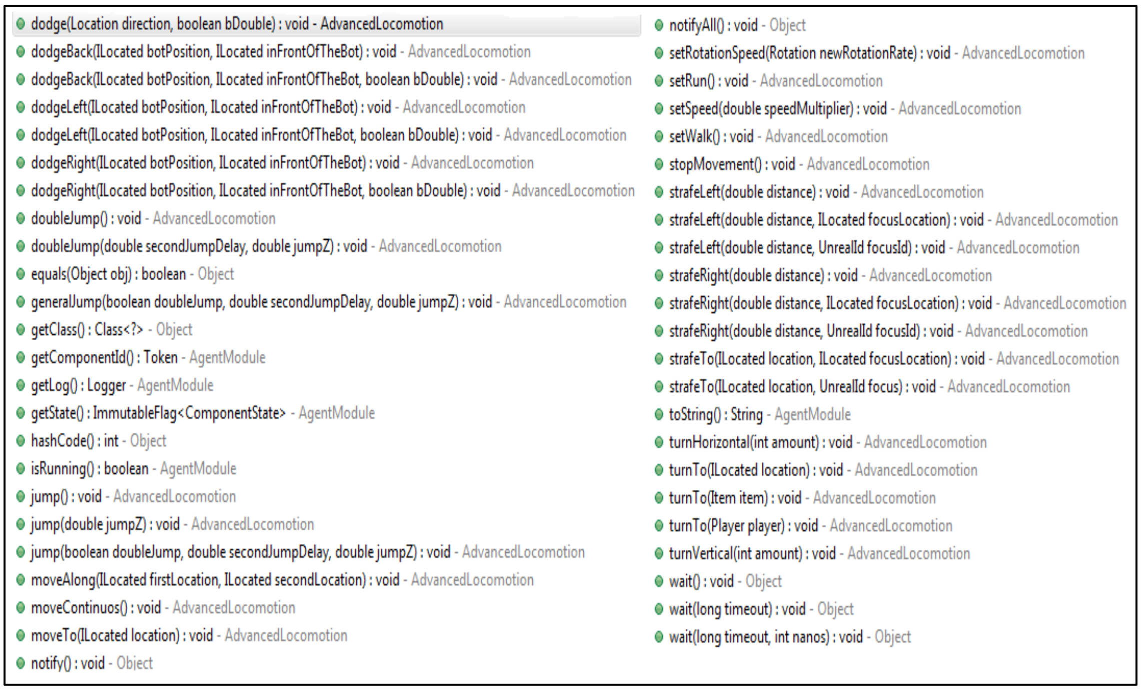
- getCommunication()



```
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getComponentId() : Token - AgentModule
getLog() : Logger - AgentModule
getState() : ImmutableFlag<ComponentState> - AgentModule
hashCode() : int - Object
isRunning() : boolean - AgentModule
notify() : void - Object
notifyAll() : void - Object
sendGlobalBubbleMessage(String text, double fadeOut) : void - Communication
sendGlobalTextMessage(String text) : void - Communication
sendPrivateBubbleMessage(UnrealId id, String text, double fadeOut) : void - Communication
sendPrivateTextMessage(UnrealId id, String text) : void - Communication
sendTeamBubbleMessage(String text, double fadeOut) : void - Communication
sendTeamTextMessage(String text) : void - Communication
toString() : String - AgentModule
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.11: getCommunication methods

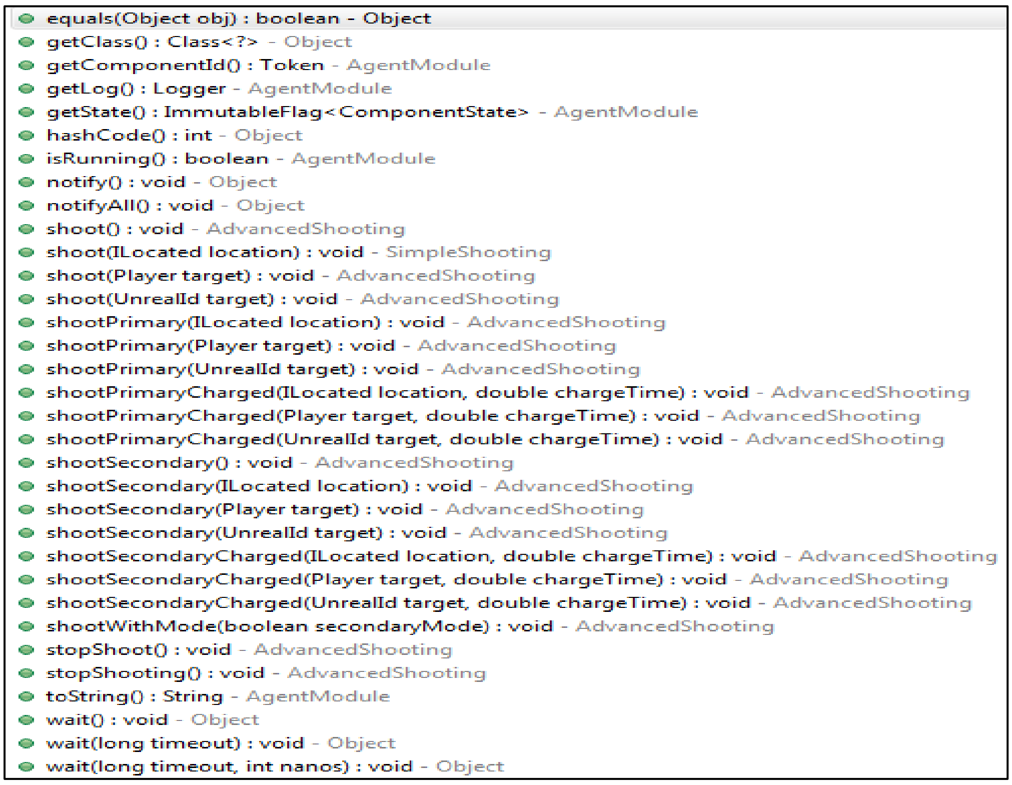
- getLocomotion()



```
dodge(Location direction, boolean bDouble) : void - AdvancedLocomotion
dodgeBack(ILocated botPosition, ILocated inFrontOfTheBot) : void - AdvancedLocomotion
dodgeBack(ILocated botPosition, ILocated inFrontOfTheBot, boolean bDouble) : void - AdvancedLocomotion
dodgeLeft(ILocated botPosition, ILocated inFrontOfTheBot) : void - AdvancedLocomotion
dodgeLeft(ILocated botPosition, ILocated inFrontOfTheBot, boolean bDouble) : void - AdvancedLocomotion
dodgeRight(ILocated botPosition, ILocated inFrontOfTheBot) : void - AdvancedLocomotion
dodgeRight(ILocated botPosition, ILocated inFrontOfTheBot, boolean bDouble) : void - AdvancedLocomotion
doubleJump() : void - AdvancedLocomotion
doubleJump(double secondJumpDelay, double jumpZ) : void - AdvancedLocomotion
equals(Object obj) : boolean - Object
generalJump(boolean doubleJump, double secondJumpDelay, double jumpZ) : void - AdvancedLocomotion
getClass() : Class<?> - Object
getComponentId() : Token - AgentModule
getLog() : Logger - AgentModule
getState() : ImmutableFlag<ComponentState> - AgentModule
hashCode() : int - Object
isRunning() : boolean - AgentModule
jump() : void - AdvancedLocomotion
jump(double jumpZ) : void - AdvancedLocomotion
jump(boolean doubleJump, double secondJumpDelay, double jumpZ) : void - AdvancedLocomotion
moveAlong(ILocated firstLocation, ILocated secondLocation) : void - AdvancedLocomotion
moveContinuos() : void - AdvancedLocomotion
moveTo(ILocated location) : void - AdvancedLocomotion
notify() : void - Object
notifyAll() : void - Object
setRotationSpeed(Rotation newRotationRate) : void - AdvancedLocomotion
setRun() : void - AdvancedLocomotion
setSpeed(double speedMultiplier) : void - AdvancedLocomotion
setWalk() : void - AdvancedLocomotion
stopMovement() : void - AdvancedLocomotion
strafeLeft(double distance) : void - AdvancedLocomotion
strafeLeft(double distance, ILocated focusLocation) : void - AdvancedLocomotion
strafeLeft(double distance, UnrealId focusId) : void - AdvancedLocomotion
strafeRight(double distance) : void - AdvancedLocomotion
strafeRight(double distance, ILocated focusLocation) : void - AdvancedLocomotion
strafeRight(double distance, UnrealId focusId) : void - AdvancedLocomotion
strafeTo(ILocated location, ILocated focusLocation) : void - AdvancedLocomotion
strafeTo(ILocated location, UnrealId focus) : void - AdvancedLocomotion
toString() : String - AgentModule
turnHorizontal(int amount) : void - AdvancedLocomotion
turnTo(ILocated location) : void - AdvancedLocomotion
turnTo(Item item) : void - AdvancedLocomotion
turnTo(Player player) : void - AdvancedLocomotion
turnVertical(int amount) : void - AdvancedLocomotion
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.12: getLocomotion methods

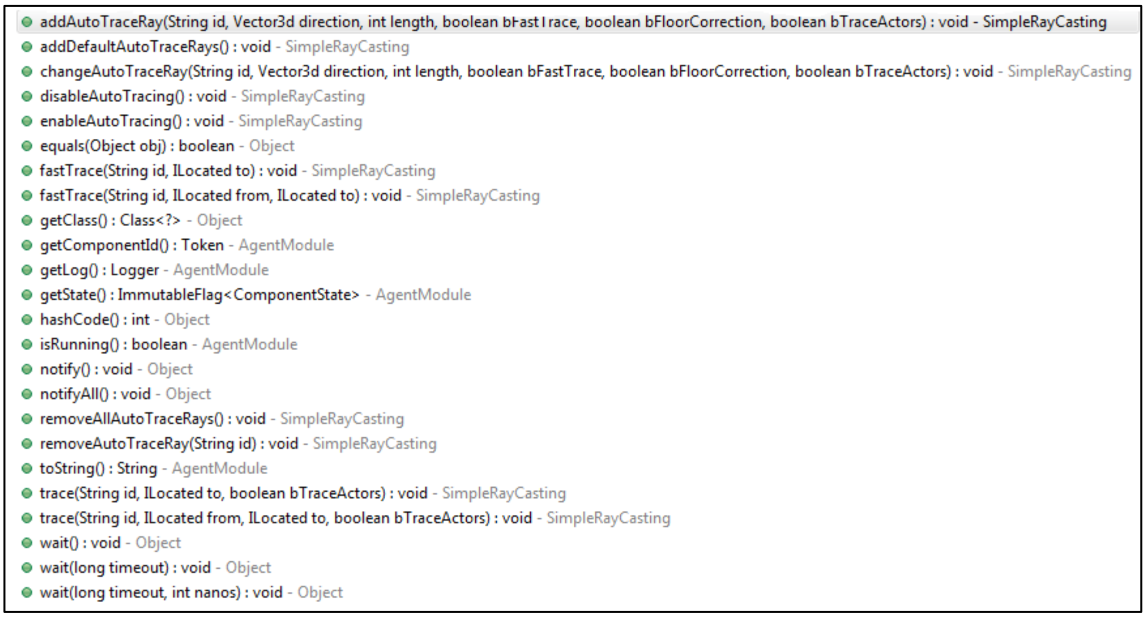
- getShooting()



```
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getComponentId() : Token - AgentModule
getLog() : Logger - AgentModule
getState() : ImmutableFlag<ComponentState> - AgentModule
hashCode() : int - Object
isRunning() : boolean - AgentModule
notify() : void - Object
notifyAll() : void - Object
shoot() : void - AdvancedShooting
shoot(ILocated location) : void - SimpleShooting
shoot(Player target) : void - AdvancedShooting
shoot(UnrealId target) : void - AdvancedShooting
shootPrimary(ILocated location) : void - AdvancedShooting
shootPrimary(Player target) : void - AdvancedShooting
shootPrimary(UnrealId target) : void - AdvancedShooting
shootPrimaryCharged(ILocated location, double chargeTime) : void - AdvancedShooting
shootPrimaryCharged(Player target, double chargeTime) : void - AdvancedShooting
shootPrimaryCharged(UnrealId target, double chargeTime) : void - AdvancedShooting
shootSecondary() : void - AdvancedShooting
shootSecondary(ILocated location) : void - AdvancedShooting
shootSecondary(Player target) : void - AdvancedShooting
shootSecondary(UnrealId target) : void - AdvancedShooting
shootSecondaryCharged(ILocated location, double chargeTime) : void - AdvancedShooting
shootSecondaryCharged(Player target, double chargeTime) : void - AdvancedShooting
shootSecondaryCharged(UnrealId target, double chargeTime) : void - AdvancedShooting
shootWithMode(boolean secondaryMode) : void - AdvancedShooting
stopShoot() : void - AdvancedShooting
stopShooting() : void - AdvancedShooting
toString() : String - AgentModule
wait() : void - Object
wait(long timeout) : void - Object
wait(long timeout, int nanos) : void - Object
```

Figure B.13: getShooting methods

- getSimpleRayCasting()



```
● addAutoTraceRay(String id, Vector3d direction, int length, boolean bFastTrace, boolean bFloorCorrection, boolean bTraceActors) : void - SimpleRayCasting
● addDefaultAutoTraceRays() : void - SimpleRayCasting
● changeAutoTraceRay(String id, Vector3d direction, int length, boolean bFastTrace, boolean bFloorCorrection, boolean bTraceActors) : void - SimpleRayCasting
● disableAutoTracing() : void - SimpleRayCasting
● enableAutoTracing() : void - SimpleRayCasting
● equals(Object obj) : boolean - Object
● fastTrace(String id, ILocated to) : void - SimpleRayCasting
● fastTrace(String id, ILocated from, ILocated to) : void - SimpleRayCasting
● getClass() : Class<?> - Object
● getComponentId() : Token - AgentModule
● getLog() : Logger - AgentModule
● getState() : ImmutableFlag<ComponentState> - AgentModule
● hashCode() : int - Object
● isRunning() : boolean - AgentModule
● notify() : void - Object
● notifyAll() : void - Object
● removeAllAutoTraceRays() : void - SimpleRayCasting
● removeAutoTraceRay(String id) : void - SimpleRayCasting
● toString() : String - AgentModule
● trace(String id, ILocated to, boolean bTraceActors) : void - SimpleRayCasting
● trace(String id, ILocated from, ILocated to, boolean bTraceActors) : void - SimpleRayCasting
● wait() : void - Object
● wait(long timeout) : void - Object
● wait(long timeout, int nanos) : void - Object
```

Figure B.14: getSimpleRayCasting methods

C. ALMA

ALMA states for A Layered Model of Affect [Geb05] and it is a Java Engine with GUI that can provide virtual agents with affects. It is based on the OCC cognitive model of emotions [Ort88] which is briefly described below. ALMA divides the affects into three different kinds depending on its duration: emotions as short-term affect, mood as medium-term affect and personality as long-term affect. The three of them are explained in more detail later in the ALMA outputs section.

OCC theory

It is a cognitive theory of emotions designed by Andrew Ortony, Gerald L. Clore and Allan Collins in 1988 [Ort88] with the purpose of implementing it in the computer systems. According to the authors the emotions are valenced[14] affective reactions whose purpose is to evaluate events that happen in the world. Three groups of emotions exist in the OCC model: emotions as reactions to events, emotions as reactions to agents and emotions as reactions to objects. A set of emotional variables are used to evaluate these events. These variables are later described in the ALMA inputs section.

The emotions in the OCC model are generated following this process: An event occurs in the environment which is evaluated by the OCC variables and those variables are associated with one of the OCC emotions (event, agent or object emotions).

ALMA inputs

ALMA's inputs are provided through emotion eliciting conditions (EECs), which are a subset of OCC theory variables. These are the EECs variables:

- **Desirability:** it measures the success of an event (pleasant or unpleasant) and affecting emotions attributed to events.
- **Praiseworthiness:** it measures the attitude towards the actions of the different agents (approve or disapprove the action) and affecting emotions attributed to agents.
- **Appealingness:** it measures the own agent attitude toward objects (like or dislike the object) and affecting emotions attributed to objects.
- **Likelihood:** it measures how likely an event will happen.
- **Liking:** it measures how much the agent is attracted to another one.
- **Realization:** it measures how much an event that occurred ended.
- **Agency:** it can be set to "self" or "other" giving the information about if the event or action is caused by itself or by another agent.
- **Elicitor:** it determines who caused the event or action.

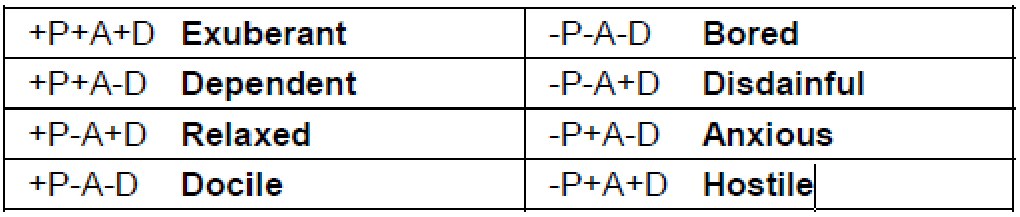
[14] Valence, as used in psychology, especially in discussing emotions, means the intrinsic attractiveness (positive valence) or aversiveness (negative valence) of an event, object, or situation

ALMA outputs

As explained before ALMA has three different affect types which are: emotions as short-term affect, mood as medium-term affect and personality as long-term affect. The three of them are here described in more detail:

- **Emotions:** They are a short-term affect. They are related to a specific event, action or object which causes the emotion. Emotions decay over the time until they completely disappear of the agents focus. The agent can have multiple emotions at the same time but the one with highest intensity is the one that is in its focus. Different emotions exist: admiration, anger, disliking, disappointment, distress, fear, fearsConfirmed, gloating, gratification, gratitude, happyFor, hate, hope, joy, liking, love, pity, pride, remorse, reproach, resentment, satisfaction and shame.

- **Mood:** It is a medium-term affect which is not generally related with a certain event, action or object. It is represented by three independent dimensions: pleasure (P), arousal (A) and dominance (D). According to the level of each dimension the 8 moods of the table in Figure C.1 are possible.



+P+A+D	**Exuberant**	-P-A-D	**Bored**
+P+A-D	**Dependent**	-P-A+D	**Disdainful**
+P-A+D	**Relaxed**	-P+A-D	**Anxious**
+P-A-D	**Docile**	-P+A+D	**Hostile**

Figure C.1: Mood's table [Geb05, p.3]

- **Personality:** It is a long-term affect. The user sets it up for each agent before the ALMA starts and later it is not possible to change it in real-time. It has to do with the emotions intensity and also with the agent’s default mood. The five dimensions of the personality model are:
    - **Openness:** reflects interest in intellectual issues, unconventional values, aesthetic sensitivity or need for variety.
    - **Conscientiousness:** reflects task-oriented characteristics such as being dependable, responsible and orderly.
    - **Extraversion:** reflects a tendency to be sociable and experience positive affect.
    - **Agreeableness:** reflects a tendency to be inter-personally pleasant and compliant
    - **Neuroticism:** reflects a tendency to experience anxiety and other negative emotions.

D. Izhikevich Model

The Izhikevich model [Izh03] is governed by these two differential equations:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I$$

$$\frac{du}{dt} = a(bv - u)$$

where **v** is the membrane potential, **u** is a recovery variable, **I** is the dendritic current, and **a** and **b** are abstract parameters of the model.

When the voltage exceeds a threshold value (preset at 30 mV) then **v** and **u** change values according to the following:

$$u \ \rightarrow c$$

$$u \ \rightarrow u + d$$

The parameter **a** describes the time scale of the recovery variable u. Smaller values result in slower recovery. A typical value is a = 0.02

The parameter **b** describes the sensitivity of the recovery variable **u** to the sub-threshold fluctuations of the membrane potential **v**. Greater values couple **v** and **u** more strongly resulting in possible sub-threshold oscillations and low-threshold spiking dynamics. A typical value is b = 0.2

The parameter **c** describes the after-spike reset value of the membrane potential **v** caused by the fast high-threshold K^+ conductance. A typical value is c = 65 mV.

Literature

[Atal98] Georgeff, Michael; Pell, Barney; Pollack, Martha; Tambe Milind; Wooldridge, Michael: The Belief-Desire-Intention Model of Agency, ATAL community, 1998.

[Baa88] Baars, Bernard J.: A cognitive theory of consciousness, Cambridge University Press, 1988.

[Baa97] Baars, Bernard J.: In the theater of consciousness. Global Workspace Theory, 1997.

[Beh09] Behrens, Tristan M.; Dix, Jürgen; Hindriks, Koen V.: Towards an Environment Interface Standard for Agent-oriented Programming, Germany, 2009

[Bid09] Bída, Michal: Artificial Emotions in computer games (Emohawk project), Charles University, Prague, 2009.

[Bra87] Bratman, Michael E.: Intention, Plans and Practical Reason, CSLI Publications, 1987.

[Bro86] Brooks, Rodney A.; A Robust layered control system for a mobile robot. Massachusetts Institute of Technology. 1986.

[Dam94] Antonio Damasio. Descartes' Error: Emotion, Reason, and the Human Brain. Penguin,1994. Published in Penguin Books 2005.

[Deu11] Deutsch, Tobias: Human Bionically Inspired Autonomous Agents, Vienna 2010.

[Die09] Dietrich, Dietmar; Fodor, Georg; Zucker, Gerhard; Bruckner, Dietmar; and et al.: Simulating the mind – A technical neuropsychoanalytical approach. Ed.Springer, Vienna, 2009.

[Dig09] Dignum, Frank; Bradshaw, Jeff; Silverman, Barry; van Doesburg, Willem: Agents for Games and Simulations, Ed. Springer. 2009.

[Dig11] Dignum, Frank: Agents for Games and Simulations, Ed.Springer. 2011.

[Doe09] Dönz, Benjamin: Actuators for an Artificial Life Simulation, Vienna 2009.

[Fou11a] Fountas, Zafeirios: Spiking Neuronal Networks for Human-like Agent Control in a Simulated Environment, Imperial College of London, 2011.

[Fou11b] Fountas, Zafeirios; Gamez, David; Fidjeland, Andreas K.: A neuronal global workspace for human-like control of a computer game character, Conference on computational intelligence and games, 2011.

[Fre15a] Freud, Sigmund: Instincts and their vicissitudes. The Standard Edition of the Complete Psychological Works of Sigmund Freud, XIV (1914-1916): On the History of the Psycho-Analytic Movement, Papers on Metapsychology and Other Works, 1915.

[Fre15b] Freud, Sigmund: Repression. The Standard Edition of the Complete Psychological Works of Sigmund Freud, 1915.

[Fre33] Freud, Sigmund: New Introductory Lectures On Psycho-Analysis., volume XXII (1932-1936): New Introductory Lectures on Psycho-Analysis and OtherWorks, of The Standard Edition of the Complete Psychological Works of Sigmund Freud. Hogarth Press and Institute of PsychoAnalysis, 1933.

[Fre40] Freud, Sigmund: An outline of psycho-analysis. International Journal of Psycho-Analysis, 1940.

[Fre72] Freud, Sigmund: Trieblehre. *Gesammelte Werke chronologisch geordnet*, 1972.

[Fre91] Freud, Sigmund: *Zur Auffassung der Aphasien*. Fischer Taschenbuch, 1891.

[Geb05] Gebhard, Patrick: ALMA – A Layered Model of Affect, German Research Center for Artificial Intelligence, 2005.

[Gem09] Gemrot, Jakub; Kadlec, Rudolf; Bída, Michal; Burkert, Ondrej; Píbil, Radek; Havlicek, Jan; Zemcak, Lukas; Simlovic Juraj; Vansa, Radim; Stolba, Michal; Plch, Tomas; Brom, Cyril: Pogamut 3 can assist developer in building AI (not only) for their videogames agents, Charles University, Prague, 2009.

[Gos96] Goss, Simon; Murray, Graeme: Artificial Intelligence Applications in Aircraft Systems. DSTO Aeronautical and Maritime Research Laboratory, Australia, 1996.

[Hin10] Hindriks, Koen V.; van Riemsdijk, Birna; Behrens, Tristan; Korstanje, Rien; Nick; Pasman, Wouter; de Rijk, Lennard: Unreal Goal Bots: Connecting agents to complex dynamic environments, 2010.

[Hin12] Hindriks, Koen V.; Korstanje, Rien; Pasman, Wouter; van Riemsdijk, Birna; de Rijk, Lennard: UT-Goal Manual, Delft University of Technology, The Netherlands, 2012.

[Ho03] Ho, Wan Ching; Dautenhahn, Kerstin; Nehaniv, Chrystopher L.: “Comparing Different Control Architectures for Autobiographic Agents in Static Virtual Environments”,2003.

[Ho04] Ho, Wan Ching; Dautenhahn, Kerstin; Nehaniv, Chrystopher L.; Te Beoekhorst, René: “Sharing Memories: An Experimental Investigation with Multiple Autonomous Autobiographic Agents“,2004.

[Hoo09] van Hoorn, Niels; Togelius, Julian; Schmidhuber, Jürgen: Hierarchical Controller Learning in a First-Person Shooter, IDSIA, Lugano, Switzerland, 2009.

[Izh03] Izhikevich, Eugene M.: Simple Model of Spiking Neurons, 2003.

[Jia07] Jiang, Hong; Vidal, Jose M.; Huhns, Michael N.: EBDI: An architecture for Emotional Agents, University of South California, Columbia, 2007.

[Kol93] Kolodner, Janet L.: Case-based Reasoning. San Francisco, 1993

[Lai87] Laird, John E.; Newell, Allen; Rosenbloom, Paul S.: SOAR: An architecture for general intelligence. 1987.

[Lai02] Laird, John E. et al.: A Test Bed for Developing Intelligent Synthetic Characters. University of Michigan, 2002.

[Lai08] Laird, John E.: Extending the SOAR cognitive architecture, University of Michigan, 2008.

[Lan08] Lang, Roland; Zeilinger, Heimo; Deutsch, Tobias; Velik, Rosemarie; Müller, Brit: Perceptive Learning – A psychoanalytical learning framework for autonomous agents, Vienna, 2008.

[Lan10] Lang, Roland: A Decision Unit for Autonomous Agents Based on the Theory of Psychoanalysis, Vienna 2010.

[LKZD10] Lang, Roland; Kohlhauser, Stefan; Zucker, Gerhard; Deutsch, Tobias: Integrating internal performance measures into the decision making process of autonomous agents. Vienna, 2010.

[Mey07] Meyers, David G.: Module 44: The psychoanalytical perspective, Worth Publishers, 2007.

[Noa84] Noam, Gil G.; Hauser, Stuart T.; Santostefano, Sebastiano; Garrison, William; Jacobson, Alan M.; Powers, Sally I.; Mead, Merrill: Ego Development and Pshychopathology, Blackwell Publishers, 1984.

[Olm11] Olmos Lanceta, Javier: Personajes con Razonamiento Basado en Casos para videojuegos en primera persona, Zaragoza 2011.

[Ort88] Ortony, Andrew; Clore, Gerald L.; Collins, Allan: The cognitive structure of emotions, Cambridge University Press, 1988.

[Ort12] Ortuño, Ernest: Game world's implementation of Artificial Recognition System model, Vienna, 2012.

[Rao95] Rao, Anand S.; Georgeff, Michael P.: Formal Models and Decision Procedures for Multi-Agent Systems, Australian Artificial Intelligence Institute, 1995.

[Sch11] Schrum, Jacob; Karpov, Igor V.; Miikkulainen, Risto: Human-like behavior via Neuroevolution of combat behavior and replay of human traces, University of Texas, 2011

[Sch12] Schrum, Jacob; Karpov, Igor V.; Miikkulainen, Risto: Humanlike combat behavior via multiobjective Neuroevolution, University of Texas, 2012.

[Sno06] Snowden, Ruth: Teach yourself Freud, Ed. McGraw-Hill, 2006.

[Tog04] Togelius, Julian: Evolution of a Subsumption Architecture Neurocontroller, Malmö, Sweden, 2004.

[Tul83] Tulving, Endel: Elements of Episodic Memory. Oxford: Clarendon Press, 1983.

[Woo99] Wooldridge, M.: Intelligent agents, in 'Multiagent systems: a modern approach to distributed artificial intelligence', MIT Press, 1999.

[Woo00] Michael J. Wooldridge. Reasoning about Rational Agents (Intelligent Robotics and Autonomous Agents). MIT Press, 2000.

[Zei10] Zeilinger, Heimo: Bionically Inspired Information Representation for Embodied Software Agents, Vienna 2010.

Internet references

[1] Pogamut home page: http://pogamut.cuni.cz/main/tiki-index.php

[2] GOAL agent programming language: http://mmi.tudelft.nl/trac/goal

[3] ARS home page: http://ars.ict.tuwien.ac.at/

[4] Botprize home page: http://www.botprize.org/

[5] Emohawk project: http://amis.mff.cuni.cz/emohawk/doku.php?id=start

[6] ALMA: http://www.dfki.de/~gebhard/alma/index.html

[7] Id, Ego, Superego: http://fdpclimentmaria.blogspot.com.es/2010/12/practica-4-mecanismes-de-defensa.html

[8] Unreal Tournament 2004: http://en.wikipedia.org/wiki/Unreal_Tournament_2004

[9] JavaBot: http://utbot.sourceforge.net/

[10] Epic games: http://epicgames.com

[11] Subsumption architecture: http://mwarnerwu.wordpress.com/research/subsumption-architecture/

[12] Homeostasis definition: http://www.biology-online.org/dictionary/Homeostasis

[13] Antonio Damasio: http://www.usc.edu/programs/neuroscience/faculty/profile.php?fid=27

[14] Janet Kolodner: http://www.ic.gatech.edu/people/janet-kolodner

[15] Sigmund Freud: http://en.wikipedia.org/wiki/Freudism