UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

INEGENIERÍA INFORMÁTICA

**Proyecto fin de carrera**

# A VIRTUAL ENVIRONMENT FOR REASONING WITH SENSORIAL INFORMATION

Author: Marta Moretón Arancón

Supervisor: Carlos Linares López

July 2010

# Contents


**1 Introduction** **10**

**2 State of the art** **13**

2.1 Background . . . . . 14

2.2 Game AI . . . . . 17

2.3 Game genres . . . . . 19

2.4 AI approaches . . . . . 24

2.4.1 Planning . . . . . 24

2.4.2 AI Navigation . . . . . 25

2.4.3 Perception . . . . . 26

2.4.4 Memory . . . . . 27

2.4.5 Decision making techniques . . . . . 28

2.5 Summary . . . . . 34

**3 Objectives** **35**

**4 Development** **37**

4.1 Description of the game . . . . . 38

4.1.1 What type of game? . . . . . 38

4.1.2 Characteristics . . . . . 38

4.1.3 Description . . . . . 39

4.2 Analysis . . . . . 44

4.2.1 Methodology . . . . . 44

4.2.2 Use cases . . . . . 46

4.2.3 Requirements . . . . . 52

4.3 Architectural design . . . . . 60

4.3.1 System overview . . . . . 61

4.3.2 System context . . . . . 66

4.3.3 Decomposition description . . . . . 67
4.3.4 Complete design . . . . . 74
4.4 Detailed design . . . . . 75
4.4.1 Server . . . . . 75
4.4.2 Player . . . . . 77
4.5 Implementation . . . . . 83
4.5.1 Simulation of perceptions . . . . . 83
4.5.2 AI techniques . . . . . 91
4.5.3 Messages . . . . . 101
4.6 Summary . . . . . 105

**5 Experimentation and results 106**
5.1 Experiments . . . . . 107
5.2 Results . . . . . 118

**6 Conclusions and future work 120**
6.1 Conclusions . . . . . 120
6.2 Future work . . . . . 122

**7 Planification and budget 124**
7.1 Planification . . . . . 124
7.1.1 Initial planification . . . . . 125
7.1.2 Real planification . . . . . 126
7.2 Technical equipment . . . . . 127
7.2.1 Hardware . . . . . 127
7.2.2 Software . . . . . 127
7.3 Economical analysis . . . . . 128
7.3.1 Methodology . . . . . 128
7.3.2 Estimated cost . . . . . 128
7.3.3 Real cost . . . . . 130

**A Reference manual 131**
A.1 Installation of work environment . . . . . 131
A.1.1 Java installation . . . . . 131
A.1.2 Erlang installation . . . . . 131
A.2 Setting up the server . . . . . 132
A.2.1 Starting the server . . . . . 132

A.2.2 Configuration file . . . . . 132
A.2.3 Defining a new scenario . . . . . 134
A.3 Playing the game . . . . . 136
A.4 Creation of a new agent . . . . . 139

# List of Figures


2.1 Pacman screenshot . . . . . 14
2.2 Civilization III screenshot . . . . . 19
2.3 Starcraft screenshot . . . . . 20
2.4 Half life 2 screenshot . . . . . 21
2.5 Thief: Deadly shadows screenshot . . . . . 22
2.6 Agent's memory . . . . . 28
2.7 Bayesian networks . . . . . 31
2.8 Production systems cycle . . . . . 33

4.1 Evolutionary prototypes . . . . . 44
4.2 Use cases . . . . . 46
4.3 Game configuration use case . . . . . 46
4.4 Login use case . . . . . 47
4.5 Taking actions use case . . . . . 49
4.6 Updating status use case . . . . . 50
4.7 Client/Server architecture . . . . . 61
4.8 Client/Server communication . . . . . 62
4.9 System context . . . . . 66
4.10 Client/Server decomposition . . . . . 67
4.11 Client subsystem . . . . . 68
4.12 Server subsystem . . . . . 71
4.13 Complete design . . . . . 74
4.14 Login sequence diagram . . . . . 76
4.15 Game sequence diagram . . . . . 76
4.16 GUI player class diagram . . . . . 77
4.17 GUI Player sequence diagram . . . . . 78
4.18 Model player class diagram . . . . . 79

4.19 Charade player class diagram . . . . . 81
4.20 Light propagation . . . . . 84
4.21 Filtering range of vision . . . . . 85
4.22 Bresenham's line algorithm . . . . . 85
4.23 Hierarchy of elements . . . . . 87
4.24 Sound propagation . . . . . 89
4.25 Collision of sounds . . . . . 90
4.26 Masking effect . . . . . 91
4.27 Avoiding walls . . . . . 101

5.1 Single mode - Configuration experiment 1 . . . . . 109
5.2 Single mode - Configuration experiment 2 . . . . . 110
5.3 Single mode - Results of configuration 2 . . . . . 111
5.4 Normal mode - Configuration experiment 1 . . . . . 112
5.5 Normal mode - Configuration experiment 2 . . . . . 114
5.6 Normal mode - Configuration experiment 3 . . . . . 116
5.7 Summary - Results . . . . . 119

7.1 Initial planification diagram . . . . . 126
7.2 Real planification diagram . . . . . 126

A.1 CreateWorld screenshot . . . . . 134
A.2 Screenshot . . . . . 138

# List of Tables


4.1 Use case definition table . . . . . 46
4.2 UC-01 - Load game configuration . . . . . 47
4.3 UC-02 - Create new game . . . . . 47
4.4 UC-03 - Login into the game . . . . . 48
4.5 UC-04 - Create a proxy player . . . . . 48
4.6 UC-05 - Register player . . . . . 48
4.7 UC-06 - Send confirmation of login . . . . . 48
4.8 UC-07 - Send action . . . . . 49
4.9 UC-08 - Insert action in the queue . . . . . 49
4.10 UC-9 - Send acknowledgement of action . . . . . 49
4.11 UC-10 - Execute actions . . . . . 50
4.12 UC-11 - Update status . . . . . 50
4.13 UC-12 - Filter status . . . . . 51
4.14 UC-13 - Send status . . . . . 51
4.15 Requirements definition table . . . . . 52
4.16 FR1 - Client/Server architecture . . . . . 52
4.17 FR2 - AI implementation . . . . . 52
4.18 FR3 - Game configuration . . . . . 53
4.19 FR4 - Updated game state . . . . . 53
4.20 FR5 - Real-Time implementation . . . . . 53
4.21 FR6 - Multiplayer . . . . . 53
4.22 FR7 - Game functionality . . . . . 54
4.23 FR8 - Player's perceptions . . . . . 54
4.24 FR9 - Signal propagation . . . . . 54
4.25 FR10 - Notification of perceptions . . . . . 55
4.26 NFR1 - Processing time . . . . . 55
4.27 NFR2 - No access to unauthorized information . . . . . 55

4.28 NFR3 - Easy to use . . . . . 56
4.29 NFR4 - System language . . . . . 56
4.30 NFR5 - Consistent game state . . . . . 56
4.31 NFR6 - Low failure rate . . . . . 57
4.32 NFR7 - Recovery from errors . . . . . 57
4.33 NFR8 - Game configuration file . . . . . 57
4.34 NFR9 - Platform independent . . . . . 58
4.35 NFR10 - Robust for invalid inputs . . . . . 58
4.36 NFR11 - Easy to add new features . . . . . 58
4.37 NFR12 - Easy to reuse . . . . . 59
4.38 Decomposition analysis . . . . . 67
4.39 Client component . . . . . 68
4.40 GUI Player component . . . . . 69
4.41 AI Player component . . . . . 69
4.42 Model Player component . . . . . 70
4.43 Communication component . . . . . 70
4.44 Server component . . . . . 71
4.45 Login server component . . . . . 72
4.46 Proxy Player component . . . . . 72
4.47 Game server component . . . . . 73
4.48 Configuration component . . . . . 73
4.49 GUI Player decomposition . . . . . 78
4.50 Model player decomposition . . . . . 80
4.51 Charade player decomposition . . . . . 82
4.52 Sound characteristics . . . . . 88

5.1 Server's performance . . . . . 108
5.2 Normal mode (Configuration 1) - Results . . . . . 113
5.3 Normal mode (Configuration 1) - Summary . . . . . 113
5.4 Normal mode (Configuration 2) - Results . . . . . 115
5.5 Normal mode (Configuration 2) - Summary . . . . . 115
5.6 Normal mode (Configuration 3) - Results . . . . . 117
5.7 Normal mode (Configuration 3) - Summary . . . . . 117

7.1 Initial planification . . . . . 125
7.2 Real planification . . . . . 126
7.3 Estimated human resources costs . . . . . 129

7.4 Estimated HW costs . . . . . 129
7.5 Estimated SW costs . . . . . 129
7.6 Estimated total costs . . . . . 130
7.7 Real human resources costs . . . . . 130
7.8 Real total costs . . . . . 130

# Acknowledgments

I would like to take this opportunity to express my gratitude to those people that, in one way or another, have helped me to be where I am.

First of all, I want to thank my family for their support during the course of my studies. Their guidance and advices have helped not only during my academic years, but also for my personal development.

I have to give a special thank you for two people that have been very important for me because maybe withouth them I would not have finished this project. Alberto, it has been a pleasure to share all these years and experiences with you. Thanks for your help, your patience and your understanding over these years. Sergio, thanks for being there when and where I needed you and for believing in me more than myself.

I have to include my most sincere gratefulness to my groups of friends, both "dementes" and "esn people", that maybe they don't know, but they have been a great support for me, specially during the last months. Thanks for being there.

Finally, I would like to thank my tutor Carlos for his time, understanding and confidence at every moment and also to Javier, for his support and assistance along the way. Their collaboration has been essential for the development of this project. It has been a pleasure to have worked with you over the past months.

# Chapter 1

# Introduction

Artificial Intelligence can be defined as *"the creation of computer programs that emulate acting and thinking like a human, as well as acting and thinking rationally"* [15]. Game AI can be seen as a subfield of it, in which the suitability of different AI techniques is studied within computer game environments. AI is a very new concept that was created in the middle of the 50s and its importance has been increased over the years, acquiring a significant role in many disciplines nowadays. Its applicability in computer games is growing exponentially, as they present a very interesting AI research area. Game AI can be used to simulate very realistic environments and therefore, facilitate the study of new AI techniques. They also create new challenges in computer games and offer more fun and entertainment.

Computers can replace people in many areas because of their high-speed processing that makes them more efficient and faster. However, they might present more problems to represent the knowledge they acquire. Human being receives the information of the world through his senses, but, which process does he follow to analyse this data and how could it be simulated by a computer? There are three main steps in a decision making process:

1. **Receive the information**: The world's state is perceived. Total or partial knowledge of the environment can be learnt, depending on each case. This fact will be decisive for the agent's behaviour.

2. **Analysis**: The information that has been obtained is structured and processed. Knowledge representation is very important for the decision making process, but it is not an easy task to do. Information can be incomplete and

the reasoning process has to be able to deal with it.

3. **Choose action**: Select one of the possible actions to perform and execute it.

These three processes constitute the called *decision cycle* and are described as *feel, think and act* [17]. They are continuously executed until the goal is achieved.

Many games nowadays present a static and deterministic environment. However, more realistic games tend to be more dynamic. Therefore, instead of just having to execute a sequence of actions to reach the goals, player must react to changes in the environment.

In summary, an efficient automatic agent needs five characteristics [17]:

- **Reactive**: Agent is able to respond quickly to changes in the environment.
- **Context Specific** : Agent's actions are consistent with past information and past actions.
- **Flexible**: Agent can choose among several actions in order to try to achieve the goal.
- **Realistic**: Agent has a human-like behaviour.
- **Easy to Develop**: There is a simple and easy knowledge representation.

The final goal of this project is the development of an extendible open source game that will allow the study of the AI techniques that can be suitable for this domain. Also, as a result of this project, the implementation of an automatic player able to manage itself in the scenario, trying to achieve its goals, will be presented.

This work has been structured as follows:

First of all, the background and the state of art will be analyzed. It is important to have a clear idea of the existing work related to this topic in order to be able to understand the contributions to the field that can be made. Starting from the analysis of the current situation and the existing solutions, it is important to learn about current tendencies in computer game development. It is also necessary to perform

a study of the requirements of the particular problem we are going to solve in order to check the suitability of solutions that already exist and solve similar problems.

Subsequently, a design of the architecture will be produced, containing the solution that has been chosen in order to solve the above mentioned problem. This approach can be based in one or more techniques previously studied in the state of the art, whether a combination or an adaptation of many of them. This design will define the structure and its components.

After the analysis and description of the solution, the results of its implementation will be shown and evaluate. The performance and feasibility of our approach will be studied from these results.

Finally, some conclusions will be given in order to understand better the solution that has been chosen, the results obtained and the benefits resulting from it.

# Chapter 2

# State of the art

Over the years, the game development industry has been growing exponentially, acquiring more and more importance and creating new ways of entertainment for a more demanding public. As a consequence, different technologies and techniques have been developed in order to allow this progress.

For this reason, interesting results can be obtained from a research study of the different technologies, ideas and solutions that have been developed within this area. In this way, it will be possible to offer a new approach and try to make a contribution into the field.

In this section, an introduction into the topic is given in order to get a closer view of the area in which the project will be developed. First of all, some background will be introduced in order to understand better the actual situation of computer games development and its relation with artificial intelligence. The basic concepts that are needed to understand this area of research will be presented and defined from both, historical and technical, points of view. This will give an idea of the advances in game development industry that have been developed so far.

Furthermore, different approaches will be studied in order to get an overview of the current situation and understand which activities have to be performed to achieve this goal of this project. Typical problems of game AI and existing solutions will be analysed and compared with the problem to be solved here.

## 2.1 Background

It is difficult to determine the origin of videogames, but it could be said that the first videogame that was created was a two-player game, *Nought and crosses*, developed by Alexander S.Douglas in 1952 for an EDSAC computer. In this game, players could compete against the machine, playing the well known game called tic-tac-toe, where they try to get three elements in a row before their opponent. For doing so, the computer made its decisions based on the last player's movement by applying a very simple AI.

Aside from this first try of adding some simple level of intelligence to the game, it was around 1970s when artificial intelligence concepts started to be included and taken into account for the game industry. Before that, most games were made for two human players, so no intelligence was needed. Some well-known examples are: *Tennis for two*, by William Higginbotham in 1957 or *Spacewar!* by Steve Russell in 1961.

Around 70s, single player games made their appearance and although they were simple games, they started needing a minimum level of intelligence in order to be able to create some level of challenge for players. Games such as *Pac-man*, *Qwak* or *Pursuit*, that were released in that decade, presented some basic AI work. They were making use of simple and **basic patterns**, that constituted the beginning of the Artificial Intelligence applied to computer games. These basic patterns were simple rules written within the code that defined a concrete behaviour by selecting the actions to perform depending on some specific preconditions. For instance, the ghosts in Pac-Man showed some predefined behaviour by making use of deterministic movements and patterns that defined their different personalities.

Figure 2.1: Pacman screenshot

Around 80s, home computers became more popular and therefore, more computer games started to be developed. Furthermore, due to important technological advances and more available resources, more complicated games were possible to be developed. For these reasons, AI techniques moved from the previous ones and new tendencies were created. The first step was the usage of **Finite State Machines (FSMs)**, composed by a finite set of states and transitions between those states. They constituted a very deterministic and simple way of defining the actions to perform in order to achieve an specific state of the game.

At the beginning of the 90s, **Real-Time Strategy games** arose. They introduced new problems as decision making, pathfinding or uncertainty. An important RTS game at that time was *Dune II*. Later on, *Warcraft (1994)* appeared and became one of the most popular games in the history of computer games. Due to its huge success, more versions have been developed during the years ( *Warcraft II: Tides of Darkness*, *Warcraft III: Reign of Chaos*,..)

Furthermore, new AI techniques were applied for computer games, such as **neural networks** , introducing new AI concepts like machine learning, pattern recognition or prediction.

Also, **First-Person Shooter games** rose in popularity with games like *Doom* (1992) or *Quake* (1996) among others. This was helped by the important improvements in computer game graphics. They also introduced new challenges for the AI development. They will be discussed later in section 2.3.

With the appearance of *The sims* (2000), another genre of games was introduced. It succeeded in the simulation of human behaviours. Players could take the role of the members of a family, and the rest of characters were controlled by the machine, which tried to act as real human players would do. Unlike other games, there was no specific goal for this game, but players had to make decisions about their characters' lives: both professional and personal ones.

The production process of computer games has been changing over the years. During the course of the history of computer games, AI has remained in the background. This fact was due to two main factors: firstly, more importance was given to the improvement of the graphical elements and secondly, it was normally

left as the last part in game development process, after everything else was finished. Nowadays, this tendency is changing: better AI techniques are being required as well in order to simulate human-like and intelligent non-player characters, introduce some certain level of challenge for players and create a differentiating element from the rest of games.

At present, computer game industry have more success than music or movies. Computer games are making use of the most advanced AI technologies: neural networks genetic algorithms and fuzzy logic and there are numerous projects working on this field [1]. Furthermore, there are even companies that are focused on specifically developing AI for computer games, such as AiLive1 [Gamasutra, 2007] or Xaitment2

[1] http://aigamedev.com,www.gamedev.net,
http://ai4games.sourceforge.net/,
http://www.ai4g.com/,
http://fear.sourceforge.net/

## 2.2 Game AI

Artificial intelligence can be defined as the research field of creation and simulation of some intelligent behaviour. But, what is it meant with the term *intelligence*? Among other definitions, it can be described as the ability to learn, adapt, understand and evaluate the environment. In [15] is defined as the ability of acting and thinking as human, but also acting and thinking rationally.

In order to play efficiently to a game and be able to compete to a human player, some intelligence is required. The main goal is for the automatic player to present some human-like behaviour or at least to create the illusion of intelligence [14]. Furthermore, it has to be taken into account the fact that game AI has to be adapted to the player. It is not fun to play against simple opponents, but neither it is to play against very intelligent ones that can never be defeated.

An important and very recurrent topic within the game AI area is *cheating*. This situation can be created when extra benefits are given to the intelligent systems, either because they are provided with additional information or because their perceptual sensors are not as limited as human ones. In order to avoid intelligent agents to take some advantage over human players, they should own the same information, trying to imitate their knowledge model and using the same way to get the information and the same actions to interact with the environment.

In particular, game AI can be seen as a subfield of the AI, that corresponds to "the code in a game that makes the computer-controlled opponents appear to make smart decisions" [16]. Many types of games are making use of AI techniques and traditionally, games have been used as a research field and a way of applying and testing AI agents. This is because computer games present interesting and challenging environments for many AI research problems: Strategical decisions, adaptation to the environment, simulation of natural behaviour, navigation, cooperation, resource allocation among others [18].

As it was explained in section 2.1, game AI has started to be considered as an important part of the game development. Over the years, game developers are getting more interested in the game AI field and it is becoming very important within the game industry. This is due to the fact that companies look for a new

element that differentiates them from the rest and this element tends to be the AI component. Furthermore, due to the important technological advances, more realistic environments are created and therefore more complex techniques are needed in order to obtain better results.

There are many examples of computer games that have stood out due to the AI techniques they were making use of. *Black and white* (Lionhead Studios, 2001) [19] is an example of an important and impressive use of artificial intelligence in games. Another outstanding example was *F.E.A.R* (Monolith Productions, 2005), whose AI was considered one of the most important contributions to the game development industry due to its advanced AI system: a real-time planning system with Goal-Oriented Action Planning (GOAP). More examples include *GoldenEye 007* (1997), *Half-Life* (1998), *Halo* (2001), *Far Cry* (2004), *Doom 3* (2004) or *Left 4 Dead* (2008)

**Game theory**

It can be defined as *the study of the different ways in which strategic interactions among rational players produce outcomes with respect to their preferences or utility values* [2]. It can also be seen as a formal study of conflict and cooperation between players, each one of them has individual preferences to achieve his personal goals. Game theory formally makes a formal definition of the strategic reasoning by making use of mathematical terms that calculate the utility value or welfare for the different players and constitutes a valuable tool of conceptual analysis.

Game theory defines three concepts [15]:*players*, that are the participants in the game, *actions* that players are allowed to perform and the *payoff matrix*, that shows utility values for each combination of action and player. Each player will use these elements to create its own strategy. They will try to maximize the outcome taking into account the different actions that other players can do and the current situation of the game.

[2] http://plato.stanford.edu/entries/game-theory/

## 2.3 Game genres

There are different types of games, according to their common characteristics.

1. **Turn-Based Strategy games (TBS)**
   Action does not take place in a continuous way, but it is divided in turns. In each one of them, the correspondent player executes his actions. When the turn finishes, the state of the game is updated with the results of these actions and another turn starts for the next player. This continues like this until the game is over.

   As TBS games allow more time to make decisions, better and more complicated strategies can arise. Furthermore, most of these games have complete information facilitating the implementation of more complicated techniques, as it happens for example in *chess* or *risk*.

   **Civilization** [3]: Game created by Sid Meier for MicroProse in 1991, that consists on the creation and expansion of an empire. Players have to decide about almost everything: when, what and where they want to build, what they want to improve and so on, increasing hugely the scope of the game. As the game became very successful, new versions were developed: *Civilization II* (1996), *Civilization III* (2001), *Civilization IV* (2005), *Civilization Revolution* (2008) and *Civilization V* (due to release in 2010), apart from other related games.

Figure 2.2: Civilization III screenshot

[3] Civilization: http://www.civilization.com/

2. **Real-Time Strategy games (RTS)**
   They can be similar to Turn-Based Strategy games, but they introduce a big difference: in RTS games, time is continuous, there are no pauses [3]. This is very important, as players cannot take very long to make their decisions because the environment keeps changing while they are deliberating. There are two main domains for this type of games [14]: economy and combat. In both of them, strategies are mainly oriented to groups, societies or armies instead of individual units

   They normally present imperfect information and contain features such as resource collection and management, terrain exploration, structure building, attacking, defending, among others. Solutions include elements like pathfinding, decision-making under uncertainty, spatial and temporal reasoning, adversarial reasoning, planning and hierarchical strategy planning. Normally, goal-oriented reasoning works better as the number of possible actions may be very large in this type of games.

   **Starcraft** [4]: One of the most well known games of this genre, it was created by Blizzard Entertainment in 1998. In this version, three races or species fight against each other for their survival and the dominance of the territory. For doing so, players can create new units or buildings, research new technologies, collect and make use of the resources among others.

Figure 2.3: Starcraft screenshot

[4]Starcraft: http://us.blizzard.com/en-us/games/sc/

Other important games of this genre are *Command and Conquer*, *Warcraft* or *Age of Empires*.

3. **First Person Shooter games (FPS)**
   Action games presented from the player's point of view. For this reason, they normally tend to work individually, that is that each player takes control of just one character [14]. Although this idea is changing and more games in which several small units are controlled by a single player are appearing. This type of games are mainly focused on attacking the enemies.

   It represents the major genre in which game AI work has more relevant [16]. This is due to the fact that users have been encouraged to create their own modifications (or "mods") of these games. Among them, bots are those that are characterized for making use of AI techniques.

   **Half-Life** [5] The main goal is the survival within a complex environment where the player is attacked by different enemies. For doing so, he has to solve several problems or puzzles. Bots, in half-life, have the particularity of cooperating with each other when exchanging weapons to maximize their attack or chasing the player to surround him.

Figure 2.4: Half life 2 screenshot

   **Quake III Arena** [6] (1999): Multiplayer game where players compete against bots that are computer-controlled (in single player modes) or against human players (in multiplayer modes). Player earns points when reaching goals. For doing so, he moves around the world, gathers weapons and attacks his enemies. Game ends when timer is over or when

[5] Half-Life: http://orange.half-life2.com/
[6] Quake: http://www.quake3arena.com/

a player reaches some specific location.

Bots have different characteristics and behaviours and use several AI techniques: FSM, fuzzy logic,.. among others, presenting different levels of difficulty. There are also several types of weapons with different characteristics that make them more suitable for some situations than for others.

4. **Stealth games**
   Also called "First-person sneaker" games, where mainly the goals are detection avoidance and hiding from enemies. They are almost identical to FPS games, but players tend to hide from their enemies, instead of fighting with them. [14].

   As this type of games give high importance to the idea of players avoiding being discovered, perceptual elements are very important.

   **Thief**: Very innovator game that includes a high amount of possible actions to take. Players have to discover the world and try not to be perceived by the guards. The latters make use of very advance AI techniques in order to discover and chase the players and fight against them. This game uses different levels of alert for the guards, indicating how suspicious they are of having a player close to them. These levels increase or decrease depending on the perceptions they get from the environment.

Figure 2.5: Thief: Deadly shadows screenshot

Due to its popularity and innovation, three different games have been

released: *Thief: Deadly shadows* [7], *Thief: The Metal Age* and *Thief: The Dark Project*

5. **Role-Playing games** (RPG)
   In these games, there is normally a story in which players represent game characters. They have different characteristics, according to the role they take. These characteristics are usually divided in three categories: attributes (own characteristics), skills (learned capabilities) and powers (extraordinary abilities). Normally, in these games, players keep growing as the story goes on. They usually include tasks or goals like learning, training, earning, recollecting or building. As RPGs represent a story, there are other people involved in the game that interact with human players and are called Non-Player Characters (NPC). Usually they are not very smart, they just help to continue the story giving for instance some information to the players when they interact.

[7] Thief, Deadly shadows: http://www.eidos.co.uk/gss/thief_ds/

## 2.4 AI approaches

This section introduces some of the existing AI techniques used in computer games. Their usability and efficiency are analysed in order to identify their suitability for this project. Five different concepts are discussed: planning, AI navigation, perception, memory and decision-making techniques.

### 2.4.1 Planning

This section introduces several techniques that deal with: presence of uncertainty, incomplete information and existence of changes in the environment. For these problems, agents cannot make use of single-agent search algorithms as they have to adapt their plans to the changes in the environment. There are two main search or planner algorithms that focus this situation: Incremental Heuristic and Real-time Heuristic.

**Incremental planning**: Incremental algorithms make use of information related to previous searches in order to find solutions for similar problems faster than if it starts from scratch. Also called continuous planning or replanning. It can provide good results for some domains in which changes are not so frequent or new states of the game turn out to be slightly different from those that the agent visited in his planning [10].

**Real-Time planning**: It needs to be dynamic and be able to handle better unexpected situations as goals and environments keep changing over time [13]. For doing so, real-time algorithms alternate planification (restricting its search to the part of the domain adjacent to the current state of the world) and plan execution. Therefore, the sequence of actions needed to satisfy some goal is computed, but when a change in the environment is detected, the current goals are checked and the plan is recalculated if necessary [11]. In order to execute this replanning process, the solution implemented in [6] and [5] is the search of intermediate goals that will lead to the final goal. Then, by making use of a heuristic process, a new plan will be generated, going through one of the intermediate goals in order to reach the final one. For instance, F.E.A.R [8] makes use of Real-Time planning to implement automatic players that combat with human or robotic enemies.

[8] http://www.whatisfear.com/

### 2.4.2 AI Navigation

It corresponds the ability of reaching specific locations of the scenario. Two important aspects are discussed: how to find the shortest path to the target (pathfinding) and how to avoid the obstacles on the way.

**Pathfinding**

Calculation of the best way between two specific locations, meaning by best, the one that gives a best utility value or has the smallest cost. The best path is not necessary the shortest one, as there might be obstacles in between or areas that introduce some cost if agents go though them. Therefore, pathfinding has to deal with four aspects [4]: how to get form one point to the other, how to avoid the obstacles, how to find the shortest path and how to do it quickly.

Search is one of the most used techniques to solve pathfinding problems:

- **Dijkstra**: It is a graph based algorithm, where nodes represent the different points or cells that agents can visit and edges define the cost of moving from one to another. The idea of this algorithm is the exploration of all the short paths that go from the origin to the rest of nodes. When the final point is reached, the algorithm stops and returns the path computed so far. Although it gives good results, it is not valid for graphs with negative costs and it can be inefficient for big domains in which the number of possible nodes grows arbitrarily.

- **A***: It uses a best-first search to find the least-cost path between two given points. Unlike Dijkstra's algorithm, it makes use of a heuristic function that not only takes into account the cost from the origin point, but it includes an estimation of the final one:

  $$f(node) = g(n) + h(n)$$

  where g(n) represents the cost from the initial node to $n$ and h(n) is an estimation of cost from $n$ to the target.

  A* is widely used nowadays to solve pathfinding problems. It obtains better results than Dijkstra's algorithm as it reduces the search space

taking into account the distance to the final point.

- **RTA* (Real-Time A*)**: Modification of A* that controls the search in real-time domains. It stores the value of each visited state in a hash table and it keeps updating them by making use of dynamic programming techniques.

**Obstacle avoidance**

Similar concept to pathfinding, but obstacle avoidance tends to be more reactive as agents need to avoid obstacles when they find them instead of precomputing them when calculating the path they are going to follow. Usually, pathfinding algorithms are used because there is complete information about the environment, therefore, the best path can be calculated. However, obstacle avoidance is more suitable for those cases where there is some uncertainty about the environment, whether it is due to the fact that agents have incomplete knowledge or the existence of dynamic obstacles.

### 2.4.3 Perception

Percepts are used to gather information in partially observed games. As players do not have a complete knowledge of the game state, they need their perceptions to learn it. As shown in [9], there are different types of perceptions that are propagated and perceived in a different way [14]. For this reason, they are treated independently:

- **Visual**

  This perception depends on several factors:

  - **Distance from the observer to the object**
  - **Blocking elements**. In case there are elements between the observer and the object, it would be more difficult to detect. A ray-trace [9] can be used to determine whether there is an element in between or not.
  - **Visual field of the player**, that corresponds to the area which the player is facing at.
  - **Visual conditions** in the object's area.

  In [16], they use a combination of these elements that gives a perception probability value (from 0 to 1) that determines the certainty for the observer to perceive the object.

- **Auditive**
  Actions can have some sound level and tone associated to them so they can be distinguished by agents. Different sounds can interfere with each other [14], being necessary to stand out those important sounds from the unimportant ones. Auditive perceptions help to get information about events that happen in an area close to the agent, but maybe not visible to him.

- **Tactile**
  It normally represents what the player feels and the reaction to pain and object collision.

### 2.4.4 Memory

The term *belief maintenance* is defined in [7] as the reliability of player's beliefs to be consistent with the state of the game. In a partially observable domain, the player does not count with complete information about the environment. Furthermore, if it corresponds to a dynamic environment, the information that players gather is not always an accurate representation of the game state. Longer it is since the player received the information, more likely it is that the current situation does not agree with it. For instance, a player might see an object in an specific position and store such information in a beliefs list. As time goes by, if the player is no longer looking at that position, the probability for that object to still be there decreases. So, the player now bases its reasoning on some facts that might not be true anymore. For instance, fig 2.6 represents two different moments of a game. In the first one, the player is able to see where the enemy is, but as he continues walking, the enemy disappears from his visual field. In the second instant, the player remembers the position in which he saw the enemy for the last time. But in this case, this fact is not true anymore, as he has also moved to another position. For this reason, it is very important to determine whether these beliefs are still reliable or not. In order to solve this problem, [7] suggests the idea of comparing the time since the information was obtained with a threshold value. In case the first one is bigger, that information is not valid anymore, so it is whether deleted from the beliefs list or it is checked again.

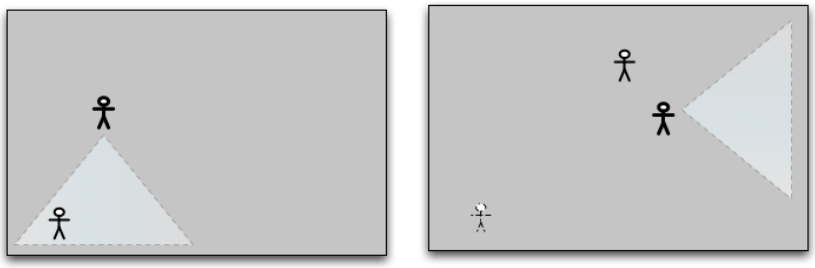


Figure 2.6: Agent's memory

### 2.4.5 Decision making techniques

They can be defined as the processes of choosing an action to perform for the current state of the world. They constitute a very important part in the AI field and try to imitate the human way of reasoning. However, it is not an easy task to do. One of the main difficulties is the way of modelling the information in order to deal with it and obtain the necessary knowledge from it. Sometimes it is also necessary to deal with uncertainty if the outcomes of the actions are not deterministic and the information is not complete.

There is no a standard solution for all the problems. Depending on their characteristics, it will be better to use one or another. Four different techniques are discussed in this section: *Finite-State Machines*, *Fuzzy logic*, *Bayesian networks* and *Production systems*

1. **Finite-State Machines (FSMs)**
   Behaviour models formed by a finite number of states and transitions between them. Each state shows a different situation of the world and transitions define those actions that are needed in order to move from one state to another. Transitions also have a set of preconditions that must be fulfilled in order to allow the movement to the next state.

   They constitute a very simple AI technique and its implementation can be pretty straightforward. It has been mainly used for FPS games, so the different states of the game may be easier to represent. For example, in *Quake2*, monsters' states can be defined as one of the following nine:

standing, walking, running, dodging, attacking, melee, seeing the enemy, idle and searching. In this way, depending on some preconditions and the actions they perform, they will behave in different ways.

On the other hand, they are no suitable for every domain, for instance in non-deterministic environments, as FSMs need to know all the possible states and actions. Furthermore, they might not work very efficiently for large systems, being difficult to manage a high number of states. Moreover, when used for videogames, it can be very predictable and therefore, not representing a challenge for the player once the latter learns its movements.

2. **Fuzzy logic**
   It can be defined as a type of logic that uses vague information to make decisions, trying to imitate the human way of thinking. In classic logic, statements are either true or false. However, this changes in fuzzy logic: there are different degrees of truth. For instance, in classic logic the statement "it is hot" can be either true or false. However, in fuzzy logic, there are different degrees: very cold, cold, normal, hot,...

   In fuzzy logic, decisions are taken according to a set of "if-then" rules that are associated to different fuzzy sets. For instance:

   *if temperature is very cold*
   *stop fan*
   *else if temperature is cold*
   *turn fan*
   *else if temperature is normal*
   *maintain level*
   *else if temperature is hot*
   *speed up fan*

   They are robust and tolerate vagueness or imprecision input values. Introducing some degrees of truth lets the player to deal not only with extreme situations, but also with intermediate ones and act more specifically for each one of them. In those situations that is not possible to classify everything as

white or black (or true or false), fuzzy logic helps to deal with all they grey areas in between [12].

3. **Bayesian networks**
   They are used to model knowledge and cause-effect relationships and to perform inference and learning, They deal with partial or incomplete information and are based on Bayes theorem, that determines dependence relationships and can be written as follows:

$$P(A|B) = P(B|A)P(A)/P(B)$$

Where P(A |B) is the probability of B knowing that A is true, P(A) is the probability of A and P(B) is the probability of B to happen.

Bayesian networks are formed by directed acyclic graphs (DAG) that represent these dependence relationships between variables. Each node represents a different variable and has a probability value assigned to it, calculated from the node's parent variables. Each arc defines the dependence relationship between the pair of variables that it connects.

Despite of the ability for the bayesian networks to provide inference under uncertain conditions, their usage has not expanded in the game AI field. It might be due to the fact that, although they can be used for a wide variety of game genres, their complexity increases hugely as the domain gets more complex [8]. However, due to technological progress, they are becoming more feasible.

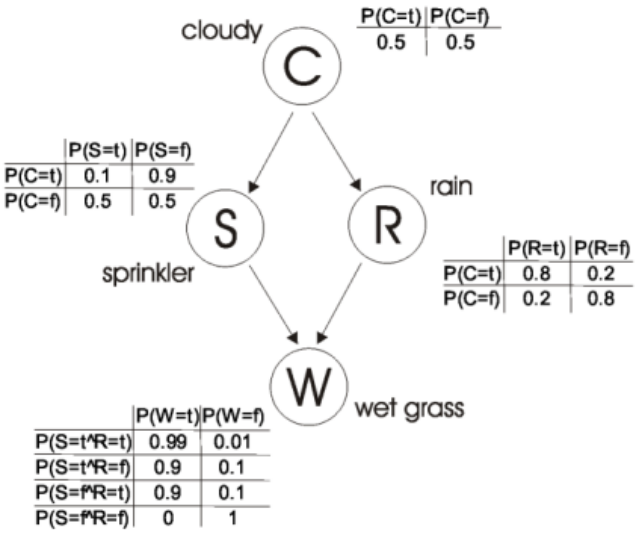



Figure 2.7: Bayesian networks

Figure 2.7 [9] shows an example of a bayesian network with one input node (C), two intermediate ones (S and R) and one output (W). Tables represent the distribution of probabilities.

Bayesian networks represent a simple and robust way of representing and dealing with uncertain information. They also can be used for prediction as they handle probability values. They are suitable for small and incomplete data sets.

4. **Production systems**
   Also called *Expert systems*, production systems try to decide which is the best action to perform from a list of conditional statements or rules that define the steps that are needed to solve a problem depending on the current state of the world.

   In doing so, problems are defined by three modules:

   - **Working memory**: Constituted by predicates or facts that describe the problem and define the agent's knowledge at each moment.

[9] http://www.ra.cs.uni-tuebingen.de

- **Rule base**: Rules or conditions that describe the reasoning process created to solve the problem. They are a set of rules that define conditional statements expressed in a "if-then" form in order to describe the preconditions that have to be fulfilled in order to execute each action.

- **Inference engine**: It executes the strategy obtained from matching the working memory and the rules that can be applied to it, acting as a rule interpreter.

In Figure 2.8, the making decision process can be seen. It consists on three important steps:

- **Matching**: Determining the active rules from the rule base, that is, the rules that can be applied to the current state of the world. Although only one action will be executed, the result of this process is a conflict set that includes all the rules than have been activated. In doing so, left-hand-side conditions are compared against the elements of working memory. In case they match and therefore preconditions are fulfilled, that rule is included into the conflict set.

- **Conflict resolution**: Choosing the rule to execute among all the rules that have been activated and therefore included into the conflict set. There is not an unique way of solving this problem, as it can be done from the simplest (e.g. random selection) to a more complex process (for instance, according to some priority values in which some specific actions are preferred over the rest).

- **Strategy execution**: Once the strategy has been decided and therefore the action to perform is known, it is executed. This will produce a modification in the world, needing to update agent's knowledge, removing or adding data to the working memory.

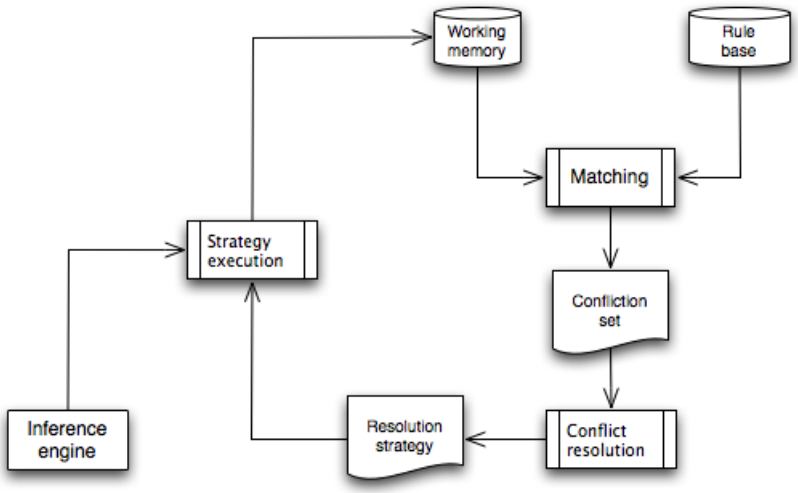



Figure 2.8: Production systems cycle

## 2.5 Summary

In this section, the computer game history has been shown, analysing the evolution of the games, their complexity and the necessity of including a better AI in their development. Moreover, it has been seen that more and more intelligence is being demanded for the NPCs (Non-Playing Characters), not only to present some level of challenge to players but also as an unique selling point to distinguish a game from the rest.

Furthermore, several game genres (TBS, RTS, FPS, Stealth games and RPG) have been defined, giving some examples of each one of them and also showing the different challenges they present to the game AI field. Furthermore, different concepts that are needed to take into account to implement AI game techniques have been shown.

All these elements will be used in next sections in order to explain the problem domain to solve and the solution that has been chosen and the reasons for having done it.

# Chapter 3

# Objectives

The goal of this project is the study and implementation of one or more artificial intelligence techniques in order to create an automatic player within a game scenario that will contain First-Person Shooter (FPS) and stealth game elements. The client will execute his actions against other players (be they humans or machines) and will have to do it in the most efficient way possible.

Firstly, the specification of the scenario will be determined. The system will just simulate a real-time game: it will be divided into small ticks of time which in their turns will be split into two parts: planification time — in which the clients are allowed to send their actions) and execution time (in which the server updates the current state of the game and informs the players about it.

The architecture will be composed of one server and several clients that will connect to it. The information of the game will be centralised within the server, being the players aware of just those parts that correspond to them.

The characteristics of the game are determined in such a way that gives enough complexity to the problem and lets the possibility of making an interesting study of the different approaches according to them. For doing so, several aspects will be taken into consideration:

**Perception:** The player will use his visual and auditive perceptions to learn about what is happening in the world.

**Non-complete information:** The player does not have a complete vision of the game, he only knows what he is perceiving in each moment. This perception will depend on several factors: his own characteristics (visual and auditive perceptions) and also the environmental ones — existence of walls, illumination, noise, etc.

**Uncertainty:** The result of his actions is not deterministic, it depends on other factors (enemies, changing elements and so on).

After the domain is defined and created, the different algorithms and approaches are studied and implemented according to it. These techniques will deal with different aspects:

**Path-finding:** Be able to find the most efficient path from one point to another, taking into account the different obstacles, doors, walls and other objects that exist within the environment.

**Decision making:** Depending on the current situation in each moment, the player will decide the action to perform: attack the enemy, pick up the objects that might be used later on, avoid being discovered or reach concrete points of the map.

**Adaptation:** The player has to be aware of the changes in the environment and be able to modify the plan he is executing in order to adapt himself.

As mentioned above, the focus of the project will be centralised in the study of the different approaches that can be used for the implementation of the player, but also in the definition and implementation of the system. It will be also interesting to determine which elements from the domain are more relevant when selecting one algorithm or another. Finally, in order to make its evaluation of its efficiency, it will be tested against other machine clients that could present or not some intelligence, but also with real players. The reason of doing that is because, in this way, the implications of the usage of artificial intelligence can be tested by analyzing its performance against simple clients (that, for example, execute random actions) and seeing the improvements they introduce. On the other hand, the fact of comparing its performance against human abilities brings back the discussion about the possible substitution of human behaviours by intelligent systems.

# Chapter 4

# Development

In this chapter, the different steps that are needed for the implementation of this project are discussed.

In first place, the definition of the game is presented: its characteristics and the main elements that compose it. This is important to understand what problem is wanted to be solved and the challenges it presents. Secondly, an analysis of the project is made, showing the different use cases that describe the behaviour of the system and defining the requirements that must be fulfilled. Then, the solution is defined by showing the architecture that has been chosen for the implementation of the system. For doing so, the main components and their subcomponents are defined and analysed. Next, the detailed design is presented, giving a more precise information of the system's implementation, showing UML class diagrams and sequence diagrams that will show the behaviour of more concrete elements and the interaction between them. Finally, more details about the implementation are given, explaining the decisions that have been made to develop the game engine and the AI techniques that have been used to create the automatic player.

## 4.1 Description of the game

In this section, the game to be implemented is introduced: its main characteristics are analysed and its key elements are defined. The game is classified according to the criteria explained in section 2.3 and the challenges it presents are introduced.

### 4.1.1 What type of game?

It is a Real-Time game where players compete against each other in order to achieve their goals. For doing so, they receive information from the environment through their perceptions: visual, auditive and tactile. They can also interact with other players: (attacking, shouting,..) and with the environment (opening and closing doors, picking up and using objects,...).

The game's world is composed by a grid of cells which are surrounded by walls and doors. The latters can be open, closed or locked. There are also some items that can be picked up and used by the player (like keys to open doors or torches to light up with fire), as well as weapons and ammunitions that can be used to attack the enemies. Furthermore, some cells contain obstacles, such as fire or bushes, that do not allow the access to the player and may cause them some damage. Every element occupies just one single cell and there is only one element per cell.

### 4.1.2 Characteristics

The game designed for this project shows the following characteristics:

**Multi-agent:** Different players are competing against each other at the same time.

**Partially Observable:** Players only have local perception about the environment. They only know about what they are perceiving in each moment, so they do not have complete knowledge about all the elements of the game.

**Stochastic:** Outcomes from players' actions are not always deterministic. Players normally know what to expect from their actions, but there could be some situations in which the result is not what they had anticipated.

**Sequential:** Current decisions can affect future ones. As the course of the game is continuos and not divided in episodes, consequences of players' actions can affect the future. For instance if a player decides to pick up a key, the actions

to perform when he finds the correspondent door to open will be different than if he had not picked it up.

**Dynamic:** Environment can change while players are deliberating. As there are more players playing the game, they can modify the game's world while the player is not performing any action.

### 4.1.3 Description

In order to understand the functionality of the game, all its key elements and the possible actions players can perform are now defined:

1. **Elements**

   - **Players**: Automatic or human players that participate in the game and compete against each other. They all have the following characteristics:
     - **Life**: Points of energy that can decrease (when the player is attacked) or increase (by getting powers).
     - **Weight**: Measure of heaviness of the objects the player is carrying. When it reaches its maximum value, the player is not able to carry anything else.
     - **Visual perception**: threshold value from which the player is able to see.
     - **Auditive perception**: threshold value from which the player is able to hear.
   - **Walls and doors**: They separate rooms and they are located between cells. They have an attenuation value, that represents the lost of sound intensity when it is being transmitted through them. Doors also have another characteristic that shows its status within the game: open, closed or locked.

- **Weapons**: They are used by players to attack each other. There are different types: *arch*, *knife* and *gun*. All of them present some characteristics that distinguish them from each other:
  - **Distance**: Number of cells that the attack of this weapon can reach.
  - **Strength**: Points of life that takes away from a player when it is hit by it.
  - **Noise**: Intensity of the sound that makes when it is used.
- **Objects**: They can be picked up and used by the players. There are different types:
  - **Key**: To be used in order to open the doors. They cannot open every door, each one of the keys is associated to an specific door.
  - **Torch**: They can be lit on with fire, giving extra light to the area where they are located.
  - **Ammunition**: Players can use them to reload their weapons. There are two types: bullets and arrows.
- **Obstacles**: They do not let players to go through the cells where they are located. There are different types of obstacles: fire, statue, bushes,.. Some of them penalize those players that are shocked against them. For instance, when touching fire, life points are taken away.

  They present two important characteristics:
  - **Damage**: The amount of life points that are taken away from the player.
  - **Attenuation**: It stops from seeing a player if he is hidden behind.
- **Powers**: They act as a reward for the players, giving them more life points.

- **Temporal elements**: They appear at some points of the game and only last for few cycles. There are two types: **footsteps** (left by players when they walk on sand) and **echo** (created by loud noises). They have three important characteristics:
  - **Duration**: Time they will remain.
  - **Intensity**: Strength of the signal (visual or auditive) to be perceived by the players.
  - **Attenuation**: Lose of intensity in each cycle that the sound is repeating.

  Temporal elements do not react or interact with the player, they just provide additional information (for instance, the fact of seeing some footprints can tell that somebody was there not very long time ago) and then they disappear.

2. **Actions**

   This is the list of actions that can be performed by the players:

- **Attack**

  **Parameters**: Weapon $w$ used to attack.

  **Preconditions**: Player has a weapon $w$ with ammunition.

  **Effects**: The player that is attacked loses life points if he is reached and the attacker spends the correspondent ammunition.

- **Close door**

  **Parameters**: Door $d$ to be closed.

  **Preconditions**: Door $d$ is open and player is situated in an adjacent cell facing at it.

  **Effects**: Door $d$ is closed.

- **Leave item**

  **Parameters**: Item $i$ to be left.

  **Preconditions**: Player has the item $i$ and the cell where he is situated is empty.

  **Effects**: The item is left and the player does not have it anymore.

- **Move** : From cell $(x_1, y_1)$ to cell $(x_2, y_2)$.

  **Parameters**: Cell $(x_1, y_1)$ and $(x_2, y_2)$.

  **Preconditions**: Player is in cell $(x_1, y_1)$ and there is no wall, obstacle or closed or locked door between the two cells.

  **Effects**: Player ends up in cell $(x_2, y_2)$

- **Open door**

  **Parameters**: Door $d$ to be opened.

  **Preconditions**: Door $d$ is closed and player is situated in an adjacent cell facing at it.

  **Effects**: Door $d$ is open.

- **Pick up**

  **Parameters**: none

  **Preconditions**: There is an item to pickup in the cell where the player is located.

  **Effects**: The player has the item.

- **Reload weapon**

  **Parameters**: Ammunition $a$ to reload weapon $w$.

  **Preconditions**: Player has the weapon to reload and the ammunition needed to do it

  **Effects**: The weapon is loaded.

- **Shout**

  **Parameters**: none

  **Preconditions**: none

**Effects**: Player has made a sound.

- **Switch torch off**

  **Parameters**: Torch $t$.

  **Preconditions**: Player has a torch $t$ and it is lit.

  **Effects**: The torch is unlit.

- **Switch torch on**

  **Parameters**: Torch $t$.

  **Preconditions**: Player has a torch $t$ and it is unlit.

  **Effects**: The torch is lit.

- **Touch**

  **Parameters**: none

  **Preconditions**: There is an element in the cell where the player is located.

  **Effects**: Player gets information about an element and is able to identify it.

- **Turn**

  **Parameters**: none

  **Preconditions**: none

  **Effects**: Player faces to a different direction.

- **Use key**

  **Parameters**: Key $k$ and door $d$

  **Preconditions**: Player has key $k$, he is facing door $d$, that is locked and can be opened with $k$.

  **Effects**: Player does not have the key $k$ anymore and door $d$ is unlocked.

## 4.2 Analysis

In this section the development process is introduced, explaining different decisions and assumptions that have been made. Furthermore, the different use cases are defined and the project's requirements are discussed.

### 4.2.1 Methodology

The methodology that has been chosen is based on evolutionary prototypes. That means that different versions (or prototypes) will be created within the system's development. In each iteration, new features are added to the previous version. First of all, a basic prototype will be implemented. Starting from it, different functionalities will be added, continuing this process until the scope of this project is reached.

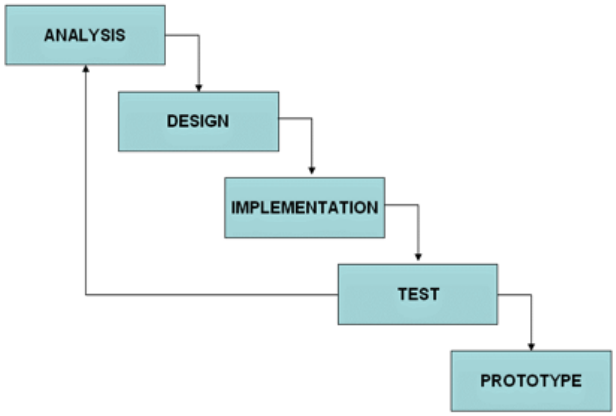



Figure 4.1: Evolutionary prototypes

Each iteration can be divided in different steps as shown in Figure 4.1:

1. **Analysis**: Identification of the main requirements for each iteration.

2. **Design**: Description of the components that are required to fulfill the requirements that have been identified. This section will be decomposed later on in this document into two sections: architectural and detailed design.

3. **Implementation**: Codification of the new classes that have been specified in the design.

4. **Test**: Confirmation of having achieved the correspondent goals for this iteration.

5. **Prototype**: Result of the iteration that corresponds to a subversion of the final system.

This methodology has many advantages and has been chosen for two main reasons: incremental development and avoidance of error propagation. In this document, the above mentioned steps will be shown for the last iteration.

### 4.2.2 Use cases

Four different use cases (Figure 4.2) are discussed in this section.

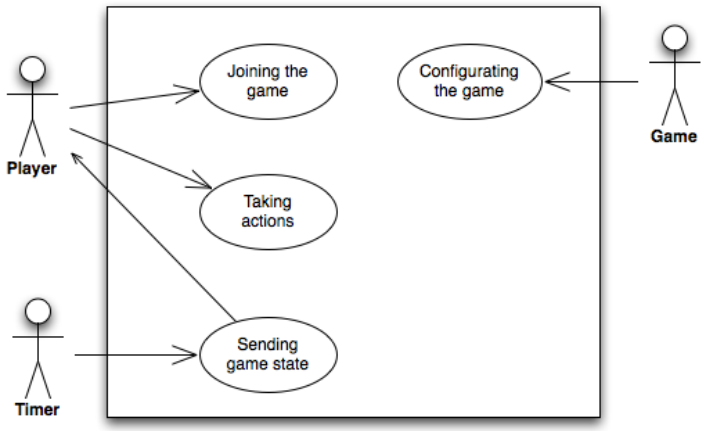



Figure 4.2: Use cases

For doing so, tables are shown, each one of them will correspond to a different use case and contain the following information:

| | |
|---|---|
| **Identifier** | Used to univocally identify the use cases. They all start with UC - XX, where XX represents the requirement's sequencial number. |
| **Name** | Identifies the use case in a clear and simple way. |
| **Preconditions** | Requirements that have to be fullfilled so the use case can be started. |
| **Description** | Brief explanation of the requirement |

Table 4.1: Use case definition table

1. **Configurating the game**
   This use case starts when a new game is going to be created and it is started by the server. Once it finishes, it passively waits for the players to login.

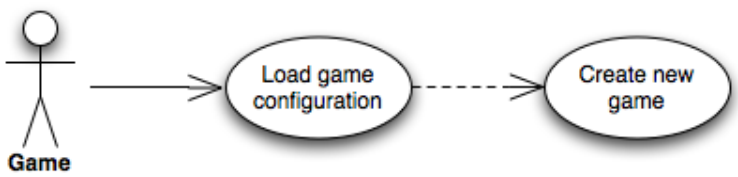



Figure 4.3: Game configuration use case

| | |
|---|---|
| **Identifier** | UC-01 |
| **Name** | Load game configuration |
| **Preconditions** | None |
| **Description** | The parameters of the game are loaded. |

Table 4.2: UC-01 - Load game configuration

| | |
|---|---|
| **Identifier** | UC-02 |
| **Name** | Create new game |
| **Preconditions** | The configuration of the game has been loaded. |
| **Description** | A new game is created from the parameters that have been loaded. |

Table 4.3: UC-02 - Create new game

**2. Joining the game**

This use case represents a scenario in which the player wants to enter the game. This only can be done before each game starts until the maximun number of players is reached.

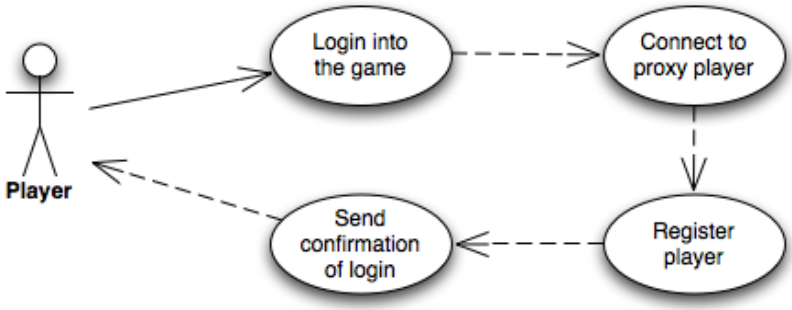



Figure 4.4: Login use case

| | |
|---|---|
| **Identifier** | UC-03 |
| **Name** | Login into the game |
| **Preconditions** | None |
| **Description** | The player joins the game by inserting his username and connecting to the login server. |

Table 4.4: UC-03 - Login into the game

| | |
|---|---|
| **Identifier** | UC-04 |
| **Name** | Create a proxy player. |
| **Preconditions** | The user has joined the game. |
| **Description** | A new process that will act as the proxy is created and the login server connects to it. |

Table 4.5: UC-04 - Create a proxy player

| | |
|---|---|
| **Identifier** | UC-05 |
| **Name** | Register player |
| **Preconditions** | The login server has connected to the proxy player. |
| **Description** | The server tries to register the player into the game. |

Table 4.6: UC-05 - Register player

| | |
|---|---|
| **Identifier** | UC-06 |
| **Name** | Send confirmation of login |
| **Preconditions** | The server has tried to register the player into the game. |
| **Description** | The player obtains the answer from the server through the correspondent proxy player. |

Table 4.7: UC-06 - Send confirmation of login

**3. Taking actions**

This use case represents a scenario in which a player wants to send his action to the server. It starts when he decides the action he wants to perform. It is repeated along the game, whenever he wants to do something and therefore, notify the server about it.

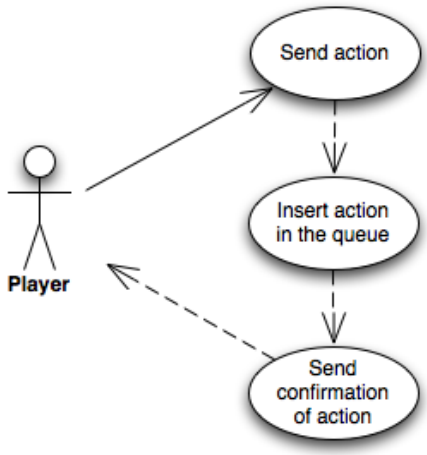



Figure 4.5: Taking actions use case

| | |
|---|---|
| **Identifier** | UC-07 |
| **Name** | Send action |
| **Preconditions** | None |
| **Description** | The player chooses the action to perform and sends it to the server. |

Table 4.8: UC-07 - Send action

| | |
|---|---|
| **Identifier** | UC-08 |
| **Name** | Insert action in the queue |
| **Preconditions** | The player has sent his action to the server. |
| **Description** | The server inserts the player's action in a queue of actions. |

Table 4.9: UC-08 - Insert action in the queue

| | |
|---|---|
| **Identifier** | UC-9 |
| **Name** | Send acknowledgement of action |
| **Preconditions** | The player has received the action and inserted it in the queue. |
| **Description** | Server sends to the player a confirmation message when it receives his action. |

Table 4.10: UC-9 - Send acknowledgement of action

4. **Sending game state**

   This use case starts when the server receives the notification of the timer, indicating the begining of the executing phase, where the server calculates the status of the game and sends it to all players.

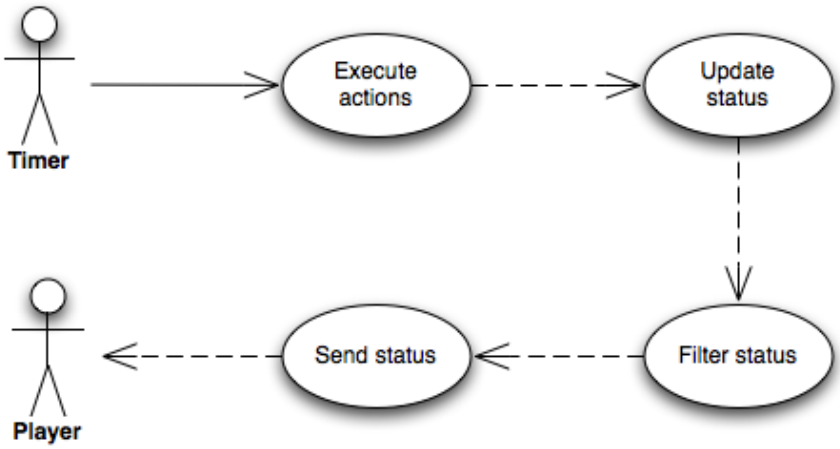



Figure 4.6: Updating status use case

| | |
|---|---|
| **Identifier** | UC-10 |
| **Name** | Execute actions |
| **Preconditions** | None |
| **Description** | The server executes all the actions it has received from all the players. |

Table 4.11: UC-10 - Execute actions

| | |
|---|---|
| **Identifier** | UC-11 |
| **Name** | Update status |
| **Preconditions** | The actions have been executed. |
| **Description** | The status is updated according to the results of the actions that have been performed by the players. |

Table 4.12: UC-11 - Update status

| | |
|---|---|
| **Identifier** | UC-12 |
| **Name** | Filter state |
| **Preconditions** | The current game state has been calculated. |
| **Description** | The perception of the world is different for each player, therefore the information is filtered according to players' conditions. |

Table 4.13: UC-12 - Filter status

| | |
|---|---|
| **Identifier** | UC-13 |
| **Name** | Send status |
| **Preconditions** | The status has been filtered |
| **Description** | The part of the world that the player perceives is sent to him. |

Table 4.14: UC-13 - Send status

### 4.2.3 Requirements

This section shows the functional requirements to fulfill. The taxonomy used to classify the different requirements follows the standard defined in Métrica [2]. For doing so, for each requirement the next table will be shown:

| | |
|---|---|
| **Id** | Used to identify the non-functional requirements. They all follow this format: FR - XX (for functional requirements) and NFR-XX (for non-functional requirements, where XX represents the requirement's sequential number. |
| **Name** | Identifies the requirement in a simple and easy way |
| **Description** | Brief explanation of the requirement |
| **Priority** | Necessity of its implementation for the developer |
| **Stability** | Possibility of being subjected to modifications, whether they are made by the client or some other reasons |
| **Necessity** | Importance of its implementation for the client or final user |

Table 4.15: Requirements definition table

**1. Functional requirements**

| | |
|---|---|
| **Id** | FR-1 |
| **Name** | Client/Server architecture |
| **Description** | A Client/server architecture has to be implemented |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.16: FR1 - Client/Server architecture

| | |
|---|---|
| **Id** | FR-2 |
| **Name** | AI implementation |
| **Description** | At least one AI technique must be implemented in both problem domains |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | Medium |

Table 4.17: FR2 - AI implementation

| | |
|---|---|
| **Id** | FR-3 |
| **Name** | Game configuration |
| **Description** | Be able to configure the game from a file |
| **Priority** | Medium |
| **Stability** | Stable |
| **Necessity** | Medium |

Table 4.18: FR3 - Game configuration

| | |
|---|---|
| **Id** | FR-4 |
| **Name** | Updated game state |
| **Description** | All players have to know the current game state at anytime |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.19: FR4 - Updated game state

| | |
|---|---|
| **Id** | FR-5 |
| **Name** | Real-Time implementation |
| **Description** | Server keeps sending the game status to the players without waiting for them to send their actions |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.20: FR5 - Real-Time implementation

| | |
|---|---|
| **Id** | FR-6 |
| **Name** | Multiplayer |
| **Description** | The system has to allow connections from several players at the same time |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.21: FR6 - Multiplayer

| | |
|---|---|
| **Id** | FR-7 |
| **Name** | Game functionality |
| **Description** | System has to be able to let players pick up, use and leave items, attach other players, open and close doors, reload weapons an move around the game world |
| **Priority** | Medium |
| **Stability** | Unstable |
| **Necessity** | Low |

Table 4.22: FR7 - Game functionality

| | |
|---|---|
| **Id** | FR-8 |
| **Name** | Player's perceptions |
| **Description** | System has to be able to simulate player's perceptions. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.23: FR8 - Player's perceptions

| | |
|---|---|
| **Id** | FR-9 |
| **Name** | Signal propagation |
| **Description** | System has to be able to simulate the propagations of, both visual and auditive, signals. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.24: FR9 - Signal propagation

| | |
|---|---|
| **Id** | FR-10 |
| **Name** | Notification of perceptions |
| **Description** | System has to notify the players about their perceptions (visual, auditive and tactile). |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.25: FR10 - Notification of perceptions

**2. Non-Functional Requirements**

**(a) Performance requirements**

| | |
|---|---|
| **Id** | NFR-1 |
| **Name** | Processing time |
| **Description** | Server has to calculate new game state and send it to the players in less than 1 second. |
| **Priority** | Medium |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.26: NFR1 - Processing time

**(b) Security requirements**

| | |
|---|---|
| **Id** | NFR-2 |
| **Name** | No access to unauthorized information |
| **Description** | Player must receive only the information he is allowed to have. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.27: NFR2 - No access to unauthorized information

### (c) Usability requirements

| | |
|---|---|
| **Id** | NFR-3 |
| **Name** | Easy to use |
| **Description** | Player does not need more than few seconds to learn how to do something. |
| **Priority** | Medium |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.28: NFR3 - Easy to use

| | |
|---|---|
| **Id** | NFR-4 |
| **Name** | System language |
| **Description** | The language used in the system will be English.. |
| **Priority** | Low |
| **Stability** | Unstable |
| **Necessity** | Low |

Table 4.29: NFR4 - System language

### (d) Reliability requirements

| | |
|---|---|
| **Id** | NFR-5 |
| **Name** | Consistent game state |
| **Description** | The game status has to be consistent at every time for all the players. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.30: NFR5 - Consistent game state

| | |
|---|---|
| **Id** | NFR-6 |
| **Name** | Low failure rate |
| **Description** | The number of failures has to be less than 2%. |
| **Priority** | High |
| **Stability** | High |
| **Necessity** | High |

Table 4.31: NFR6 - Low failure rate

| | |
|---|---|
| **Id** | NFR-7 |
| **Name** | Recovery from errors |
| **Description** | The system has to be able to recover from errors. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.32: NFR7 - Recovery from errors

#### (e) Operational requirements

| | |
|---|---|
| **Id** | NFR-8 |
| **Name** | Game configuration file |
| **Description** | Be able to read a configuration file of the game written in XML. |
| **Priority** | Medium |
| **Stability** | Unstable |
| **Necessity** | Low |

Table 4.33: NFR8 - Game configuration file

#### (f) Portability requirements

| | |
|---|---|
| **Id** | NFR-9 |
| **Name** | Platform independent |
| **Description** | The system has to work under Mac Os X, Linux and Windows. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | Medium |

Table 4.34: NFR9 - Platform independent

#### (g) Robustness requirements

| | |
|---|---|
| **Id** | NFR-10 |
| **Name** | Robust for invalid inputs |
| **Description** | Works in presence of invalid inputs made by the human player when using the system. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | High |

Table 4.35: NFR10 - Robust for invalid inputs

#### (h) Modifiability requirements

| | |
|---|---|
| **Id** | NFR-11 |
| **Name** | Easy to add new features |
| **Description** | It has to be modular and flexible enough to make easier the addition of new actions or elements to the game. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | Medium |

Table 4.36: NFR11 - Easy to add new features

### (i) Reusability requirements

| | |
|---|---|
| **Id** | NFR-12 |
| **Name** | Easy to reuse |
| **Description** | Easy to reuse actual components to create new AI players. |
| **Priority** | High |
| **Stability** | Stable |
| **Necessity** | Medium |

Table 4.37: NFR12 - Easy to reuse

## 4.3 Architectural design

The architectural design will be divided in three sections:

**System overview:**
The system context and the design will be introduced. Furthermore different decisions, choices and assumptions will be explained according to the background, the problems and the requirements that define the scope of the project.

**System context:**
The system will be described from the point of view of the user interaction. The relations with external objects will be shown and the system would be treated like a black box.

**System decomposition:**
This section will show the design of the components that constitute the whole system. The system will be specified in terms of smaller subcomponents and the relations among them. Finally, these components will be analyzed in detail.

### 4.3.1 System overview

The whole system is designed to be structured as a client-server architecture as shown in Figure 4.7. This solution offers a centralised game server which allows players to connect to it. Furthermore, it will be implemented as a **thin client architecture**, where the main functionality of the system is executed in the server.

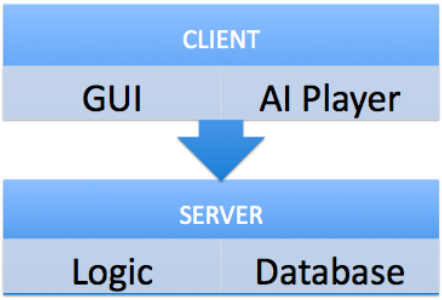



Figure 4.7: Client/Server architecture

The idea is to develop two independent applications that will compose the whole system: the client application and the server application. They can be executed from the same computer or from different ones, connected to the same local network.

**Client:**

It corresponds to each one of the players of the game. As a thin client, it does not perform any logic operation within the system. It just contains the modules corresponding to the graphical user interface, the model of the world that the player is perceiving and those modules related to the AI processing. It only stores the relevant information to each player.

**Server:**

It executes the logical functionality of the game and stores the whole information that constitutes the game state in each moment that is common for all players. It receives the actions from the players, computes the new state and sends the results back to the players.

The fact of having chosen a thin client architecture rather than any other possibility is due to the following reasons:

**Non duplication of information:** Although players have different perceptions of the world, the game status must be unique. For this reason, it seems more suitable to centralise this information in one component, that is the server, instead of having the same information replicated in different places.

**Consistent information:** In addition, the fact of having just one component in charge of updating the status of he world for all players ensures that the information they share is the same.

**Improvement in performance:** The actions and their results are executed and calculated in a centralised component, reducing in this way the number of operations that have to be executed and the work the client has to perform. Furthermore, as the players are only aware of some concrete information about the game, the server does not need to send them all the data, minimizing in this manner, the amount of information that has to be transmitted.

**Security:** The fact that only some specific information is sent to the clients gives security to the game against cheating players, as they only receive the data they are allowed to use and it is not possible any access to extra information related to the game in order to gain some advantage of it.

As shown in 4.8 the communication between the two modules is bidirectional: The client has to send his actions to the server, but he also has to receive an update of the game status from it.

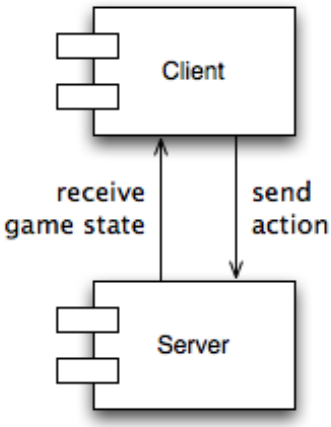



Figure 4.8: Client/Server communication

It is important to notice that players should know at every moment of the game what the current scenario is. For this reason, it is necessary that this communication works in both ways and the status is correctly sent to the players.

This communication process is carried out in two steps, that will be repeated in each cycle:

- **Receive actions:** Server lets some specific time in each cycle to receive player's actions.

- **Send update status:** Once the established time period is over, the server sends an updated status of the game to all players (to those who have performed some action in this turn, but also to those who haven't).

For doing so, when the player logins, he is actually subscribing himself to the game and offering an interface to the server in order to be notified about changes that will occur later on. Once the sending actions period has finished, the server will notify about the new situation to those players that are subscribed (i.e. those who are playing).

Therefore, the sequence of steps that both, server and client, follow during the course of the game are the following ones:

**Server**

1. **Configuration of the game**: Loads the configuration and creates the initial game state.

2. **Login into the system**
   - **Wait for clients to login**: Server receives login petitions from the players that want to join the game and signs them up.

   - **Send initial state state**: When the number of players reaches the maximum allowed for the game, the server sends the initial state to all of them.

3. **Execute game**: During the course of the game, three steps will be continuously repeated until the end is reached:
   - **Receive actions**: During the first period of each cycle, the server receives actions from the players and stores them in a queue.

   - **Execute actions**: The actions sent by the players are executed in order. The preconditions are checked and if they are correct, the effects are calculated and apply to the game status.

     This step is very important and there are two things that have to be taken into account:

     - **Resolve conflicts**: It could happen that due to the fact that several actions are received in the same cycle, those that are processed later than others cannot be executed as their preconditions are not valid anymore. In these cases, it has to be notified to the player that his actions could not be executed and why not.

     - **Compute new state**: Those actions that are executed have an impact in the state of the game, so when they are processed, the effects that they originate are applied in order to produce the next state. This task is centrally computed for all the players and will be identically for all of them.

   - **Send new game state to players**: When all actions have been executed and the new game state has been calculated, it is sent back to the players. As mentioned above, only that part the player can be aware of is sent to each one of them.

**Player**

1. **Login into the system**
   - **Send login petition to the login server**: Sends to the server his username within the request to enter the game.

- **Receive initial game state**: When the game starts, he receives the initial status of the game

2. **Play the game**: During the course of the game, players will repeat two steps:

   - **Plan actions**: According to the current status of the game in each time, players will select an action to perform among all the possible ones. For automatic players, decision-making algorithms choose the most suitable action to perform depending on the game conditions. These techniques that have been implemented are detailed later on in section 4.5.

   - **Receive state of the game**: Players receive the new game state with the results of all players actions.

### 4.3.2 System context

The system functionality will be centralised into the server that can be connected to several clients that can be either machines or real players.

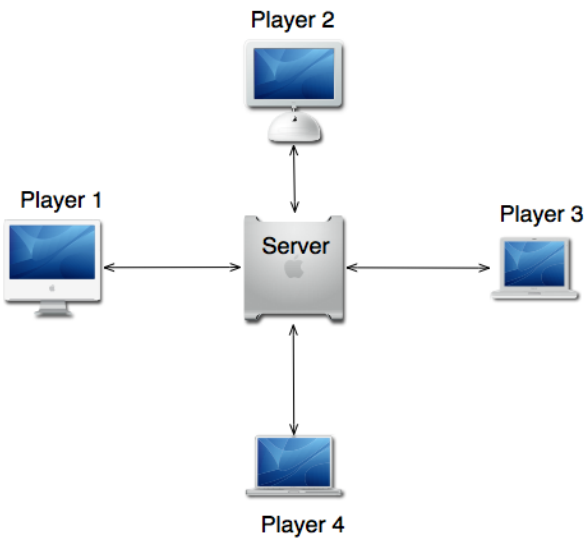



Figure 4.9: System context

For being a client/server architecture, two important elements to define are:

- **Server**: Executing the correspondent processes that allow players to connect and play the game.

- **Player**: It can either be:
  - Human player: He will be able to play through the game interface.
  - AI player: Implementation of one or more AI techniques.

As mentioned before, they are independent applications and they can run at the same or different computers, connected through a local network.

### 4.3.3 Decomposition description

This section analyzes each subcomponent with a table that contains the following information:

| | |
|---|---|
| **Type** | It can take two values: subsystem (independent component) or module (concrete functionality within a subsystem). |
| **Purpose** | Explanation of why this component was implemented. |
| **Function** | Explanation of what this component is performing in the system. |
| **Subordinates** | Subcomponents that are integrated in this component. |
| **Dependences** | Interfaces that are required by this component. |
| **Interfaces** | Interfaces that are supported by this component. |
| **Processing** | Definition of the way it works. |
| **Data** | Data that is used by this component. |

Table 4.38: Decomposition analysis

Starting from the client/server architecture defined in section 4.3.1, two subcomponents are obtained: client and server, that, communicate with each other.

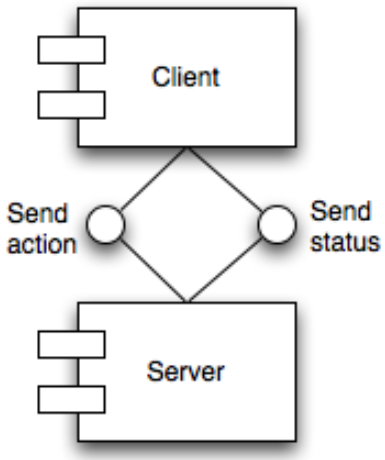



Figure 4.10: Client/Server decomposition

**1. Client**

| | |
|---|---|
| **Type** | Subsystem |
| **Purpose** | Decoupling game logical functionality from its presentation. |
| **Function** | Execution of the game in local computers, presentation of the graphical interface to real players, interaction with real players, implementation of an automatic player and connection to the server. |
| **Subordinates** | GUI Player, AI Player, Model Player, Communication. |
| **Dependences** | Send action (from the player to the server). |
| **Interfaces** | Send status (from the server to the player). |
| **Processing** | Receiving actions from players and sending them to the server, obtaining the status of the game, representing local information, generating automatic plans and strategies and updating players' interfaces. |
| **Data** | Tuples that represents the actions that players want to perform and data sent from the server that represents the current situation of the game at each moment. |

Table 4.39: Client component

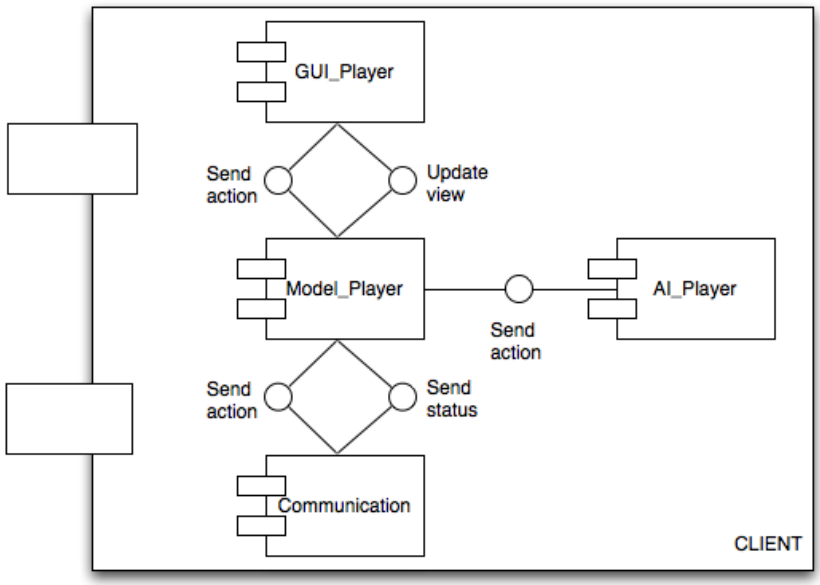



Figure 4.11: Client subsystem

#### GUI Player

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling game logical functionality from its presentation. |
| **Function** | Presentation of the graphical interface to real players and interaction with real players. |
| **Subordinates** | None |
| **Dependences** | Update view |
| **Interfaces** | Send action |
| **Processing** | Receiving actions from the player and sending them to the server and updating players' interfaces. |
| **Data** | Gets data from the model that represents the current situation of the game at each moment and extra information that has to be shown and messages that are sent by the player and represent the actions he wants to perform. |

Table 4.40: GUI Player component

#### AI Player

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling the AI implementation. |
| **Function** | Implementation of an automatic player. |
| **Subordinates** | None |
| **Dependences** | None |
| **Interfaces** | Send action |
| **Processing** | Generating automatic plans and strategies from the current game state. |
| **Data** | Gets data from the model that represents the current situation of the game at each moment and extra information that has to be shown and messages that are sent by the player and represent the actions he wants to perform. |

Table 4.41: AI Player component

#### Model Player

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling the logic functionality of the client from the presentation. |
| **Function** | Representing local information in an structured way, so it can be understood by GUIPlayer and AIPlayer. |
| **Subordinates** | None |
| **Dependences** | Send action and update view. |
| **Interfaces** | Send status (from the server to the player). |
| **Processing** | Processes the information received from the server to update the representation of the game state and creates the messages to be sent to the server. |
| **Data** | The client needs data that represents the current situation of the game at each moment. |

Table 4.42: Model Player component

#### Communication

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling player's presentation and AI implementation of the communication with the server |
| **Function** | Providing an interface that allows the communication between the client and the server |
| **Subordinates** | None |
| **Dependences** | Send action (from the player to the server) |
| **Interfaces** | Send status (from the server to the player) |
| **Processing** | Creating of messages that are sent between client and server in such a way that makes possible the communication |
| **Data** | Information that is sent between client and server |

Table 4.43: Communication component

**2. Server**

| | |
|---|---|
| **Type** | Subsystem |
| **Purpose** | Decoupling game logical functionality from its presentation |
| **Function** | Execution of the logic functionality of the system: create the game and |
| **Subordinates** | Login server, Proxy player, Game server |
| **Dependences** | Send status (from the server to the player) |
| **Interfaces** | Send action (from the player to the server) |
| **Processing** | Receiving messages from the players, execute their actions, update the game state and send it back to the players |
| **Data** | Tuples that represents the actions that players want to perform and data sent from the server that represents the current situation of the game at each moment |

Table 4.44: Server component

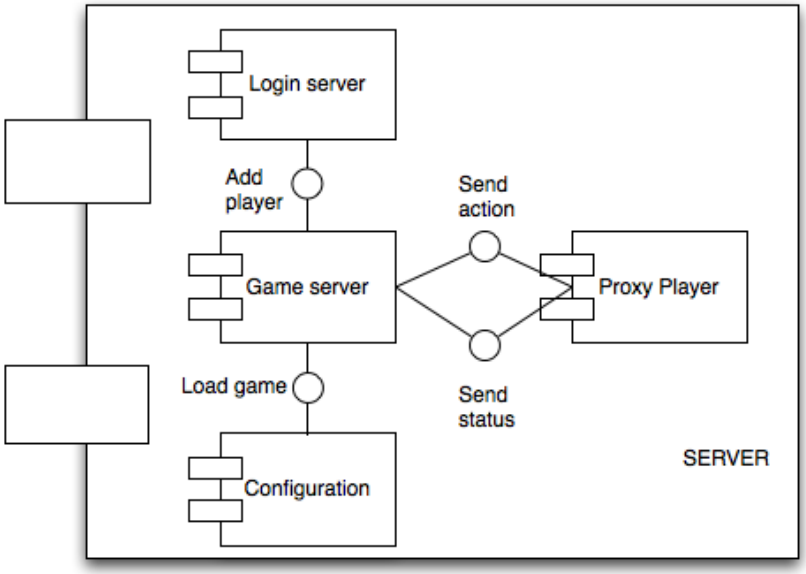



Figure 4.12: Server subsystem

**Login server**

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling game logical functionality from its presentation |
| **Function** | Executing the login process of the player into the game |
| **Subordinates** | None |
| **Dependences** | None |
| **Interfaces** | Add player |
| **Processing** | Receiving the login petition from the client and creating a new process that will execute the proxy player |
| **Data** | Information needed for the registration of the player, such as his username, and the pid of the two processes that execute the client and the proxy player |

Table 4.45: Login server component

**Proxy player**

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling game logical functionality from its presentation |
| **Function** | Acting as an intermediary component between the player and the game server |
| **Subordinates** | None |
| **Dependences** | Send status (from the server to the player) |
| **Interfaces** | Send action (from the player to the server) |
| **Processing** | Receiving messages from the player and filtering the information received from the game server before sending it to the correspondent player |
| **Data** | The data that represents the current situation of the game at each moment and messages that are sent by and to the player and represent |

Table 4.46: Proxy Player component

#### Game server

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling game logical functionality from its presentation |
| **Function** | Execution of logical functionality |
| **Subordinates** | None |
| **Dependences** | Send status (from the server to the player) and load game (to the configuration component) |
| **Interfaces** | Send action (from the player to the server) and add player (from the login server) |
| **Processing** | Executing players' actions, resolving conflicts and updating game status |
| **Data** | Information related to the game, that includes the layout (cell, walls and doors), elements and players' positions and data |

Table 4.47: Game server component

#### Configuration

| | |
|---|---|
| **Type** | Component |
| **Purpose** | Decoupling the configuration of the game of the rest of functionalities of the server |
| **Function** | Creating a new game from the configuration file |
| **Subordinates** | None |
| **Dependences** | None |
| **Interfaces** | Load game |
| **Processing** | Loading the configuration file and creating the necessary structures to initialize the game |
| **Data** | Data that defines the initial state of the game and parameters that are used to created |

Table 4.48: Configuration component

### 4.3.4 Complete design

As summary, it is shown the complete design of the architecture and its decomposition:

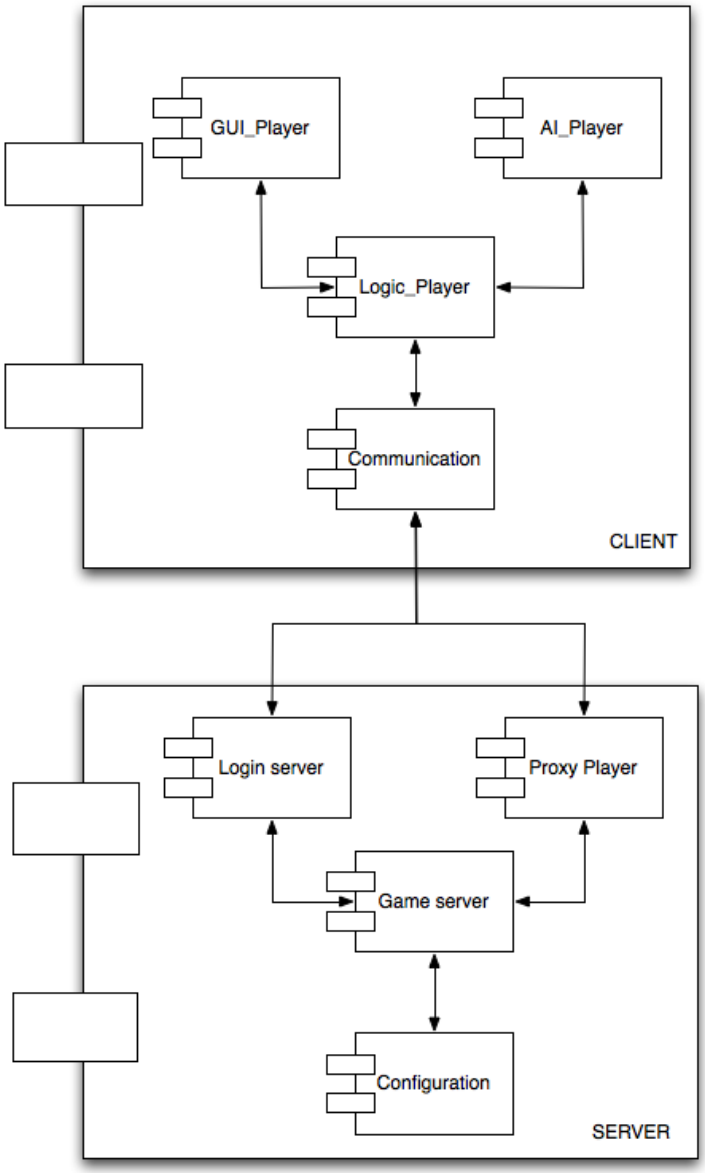



Figure 4.13: Complete design

## 4.4 Detailed design

### 4.4.1 Server

**Decomposition**:

As mentioned above, the logic functionality is executed in the server. In this design, it has been divided in four main modules (see Figure 4.12):

- **Login server**: It is only in charge of the login process. Players will connect to it and it will inform the game server.

- **Game server**: It is the core of the server. It receives and processes players' actions. It stores the information related to the game and it keeps the game state coherent at every time.

- **Proxy server**: There will be several proxies, acting each one of them as an intermediary between each player and the game server. They are in charge of:

  - receiving the actions from the players and sending them to the server

  - receiving the game status from the server and sending to each player the information that each one is allowed to have.

- **Configuration**: It is in charge of loading the game configuration in order to create a new game according to it.

**Sequence diagrams**:

1. **Login**: As it can be seen in Figure 4.14, during the login process, players send their petitions to the login server. The latter creates a process for each one of the players, that is the proxy player, that will connect to the game server in order to add the player to the game. From this moment, the player will not communicate with the login server during the course of the game, but with the proxy server. Finally, the game server will notify the success or failure to the player through the proxy server. Once

all the players have signed up, the game server will send the initial state to each one of the proxy servers, and these latters will send it to each correspondent player.

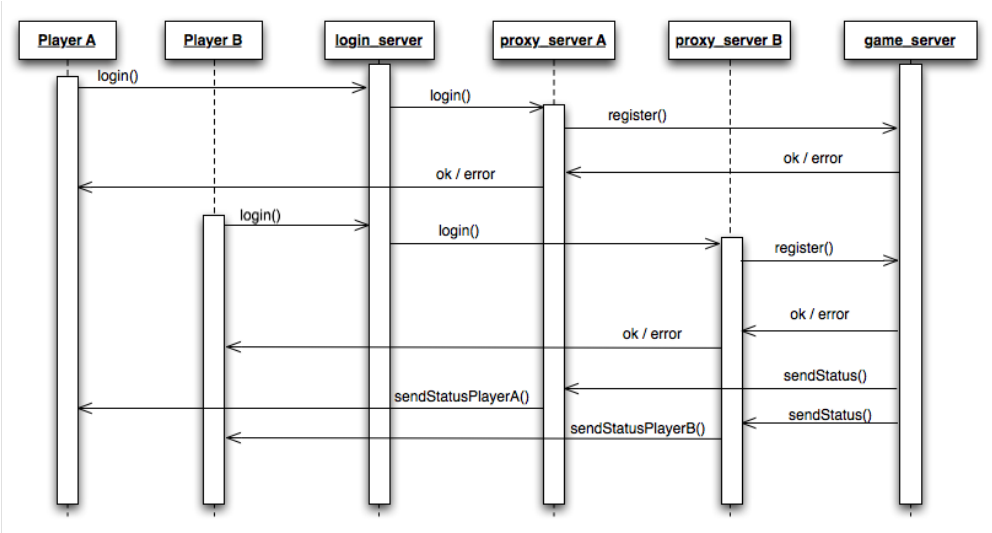



Figure 4.14: Login sequence diagram

2. **Execution**: In Figure 4.15, it can be seen the sequence of steps that are carried out in each cycle in a game for two players.

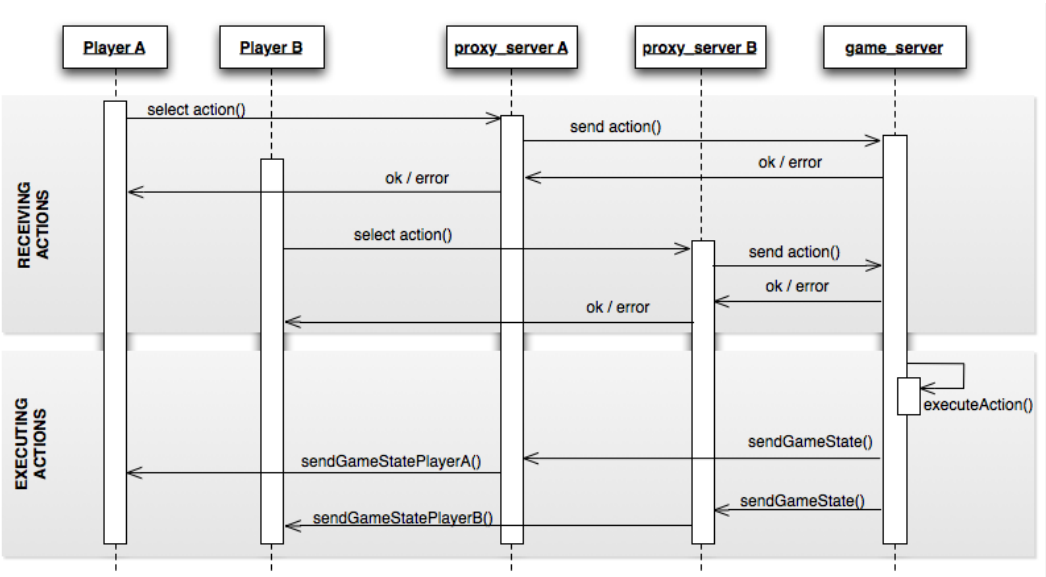



Figure 4.15: Game sequence diagram

Two different phases are followed in each cycle:

- **Receiving actions**: Period in which players can send their actions through their correspondent proxy servers.

- **Executing actions**: Game server executes players' actions and calculates the new state from their outcomes and sends back the result.

### 4.4.2 Player

As it was shown in the architectural design (section 4.3.3), the player component has four subcomponents: GUI Player, Model Player, Communication and AI Player.

1. **GUI Player**

   **Definition**

   It is in charge of the presentation of the application to the players: showing the elements to the users and capturing the events from them in order to perform the actions.

   **UML diagram**

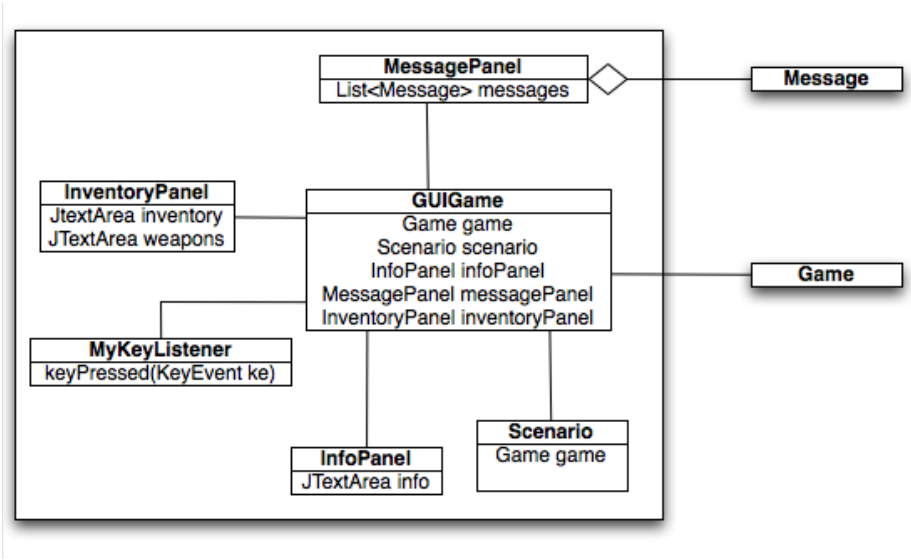



Figure 4.16: GUI player class diagram

**Participants**

| Class | Description |
|---|---|
| `GUIGame` | Panel that acts as a container for the rest of panels. |
| `InfoPanel` | Panel that shows the characteristics of the player, such as life points. |
| `InventoryPanel` | Panel that shows the elements and weapons that the player has. |
| `MessagePanel` | Panel that shows the results of players' actions and the elements they perceive. |
| `MyKeyListener` | Extends `KeyAdapter` class, implementing `keyPressed` method in order to assign different actions to the buttons or keys for the player to use. |
| `Scenario` | Main panel of the game that shows the cells and the elements that are situated in them. |

Table 4.49: GUI Player decomposition

**Sequence diagram**:
When the player wants to do some action, he presses the correspondent key. This event is handled by the class `MyKeyListener` and the action is sent to the model (defined by the class `Game`), that will execute it. Once the model has been updated, it will send the information about the changes to the GUI, that will modify its components.

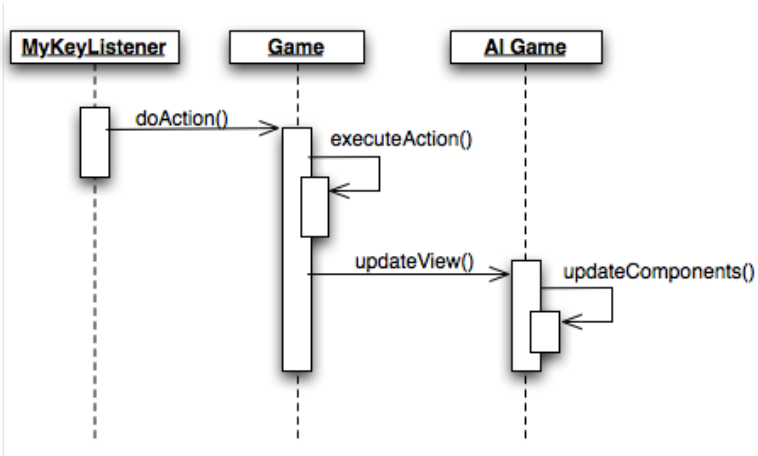



Figure 4.17: GUI Player sequence diagram

## 2. Model Player

### Definition

It represents the information that the player knows about the environment. Its definition is important, as it constitutes the working memory that will be used by the AI algorithms. Therefore, it has to represent the game state in such a way that allows the player to extract the needed knowledge from it.

### UML diagram

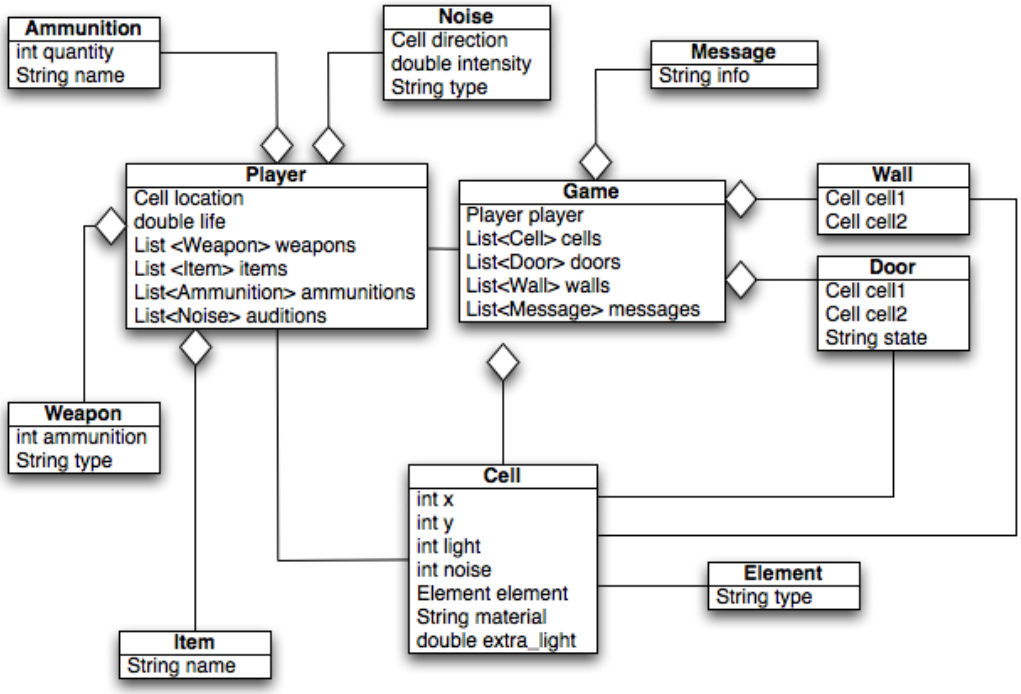



Figure 4.18: Model player class diagram

**Participants**

| Class | Description |
|---|---|
| `Ammunition` | Item that is carried by the player. It is defined by the `name` attribute and the `quantity` represents the amount of ammunition the player has. |
| `Cell` | Each one of the squares or units of the game world that the player is seeing at each moment. |
| `Door` | Door that is situated between two cells ($cell_1$ and $cell_2$) and it is observed by the player. Its `state` defines if it is open or closed. |
| `Element` | Object that is situated in the environment and has been detected by the player. It has an attribute `type` that defines what kind of object it is. |
| `Game` | Representation of the state of the game that is shown to the player. |
| `Item` | Item that is carried by the player. It is defined by the `type` attribute. |
| `Message` | Extra information sent to the player |
| `Noise` | Auditive signal that is perceived by the player. Its `direction` defines where the sound comes from, its `intensity` represents the energy value of the noise and the type shows what kind of sound it corresponds to. |
| `Player` | Representation of the current player: the cell where he is located, the list of weapons, items an ammunitions he has and his life value. |
| `Wall` | Wall that is situated between two cells ($cell_1$ and $cell_2$) and it is observed by the player. |
| `Weapon` | Weapon that is carried by the player and has been detected by the player. It has a type (gun, knife, arch,...) and a number that represents the ammunition it has. |

Table 4.50: Model player decomposition

**3. Communication**

It is in charge of the communication with the server: receiving and sending messages. It is done by using the interface `JInterface`[1], a set of methods that allows the communication between Java and Erlang.

[1] JInterface: http://erlang.org/doc/apps/jinterface/index.html

### 4. AI Player

#### Definition

Player that automatically participates in the game, deciding the best movements to perform. This package is composed by two main classes `MyGame` and `MyGameGUI`. The difference between them is that the latter uses a graphical interface to show the behaviour of the player. Both of them make use of some specific player, being this one an implementation of `MyPlayer`. For this project, the player `CharadePlayer` has been implemented, making use of the techniques explained in section 4.5.2.

#### UML diagram

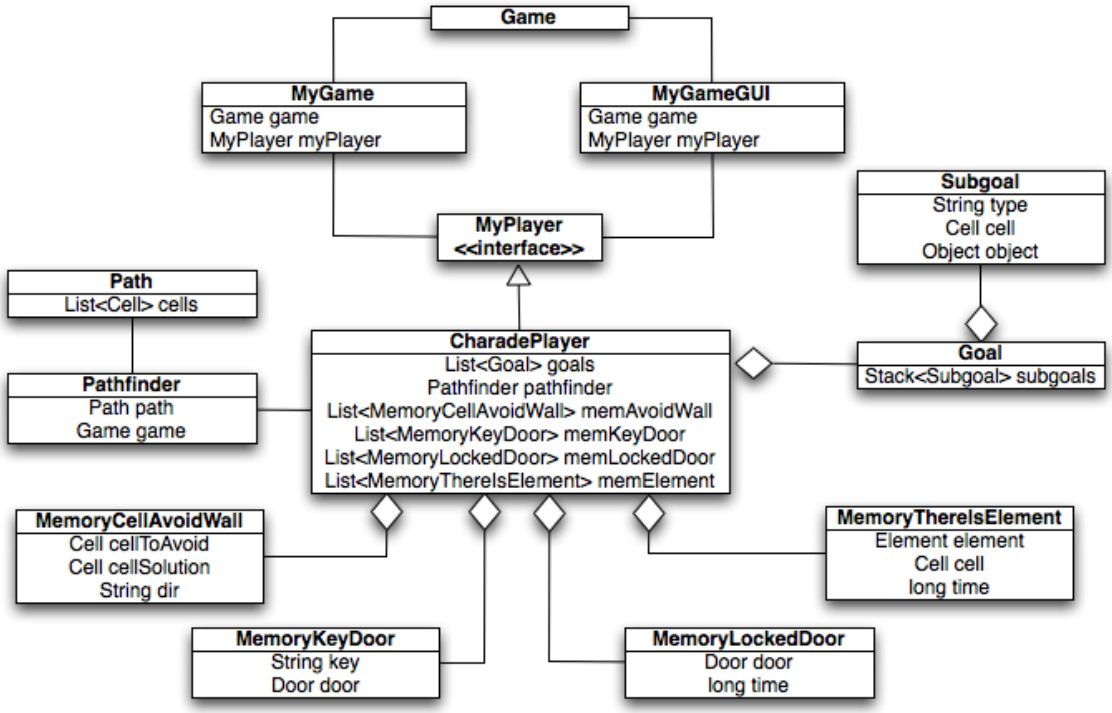



Figure 4.19: Charade player class diagram

**Participants**

| Class | Description |
|---|---|
| `CharadePlayer` | Implementation of an AI player (more information in section 4.5.2 ) |
| `Goal` | Targets that the player has to reach. |
| `MemoryCellAvoidWall` | Representation of a player's memory: which cell he has to go to to avoid some wall. |
| `MemoryKeyDoor` | Representation of a player's memory: which key has been tried with which door. |
| `MemoryLockedDoor` | Representation of a player's memory: which doors have been found locked. |
| `MemoryThereIsElement` | Representation of a player's memory: which elements have been seen and where. |
| `MyGame` | Main class that will be executed to start the application. |
| `MyGameGUI` | Main class that will also show the behaviour of the player through the application's interface. |
| `MyPlayer` | Interface that defines the methods for the automatic players. |
| `Path` | Contains the list of cells that constitute the path. |
| `Pathfinder` | Implementation of Dijkstra's algorithm to find the shortest path between two given points. |
| `Subgoal` | Subtasks that compose the different goals. |

Table 4.51: Charade player decomposition

## 4.5 Implementation

In this section, more concrete details about the implementation are given. The simulation of the perceptions and the signal propagation is explained, as well as the assumptions that have been made and the parameters that have been taken into account. On the other hand, the AI technique implemented is analysed and the decision making algorithm is defined. Also, details about the communication between server and players are given.

### 4.5.1 Simulation of perceptions

Perceptions constitute a very important concept for this project, as they are the only way players have to learn about what surrounds them. They simulate how players would perceive the environment in the most realistic way possible. As it happens in real life, perceptions do not give complete information about what it is happening, but it gives more or less precise and accurate data depending on agent's abilities and environmental conditions.

- **Visual**
  As in real life, it is the most important perception as it will give the most amount of information.

  **Characteristics**:
  So players are able to see the different elements in the world, there are many lights that introduce some visibility in the game. Cells can have their own natural light, but there are some other elements such fire or torchs that give some extra light to the cells where they are situated and also to the adjacent ones. In addition, torches can be lighted and put out as well as carried by players as they are moving. In this environment, lights are mainly characterized by their intensity. No differences of type of light have been made, as this aspect has not been considered important for this game. The main reason of doing that is because lights are not so important as such, they just help players to perceive the important elements of the game. The only thing that might be of interest to the players is the fact that when one light is moving, they may think that it corresponds to a torch that it is being carried by another player.

**Propagation**:

As mentioned before, it is wanted to create the most realistic environment as possible. For doing so, the following algorithm has been implemented in order to perform the propagation of lights and determine the perception of visual elements:

1. **Computation of light propagations**

   (a) **Propagation**: As shown in Figure 4.20, the propagation is carried out from the light focus to the four adjacent cells. Then, from these four is propagated to the adjacent ones and continues like this until the intensity of the light is zero and cannot be propagated anymore. When propagating from one cell to another, the intensity is reduced, but in case that there is a closed door or a wall in between, the light is not transmitted in that direction.

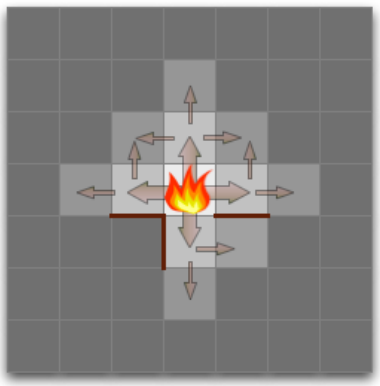


Figure 4.20: Light propagation

   Furthermore it is also checked that there are no blocking elements between the light focus and each cell where it is tried to be transmitted.

   (b) **Computation of new state in the scenario**: After the propagation process, the intensity of each cell is calculated. As types of lights are not differentiated from each other, in case of a collision of several lights, their intensities are added. However, the final value of the intensity will always be between

the minimum (0) and the maximum (9).

2. **Computation of each player's visual perception**
   (a) **Computation of the range of vision for each player** The set of cells the player is able to see depends on his location and the direction he is facing at.

   (b) **Reducing range of vision**The cells that belong to the range of vision are filtered depending on the existence of doors, walls or objects in between. Figure 4.21 shows an example of how cells that are behind walls or closed doors are deleted from player's range of vision.

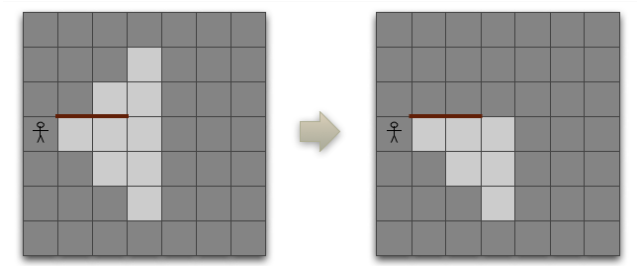


Figure 4.21: Filtering range of vision

For doing so, the idea of Bresenham's line algorithm [2] has been used in order to determine whether it exists or not a line that connects two given points of the board without going through any wall, any closed door or any object.

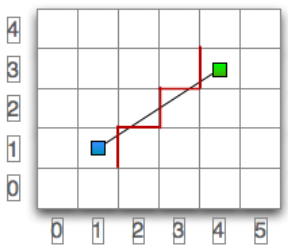



Figure 4.22: Bresenham's line algorithm

[2]Bresenham's line algorithm: http://www.gamedev.net/reference/articles/article767.asp/

As shown in Figure 4.22, the line between the observer and the element is computed. Then, the edges between cells (shown in red in the picture) are checked. If any of them is either a wall or a closed door, the cell is not within the visual range of the player.

3. **Notification to the players of what they are able to see**: At this point, the set of cells that the player is able to see has been calculated, and therefore the elements that are situated in that area is known. However, a last process has to be performed: the perception of the objects does not just depend on being or not in the visual field of the player, there are several factors that have to be taken into account:
   - **Illumination** of the cell where the element is.
   - **Distance** between the player and the element.
   - **Attenuation** from the obstacles that can be in between the observer and the element.
   - **Element's size**: Smaller objects will be more difficult to be perceived.

Therefore, the perception of the elements will be a combination of these four characteristics. This is done in this way because all of them are somehow important to determine how well the element can be perceived. For instance, a player might not see an object that is very close to him if the area is dark. On the other hand, if it is situated a little bit further, but there is more light and the element's size is bigger, the visibility will be better. So, the probability of perceiving an element is calculated from these values in the following way:

$$Probability = \frac{light + distance + size}{3} \ * \ (1\text{- } attenuation)$$

being:

- **Light**: Values from 0 (darkest) to 9 (brightest).
- **Distance**: Values from 0 (the element is the same cell) to the 4 (the furthest point).
- **Attenuation**: Values from 0 (where there are no obstacles in

between) to 1(completely blocked).

– **Size**: It can take one of these three values: small (0.4), medium (0.70), big (1).

When the final value is calculated, the perception will be notified to the player. In order to make the notifications in a consistent way and trying to be as more realistic as possible, the different elements have been organized in a hierarchical structure based on physical similitudes (Figure 4.23). The idea is to give more concrete and better information of the perception (i.e. going down the hierarchical tree) as the value that has been calculated becomes bigger.

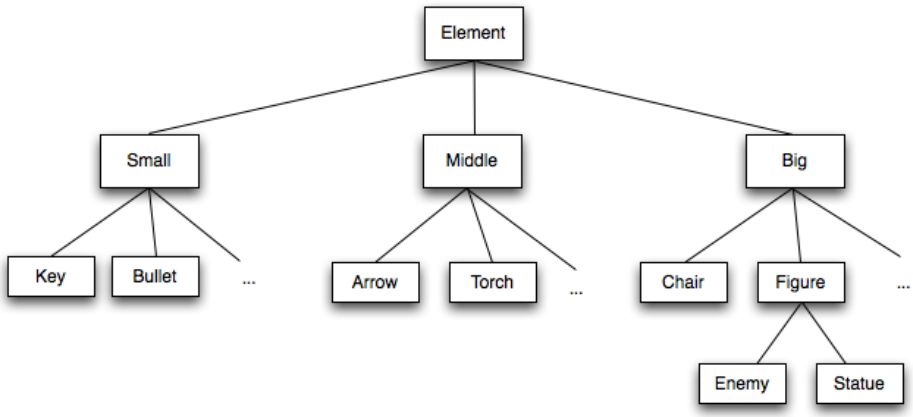



Figure 4.23: Hierarchy of elements

In this way, if the perception of the element is low, it will only be notified to the player whether it is a big, middle or small element. If the perception is better, more concrete information will be given.

$$Perception \geq 0.2 \rightarrow nothing$$
$$0.2 < Perception < 0.5 \rightarrow small/middle/big$$
$$0.5 < Perception < 0.7 \rightarrow small/middle/big/figure$$
$$Perception \geq 0.7 \rightarrow key/bullet/knife/gun/..$$

For instance, if there is an enemy in the visual range of the player, the latter might perceive that there is a big element at some specific location in the

scenario, but he might be unaware of what it is exactly. As he gets closer, he can get a better perception and be able to distinguish a figure. However, he still cannot tell what the element is, as it can whether be an enemy or a statue. However, if the player moves closer, the perception he gets is better, being able to correctly identify what he is perceiving.

- **Auditive**

  It is also an important way for players to learn about the environment. It helps to realize about other player's actions (opening or closing doors, attacking, shouting,..) or getting extra information from places that are not within the range of vision.

  **Characteristics**:

  In this case, three factors have been used in order to represent sounds:

  - **Frequency**: Number of occurrences per second.
  - **Intensity**: Amount of energy that the sound produces
  - **Type**: Distinguishes the source of the sound. It identifies whether is a person walking, a gunshot, a door,...

  For the simulation of the game environment, the following values for these characteristics have been assigned:

| Sound | Frequency | Intensity |
|---|---|---|
| Arrow | 60 | 10 |
| Door | 50 | 20 |
| Gunshot | 100 | 150 |
| Shout | 150 | 90 |
| Steps | 50 | 15 |

Table 4.52: Sound characteristics

  **Propagation**:

  In order to propagate sounds and notify players about them, the algorithm implemented is:

1. **Propagation of sounds as consequence of players' actions**: As it happened with the visual signals, there is a non-directional propagation (fig 4.24).

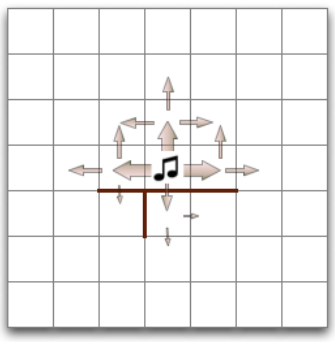
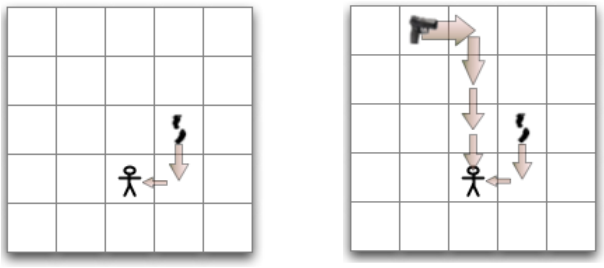


Figure 4.24: Sound propagation

For each sound that is produced, it is propagated to its adjacent cells and from these ones to the adjacent ones and so on. In contrast to the visual signals, sounds also propagate through walls and doors, being the attenuation bigger than between cells with no walls or closed doors.

If the sound has to remain in time (e.g. an echo of a shout), it is inserted into a queue, so the sound can be propagated in the next cycle with reduced intensity.

2. **Computation of new state in the scenario**: As it happened with the light propagation, more than one sound can be perceived from the same position. In this case, a more complex simulation has been made, as different types of auditive signals are taken into account. Figure 4.25(a) shows how the sound from some footsteps is perceived by the player, as he is situated very close to where they are produced. However, in Figure 4.25(b), a gunshot is produced at the same time and although the sound comes from a further point, the player perceives it with higher intensity. So, it has to to be determined whether he is able to hear both or just one of the sounds.



(a) Footsteps propagation (b) Gunshot propagation

Figure 4.25: Collision of sounds

In order to solve this problem and trying to make the model as much realistic as possible, a masking effect has been introduced. This would happen in the real world, as we would not hear somebody's steps if there is a gunshot at the same time. On the other hand, it may be possible to hear both footsteps and a door closing at the same time.

The masking effect can be defined as the fact that auditive signals can be masked those that present a similar frequency and have lower intensity. So, if two sounds (A and B) are propagated to the same point, their perception would be calculated as follows:

*if similar frequencies*
    *if A's intensity > B's intensity*
        *Only A is heard*
    *else if A's intensity < B's intensity*
        *Only B is heard*
    *else*
        *Both are heard*
*else if different frequencies*
    *Both are heard*

The masking effect can be seen in Figure 4.26, in which only signal B is masked by A. Although C has an intensity value that is lower than B's, as its frequency is not very similar to A's or B's, it is not masked by neither of them.

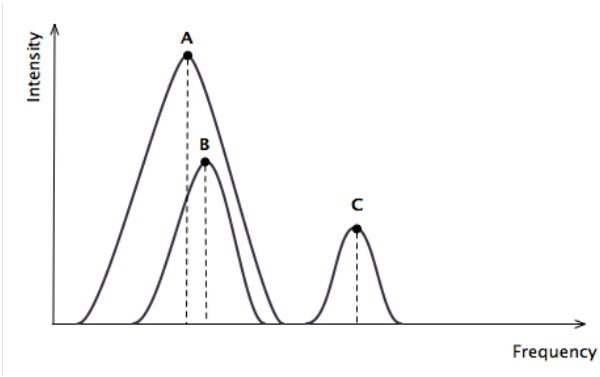



Figure 4.26: Masking effect

3. **The new perceptions are notified to the players**: Depending on the intensity they are heard, they are notified in a different way:
   - **Strong**: If the intensity is high, the player will be notified about the type of noise and direction it comes from
   - **Normal**: If the intensity is medium, only the type of sound will be notified.
   - **Low**: If the intensity is low, the player will only now that there has been a sound.

- **Touch**
  This perception has been added in order to give some extra information when conditions for visual and auditive conditions do not let a good perception of the environment. When a player is located in a cell, he can touch what is at that position in order to learn what type of element it is.

### 4.5.2 AI techniques

1. **Production system**

   For the development of the automatic player, a production system has been implemented. In this section, its main characteristics are explained:

   - **Description**
     Production systems are rule-based. That means that are constituted by

a set of rules composed by a set of preconditions and a set of effects. Once the preconditions are fulfilled, the rules are activated and the respective actions are executed. As shown in section 2.4.5, they represent a simple way of structuring knowledge and can be similar to the human way of thinking. For this project, the technique implemented is based in behaviour modes, allowing the activation of some rules, but not others, depending on the state of the player at each moment and his memory. On the other hand, its simplicity introduces some limitations for some specific situations. For instance, implementation of backtracking algorithms is not allowed as they cannot be implemented within a rule-based technique.

Furthermore, it has been assumed the linearity of the problem when trying to find the solution of this game. That means that different subgoals that belong to the same goal must be satisfied in some specific order. For instance, the sequence: *pick up key, go to door, open door, go to goal cell.* On the other hand, different goals are independent from each other, being possible to try to satisfy one or more at the same time.

- **Decision making**

  The decision-making process is done as follows: once the game has started, the goal list $G$ contains just one goal with only one subgoal: *go_to_goal_cell.* Then, he keeps analysing the preconditions of the rules and activating those that are fulfilled and executing the corresponding action. The pseudocode can be expressed as follows:

```
G = {g0:go_to_goal}
if gi is empty, 0 ≤ i ≤ n
    exit
else
    if precondition_rule1
        choose actionA
    else if precondition_rule2
        choose actionB
    ...
```

When player chooses one action to perform, he might need to update

his behaviour mode or the list of goals. For instance, if the precondition *see_enemy* is fulfilled, the player might set his mode to *attack_enemy*.

Furthermore, the player may need to add new subgoals. For instance, if the precondition *see_key* is true, he would add a new subgoal: *pick_up_the_key*. In doing so, when the player wants to add new subgoals, the list of goals is augmented as follows: for each one of the existing goals, a new goal is created containing the subgoals of that goal and the new subgoal. If the latter is not already included and the resulting goal is not already in the goal's list, it is included. The pseudo-code for this process is the following:

being $sg$ the new subgoal to be added in the goal list $G$:

$\forall g_i \subset G = \{g_0, g_1, ....g_{n-1}\}$
$\quad$ *if* $sg \not\subset g_i = \{sg_0, sg_1, ....sg_{m-1}\}$
$\quad\quad$ $new_g = g_i \cup sg$
$\quad\quad$ *if* $new_g \not\subset G$
$\quad\quad\quad\quad$ $G = G \cup new_g$

For instance, starting from the initial goal, the player might reach a position in which he has to open a door. Therefore, a new subgoal is created. However, it cannot be said that the new goal of the player is just to open the door and go to the goal cell. Instead of that, player will have two goals: the first of them would correspond to *"open the door and get to the goal cell"* and the second goal would just be *"go to the goal cell"*. This is done in this way because player might find a solution without having to go through that door and the fact of having just one goal would imply the player to do both things even if it is not necessary.

The following aspects are also taken into account:

- **Resolving conflict set**: As there might be different rules that can be activated with the same game state, priority values have been given to all of them. So, those that contain a higher priority will be executed before others at same conditions.

- **Cell to go**: A variable `cellToGo` is used in order to determine, in each moment, the cell that player desires to reach. For instance, if player's mode is *go_to_pick_up_item*, it would correspond to the cell where the item is located. In *default_mode* it takes the value of the goal cell.

- **Achievability of goals**: As it corresponds to a dynamic environment, subgoals might not be achievable at some point, so they have to be cancelled. For instance, if the player is going to pick up some item and then finds out that it is not where he thought because somebody already took it, he would cancel this subgoal from his goals list.

- **Replanning**: Sometimes, the player can be more interested in stopping what he is doing to change his path and pick up some item that he suddenly sees. In order to determine if it is worthy for the player to go to pick up weapons, ammunition or power, it has been done by taking into account the distance to the element and its necessity. For instance, if player has a low life value, it will consider to go to a further point to pick up some power. On the other hand, it will not do it if his life value is much bigger, as he would only move to a closer point.

- **Randomness**: There are some situations in which the player has to take some random action as he does not know which action should perform better with the information he owns. In order to avoid situations in which the player moves around the same cells for several cycles, a closed list that contains the last visited cells is used. So, the random movement is limited only to new cells that the player has not recently visited.

  Sometimes, the fact of acting randomly can be the most rational thing to do. Even if a human player was playing, he would end up performing random actions at some points. On the other hand, the necessity of including this randomness in player's behaviour is due to the inefficiency of production systems for particular environments.

- **Elements**
  Two important elements have been defined:
  - **Working memory**
    It is represented by the classes of the *model package* (see section 4.4.2). It represents information related to the elements that the player percieves around the game but also those items he owns. Few examples of these statements are the following:
    * `is_there_element (t)`: Indicates if there is an element of type *"t"* in player's position.
    * `has_ammunition(w)`: Tells whether the weapon *"w"* is loaded or not.
    * `is_door_closed (d)`: Determines if the door *"d"* is closed or not.
    * `has_picked_up (i)`: Indicates if the player has just picked up an item *"i"*.
  - **Rules**
    The following rules have been implemented:
    * if *mode_attack* and *has_ammunition* then **attack**
    * if *mode_go_to_goal* then **move**
    * if *has_empty_weapon* and *has_ammunition* then **reload weapon**
    * if *there_is_element* then **pickup**
    * if *has_torch and torch_is_off* and *in_front_of_fire* then **use torch**
    * if ( *mode_default* or *mode_random* or *mode_avoid_walls*) and *see_weapon_that_needs* then **go to pick up weapon**

  * if ( *mode_default* or *mode_random* or *mode_avoid_walls*) and *see_power_that_needs* then **go to pick up power**

  * if ( *mode_default* or *mode_random* or *mode_avoid_walls*) and *see_ammunition_that_needs* then **go to pick up ammunition**

  * if *follows_path* and *subgoal_still_possible* then **move**

  * if ( *mode_find_key* or ((*mode_random* or *mode_default*) and *has_subgoal(find_key)*) ) and *remembers_key* then **go to pick it up**
  * if ( *mode_find_key* or ((*mode_random* or *mode_default*) and *has_subgoal(find_key)*) ) and *remembers_small_element* then **go to pick it up**
  * if *not mode_unlock_door* and *door_is_closed* then **open door**

  * if *mode_unlock_door* and *has_key_to_try* then **use key**

  * if *mode_default* and *goal_is_at(dir)* and *player_is_at(dir)* then **move**

  * if *mode_default* and *goal_is_at(dir)* and *not player_is_at(dir)* then **turn**

- **Player's behaviour**
  There are three main concepts that define the behaviour of the player:
  - **Subgoal**: Single and concrete purpose of the game. The following have been designed:

    * *Go to goal cell*: Reach the final cell of the game.

    * *Open door*: The player has determined that need to open a door to continue his way.

  * *Find key*: The player has learnt he needs to find a key to open a particular door.

  * *Pick up item*: He needs an specific item from a known location.

  * *Avoid wall*: He has to avoid some wall that is on his way.

  * *Go to cell*: Reach some specific cell of the scenario.

  The most important one is *"go to goal cell"* as once the player has achieve it, he wins the game. The rest of them are used as landmarks, that is, intermediate goals that have to be fulfilled meanwhile he tries to find the solution.

- **Goal**: Combination of different subgoals that lead to the solution of the game. For instance: *"pick up key, open door and go to goal"*. Each goal is composed by a stack of subgoals, in which the last ones must be the first ones to be achieved. Different subgoals are inserted and deleted when they are created and achieved, respectively. Each goal represents a way to achieve the final purpose of the game, but with different subgoals appearing on the way.

- **Modes**: They represent the current behaviour of the player at each moment. While he can have several goals at the same time, his behaviour mode is just one. This concept has been introduced in order to avoid problems when trying to resolve more than one subgoal at the same time.

  * *Default*: Player continues getting closer to the cell `cellToGo`.

  * *Go to goal*: Once the player is able to see the goal cell and has found a path to get to it, he starts moving towards it.

  * *Unlock door*: Player enters this mode when he finds out that the door he wants to open is locked, so he starts trying to unlock it

with all the keys he has. Once he has tried with all of them, he enters the mode *find key*. If the door is unlocked, the mode is set to default again.

* *Find key*: Player tries to find a key around him or remember if he has seen any before. In those cases, the mode is set to *go to pick up item*.

* *Go to pick up item*: Player has decided to pick up some specific item.

* *Attack enemy*: The player enters this mode when he has been attacked or has seen some enemy closer and has ammunition to attack him.

* *Avoid wall*: The player has found a wall on his way, so he is set to this mode in order to find a way to avoid it. The way that this process has been implemented is explained later on in this section.

In this way, a player might have two different subgoals to achieve: *pick up key, go to cell goal*, but as he cannot do both things at the same time. The *behaviour mode* will define the action he is performing at the moment.

In order to choose the next behaviour mode, the list of goals is checked. For instance, if a key is picked up and the player has a subgoal which type is *"open door"*, the new behaviour mode will be *"going to the cell where the door is located"* and then it will probably change to *"unlock door"*.

Furthermore, some modes are more important than others. For example, if player's mode is *pick up an item*, but he is suddenly attacked by another player, his mode will be set to *attack enemy*. This will not happen on the other way around: if the player is decided to attack one of his enemies, he will not stop attacking his enemies because he has found something to pick up. The same

happens with the *go to goal cell* mode. When is set into this mode, he will not stop in order to open some door or pick up something, because if he is in *go to goal cell* mode is because he is following the right path to win the game.

**2. Other implemented techniques**

**(a) Pathfinding**

Due to the fact that player has incomplete information about the environment, he cannot always compute a path to the cell he wants to go. So, the best he can do, as he knows his position and the cell he's going to, is moving towards it. However, when he gets closer and this cell gets inside his range of vision, he is able to determine the best path to get to it. For doing so, **Dijkstra's algorithm** has been implemented. Starting from the position where the player is, he tries to find the shortest path to the destination, taking into account the obstacles that exist on his way. If a path is found, the cells that compose it are stored in a list called `cellsToFollow`, so the player can make the movements following the chosen path. Dijkstra's algorithm has been chosen for this process because the range of possible cells is so reduced, that Dijkstra's implementation has been considered sufficient.

**(b) Player's memory**

Player needs to make use of knowledge that he is learning through the game's course. It might be useful for him to remember where he found a key, a weapon or a power because they can be useful for him later on. It is also important to take note of when it happened, as it is needed in order to determine whether the fact is more likely to continue being true. For this reason, several game facts are going to be stored and some of them will also include the time label in which they are produced. In other cases, it will not be necessary. For instance, if the player learns that a key does not unlock one specific door, it does not matter when he finds out, as it will not stop being true.

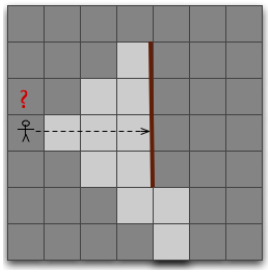
The following situations are stored in player's memory:

- **There is an element**: Player remember where the element was and the moment in which he saw it. For doing so, the next tuple is stored:{*element,x,y,time*}. These memories are updated when they are perceived again.

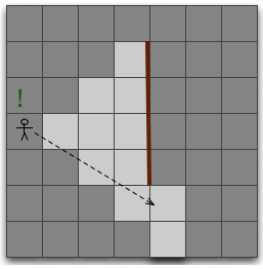
- **The door is locked**. As doors might seem always closed for the player, the latter has to remember whether the formers are closed or locked. In doing so, when player tries to open the door and finds out it is locked, he stores this fact in his memory. As the door can be unlocked by any player, this fact can change, being important to remember the moment in which the player learnt about the locked door. So, the next tuple is stored:{*door,time*}, representing the door that is locked and the moment it was found out.

- **The key does not unlock the door**: Each time the player tries to use a key to unlock a door, the next tuple is stored:{*key, door*}. In this case, the time is not stored, as it a key does not open a door, it will never do.

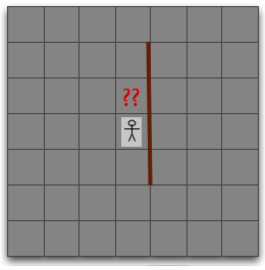
- **Way to avoid a wall**: The player remembers possible escape ways for possible situations in which he must avoid a wall that finds on his way. In Figure 4.27(a), player is walking to the right part of the board and he perceives a wall further on his way. At that moment, as it can be seen in 4.27(b), he looks for a possible way to avoid this wall and stores this information in his memory. In case he reaches that wall, as he does not how to continue, he has to remember the way he learnt before (Figures 4.27(c) and 4.27(d)).



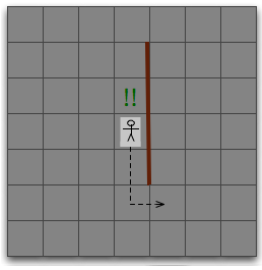

(a) Player sees a wall in front




(b) Player detects a way to avoid it




(c) Player reaches wall




(d) Player remembers what he learnt

Figure 4.27: Avoiding walls

### 4.5.3 Messages

In order to establish the communication, both, server and client, exchange messages. They are formed by tuples and lists that contains the information that is transmitted. Tuples are inserted into "{ }" and lists into "[ ]". Two types are distinguished: those messages that are sent from the client to the server and vice versa.

- **From client to server** As clients are implemented in Java, they make use of the `JInterface` package and send the data to the server as tuples, that are instances of the class `OtpErlang`. Clients need to create two types of tuples: one for the login process and another when sending his action to the server.
  - **Sending login petition** When player wants to connect to the game, he has to send this tuple:
    - login: {*login, username*}
  - **Sending action** When players want to send their actions to the server, they use the same structure for all the different actions: {do username

action} where *username* is replaced by his username and *action* corresponds to a tuple whose structure will depend on the action that is sent:

* attack: {*attack, weapon,direction* }
  where *weapon* represents the weapon that is used to attack and *direction* the direction of the attack.

* close door: {*closeDoor*}

* move: {*move, x,y*}
  where $x$ and $y$ represent the cell where the player is moving to.

* open door: {*openDoor*}

* pickup: {*pickup*}

* reload: {*reload, weapon*}
  where *weapon* represents the weapon that is reloaded

* shout: {*shout*}

* touch: {*touch*}

* turn: {*turn*}

* use object: {*use, item*}
  where *item* represents the item that is used

- **From server to client**
  - **Confirmation messages**: They are sent to the player in order to notify whether it correctly received or not the message from the him.
    * ok: {*ok, message*}

  * error: {*error, message*}
    where message gives more detailed information about server's response.

- **Sending status**: When server sends the game state to the players, it just sends a tuple of tuples as follows:
  {*Message, PlayerInfo, Walls, Doors, Cells, Elements, Noise, HasItems, HasWeapons, Lights,TemporalElements,Messages*}
  where:

  * Message: Represents the state of the game. It can be whether *status* to indicate that the game state is being sent or endgame to specify the game has finished.

  * PlayerInfo: [*x,y,life,direction* ]
    where $x$ and $y$ represent the current location of the player, *life* is the current life value and *direction* is where the player is looking (up, down, left, right).

  * Walls: $\{x_1, y_1, x_2, y_2\}$
    where $(x_1,y_1)$ and $(x_2,y_2)$ represent the cells that are separated by the wall.

  * Doors: $\{\{x_1, y_1, x_2, y_2\}, \{state,material\}\}$
    where $(x_1,y_1)$ and $(x_2,y_2)$ represent the cells that are separated by the door, *state* is the current situation of the door(closed, locked or open) and *material* is the material of the door (wood or metal).

  * Cells: {{*x,y*}, *material, light*}
    where $x$ and $y$ represent its coordinates, *material* is the type of floor (sand, wood or stone) and *light* is the amount of light in that cell.

  * Elements: {*x, y, type*}
    where $x$ and $y$ represents the cell where the element is located and *type* defines which kind of element it is (see Figure 4.23).

* Noise: {{*x,y*}, {*type, ownNoise*,{$from_x$,$from_y$}}}
  where $x$ and $y$ represent the coordinates of the cell where the noise is heard, *type* indicates which kind of noise it is (footsteps, gunshot, shout,...), *ownNoise* indicates whether the noise has been made by this player or not (taking "true" and "false" values respectively) and *fromx* and *fromy* show the direction the noise is coming from.

* HasItems: [{{*name*},*type, info*}]
  where *name* is the name that identifies the item, *type* defines which kind of object it is (key, torch,..) and *info* gives more detailes about it when necessary (in case of the torch, indicates whether is "on" or "off").

* HasWeapons: [{{*name*},*type,ammo*}]
  where *name* is the identifier of the weapon, *type* defines which weapon it is (gun, arch or knife) and *ammo* indicates the ammunition of that weapon (in case of knifes, it is represented as the string *inf*).

* Lights: [{{*x,y*},*intensity*}]
  where $x$ and $y$ represent the coordinates of the cell and *intensity* is the amount of light that is added that cell.

* TemporalElements: [{*x,y,type,info*}]
  where $x$ and $y$ represents the cell where the element is located, *type* defines which kind of element it is (footsteps, smoke,..) and *info* gives more details about it.

* Messages: *message*
  where *message* contains the extra information corresponding to the result of the last action.

## 4.6 Summary

In this section, the proposed game has been defined, as well as its characteristics and problems to be solved. The different elements of the game have been described, as well as the different actions that can be performed by the players.

Furthermore, the architecture of the system has been been described. As it was said, it corresponds to a client/server solution in which both parts are independent from each other and are executed separately. In this section, the description of the different components that constitute both subsystems has also been made

Also, a more detailed design of the implemented solution has been made, showing more concrete elements of the components that were defined before. The different elements and their interactions have been analysed by using UML class diagrams and sequence diagrams respectively.

As a very important part of this section, the development of the system and the implementation of the AI techniques that have been chosen have been described. First of all, the way in which perceptions are simulated has been explained: the algorithm that has been implemented, as well as the parameters that have been used for it. Moreover, the production system that has been implemented for the AI player has been also defined, explaining how the algorithm works and the aspects that have been taken into account for its implementation.

In next section, different experiments will be performed in order to test the efficiency of these implementations.

# Chapter 5

# Experimentation and results

Once the development process has been detailed, in this section a study of the performance is made. For doing so, a few games are executed, facing an automatic player that has been implemented by using the AI techniques explained in section 4.5 against simple automatic players and human players.

For the evaluation of the whole system, two different studies have been made:

- **Performance of the game**
  As the system's implementation is an important part of this project, its performance will be tested. Aspects like *time to execute players' actions* or *time to update the game state* are analysed in this section.

- **Intelligence of the automatic players**
  The AI techniques that have been implemented for this project are evaluated in this section, analysing their performance and suitability for this domain.

For the experimentation, the server is going to be executed in an Intel Core 2 Duo, 2GHz and 2 GB of memory. Players will execute the application from the same computer and from a different one, but connected to the local network.

For the experimental process, different scenarios are defined and several games are executed in each one of them. Different tables and graphs will gather the parameters and results that are obtained in them.

## 5.1 Experiments

In this section, the different experiments used to test the performance of the system and the efficiency of the AI technique are defined. The parameters needed and the results obtained from each one of the experiments are also shown and analysed.

1. **Game's performance**
   In order to evaluate the game's performance, these parameters have been used:

   - Board's size (width × height): Number of cells that the board contains.
   - Number of players that are participating in the game.
   - Number of items than can be picked up by the players.

   The following aspects have been measured:

   - **Initialization**: Configuration of the elements of the game and structuration and initialization of the information. It includes the initial propagation of the lights (such as fires or torches) and it is done once at the beginning of each game, before it has started.
   - **Actions execution**: Process that calculates the next state of the game by executing player's actions, including signal propagations for those actions requiring it.
   - **Status sending**: Period in which the server sends the new game state to all players.
   - **Total**: Time in execute player's actions and return the new state to the players. It includes the two previous operations (actions execution and status sending), as well as some other checking operations.

   For doing the different experiments, several games are going to be simulated combining the different parameters: board's size, number of players and number of items. Simple players that execute a random action in each cycle will

be used for this purpose. The idea is to maximize the number of actions in each cycle, so the worst time for each one of the cases is obtained. Each configuration will run for 100 cycles, measuring the time for each specific task and calculating the average of them all.

**Results**

The following table shows the results that have obtained from the different simulations, in which all parts involved (server and players) have been executed from the same computer:

| Size | Players | Items | Init. (ms) | Exec. (ms) | Sending (ms) | Total (ms) |
|---|---|---|---|---|---|---|
| 15x15 | 2 | 10 | 289,60 | 1,26 | 1,31 | 2,97 |
| 15x15 | 4 | 10 | 222,78 | 1,83 | 2,10 | 4,15 |
| 15x15 | 8 | 10 | 301,12 | 4,02 | 5,30 | 9,54 |
| 20x20 | 4 | 20 | 329,86 | 1,42 | 1,61 | 3,25 |
| 20x20 | 8 | 20 | 365,16 | 5,52 | 3,65 | 9,42 |
| 40x40 | 8 | 30 | 860,91 | 4,50 | 7,9 | 12,64 |
| 40x40 | 12 | 30 | 968,64 | 5,00 | 9,03 | 14,26 |
| 60x60 | 8 | 50 | 926,48 | 5,96 | 5,66 | 11,97 |
| 60x60 | 12 | 50 | 1.034,00 | 6,20 | 6,16 | 12,60 |
| 80x80 | 8 | 75 | 1.625,43 | 7,37 | 5,25 | 12,91 |
| 80x80 | 12 | 75 | 1.721,91 | 15,67 | 8,22 | 27,87 |
| 80x80 | 16 | 75 | 1.801,11 | 526,78 | 48,77 | 667,66 |
| 100x100 | 12 | 100 | 1.863,44 | 708,78 | 75,46 | 923,75 |
| 100x100 | 16 | 100 | 1.832,78 | 1.016,88 | 155,95 | 1.294,31 |
| 100x100 | 20 | 100 | 1.933,12 | 878,25 | 254,67 | 1.388,94 |

Table 5.1: Server's performance

2. **AI techniques**

   In order to evaluate the performance of the implemented automatic player, several games have been executed. The experiments have been divided into two categories: *Single mode* and *Normal mode*. The former consists on executing just the automatic player in some specific scenario in order to check its behaviour and the latter faces the automatic player against simple automatic players and human players to compare their efficiency.

In order to make this evaluation, the number of actions that are performed will be analysed and in some cases, the players' life values will be also taken into account in order to determine their behaviour and performance.

### (a) Single mode

#### i. Configuration 1

**Domain definition**

For this first domain, a simple scenario is created, in which only one door is locked, but the key that opens it is situated very close to it. Next picture shows the scenario that has been used, where the locked door is shown in red and closed ones in orange:

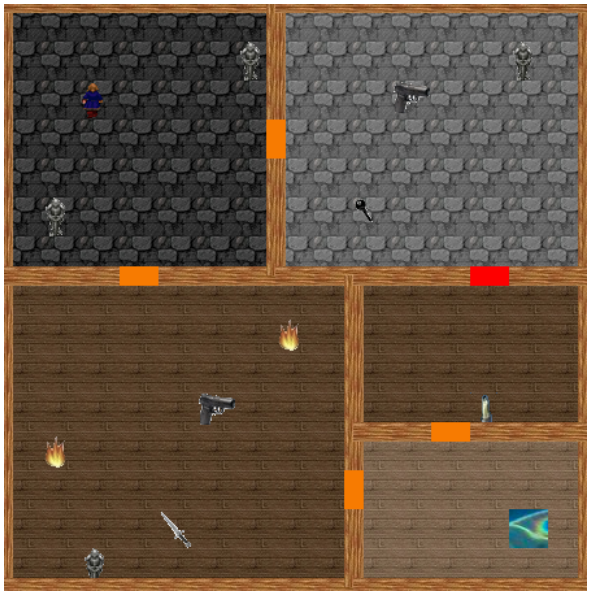


Figure 5.1: Single mode - Configuration experiment 1

**Results**

This game has been executed several times, but the number of actions has remained constant in all of them. It has been **61** the number of steps needed to achieve the goal. For all games, the

same sequence of actions has been carried out.

Furthermore, the initial position of the player has been changed for a second test, facing him to the cell that is below him. This fact has reduced the number of actions to **41**, being kept constant and repeating the same sequence of actions in all the cases that has been executed. This change is due to the fact that for the second case, player starts walking to the cells that are located below him, achieving his goal in less number of steps as he does not find a locked door on his way.

### ii. Configuration 2

#### Domain definition

The domain that has been used for the second configuration is a little bit more complex, as it can be seen in the next picture:

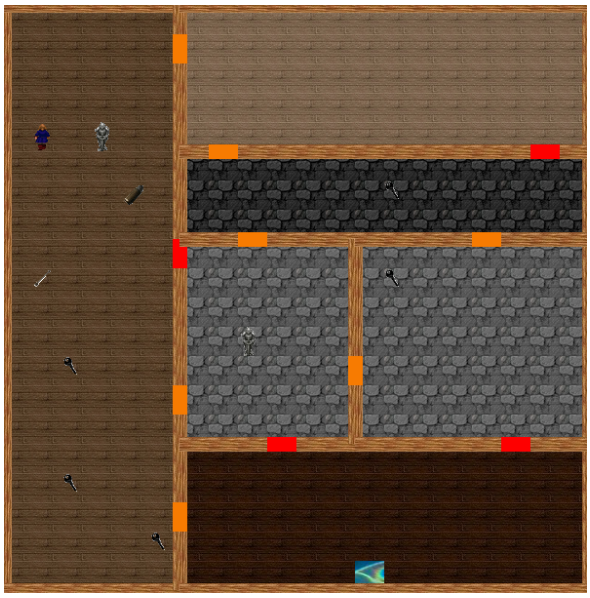


Figure 5.2: Single mode - Configuration experiment 2

In this case, the number of cells and rooms is larger and there are three locked doors that must be opened using the corresponding

keys.

**Results**

A total of 20 games have been played by the automatic player. Contrary to what happened in the previous configuration, different results have been obtained for each game. The next graph shows the number of actions that have been needed to reach the goal for each one of the cases. The green line represents the average.

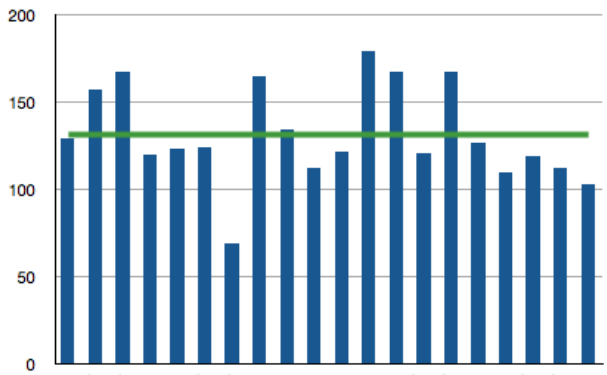



Figure 5.3: Single mode - Results of configuration 2

The differences between the games' results are due to the existence of some randomness in player's behaviour. This makes him to go to different areas of the scenario, needing to perform more or less number of actions depending on what he is perceiving at each moment and the obstacles he is finding on his way. No big differences between the results are appreciated, but this can be due to the simplicity of the game, that does not allow the player to go much further before finding a way to reach the solution.

**(b) Normal mode**

In this mode, the automatic player has competed against a simple automatic player and a also against a human player. For doing the experiments, four different people have played. For each one of the configurations, each human player has played three times. That means

that a total of 12 games have been played in each scenario. It has decided to test with several people in order to get a wider variety of results and also to avoid human players to remember the scenario of the game and therefore take advantage over the automatic player. Moreover, for each game, the starting points of the players were rotating in order not to play the exact game twice and also to give the same chances to all players.

### i. Configuration 1

#### Domain definition

The domain for the first game is very simple. It contains few elements to be picked up and only four rooms without locked doors. The challenge is just to find the path to get to the goal, avoiding obstacles and walls on the way. Figure 5.4 shows a screenshot of the initial state of this scenario, where the three characters represent the initial position of the players and the direction they are facing and the orange elements correspond to the closed (but unlocked) doors.

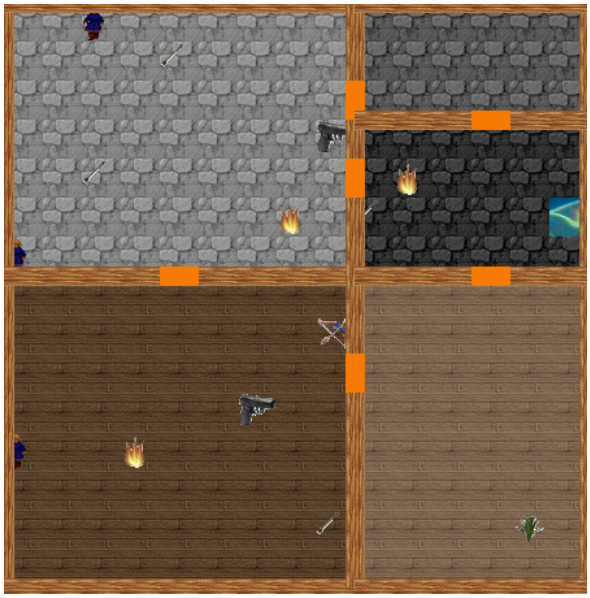


Figure 5.4: Normal mode - Configuration experiment 1

### Results

Table 5.2 shows the results of the twelve played games. Each cell represents the number of actions that each one (AI player, simple player and human player) has taken. The columns indicate the number of the game and the human player (players *a*, *b*, *c* or *d*) that has participated. Those results in boldface represent the winner of each game.

| | **Player a** | | | **Player b** | | | **Player c** | | | **Player d** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** |
| **AIPlayer** | **26** | **30** | 22 | **28** | **26** | 26 | **28** | **26** | **30** | **31** | **26** | 30 |
| **SimplePlayer** | 27 | 31 | 22 | 29 | 27 | 21 | 30 | 27 | 31 | 32 | 27 | 30 |
| **Human player** | 26 | 29 | **21** | 44 | 34 | **28** | 29 | 26 | 30 | 31 | 22 | **35** |

Table 5.2: Normal mode (Configuration 1) - Results

For all the games, players have finished with the same life values as they started, that is 100 points. It has been observed during the experiments that human players tended to discover what they had around them while the automatic player kept following a path to achieve the goal. As there were no interactions between players (attacking or blocking each other) and the solution was pretty straight forward, the AI player managed to win most of the games. It also has to be noted, that those games that the automatic player lost were the third time that human players (a, b and d) were playing. This could have been helped by the fact that players had gained some experience from the two previous games. These results can be summarized in the following table:

| **Player** | **Winning games** | **Average of actions** |
|---|---|---|
| AIPlayer | 9 | 27,42 |
| SimplePlayer | 0 | 27,83 |
| Human player | 3 | 29,58 |

Table 5.3: Normal mode (Configuration 1) - Summary

It can be seen that, although the automatic player has won a higher number of games, the human player has performed a higher number of actions. This could be due to the fact that, as said above, human players tended to go around the scenario, analysing the different ways to go and on the other hand, every action the automatic player took was carrying him closer to the goal.

### ii. Configuration 2

#### Domain definition

The domain in this case is a little bit more complex, as the number of rooms is higher, and also the number of weapons and ammunitions. Figure 5.5 shows a screenshot.

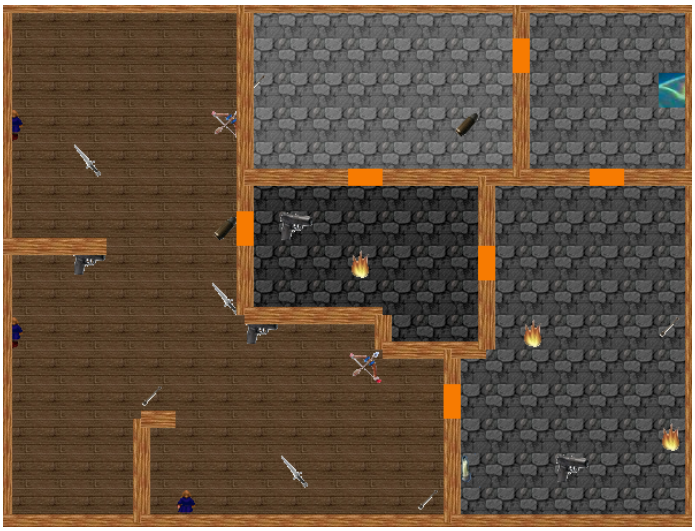


Figure 5.5: Normal mode - Configuration experiment 2

#### Results

The results that have been obtained from the twelve games are shown in the next table, where as shown before, each cell represents the number of actions of each player and those results in boldface represent the winner of each game:

| | Player a | | | Player b | | | Player c | | | Player d | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** |
| **AIPlayer** | * | 57 | * | 90 | * | 75 | **57** | 72 | **73** | 81 | 54 | **72** |
| **SimplePlayer** | 59 | 40 | 71 | 83 | 73 | 76 | 58 | 69 | * | 79 | 52 | 72 |
| **Human player** | **81** | **53** | **91** | **103** | **110** | **94** | 60 | **52** | 75 | **62** | **47** | 64 |

Table 5.4: Normal mode (Configuration 2) - Results

Those cells that contains "*" represent those cases in which the player has lost all his life points and therefore, the number of actions has not been taken into account for the calculations. It can be observed that three out of the nine times that the automatic player lost have been due to being attacked by the enemies. It is worth noting the difference between the number of actions in the different games. In case of the automatic player, this can be due to the existence of some modifications in the environment that led the player to change his path and therefore, take a higher number of actions. On the other hand, in case of the human player, it can be due to differences in players' abilities.

Another important issue is the fact that the AI player has lost all his life points in three out of the twelve games, while this has not happened to he human player in any of the games.

A summary of the results of this configuration is shown in the following table:

| Player | Winning games | Average of actions |
|---|---|---|
| AIPlayer | 3 | 70,11 |
| SimplePlayer | 0 | 66,55 |
| Human player | 9 | 74,33 |

Table 5.5: Normal mode (Configuration 2) - Summary

The number of actions of the human player is also higher than the number of actions needed by the AI player, but in this case the difference has been bigger.

### iii. Configuration 3

#### Domain definition

The next domain is more complex, having added some locked doors and the corresponding keys to open them. A higher number of weapons, elements that can be picked up and obstacles have been inserted in order to introduce a bigger challenge in the game.

Figure 5.6 shows the scenario that has been used, representing with red those doors that are locked and with orange those that are closed.

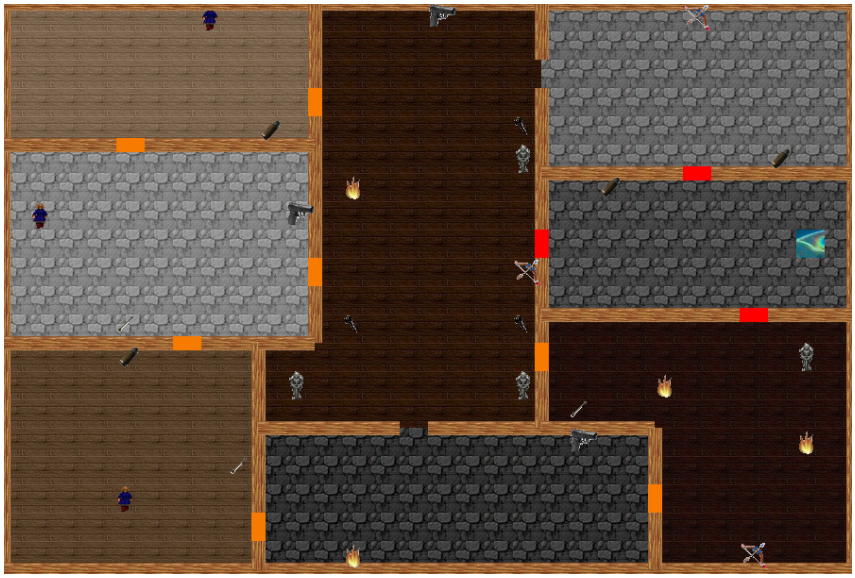


Figure 5.6: Normal mode - Configuration experiment 3

#### Results

Table 5.6 shows the results of the different games played for this configuration:

| | Player a | | | Player b | | | Player c | | | Player d | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** | **1** | **2** | **3** |
| **AIPlayer** | **55** | 80 | **65** | 92 | **108** | **70** | **73** | **110** | **84** | **62** | **116** | **65** |
| **SimplePlayer** | 57 | 53 | 58 | 73 | 94 | 71 | 60 | 74 | 72 | 63 | 117 | 66 |
| **Human player** | 40 | **70** | 46 | **97** | 103 | 80 | 59 | 98 | 69 | 59 | 109 | 64 |

Table 5.6: Normal mode (Configuration 3) - Results

In this game, the number of actions has been a little bit higher, compared to the two previous examples. This is due to the higher complexity of the problem. However, the difference between the number of actions between different games in this configuration is also bigger. This can be caused by the fact of the automatic player needing to perform more random actions, as there is no a straight forward path to the goal cell. It can also be noted that automatic player tends to need more steps for the second game of each human player.

| Player | Winning games | Average of actions |
|---|---|---|
| AIPlayer | 10 | 81,67 |
| SimplePlayer | 0 | 71,50 |
| Human player | 2 | 74,50 |

Table 5.7: Normal mode (Configuration 3) - Summary

Contrary to what happened in the two previous configurations, in this case the average of actions that are needed by the AI player is much higher than the number of actions performed by the different human players. However, he still gets better results, as he has managed to win 10 out of 12 games under this configuration.

## 5.2 Results

1. **Game's performance**

   In table 5.1, it can be seen the good performance of the server. The task that needs most time is the initialization, but it only takes almost 2 seconds for a 100x100 scenario. As it is only executed once, before the game has started it has been considered that it is reasonable and adequate time.

   In order to create a continuous game, players should not notice the existence of different cycles of execution. For this reason, the time of receiving actions and sending back the results should not take too long. It has been determined that this time should not exceed 500 ms. For this reason, the maximum size that can be accepted is 80×80 with 8 games, as it maintains low time values.

2. **AI techniques**

   The results of the automatic player in the *single mode* games show some of the advantages and deficiencies of the production systems.

   In the first example, it is proved that player can find a good solution to reach the final goal. On the other hand, it can be observed, that for the same scenario, with the same information, the player will always act in the same way, being totally predictable for similar situations.

   In the second game, it can be seen how the player is able to deal with situations in which the uncertainty is higher (for instance, when he reaches a wall and has to go around to find a way to avoid it), but his efficiency depends on how lucky he is when performing random actions. However, this would also happen to a human player, when he finds some problem and has to move around until he finds a way to solve it.

   Despite of having implemented a simple AI technique, its results against human players are competitive. In the first example, corresponding to a simple domain, the goal has been achieved in a quite straight forward way, obtaining good results in most of the games.

In the second game, there have been more problems. This can be due to a bigger interaction between players, creating a very dynamic environment in which the AI player did not know how to react.

In the third game, he has obtained good results, even though the domain was more complex, having to avoid more number of walls, obstacles and locked doors.

The next diagram shows the number of games that each automatic and human players have won in each of the three scenarios that have been used for experimentation:

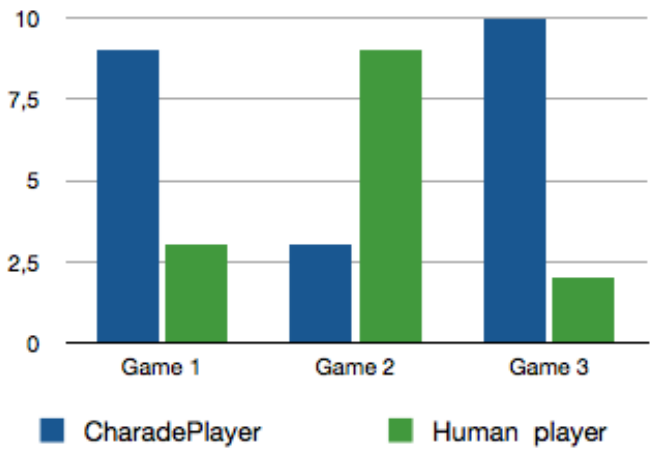



Figure 5.7: Summary - Results

In summary, despite of the simplicity of the technique, it has obtained good results, mainly in the first and third configuration, in which has won most of the games.

# Chapter 6

# Conclusions and future work

This section presents the conclusions obtained after this project's development, as well as a list of lines of future research.

## 6.1 Conclusions

One of the main characteristics of the game developed for this project is the fact that it contains a huge amount of incomplete and imprecise information. Players have to discover the scenario as they interact with it, receiving the information by their perceptions. Furthermore, this knowledge might not be very accurate, as it depends on several factors such as light, distance, obstacles,...(as explained in section 4.5) in case of the visual perception. So, players need to be able to deal with imperfect and incomplete data in order to choose the best action to perform.

It also has been very important for its proper performance the periodic update of information, so players know at each moment the correct state of the game in order to try to make the best action as possible according to it. For this reason, the server has to keep sending the current status of the game and cannot take too long in doing it.

Few simplifications have been made:

- Dealing with temporal information. It has been determined that it is not important to remember everything that the player is perceiving at each moment: some might not be relevant to be remembered (such as fire or statues' locations) and some could be very dynamic, such as the enemies' situation.

- Dealing with uncertain situations. At some points of the game, the player has to perform a random action. Sometimes it can be the best thing to do, but some other times it is due to the inefficiency of the production systems.

In summary, the system implemented can be described as:

- Portable system: A multiplatform system has been developed. Both, server and client, can run in any of the different operative systems.

- Expandable system: It can be very easy to add new automatic players, just by implementing the `AIPlayer` and `GUIPlayer` interfaces as it is explained in Appendix A.4 and also to add new elements and functionalities to the game.

- System's performance: According to the results obtained in section 5.1, the performance of the server is quite good. It takes an appropriate amount of time to perform the different operations in each cycle. On the other hand, the interface implemented for the players is not very optimal, as it takes too long to refresh. However, this fact has not been taken into account for the development of this project, as it corresponds only to a prototype, being the game's engine the most important part.

- Parallel execution: The task of filtering the information to be sent to the players is done in different processes, one for each player, being each one of them completely independent from the rest.

- Realistic environment: Visual and auditive propagations have been implemented in a simple, but the most realistic possible way.

On the other hand, a minimally competitive agent has also been created. Implemented as explained in section 4.5.2, it corresponds to a simple technique that deals with the information that receives from the server and is able to determine the next action to take according to the current game state. It shall be mentioned

that the player has obtained good results for the experiments, as it has managed to win 61% of the games against human players.

One important aspect on player's functionality has been the way of dealing with multiple goals and subgoals and also the fact of setting the current behaviour mode once a subgoal was achieved. Lineality was assumed in this case in order to simplify the process of goal selection.

In summary, production systems constitute a simple AI technique that can deal with many of the situations that have appeared during the experiments. However, they might not be the most suitable technique for a domain like this one, which can be very dynamic and due to production systems' deficiencies (not very fast to react and response to sudden events, completely predictable, very deterministic and not very easy to expand) can be difficult to handle.

## 6.2 Future work

In this section, a few ideas or guides that can be used to extend and improve this work are going to be given:

**System implementation**

For this project, a protoype has been implemented, but there are many aspects that can be added or improved:

- Creation of new elements, including new weapons, new items that can be used, etc

- Adding new characteristics to the players, such as strength, speed or stealth.

- Improvement of the interface. For this prototype, AWT has been used, but a better one should be used for a proper version of the game as it can get very slowly.

### AI implementation

The creation of a player for this domain can include many different aspects and their implementation was out of the scope of this project, as it had to be simplified.

For this reason, there are many lines that can be followed to improve the performance of the player. Some of them can be:

- **Planification**: Improvement of the way that goals and subgoals are handled and implementation of a proper planner algorithm.

- **Adversarial search**: Instead of acting just to achieve their personal goal, players can realize that in fact they are competing against other players and therefore, try to prevent them from wining the game.

- **Time inference**: It can be taken into account the time in which past events happened in order to determine the credibility of the facts that the player has stored in memory.

- **Handling uncertainty** : Development of a more sophisticated probabilistic method of reasoning.

# Chapter 7

# Planification and budget

In this section, the planification of the development of this project is analysed. For doing so, the initial planification is shown and subsequently compared to the real evolution of the development of this project, pointing out those aspects that were not carried out within the expectations. Finally, an economical anlysis is made.

## 7.1 Planification

This is an analysis of the tasks that have been performed, their time distribution and the comparison between their real development and the initial one. In doing so, Gantt's diagrams have been used to show the time periods for each one of the following tasks:

1. **Game definition and design**: Definition of the problem to solve: the scenario, the elements and the characteristics of the game

2. **Game implementation**: Development of both, server and client, applications. It involves the implementation of the different components of each one and the communication between them.

3. **Game testing**: During this process, the system's implementation is checked, detecting if there is any error or anything that could be improved. Different domains are tested to check the existence of errors and prove that the objectives have been fulfilled.

4. **AI techniques research**: Study of the existing technologies that solve similar problems and their suitability for the domain to be solved.

5. **AI techniques implementation**: Development of the AI techniques that have been chosen.

6. **AI testing**: Comprobation of the AI technique efficiency in order to detect errors or features that could be improved.

7. **Experimentation**: Study of the performance of the system's performance and the efficiency of the AI techniques that has been developed. It includes the definition of the different domains to be used and the correspondent tests of each one of them.

8. **Documentation**: Composition of this document.

9. **Presentation** Preparation of presentation of this project.

It is worth noting that the methodology followed has been based on evolutionary prototypes. Therefore the initial idea has been the repetition of the steps 1, 2 and 3, obtaining as a result of them an incremented prototype, that could be fixed or extended in next iterations. That is the reason why, for the initial planification, these three steps seem to be carried out at the same time. Furthermore, the implementation of the AI techniques is also executed along with the testing process (step 6).

### 7.1.1 Initial planification

| Id | Name | Duration | Start date | End date |
|---|---|---|---|---|
| 1 | Game definition and design | 4 weeks | 01/12/2009 | 31/12/2009 |
| 2 | Game implementation | 8 weeks | 01/01/2010 | 28/02/2010 |
| 3 | Game testing | 2 weeks | 15/02/2010 | 07/03/2010 |
| 4 | AI techniques research | 2 weeks | 08/03/2010 | 21/03/2010 |
| 5 | AI techniques implementation | 6 weeks | 22/03/2010 | 07/05/2010 |
| 6 | AI testing | 2 weeks | 01/05/2010 | 14/05/2010 |
| 7 | Experimentation | 1 week | 15/05/2010 | 21/05/2010 |
| 8 | Documentation | 7 weeks | 07/05/2010 | 30/06/2010 |
| 9 | Presentation | 2 weeks | 01/07/2010 | 15/07/2010 |

Table 7.1: Initial planification

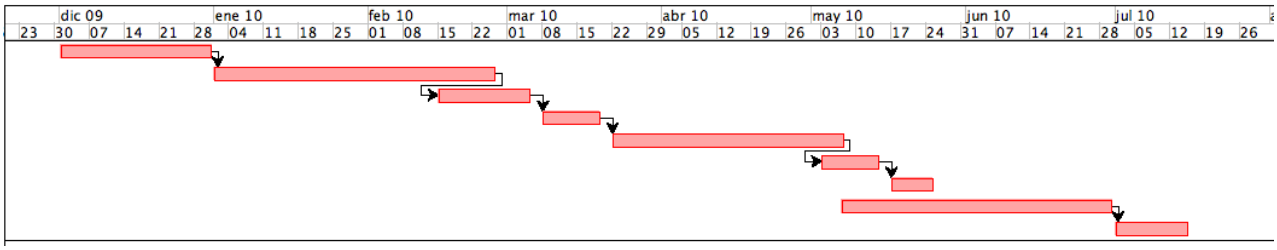



Figure 7.1: Initial planification diagram

### 7.1.2 Real planification

| Id | Name | Duration | Start date | End date |
|---|---|---|---|---|
| 1 | Game definition and design | 6 weeks | 01/12/2009 | 14/01/2009 |
| 2 | Game implementation | 12 weeks | 15/01/2010 | 15/04/2010 |
| 3 | Game testing | 3 weeks | 01/04/2010 | 21/04/2010 |
| 4 | AI techniques research | 2 weeks | 22/04/2010 | 07/05/2010 |
| 5 | AI techniques implementation | 4 weeks | 08/05/2010 | 07/06/2010 |
| 6 | AI testing | 2 weeks | 01/06/2010 | 14/06/2010 |
| 7 | Experimentation | 2 weeks | 15/06/2010 | 30/06/2010 |
| 8 | Documentation | 14 weeks | 01/04/2010 | 15/07/2010 |
| 9 | Presentation | 2 weeks | 07/07/2010 | 21/07/2010 |

Table 7.2: Real planification

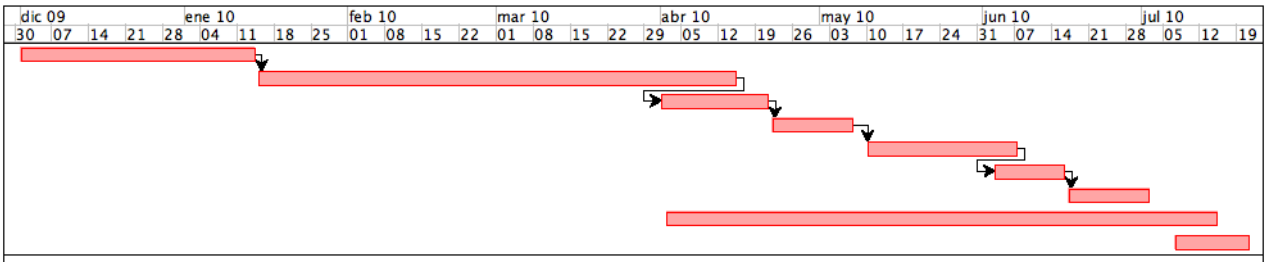



Figure 7.2: Real planification diagram

The development of this project has changed considerably from the initial planification. The implementation of the game required much more time that expected. However, the implementation of the AI techniques did not take as much time as it was thought in the beginning. Furthermore, the composition of this document

started before planned, needing less time at the end of the project development.

## 7.2 Technical equipment

In this section, different resources, software as well as hardware, that have been used for the development of this project are detailed.

### 7.2.1 Hardware

- Macbook 2 Ghz Intel Core 2 Duo, 2GB DDR3 RAM
- PC 2,2 Ghz, 3Gb RAM

### 7.2.2 Software

- **Operative Systems:**
  - *Microsoft*® Windows XP
  - *Apple*® *Mac*™ OS X Leopard
- **Programming environment:**
  - *Eclipse*™ Ganymede (The Eclipse Foundation)
- **Programming language:**
  - *Java*™ (Sun Microsystems)
  - Erlang
- **Additional libraries:**
  - JInterface
- **Text processor:**
  - TeXshop
- **Diagrams and figures:**
  - *OmniGraffle*™ Professional (*TheOmniGroup*®)
  - *GIMP*™
  - *OpenProj*™

- **Presentation:**
  - *Apple*® *Keynote*™ 09

## 7.3 Economical analysis

This sections presents an economical analysis of this project, presenting a comparison between the initial estimation and the initial one.

### 7.3.1 Methodology

In order to calculate the costs of this project, the resources are divided in three groups:

- **Human resources**: Personal costs will be calculated taking as reference for the salaries the studied done by Hays Group, whose results have been provided by *"Asociación de Ingenieros e Ingenieros Técnicos en Informática"* [1].
- **Hardware resources**: In order to make an estimation of the cost of the hardware components that have been used, it will be done by taking into account the cost of the acquisition of the material, its estimated useful life and the time that has been used for this project.
- **Software resources**: As it happened with hardware resources, the acquisition cost of the licenses, the useful life and the time that has been used will taken into account.

### 7.3.2 Estimated cost

1. **Human resources**
   Although this project has been developed by one person, it has been also taken into account the job of supervision carried out by the supervisors, that will be considered as the 5% of the development time. For the latter, it has been considered two different work timetables: during the first four months, 20 hours per week were devoted for this project and during the second four months, it was increased up to 40 hours per week.

| Person | Estimated hours | Cost (euros/hour) | Total cost |
|---|---|---|---|
| Developer | 960 | 17,32 | 16.627,20 |
| Supervision | 48 | 24,35 | 1.168,80 |
| | | **Total** | **17.796 €** |

Table 7.3: Estimated human resources costs

2. **Hardware resources**
   Next table shows the cost of the hardware used for this project (see section 7.2.1):

| Resource | Cost(€) | Estimated useful life (months) | Estimated use in the project (months) | Cost for the project |
|---|---|---|---|---|
| Macbook | 1.099 | 48 | 3 | 68,68 |
| PC | 500 | 36 | 3 | 41,66 |
| | | | **Total** | **110,34€** |

Table 7.4: Estimated HW costs

3. **Software resources**
   The cost of the software used for this project (see section 7.2.2) is shown in the following table:

| Resource | Cost(€) | Estimated useful life (months) | Estimated use in the project (months) | Cost for the project |
|---|---|---|---|---|
| *Apple*® *MAC*™ OS X Leopard | 108.36 | 48 | 3 | 6.77 |
| *Eclipse*™ | 10 | 12 | 5 | 4,17 |
| *Microsoft*® Windows 7™ | 300 | 36 | 5 | 25 |
| *OmniGraffle*™ Professional (*TheOmniGroup*®) | 140 | 12 | 3 | 35 |
| GIMP | 10 | 12 | 3 | 2,50 |
| *OpenProj*™ | 10 | 12 | 3 | 2,50 |
| | | | **Total** | **75,94€** |

Table 7.5: Estimated SW costs

4. **Summary**
   Table **??** shows the predicted total costs for this project, according with what has been calculated before:

| Concept | Cost(€) |
|---|---|
| Human resources | 17.796 |
| Hardware resources | 110,34 |
| Software resources | 75,94 |
| **Total estimated cost** | **17.982,28€** |

Table 7.6: Estimated total costs

### 7.3.3 Real cost

The difference between the estimated and real budget is due to the increase of the cost in human resources. This is due to the increase of the number of hours used in the development of this project. Both, software and hardware resources, have not been affected from the estimated costs as there have been no changes in the resources that have been used. Therefore, the increase of the budget does not differenciate much with the estimated one.

**Human resources**

| **Person** | **Estimated hours** | **Cost (euros/hour)** | **Total cost** |
|---|---|---|---|
| Developer | 1200 | 17,32 | 20,784 |
| Supervision | 60 | 24,35 | 1.461 |
| | | **Total** | **22.245€** |

Table 7.7: Real human resources costs

**Summary**

| **Concept** | **Cost(€)** |
|---|---|
| Human resources | 22.245 |
| Hardware resources | 110,34 |
| Software resources | 75,94 |
| **Total estimated cost** | **22.431,28€** |

Table 7.8: Real total costs

# Appendix A

# Reference manual

In this section, the steps needed to execute the system are explained.

## A.1 Installation of work environment

In order to run the system, it is needed to install Java for the client's application and Erlang to execute the processes on the server's side.

### A.1.1 Java installation

In order to install Java, it can be downloaded from Sun's website[1], following the instructions that are specified

- In order to execute the client's application, only JVM 5 or later will be needed.
- If a new player wants to be created for the game, also JDK 1.5 or later has to be downloaded and installed.

### A.1.2 Erlang installation

For this version, R13B03 release has been used. It can be downloaded from Open Source Erlang's website [2] and follow the steps that are indicated to proceed with the installation.

[1] Sun's website: http://java.sun.com
[2] Erlang's website: http://www.erlang.org/download.html

## A.2 Setting up the server

### A.2.1 Starting the server

In order to execute the server, the following steps must be carried out:

1. Download *"server.rar"*

2. Uncompress the *rar* file.

3. Go to the server directory *"org/charade/game/server"*

4. Execute *erl -setcookie galleta -sname sakonia*

5. Once Erlang has been started, the following lines have to be inserted:

   *Pid=spawn(login_server,start,[]).*
   *Pid2=spawn(game_server,start,[]).*
   *register(connection,Pid).*
   *register(server,Pid2).*

At this point, the server is ready to receive petitions from the players and once they have joined, start the game.

### A.2.2 Configuration file

In order to set the parameters of the game, an XML file is used. This file is stored as *"org/charade/game/server/map.xml"*. The structure of this document has to be specified as follows:

```
<map>
    <walls>
        <wall x1="" y1="" x2="" y2="">
    </walls>
    <doors>
        <door state="" material="" x1="" y1="" x2="" y2="">
    </doors>
    <items>
        <item x="" y="" name="" type="" opt="">
    </items>
    <weaponInfos>
        <weaponInfo type="" distance="" strength="" noise="">
    </weaponInfos>
    <ammunitions>
        <ammunition x="" y="" name="" weapon="" amount="">
    </ammunitions>
    <powers>
        <power x="" y="" type="" amount="">
    </powers>
    <weapons>
        <weapon x="" y="" name="" type="" amount="">
    </weapons>
    <obstacles>
        <obstacle x="" y="" type="" atenuation="" light="" damage="">
    </obstacles>
    <useItems>
        <item="" x1="" y1="" x2="" y2="" action="">
    </useItems>
    <cells>
        <cell light="" noise="" material="" x="" y="">
    </cells>
</map>
```

### A.2.3 Defining a new scenario

A graphical tool has been developed in order to easily define new domains for the game. For doing so, the following steps have to be followed:

1. Download *"CreateWorld.rar"*.

2. Uncompress the *rar* file.

3. Go to *CreateWorld/* directory.

4. Execute *java -jar CreateWorld.jar width height*
   where *width* and *height* represent the size of the board

   The following window will be displayed:

Figure A.1: CreateWorld screenshot

5. Load an existing configuration: If the scenario is not created from scratch and it is wanted to load an existing one, this is done by clicking on the *Load XML* button and choosing the correspondent XML file.

6. Create the elements by using the interface: In order to add new elements to the scenario, different options can be used:

   - Add normal cell:
     (a) Select *"Add normal cell"*
     (b) Choose light value
     (c) Choose material
     (d) Click into the scenario at the point where it wants to be added.

   - Add goal cell:
     (a) Select *"Add goal cell"*
     (b) Click into the scenario at the point where the goal cell wants to be added. It has to be taken into account that only one goal cell can be created, so if it has been defined before, it will be deleted.

   - Add a door or wall:
     (a) Select one option: *"Add wall"* or *"Add door"*(*open*, *closed* or *locked*)
     (b) Click into the scenario at the point to create the element. It has to be taken into account that only one element is created at the same position. Therefore, if an element is added, but there is another one at the same location, this one will be deleted.

   - Delete a door or wall:
     (a) Select the option *"Delete wall or door"*
     (b) Click into the scenario at the point to delete the element.

   - Add an element:
     (a) Select the correspondent option *"Add element"*, *"Add weapon"*, *"Add ammunition"*, etc.

(b) Choose the specific item to add( *key*, *torch*, etc)

(c) Click into the scenario at the point to create the element. It has to be taken into account that only one element is created at the same position. Therefore, if an element is added, but there is another one at the same location, this one will be deleted.

- Delete an element:
  (a) Select the option *"Delete element"*
  (b) Click into the scenario at the point to delete the element.

7. Once the game is defined, by pressing the *"Save XML"* button and selecting where to save it, the configuration file is automatically generated.

## A.3 Playing the game

1. **Installing the application**
   In order to install the game, the following steps must be carried out:

   (a) Download *game.rar*.

   (b) Uncompress the *rar* file.

   (c) Go to *charadeGame/* folder.

   (d) Depending whether a human player wants to play or an automatic player is used, one of these three commands are executed

      i. To run the application for a human player to use it:
      *java -jar Game.jar username*

      ii. To execute an automatic player implemented in the class *ClassPlayer*.
      *java -jar AIGame.jar ClassPlayer username*

iii. To execute an automatic player implemented in the class *ClassPlayer* and see it through the graphical interface.
*java -jar AIGameGUI.jar ClassPlayer username*

**2. Executing the application**

In Figure A.2 a screenshot is shown. Different elements can be appreciated:

**Scenario**: Main panel in which the action is shown. It can be seen the current player's icon and the rest of elements. Furthermore, there is a blue cell that will be visible at every time and represents the goal point that the player has to reach to win the game.

**Information panel**: Situated at the top of the screen, shows the life points of the current player and the weight he is carrying.

**Inventory panel**: Located at the right-hand side of the screen, it includes two different lists:

- **Items** that player has and can use. Some of them show extra information, like torches, that indicate whether they are on or off.
- **Weapons** that player has and can use to attack. The number of ammunition of each one of them is shown in brackets.

It has to be notice the symbol "X", that indicates the item or the weapon that are selected to be use.

**Message panel**: Located at the bottom of the screen, it informs the player about its auditive perceptions and other events that he has to be aware off as they correspond to the result of his actions.

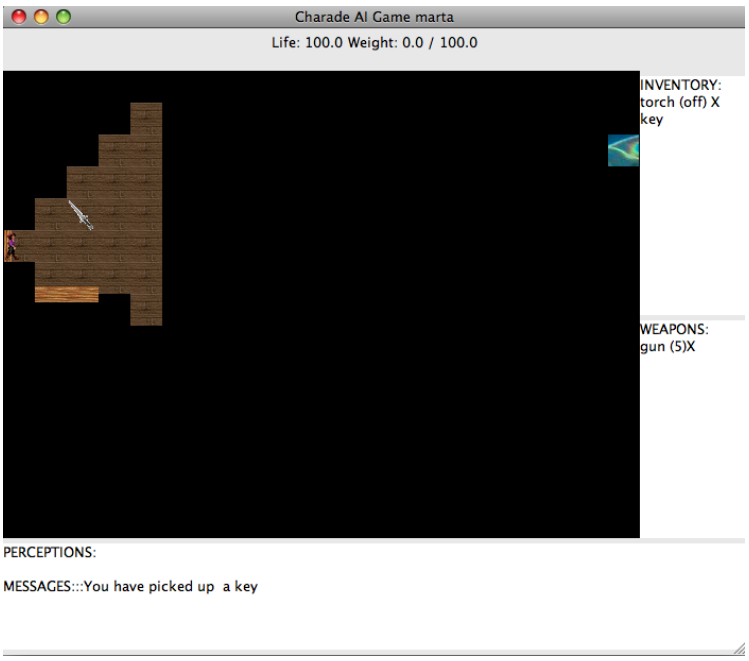



Figure A.2: Screenshot

3. **Actions**

Players **move** by using the **arrows** and can perform the different actions by using the following keys:

**A**: Attack to the direction player is facing. If it is wanted to attack to some specific point, the **mouse** has to be used by clicking where the attack wants to be made.

**C**: Close door

**Enter**: Touch

**L**: Leave item

**I**: Choose from inventory (selects among the items from the inventory)

**O**: Open door

**R**: Reload weapon

**Space**: Pick up object

**Turn**: Turn

**U**: Use item

**W**: Choose weapon (selects among the weapons from the list)

## A.4 Creation of a new agent

For each new agent, a new class has to be created. This class would be an implementation of the interface `MyPlayer`, needing to implement the methods `chooseAction(Game g)` and `start(Game g)`.

# Bibliography


[1] Asociacin de ingenieros e ingenieros tcnicos en informtica. `http://www.ali.es`. [Online; accessed May-2010].

[2] Métrica versión 3. metodología de planificación, desarrollo y mantenimiento de sistemas de información. `http://www.csi.map.es/csi/metrica3`. [Online; accessed April-2010].

[3] Blackrose: Un modelo de razonamiento con incertidumbre en juegos de estrategia. Master's thesis, Universidad Carlos III de Madrid, 2009.

[4] David Brackeen, Bret Barker, and Laurence Vanhelswue. *Developing Games in Java*. New Riders Games, 2003.

[5] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Extending online planning for resource production in real-time strategy games with search.

[6] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Online planning for resource production in real-time strategy games.

[7] John Funge. *Artificial intelligence for computer games: an introduction*. Sales and Customer Service Office.

[8] Stephanie Elzer Jarett Cummings and Gary Zoppetti. Bayesian networks in video games.

[9] Donald Kehoe. Designing artificial intelligence for games. 2009.

[10] Sven Koenig, Maxim Likhachev, Yaxin Liu, and David Furcy. Incremental heuristic search in ai. *AI Mag.*, 25(2), 2004.

[11] Ronald A. Metoyer, Simone Stumpf, Christoph Neumann, Jonathan Dodge, Jill Cao, and Aaron Schnabel. Explaining how to play real-time strategy games.

In Max Bramer, Richard Ellis, and Miltos Petridis, editors, *SGAI Conf.*, pages 249–262. Springer, 2009.

[12] Ian Millington. *Artificial intelligence for games.* Morgan Kauffman, 2006.

[13] Jeff Orkin. Agent architecture considerations for real-time planning. In *in Games. Artificial Intelligence and Interactive Digital Entertainment (AIIDE.* AAAI Press, 2005.

[14] Steve Rabin. *AI Game Programming Wisdom.* Charles River Media, Inc., Rockland, MA, USA, 2002.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[16] Brian Schwab. *Ai Game Engine Programming (Game Development Series).* Charles River Media, Inc., Rockland, MA, USA, 2004.

[17] Michael van Lent and John Laird. Developing an artificial intelligence engine.

[18] Michael van Lent and John Laird. Human-level ai's killer application: Interactive computer games.

[19] James Wexler. Artificial intelligence in games: A look at the smarts behind lionhead studios black and white and where it can and will go in the future.