UNIVERSITY OF MACEDONIA

POSTGRADUATE STUDIES

DEPARTMENT OF APPLIED INFORMATICS

BDI AGENTS IN GAMES

Thesis of

Evaggelos Raptis

Thessaloniki, October 28, 2014

BDI AGENTS IN GAMES

Evaggelos Raptis

Bachelor degree in Applied Informatics, University of Macedonia, 2012

Thesis

submitted for the partial fulfillment of the requirements of the

POST GRADUATE DEGREE IN APPLIED INFORMATICS

Supervisor
Ilias Sakellariou

Approved by the three-member inquiry committee at .....

....

....

Evaggelos Raptis

.....

Abstract

Intelligent agents are the focus of interest by many fields of computer science. Agent-based systems are being used in an wide variety of applications from small scale systems such as email filters to large scale systems such as air traffic control. Rather recently, video games have attracted a lot of attention, however, there are still a large number of open issues regarding the application of AI to the field. Although, these areas appear to be different, they share one common factor; intelligent behavior.

Intelligent behavior in video games is accomplished in many ways these days, from finite-state machines to neural networks, even more ad-hoc approaches. In this work we address the problem of devising a simple way to create game characters, using agent-based architecture, so that these characters (agents) are able to achieve their goals in an intelligent manner. A simple way here means an easy way to:

- specify and manage agent behavior.
- provide the means to easily adjust agent behavior to specific conditions.
- allow to easily encode complex behavior.

To address the problem stated above, this work reports on the implementation of a well defined and documented web application that generates agent simulation projects for experimentation. As it turns out, BDI architecture is perfectly suited for the task. Not all genres of games are appropriate, but most of modern games such as role-playing-games (RPGs) or first-person-shooters (FPSs) are most likely to benefit from this approach.

Keywords: video game, game character, agent, intelligent behavior, complex behavior, agent-based architecture, BDI, NetLogo

Preface - Acknowledgments

This thesis would have not been possible without the help and support of my supervisor Ilias Sakellariou, who has guided me through more than this dissertation.

My mother Eleni, my father Paulos, my brother George and my friends, thank you all for your love, support and patience.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem - Topic significance

Over the course of the last few decades, the gaming industry has seen great strides. Beginning with simple games like Pong and Pac-Man which offered players a short escape from reality and growing into more involved games like World of Warcraft and Call of Duty 4 which are serious hobbies and even professions to those that play them. Today's gamers have grown accustomed to seeing each new game becoming increasingly complex, engaging, and intelligent. For developers, the challenge becomes pushing the envelope to create games that are increasingly compelling. Computer-controlled Artificial Intelligence has evolved in many forms to meet the challenge. However, creating an adaptive foil for the player that can match their moves and encourage player evolution is no simple task.

In video games, intelligent behavior can be naturally captured through interaction with the environment, and biologically inspired techniques such as evolutionary computation, neural networks etc.. Belief, Desire Intention (BDI) agents also are well suited for this task. Yet, while much research focused on utilizing search and logic in board games, since the dawn of AI, it is only recently that AI research has turned its focus to addressing the problem of BDI agents in video games.

1.2 Purpose - Aims

This thesis, aims to present a way, to apply BDI agent techniques in games. In other words, to trigger some attention in using BDI architecture, to architect game characters' behavior. To be more specific, this document describes a web application, named NetLogo Game Bots (N.G.B.), that makes use of BDI-like architecture for creating games.

N.G.B. can be used to create games in an agent simulation environment using BDI architecture. The agents of the game are modeled as BDI agents, i.e. have beliefs, intentions and goals that they will try to achieve according to their specification. Agents are also able to communicate using the KQML protocol. A rich user interface and documentation guides the user to architect the agents to get the desired results.

Finally, N.G.B. aims that the user will have to code as little as possible, so that he may focus on the behavior of the agents. After agent specification, the user will be able to export the project, as NetLogo project and test the results. This can be seen as a prototyping phase, after which, the project can be used as a guide to transfer the agents' behavior to another platform or game engine.

1.3 Structure of the study

The rest of the thesis is structured as follows. Chapter 2 introduces basic concepts of artificial intelligence (AI) in games as well as previous efforts applying BDI techniques in video games. In Chapter 3 the methodology used for this approach is analyzed in details. The analysis of the problem takes place in Chapter 4 where the main goals of this study are analyzed, highlighting the benefits of such an approach. The implementation of the N.G.B. web application follows on Chapter 5 in an effort to highlight the key parts of such an implementation. Chapter 6 provides a case study, in which we make use of the web application, in order to create and deploy a NetLogo project, using the most significant features of the web application. Chapter 7 concludes the work analyzing the outcome of the case study finally drawing conclusions and future works.

2 Bibliographic Overview - Theoretical Background

2.1 Video Games

The first video games (Wikipedia [2014a]) were developed in the 1950s. The first game was designed exclusively for CRT monitors[1]. With funding problems and under those conditions, many were those who continued their efforts and their research on the technology of computer games. So far the history of the art electronic numbers seven generations, which are briefly described below.

First generation (1970 - 1976) The first generation on the early 1970s is characterized by the lack of resources and the high cost of acquiring these resources while those games were written in large data centers of many universities. There was no commercial development nor an incentive to implement such programs.

The first mass-produced computer game came from Nolan Bushnell, who founded the company known Atari and launched on the market the game Pong, which is an attempt to imitate Tennis. Pong enjoyed great commercial success for its time as it sold 19,000 electronic machines.

Along with Pong started the rise of video games as there were other video games like PacMan, Arkanoid and the Asteroids as video games installed in restaurants, shopping centers and recreational areas.

Second generation (1977 - 1984) The second generation from 1977 to 1984 uses discrete logic chips and compiled code. The first consoles were implemented and by mid-1970 games were found in cassettes. Consumers now have the possibility of a console and a lot of games.

Typical phenomenon of the era is the great variety of games that were available (often of poor quality) but not adopted by the general public despite the surge in

[1] CRT monitors are monitors that use cathode ray tubes (CRT), which are glass vacuum tubes into which an electron gun emits a flow of electrons guided by an electrical field towards a screen covered in small phosphorescent elements.

computer games, which led to a large stock and many companies being forced to close due to either bankruptcy or lack of funding.

Third generation (19785 - 1988) The third generation of computer games is characterized by a return of gaming consoles although there was no progress in electronics as the games were still written and run on machines in the form of 8bit, while computers start competing consoles. Levers give way to controls as the latter were integrated into most consoles.

Fourth generation (1989 - 1994) From 1989 to 1994 is the fourth generation of computer games and is characterized by great technological progress in this field. CD-ROMs with greater capacity paved the way for games with better graphics and of course more playing time as the games are not terminated early! The second era of technological success is 16bit games and the introduction of three-dimensional graphics.

Fifth generation (1994 - 1999) The fifth generation is characterized by the new 32-bit technology while consoles like Playstation and Sega Saturn are launched to the market. Nintendo makes the difference by creating an 64bit console, the Nintendo64.

Sixth generation (2000 - 2004) The sixth generation converges with DVD technology and the advancement of new technology digital displays. Many games require 2 and 3 DVDs and the graphics are more realistic than ever, with stunning clarity and detail. Great importance is now given to the speed of processors and memory.

Seventh generation (2004 - today) The seventh generation counting from 2004 to today shows the highest technology available in the market with gaming consoles like Playstation 3 and XBox360 having features and capabilities of a computer, while cable is past with Bluetooth and WiFi technologies to incorporate into many other. Another gaming device, Nintendo Wii uses motion sensors instead of the remote control and works with the body movements!

2.2 Video Game Genres

Video game genres (Wikipedia [2014b]) are used to categorize video games based on their gameplay interaction. A video game genre is defined by a set of gameplay challenges. They are classified independent of their setting or game-world content, unlike other works of fiction such as films or books. For example, an action game is still an action game, regardless of whether it takes place in a fantasy world or outer space. Following is a listing of the most commonly used video game genres.

Action Games An action game requires players to use quick reflexes, accuracy, and timing to overcome obstacles. It is perhaps the most basic of gaming genres, and certainly one of the broadest. Action games tend to have gameplay with emphasis on combat. There are many sub-genres of action games, such as fighting games and first-person shooters.

Role-playing Games Role-playing video games (RPGs) draw their gameplay from traditional role-playing games like Dungeons & Dragons. Dungeons & Dragons (D&D) is a fantasy tabletop role-playing game, first published in 1974. Most of these games cast the player in the role of one or more "adventurers" who specialize in specific skill sets (such as melee combat or casting magic spells) while progressing through a predetermined storyline. Gameplay elements strongly associated with RPGs, such as statistical character development through the acquisition of experience points, have been widely adapted to other genres such as action games.

Simulation Games Simulation video games is a diverse super-category of games, generally designed to closely simulate aspects of a real or fictional reality. Flight simulators belong to this genre.

Strategy Games Strategy video games focus on gameplay requiring careful and skillful thinking and planning in order to achieve victory and the action scales from world domination to squad-based tactics. In most strategy video games, the player

is given a godlike view of the game world, indirectly controlling the units under his command.

Sports Games Sports are games that play competitively one team, containing or controlled by the player, and another team that opposes the player. This opposing team(s) can be controlled by other real life people or artificial intelligence.

Massively Multiplayer Online Games A massively multiplayer online game (MMO) is a multiplayer video game which is capable of supporting large numbers of players simultaneously. By necessity, they are played on the Internet. MMOs can enable players to cooperate and compete with each other on a large scale, and sometimes to interact meaningfully with people around the world. They include a variety of gameplay types, representing many video game genres.

Board Games Many popular board games have computer versions. AI opponents can help improve one's skill at traditional games. Chess, Checkers, Othello (also known as Reversi), and Backgammon have world class computer programs.

2.3 Game Development

Game development is a complex software development process, as a video game is software with art, audio, and game-play. Formal software development methods are often overlooked. Games with poor development methodology are likely to run over budget and time estimates, as well as contain a large number of bugs. Planning is important for individual and group projects alike.

2.3.1 Development Team

2.3.1.1 Designer

A game designer is a person who designs game-play, conceiving and designing the rules and structure of a game. Development teams usually have a lead designer who coordinates the work of other designers. They are the main visionary of the game.

2.3.1.2 Artist

A game artist is a visual artist who creates video game art. The artist's job may be 2D oriented or 3D oriented. Artists involved in 2D may produce concept art, sprites, textures, environmental backdrops or terrain images and user interface. Artists involved in 3D may produce models or meshes, animation, 3D environment, and cinematics.

2.3.1.3 Sound engineer

Sound engineers are technical professionals responsible for sound effects and sound positioning. They sometimes oversee voice acting and other sound asset creation. Composers who create a game's musical score also comprise a game's sound team, though often this work is outsourced.

2.3.1.4 Tester

The quality assurance is carried out by game testers. Testing is a highly technical field requiring computing expertise, and analytic competence. The testers ensure that the game falls within the proposed design: it both works and is entertaining. This involves testing of all features, compatibility, localization, etc.

2.3.1.5 Programmer

A game programmer is a software engineer who primarily develops video games or related software (such as game development tools). The game's codebase development is handled by programmers. There are usually one to several lead programmers, who implement the game's starting codebase and overview future development and programmer allocation on individual modules. Individual programming disciplines roles include:

- Physics - the programming of the game engine, including simulating physics, collision, object movement, etc.
- Graphics - the managing of graphical content utilization and memory considerations; the production of graphics engine, integration of models, textures to work

along the physics engine.

- Sound - integration of music, speech, effect sounds into the proper locations and times.
- Gameplay - implementation of various games rules and features.
- Scripting - development and maintenance of high-level command system for various in-game tasks, such as AI, level editor triggers, etc.
- UI - production of user interface elements, like option menus, help and feedback systems.
- Input processing - processing and compatibility correlation of various input devices, such as keyboard, mouse, gamepad, etc.
- Network communications - the managing of data inputs and outputs for local and internet gameplay.
- AI - producing computer agents using game AI techniques, such as scripting, planning, rule-based decisions, etc.
- Game tools - the production of tools to accompany the development of the game, especially for designers and scripters.

2.3.2 Programming AI

An AI programmer develops the logic of the game to simulate intelligence usually in enemies and opponents. It has recently evolved into a specialized discipline, as these tasks used to be implemented by programmers who specialized in other areas until today. An AI programmer may program path-finding, strategy and enemy tactic systems. This is one of the most challenging aspects of game programming and its sophistication is developing rapidly. Contemporary games dedicate approximately 10 to 20 percent of their programming staff to AI.

Some games, such as strategy games like Civilization V or role-playing video games such as The Elder Scrolls IV: Oblivion, use AI heavily, while others, such as puzzle games, use it sparingly or not at all. Many game developers have created entire languages that can be used to program their own AI for games via scripts. These languages are typically less technical than the language used to implement the game, and will often be used by the game or level designers to implement the world of the game. Many studios also make their games' scripting available to players, and it is often used extensively by third party mod developers.

The AI technology used in games programming should not be confused with academic AI programming and research. Although both areas do borrow from each other, they are usually considered distinct disciplines, though there are exceptions. In recent years, more effort has been directed towards intervening promising fields of AI research and game AI programming. More about AI in games is provided in the section that follows.

2.3.3 Programming Game Tools

A game tool is a specialized software application that assists or facilitates the making of a computer or video game. Some tasks handled by tools include the conversion of assets (such as 3D models, textures, etc.) into formats required by the game, level editing and script compilation. Game tools can make game development heaven or unbearably difficult. Tools are used on almost every game for tasks such as scripting, importing or converting art, modifying behaviors or building levels. Some tools, such as an IDE, 3D graphics modeling software and Photoshop are COTS products, but many tools are specific to the game and are custom programmed.

It is the programmer's job to write the tools that handle these game-specific tasks. Some tools will be included in the final production version of the game, as part of the runtime engine, but most will not. Most tools evolve with the game and can easily consume all of several programmers' time. Well written and fairly bug-free tools make

everyone's development tasks easier whereas poorly written or poorly documented ones can seriously hamper development and jeopardize the project. Due to time constraints, however, many tools are not carefully implemented. Some of the most famous game tools are those of the *RAD Game Tools* company, like the game tool *Oodle* Radgametools [2014a] used for data compression or *Telemetry* Radgametools [2014b] used for profiling [2].

2.4 Artificial Intelligence in Video Games

The term artificial intelligence (AI) refers to the branch of computer science that deals with the design and implementation of computing systems that mimic aspects of human behavior which imply even rudimentary intelligence, learning, adaptability, conclusions, understanding of contextual, problem solving etc.

AI is an intersection between several fields such as computer science, psychology, philosophy, neuroscience, linguistics and science engineering, with the aim of intelligent behavior composition with elements of reasoning, learning and adapting to the environment, and is usually applied to machines or computers specially designed.

AI in games, refers to the techniques used by computers, to produce the illusion of intelligence in the behavior of characters, who are not manipulated by people. Games were an area of Artificial Intelligence research since its birth. Most AI research so far has focused on games that can be described in a compact form using symbolic representations, such as board games and card games. The so-called good old-fashioned artificial intelligence (GOFAI J. [1985]) techniques work well with symbolic games, and to a large extent, GOFAI techniques were developed for them.

Since the 1990s, the field of gaming has changed tremendously. Inexpensive yet powerful computer hardware has made it possible to simulate complex physical envi-

[2] In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

ronments, resulting in tremendous growth in the video game industry. From modest sales in the 1960s, sales of entertainment software reached $25 billion worldwide in 2004. Video games are now a regular part of many people's lives, and the market continues to expand.

2.4.1 AI in Board Games

Video games were among the first projects dealt with by AI. From 1950 to date, there has been significant progress so that machines have beat man at *checkers* and *othello* and earn even champions in *checkers* and *chess*. The only exception, among widely known games, is in the game *go* (Figure 1), where their performance is still in beginner level.

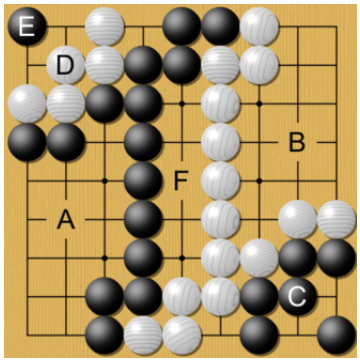

Figure 1: Go Game
Source: wikipedia

In 1951 at the University of Manchester, *C. Strachey* (ChessProgramming [2014b]) wrote a *checkers* program and *D. Prinz* (ChessProgramming [2014c]) a *chess* program. These were among the first programs ever written. *A. Samuel* (ChessProgramming [2014a]) developed a *checkers* game in the 60s, giving a satisfactory level to complicate an amateur player. All this work in *chess* and *checkers*, would result in the defeat of *G. Kasparov* (ChessProgramming [2014d]) by the IBM computer *Deep Blue* in 1997 (Figure 2).

Figure 2: G. Kasparov vs IBM Deep Blue
Source: chessprogramming

2.4.2 AI in Modern Games

Smartness and complexity are two of the key characteristics every modern video game must contain in order to succeed. Both depend on the game's underlying AI. Hence, design and development of AI is becoming more and more significant.

First step would be to define, where AI is "hidden" in modern games. Modern video games consist of various elements such as graphics and sound but the most important are the characters and their behavior. The player of the game controls a character and interacts with other characters that are not always handled by other people. Those computer-controlled characters can be divided in two main categories, those being *NPCs* and *Bots*. The characteristics that define if a character is an NPC or a Bot are not always clear.

NPC In general, an NPC (Janssen [2014]) is placed in the environment and exists to interact with the player. Examples would include vendors, quest givers, or enemies. They are often distinguished by having distinct personalities and dialog, whereas a "Player Character" is meant as more of an extension of the player themselves.

Bot A Bot, is essentially a player of the game demonstrating intelligence. This can be populated within a single instance, as a feature in some games (i.e. AI opponents

in a multiplayer game).

The key distinction is that a *Bot* represents an automated player but an *NPC* is a passive character. Conclusively, AI applies to Bots, so Bot is the keyword that will be used to define a computer-controlled game character in modern games.

Bots can be found in all game genres from action games to RPGs. Bots enhance game experience providing the complexity required. Movement, path finding and decision making are the most significant features, Bots must present for the game to be enjoyable. Various algorithms and game tools (see 2.3.3) are used in order to implement complex behavior in a single game. Some of the most popular Bots used in modern games are presented below.

Half Life 1998 Half-Life is a sci-fi first-person shooter featuring Dr. Gordon Freeman, a theoretical physicist stuck inside an underground research facility when teleporting experiments go wrong unexpectedly.

- The cut scenes[3] are entirely interactive. A combination of scripting and AI is used to avoid interrupting the gameplay (i.e. you can use the crowbar to destroy things while being told what to do next).
- An AI security guard accompanies the player through some levels early in game.
- Despite not using any revolutionary technology, the AI is very well tweaked and integrated smoothly into the storyline.

[3]A scene that develops the story line and is often shown on completion of a certain level, or when the player's character dies.

Figure 3: Half Life
Source: aigamedev

Total War 2000 Total War is a series of games combining turn-based strategy on a Risk-like map, with real-time tactical control of battles on a 3D terrain.

- Thousands of AI-controlled soldiers are featured for the first time in a fun and interactive game, without noticeable performance problems.
- The game models the emotions of groups of soldiers, essential for simulating battles accurately. This logic is inspired by the book, The Art Of War.
- The Total War engine is used on TV by the History Channel as part of the Decisive Battles series.

Figure 4: Total War
Source: aigamedev

Thief 1998 Thief: The Dark Project is a single player stealth-based game played from a first person perspective. The main character is Garret, a master thief applying his trade in a medieval/Victorian setting.

- The game uses an accurate sensory model, allowing the AI actors to respond realistically to light and sounds. The whole game is based around this technique.
- The AI actors use audio recordings to voice their current state, letting the player understand what's going on.

Figure 5: Thief
Source: aigamedev

Halo 2001 Halo: Combat Evolved is a first-person shooter where the player assumes the role of the Master Chief, battling various aliens on foot or in vehicles.

- Enemies use cover very wisely, and employ suppressive fire and grenades.
- The squad situation affects the individuals, so certain enemies flee when their leader dies.
- A lot of attention is paid to the little details, with enemies notably throwing back grenades or team-members responding to you bothering them.
- The underlying behavior tree technology has become very popular in the games industry (especially since Halo 2).

Figure 6: Halo
Source: aigamedev

Facade 2005 Facade is an interactive story where the couple is invited to the apartment of Grace and Trip, a couple who has a relationship breakdown. The player can coach them using full typed sentences.

- The player interacts with the game by tying text into a natural language parser.
- The underlying behavior language provides ways to specify the behavior of characters in terms of a dynamic story.

Figure 7: Facade
Source: aigamedev

F.E.A.R. 2005 F.E.A.R. First Encounter Assault Recon is a first-person shooter where the player helps contain supernatural phenomenon and, naturally, armies of

cloned soldiers.

- The AI uses a planner to generate context-sensitive behaviors, the first time in a mainstream game. This technology used as a reference for many studios still today.
- The enemies are capable of using the environment very cleverly, finding cover behind tables, tipping bookshelves, opening doors and crashing through windows.
- Squad tactics are used to great effect. The enemies perform flanking maneuvers, use suppression fire, etc.

Figure 8: F.E.A.R.
Source: aigamedev

2.5 Intelligent Agents

An agent is an autonomous entity which observes through sensors and acts upon an environment using actuators and directs its activity towards achieving goals. Agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex: a reflex machine such as a thermostat is an agent, as is a human being, as is a community of human beings working together towards a goal. An agent is a computer system (hardware or software) with the following properties (Wooldridge and Jennings [1995]):

Autonomy Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state.

Social Ability Agents interact with other agents and humans, via some kind of agent-communication language (Genesereth and Ketchpel [1994]).

Reactivity Agents perceive the environment and respond to changes that occur in it.

Pro-activeness Agents do not simply react to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

One of the best-known and most influential contributions to the area of agent theory is due to Cohen and Levesque (Cohen and Levesque [1990]). Their formalism used to develop a theory of intention (as in "I intend to..."), which the authors required as a prerequisite for a theory of speech acts. Cohen and Levesque identify seven properties that must be satisfied by a reasonable theory of intention:

- Intentions pose problems for agents, who need to determine ways of achieving them.
- Intentions provide a "filter" for adopting other intentions, which must not conflict.
- Agents track the success of their intentions, and are inclined to try again if their attempts fail.
- Agents believe their intentions are possible.
- Agents do not believe they will not bring about their intentions.
- Under certain circumstances, agents believe they will bring about their intentions.
- Agents need not intend all the expected side effects of their intentions.

2.5.1 Applications of Agents

Here are some examples that use intelligent agents and illustrate some ways that agents can help solve real problems and make today's computer systems easier to use.

Google Parallel Search Engine The core of the Google search engine, is a suite of modular software engines, including: an intelligent agent parallel search and comparison engine, a proxy engine that registers saved queries on host sites, and an agent-based engine that pushes data to online forms, web sites, or databases (Martin and Boyle [2004]).

Air Traffic Control Modem air traffic control is carried out primarily with the aid of intelligent agents. Agents receive, organize, present and transmit flight information from one air traffic controller to another. The role of an air traffic controller is to coordinate the airspace according to this information, so that airplanes can navigate safely and efficiently.

Customer Help Desk Customer help desk job is to answer calls from customers and find the answer to their problems. When customers call with a problem, the help desk person manually looks up answers. An agent can automatically search the appropriate databases, then present a consolidated answer with the most likely first.

Personal Shopping Assistant IBM's Personal Shopping Assistant uses intelligent agent technology to help the Internet shopper or the Internet shop owner to find the desired item quickly without having to browse page after page of the wrong merchandise. With the Personal Shopping Assistant, stores and merchandise are customized as the intelligent agent learned the shopper's preferences as he/she enters in any on-line mall or stores or looking at specific merchandise. It could also arrange the merchandise so that the items you like the most are the first ones you see.

2.5.2 Classes of Agents

Russel and Norvig [2003] group agents into five classes based on their degree of perceived intelligence and capability.

Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history and usually lacking an explicit world representation. The

agent function is based on the condition-action rule: if condition then action.

This agent function only succeeds when the environment is fully observable. Some reflex agents can also contain information on their current state which allows them to disregard conditions whose actuators are already triggered. Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments, but if the agent can randomize its actions, it may be possible to escape from infinite loops.

Model-based reflex agents can handle a partially observable environment. Their current state is stored inside the agent maintaining some kind of structure which describes the part of the world which cannot be seen. This knowledge about "how the world works" is called a model of the world, hence the name "model-based agent". A model-based reflex agent should maintain some sort of internal model that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. It then chooses an action in the same way as the reflex agent.

Goal-based agents Goal-based agents further expand on the capabilities of the model-based agents, by using "goal" information. Goal information describes situations that are desirable. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. Search and planning are the sub-fields of artificial intelligence devoted to finding action sequences that achieve the agent's goals. In some instances the goal-based agent appears to be less efficient; it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

Utility-based agents Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is. This measure can be obtained through the use of a utility function which maps a state to a measure of the utility of the state. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. The term utility, can be used to describe how "happy" the

agent is.

A rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes- that is, the agent expects to derive, on average, given the probabilities and utilities of each outcome. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning.

Learning agents have an advantage that allows the agents to initially operate in unknown environments and to become more competent than their initial knowledge alone might allow. The most important distinction is between the "learning element", which is responsible for making improvements, and the "performance element", which is responsible for selecting external actions.

The learning element uses feedback from the "critic" on how the agent is doing and determines how the performance element should be modified to do better in the future. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

The last component of the learning agent is the "problem generator". It is responsible for suggesting actions that will lead to new and informative experiences.

2.5.3 Agent Architectures

The architecture of an agent is a collection of software modules, usually designed as boxes, which are connected by arrows, which indicate the flow data and control between modules (Figure 9).

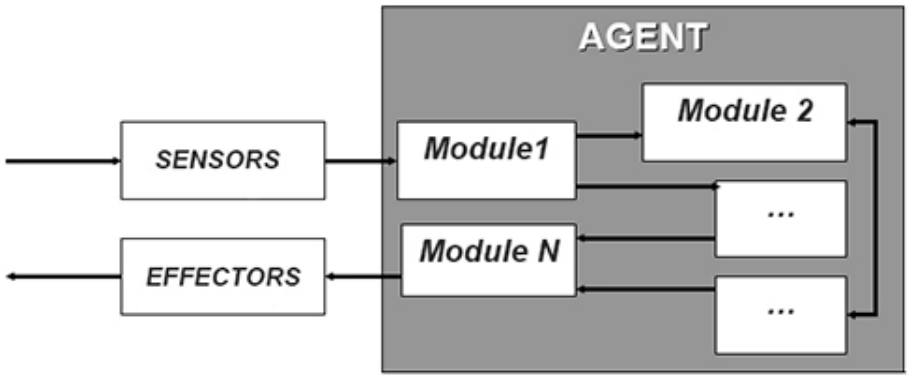

Figure 9: Abstract Agent Architecture

A specific methodology for developing agents, specifies how an agent can be analyzed (degraded) in a kit of parts and how these parts should interact. It whole should answer the question how sensor data and internal state of the agent determine the current and future energy situation agent. An architecture includes techniques and algorithms to support a such methodology. Below are the most popular between agent architectures.

Logic based architecture A logic-based agent is a software agent defined by:

- a language L
- a logic theory T over the language L
- a declarative semantics DS(T) of T meaning of T: set of logical consequences of T
- a procedural semantics PS(T) how to compute the logical consequences of T

Reactive architecture Reactive agents simply retrieve pre-set behaviors similar to reflexes without maintaining any internal state.

Belief-Desire-Intention architecture Belief-Desire-Intention (BDI) agents consist of

- Beliefs: Environment information (could be inaccurate).
- Desires: The agent's desires.
- Intentions: The goals for which the agent is committed.

Hybrid architecture A hybrid agent consists of at least 2 levels (layered architectures)

- one level responsible for reactive behavior
- one level responsible for the proactive behavior

2.5.4 Agent Communication Languages

In a Multi-Agent System[4], agents are social, this means that they communicate with other agents. Some agents learn or change their behavior based on their previous experiences. Finally, some agents attempt to be believable, such that they are represented as an entity visible or audible to the user and may even have aspects of emotion or personality. Some of the agent communication languages include KQML (Knowledge Query and Manipulation Languages), AOP (Agent Oriented Programming) and Agent Talk.

KQML is a language and protocol used for exchanging information and knowledge. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to share knowledge in support of cooperative problem solving.

AOP is an interpreter for programs written in a language called AO. AO is a programming language for the paradigm of Agent-Oriented Programming.

Agent Talk is a coordination protocol description language for multi-agent systems. Agent Talk allows coordination protocols to be defined incrementally and to be easily customized to suit application domains by incorporating an inheritance mechanism.

[4] A multi-agent system (M.A.S.) is a computerized system composed of multiple interacting intelligent agents within an environment.

2.5.5 Agents In Modern Video Games

The use of game applications has a long tradition in AI. Games provide high variability and are generally easy to evaluate. Many "traditional" games, such as card/board/puzzle games have been solved by AI techniques. However, it is highly questionable whether and to what extend the techniques used in this field of research can be applied to today's modern games. Such games pose problems for AI that are infinitely more complex than those of traditional games. Nareyek [2002] states that, modern computer games feature:

- Real Time: There is only very limited time for reasoning.
- Dynamics: Computer games provide a highly dynamic environment.
- Incomplete Knowledge: A game character has only incomplete knowledge of the world.
- Resources: The game's resources may be restricted.

Techniques from the AI fields of autonomous agents, planning, scheduling, robotics and learning would appear to be much more important than those from traditional games. The standard procedure followed in modern games is to implement a goal-directed behavior using predetermined behavior patterns. This is normally done using simple if-then rules. In more sophisticated approaches using neural networks, behavior becomes adaptive, but the purely reactive property is still present.

3 Methodology

In this chapter we will review the problem that this thesis addresses. Related work will also be discussed. Finally, the methodology and development process followed will be presented.

3.1 Problem Review

The problem this dissertation is trying to solve has been examined by other works as well. Intelligent behavior is sought in many fields in computer science since it's birth. Video games is not an exception. On the contrary, the gaming community is most of the times the leader in such an effort. One reason is because the results are visualized directly. In games, one single question determines success; do i enjoy this game? Of course we are talking about all game genres, from board games to RPGs. But in some game genres, just to write software that wins the game is not the one and only goal.

In games like chess, victory is an end in itself. Even if the software is making moves that do not seem reasonable, if it wins, it's a successful software. On the contrary, games like Counter-Strike [5] do not seek for victory itself. Such games seek for human-like behavior. The human player, should have the illusion that he is playing against another human player. Below are listed some behaviors for the game Counter-Strike, that could be winning, but are not comparable to what a human player would do:

- A Bot always rushes through the enemy indifferent if it dies.
- A Bot shoots for the enemy ignoring any allies in front.
- A Bot attacks the enemy with a knife, while carrying a rifle.
- A Bot ignores an ally calling for help, while it could help.

In the gaming industry, a lot of work has been done when it comes to AI. This work is of course limited in time and money spent. On the other hand, almost every

[5] Counter-Strike is a first-person shooter video game developed by Valve Corporation.

single game uses it's own game engine, it's own game tools (2.3.3) and it's own AI implementation. This is somewhat unavoidable since each game describes a totally different world and research has not progressed enough to provide general solutions. The result though is that most of the times, all this code is not reusable or even customizable.

Of course, no one can compare human complexity in thinking with a computer software. This is why human-like behavior in games is so difficult to achieve. Our effort will focus in creating agents that present complex behavior. For this to happen, agent creation process should closely imitate the human reasoning process. This is one of the strong points of the BDI architecture. If a human believes something, an agent believes something. If a human has desires that intends to achieve, an agent should have also. If a human has alternative plans for an intention of his, an agent should have also. If it's easy for a human to alter his intentions, an agent should as easy alter it's intentions.

3.2 Software Development Process

In terms of Software Development Process, the spiral model (Boehm [1986]) was used to design and implement N.G.B.. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation (Figure 10). Our software project repeatedly passed through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements were gathered and risk were assessed. Each subsequent spirals were build on the baseline spiral.

Planning Phase Requirements were gathered during the planning phase.

Risk Analysis In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype was produced at the end of the risk analysis phase. If any risk was found during the risk analysis then alternate solutions were suggested and implemented.

Engineering Phase In this phase software was developed, along with testing at the end of the phase. Hence in this phase the development and testing was done.

Evaluation phase This phase allowed us to evaluate the output of the project to date before the project continues to the next spiral.

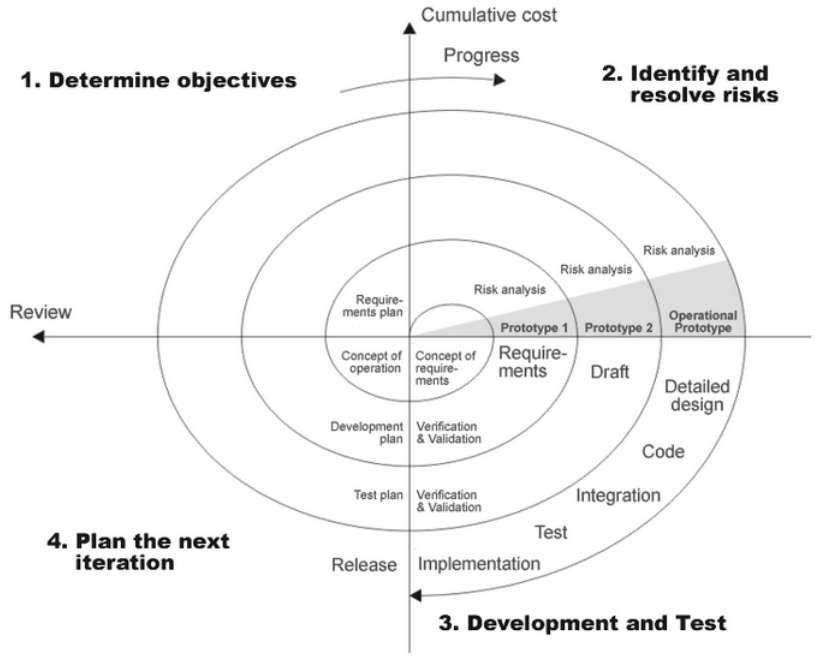

Figure 10: Spiral Model
Source: Wikipedia

3.3 Related Work

Various projects have connected agents to games. Two of the most relevant are discussed below.

3.3.1 Robocode

Robocode (Robocode [2014]) is a programming game where the goal is to code a robot battle tank to compete against other robots in a battle arena. So the name Robocode is a short for "Robot code". The player is the programmer of the robot, who will have no direct influence on the game. Instead, the player must write the AI of the robot telling it how to behave and react on events occurring in the battle arena. Battles are running in real-time and on-screen.

Figure 11: Robocode
Source: sourceforge.net

Competitors write software that controls a miniature tank that fights other identically-built (but differently programmed) tanks in a playing field. Robots can move, shoot at each other, scan for each other, and hit the walls or other robots if misprogrammed. Though the idea of the game is simple, the strategy needed to win is not. Robots can have thousands of lines in their code dedicated to strategy. Some of the more successful robots use techniques such as statistical analysis or attempts at neural networks in their designs. Robocode programming attempts to achieve:

- Movement: Avoid getting hit too much.
- Targeting: Try to predict where the opponents will move and hit them as much as possible.

Robocode agents are mostly reactive agents, since their behavior is triggered by events, followed by actions. Some of these events are:

- HitByBulletEvent. Triggered when the robot is hit by a bullet.
- HitRobotEvent. Triggered when the robot hits another robot.
- HitWallEvent. Triggered when the robot hits a wall.

3.3.2 GameBots

Gamebots (GameBots [2014]) is a project started at the University of Southern California's Information Sciences Institute that seeks to turn the game Unreal Tournament (UT) into a domain for research in artificial intelligence. The Gamebots domain allows both human players and agents, or bots, to play simultaneously; thus providing the opportunity to study human team behavior and to construct agents that play collaboratively with humans. The Gamebots system provides a built-in scripting language giving interested researchers the ability to create their own multi-agent tasks and environments for the simulation.

The core of the Gamebots project is a module for UT that allows characters in the game to be controlled via network sockets, connected to bot clients (Figure 12). The Gamebots server feeds sensory information for the characters over the network connections. Based on this information, the client (bot or human player) can decide what actions the character should take and issues commands back over the network to the game to have the character move, shoot, or talk. Agents must display advanced AI and MAS capabilities to play successfully, such as planning paths, learning a map of their 3D environment, using resources available to them, coordinating with their teammates, and engaging in strategic planning which takes their adversaries into account. Unlike other standard test-beds, the Gamebots system allows human players to play with the agents, thus providing opportunity to study human team behavior, and to construct agents that play collaboratively with humans.

The Gamebots interaction protocol is a simple text based protocol of single-line messages sent over the network between the server and bots. The Gamebots server sends sensory information messages to the bots containing the current state of the virtual world. The bots interact in the environment by sending action commands or player communication messages back to the server.

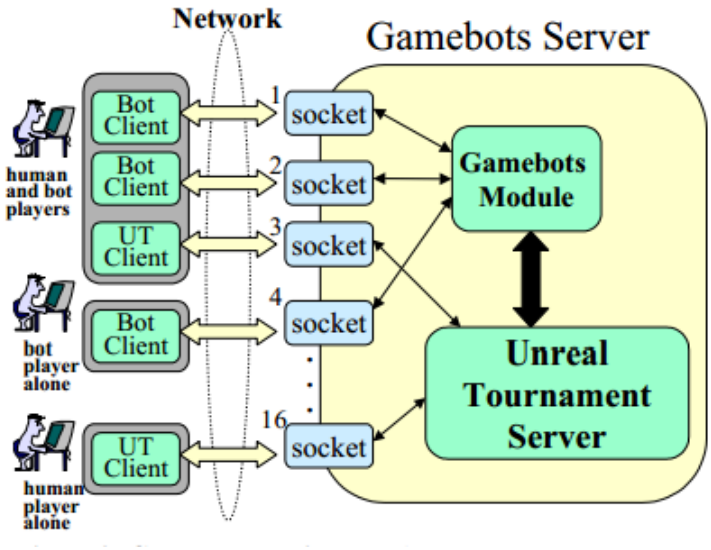

Figure 12: Gamebots
Source: Adobbati et al.[2001]

3.4 Specifications

Once an understanding is reached of how agent theory is currently applied into modern video games, a precise statement is required of what we intend to build.

First of all, an agent implementation is not usually used in "traditional" games such as chess and backgammon. Our implementation is not as well. We aim for modern games that provide the player an environment with characters to interact with (NPCs and Bots see 2.4.2). The player of such a game can be an active character as in an action game (see 2.2 Action Games). The player could also act as an observer-commander that controls his/her units as in a strategy game (see 2.2 Strategy Games). The current implementation supports the latter scenario (Strategy Games). Future plans include that the player will be able to choose one of the agents as the character

that he/she controls (see 7.2).

NetLogo Game Bots (N.G.B.) is a web application. The reason for that is not only to avoid software installation (which is a requirement these days) but also to promote project sharing between users. A single click is required from a user, to share a project with other users. ASP.NET MVC, which is used to develop N.G.B., provides the necessary facilities to treat a web application as a desktop application. Through the interface provided, the user is able to create-modify-delete agents and other GameEntities, drag & drop into the game environment, share projects and get help online.

N.G.B. is trying to standardize a way to build agents for games. We tried to deliver a simple development process that results in complex agent environments. BDI architecture along with KQML messaging protocol provide all the necessary functionality to achieve this result.

4 Analysis and Design

This chapter is an in depth analysis of the architecture used in the N.G.B. web application that is going to be presented in the next chapters. At first we will discuss about the concept of the project. Then we will take a look on the GameEnities and how those GameEnities are formed. Next, we will see how those GameEntities are mapped in an exported NetLogo project. Finally, we are going to discuss about logical errors/warnings concerning the project and how those may affect the final outcome.

The following table (Table 1) is a list of the terms that are going to be used in the sections that follow. Each term in the list is further analyzed further in the sequel.

Table 1: Main Terms

GameAgent
GameObject
GameRegion
GamePath
GameEntities
GameEnvironment
GameBoard
GameEntity
Custom Procedure
AgentSet

4.1 Project

The first concept we need to clarify is that of the project. A project can be seen as the environment of the game. It's the world that the game's GameEntities "live" in. A project consists of data and functionality.

Project data

- GameAgents
- GameObjects
- GameRegions
- GamePaths
- GameBoard

Project functionality

- Create, edit and delete GameEntities
- Add, move, resize and remove GameEntities on the GameBoard
- Detect logical errors/warnings on a project
- Export to NetLogo project

The workflow for a complete project is shown in Figure 13.

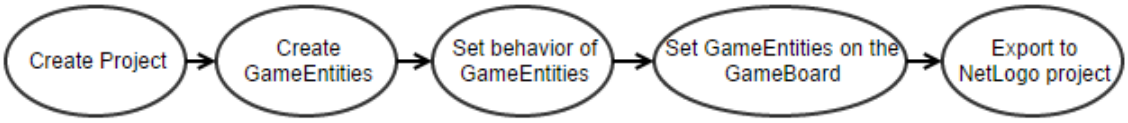

Figure 13: Project Work-flow

4.2 GameAgent

GameAgents are the pillars of the architecture. GameAgents are the intelligent agents of the game. A fighter character, a wizard character even a pet character could be a GameAgent. They have all common properties like name, color and shape, but they are mostly characterized by their behavior. They are hybrid agents owning BDI and reactive behavior. GameAgent properties are:

Name is the signature of the GameAgent. The Name property in plural is used to refer to the AgentSet. For example, if the Name of a GameAgent is "Friend", the keyword "Friends" refers to all GameAgents with Name "Friend".

Color property is used to draw the GameAgent.

Shape along with the Color property is used to graphically distinguish GameAgents among themselves.

Die Situation is a condition under which a GameAgent is considered to be dead. A periodical task will check if that condition is true. If it is true, the corresponding GameAgent will leave the game. An example follows on Figure 14.

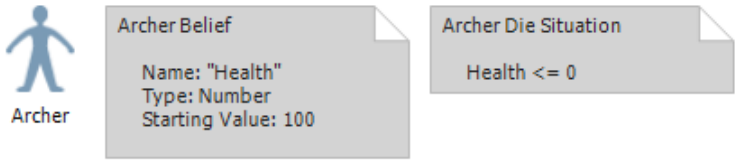

Figure 14: Die Situation

Win Situation is a condition under which a GameAgent is considered to have won the game. A periodical task will check if that condition is true. If it is true, the game will end and the corresponding AgentSet will be declared as winners. An example follows on Figure 15.

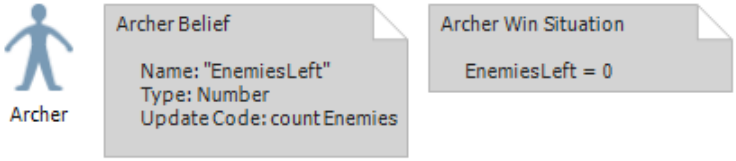

Figure 15: Win Situation

Beliefs is a list that forms the knowledge base of a GameAgent. A Belief can represent a numeric value or point to another GameAgent. An example could be the GameAgent's remaining hitpoints or ammunition. Another Belief could point be the nearest GameAgent of an opponent AgentSet.

The value of a Belief is not always fixed. A GameAgent's hitpoints for example are reduced every time the GameAgent is attacked. The nearest GameAgent of the opponent AgentSet is not always the same. So we need a way to update the value of the Belief every time needed. The best way is to update the value of the Belief, every time

that value is asked.

But not all Beliefs update in the same way. For that reason Beliefs are divided in 5 categories depending on the way they need to be updated. Those categories are the following:

- **Static** The value of the Belief is set once when the game starts.
- **Every time** The value of the Belief is updated every time asked.
- **Only if nobody** The value of the Belief is updated every time asked but only if the Belief value is equal to null.
- **On every tick** The value of the Belief is updated on every game tick.
- **On message** The value of the Belief is updated when a related message is received (see Messaging 4.7).

Interactions is a list of actions that displays what a GameAgent can do. An Interaction is typically the code that is executed when an action is taking place. Also, an Interaction may or may not be related to another GameAgent or GameObject. For example, the Interaction "Heal" could be taking place in the GameObject "Base". So, the GameAgent needs to be standing next to his "Base" in order to "Heal". For that reason, the user will be able to define if the Interaction requires contact with a given GameAgent or GameObject. Finally, Interactions provide a clean view of what a GameAgent can do.

Situations is a list of conditions. Each Situation is typically a procedure that returns true if the condition is valid, and false otherwise. Situations also provide a clean view of the situations a GameAgent can be involved in, that need special treatment. For example, a typical reactive behavior of a GameAgent should be: if Situation then Interaction.

Goals is a list of goals that a GameAgent is committed to achieve. Each Goal owns a list of Plans and each Plan owns a list of Actions. An example is shown in

Figure 16. Plans and Actions are explained below:

- **Plan** is a group of Situations, meaning that this particular Plan will take place only if all the Situations associated are valid. A Goal can have multiple Plans with different combinations of Situations in order to cover every situation needed.
- **Action** can be an Interaction or another Goal. Actions of a Plan are executed if the Plan is taking place. Also, every Action is related to a Situation, meaning that the Action is executed while this particular Situation is not valid. Since the Situation is valid, the Action stops executing. For example, the Interaction "Heal" should be executed until Situation "IsHealed" returns true.

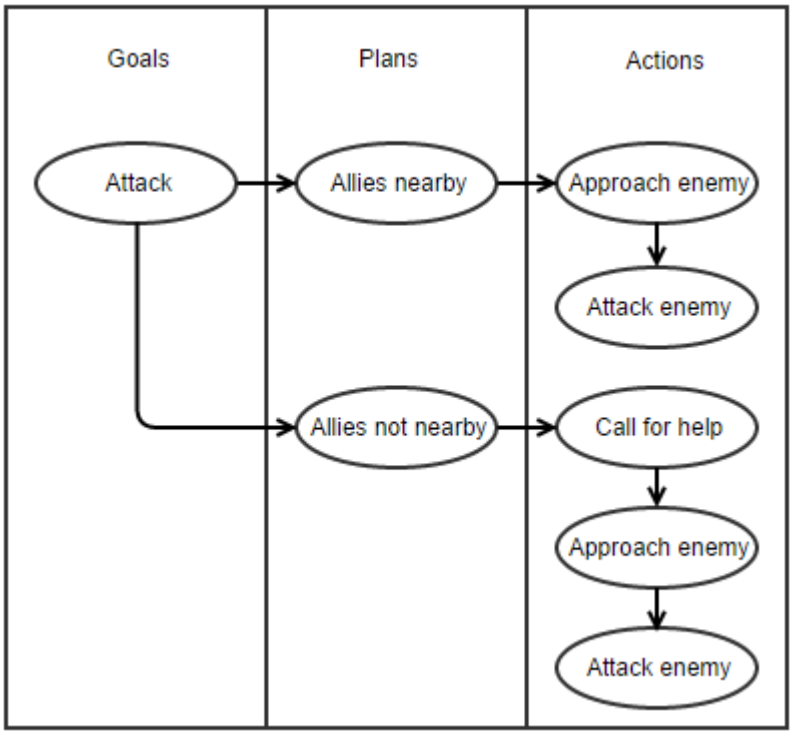

Figure 16: The Goal of Attack

Goals are also divided in three categories. A Goal can be "Simple Goal", "Maintenance Goal" or "Top Level Goal" as follows:

- **Top Level Goal** can be considered as the main obligation of a GameAgent. A GameAgent must have exactly one "Top Level Goal", since this is the Goal that the

GameAgent will try to achieve since the game starts. Also, this Goal is recycled forever meaning that this Goal is always going to be on top of the intention stack of the GameAgent.

- **Maintenance Goal** is a representation of a reactive behavior of a GameAgent. A GameAgent can have zero or multiple Maintenance Goals.
- **Simple Goal** is a Goal that can be triggered from another Goal as an Action of a Plan of the last.

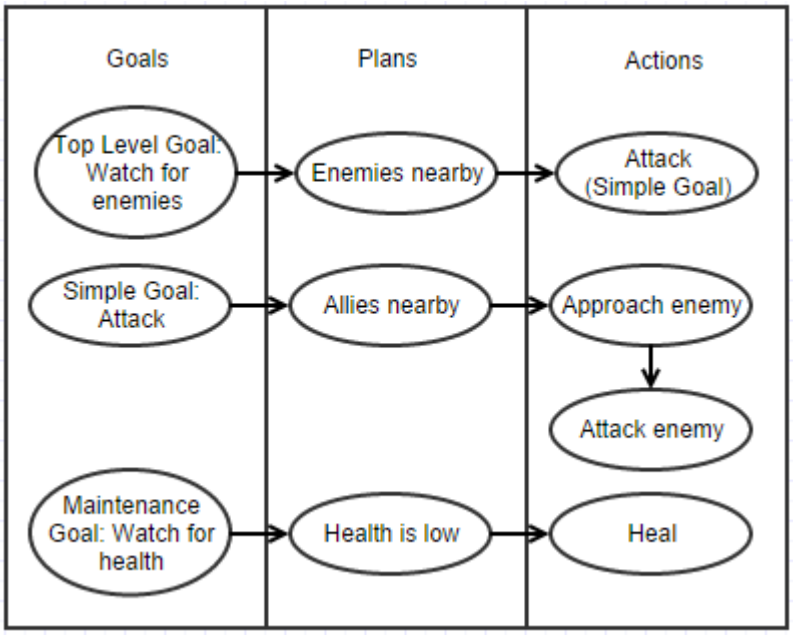

Figure 17: Goal Types

4.3 Other GameEnvironment Entities

The following GameEntities constitute the environment of the game, thus their role is limited to help the game evolve.

4.3.1 GameObject

A GameObject might be a house, an obstacle, a gun or even a bicycle. GameObjects are the objects that are part of the scene and participate in the game plot. For example, a temple in which a GameAgent heals himself, is a GameObject, or a gun

that the GameAgent picks up, is a GameObject. A ladder that a GameAgent climbs is a GameObject, yet a frame in the wall is not, if there is not any kind of interaction with it. GameObjects have their own properties, those being:

Die Situation is the condition under which the GameObject is dead-destroyed.

Object Properties is a list of properties that characterize this GameObject. For example, a frame in the wall owns an Object Property named "Hitpoints" and a GameAgents reduces this property's value throwing knives on the frame. An example is shown in Figure 18

Figure 18: GameObject

4.3.2 GameRegion

A GameRegion is a region on the scene of the game. A GameRegion owns a color property to graphically distinguish it from other GameEntities. The user can set a GameAgent to move on a GameRegion, or avoid a GameRegion through the use of custom procedures (see 4.6). For example, a GameAgent could be set to approach the GameRegion "FriendlyRegion" using the custom procedure "ApproachRegion".

4.3.3 GamePath

A GamePath is a path that a GameAgent can follow. GamePaths own a color property to graphically distinguish one from another. GamePaths also have checkpoints. The user can add checkpoints on the GameBoard to set the desired path. Then the user can set the GameAgent to follow the GamePath or cycle through the GamePath, using custom procedures (see 4.6).

4.4 NetLogo Code Generation

Next we are going to see how GameEntities and their properties are going to be formed in order to create the NetLogo project.

4.4.1 Set-Up NetLogo Procedure

The first procedure created is the Set-up procedure. In this procedure, all the GameEntities are initialized. GameAgents and GameObjects are created, GameRegions are drawn on the GameBoard and the checkpoints of the GamePaths are stored into lists. Ticks are reset and finally the Beliefs of the GameAgents are set to their initial value. This action takes place at the end of the procedure, because some Beliefs might need other GameAgents to have been created before initialization.

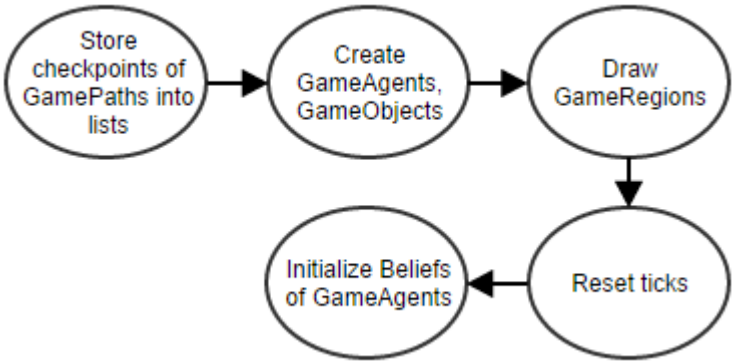

Figure 19: Set-up Procedure

4.4.2 GameAgent NetLogo Procedures

Each GameAgent owns a procedure named Set-up-(GameAgent Name). This procedure is called in the Set-up procedure (see 4.4.1). In this procedure, every instance of the GameAgent in the GameBoard is placed to it's coordinates and it's properties are initialized. Finally the "Top Level Goal" of the GameAgent is added as the first of it's intentions.

Figure 20: Set-up(GameAgent Name) Procedure

Each GameAgent owns a procedure named (GameAgent Name)-behavior. This procedure captures the behavior of the GameAgent. First thing in this procedure is to check if the GameAgent is dead. Next step for the GameAgent is to receive messages. Then the GameAgent proceeds to it's reactive behavior. Finally the GameAgent executes it's intentions. The procedure (GameAgent Name)-behavior is executed in every tick of the game. One and only action is taking place on every tick, meaning that if one of Maintenance Goals is executed, the procedure stops. If none of the reactive actions is executed, the GameAgent proceeds executing his next intention.

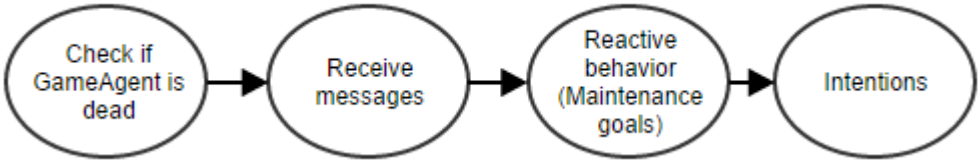

Figure 21: Set-up((GameAgent Name)-Behavior) Procedure

The reactive behavior of a GameAgent is his "Maintenance Goals". Those Goals are translated from Plans and Actions to if condition then action statements as shown in Figure 22.

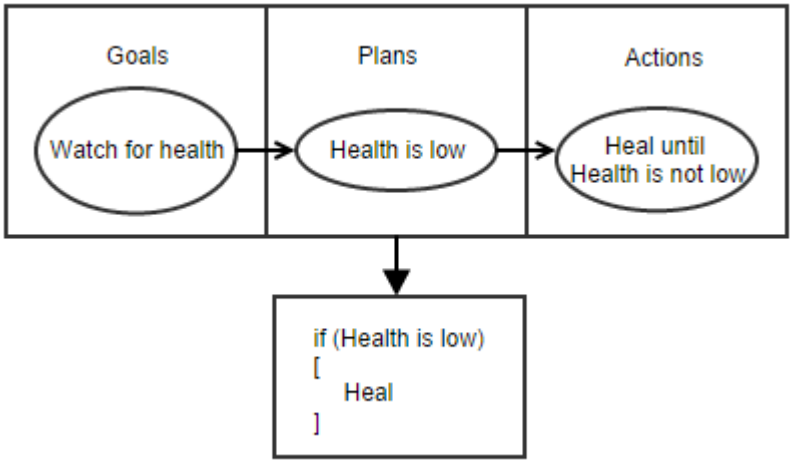

Figure 22: Maintenance Goal

Next step for a GameAgent is to create the procedures that map to his Goals. These procedures add intentions to the GameAgent's intention stack as shown in Figure 23. One procedure is created for every Goal named after the Name property of the Goal.

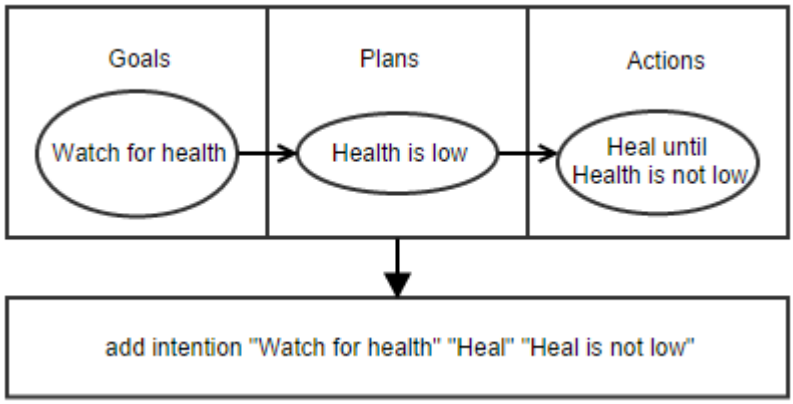

Figure 23: Goal

Then the Beliefs of the GameAgent are formed in the code. For every Belief a procedure and a reporter are created. A reporter that reports the value of the Belief named Get.(Belief Name) and a procedure that sets the value of the Belief named Set.(Belief Name). The reporter sets the value of the Belief according to the update code and update type and then reports that value.

Finally, Interactions and Situations are added to the code regarding the GameAgent. Interactions are simple procedures while Situations are reporters that report true or false.

4.4.3 GameObject NetLogo Procedures

Each GameObject owns a procedure named Set-up-(GameObject Name). This procedure is called in the Set-up procedure (see 4.4.1). In this procedure, every instance of the GameObject in the GameBoard is placed to it's coordinates and it's properties are initialized.

Each GameObject owns a procedure named (GameObject Name)-behavior. The procedure (GameAgent Name)-behavior is executed in every tick of the game. This procedure is used only if the GameObject owns an Object Property that needs to alter it's value on every tick.

For every Object Property of the GameObject a procedure and a reporter are created. A reporter that reports the value of the Object Property named Get.(Object Property Name) and a procedure that sets the value of the Object Property named Set.(Object Property Name). The reporter sets the value of the Object Property according to the update code and update type and then reports that value.

4.4.4 Check-game-end NetLogo Reporter

The reporter Check-game-end reports true if the game has ended, otherwise false. The game has ended if the Win Situation of a GameAgent reports true. All Win Situations are checked and if one of those is valid, then the game ends as shown in Figure 24.

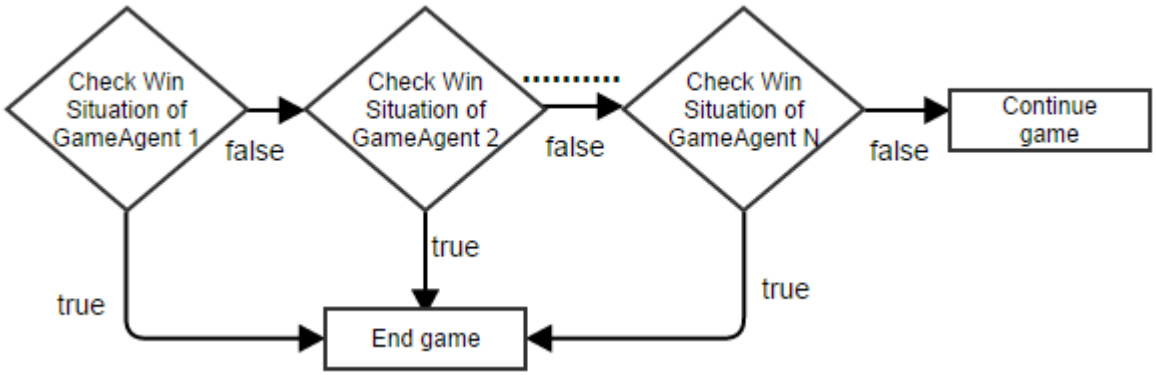

Figure 24: Check-game-end Reporter

4.4.5 Run-game NetLogo Procedure

Run-game procedure is the procedure that is executed in every tick of the game. In this procedure, Check-game-end procedure is called to determine if the game has ended. If not, all of the GameAgent and GameObject behavior procedures are called and tick increases by one.

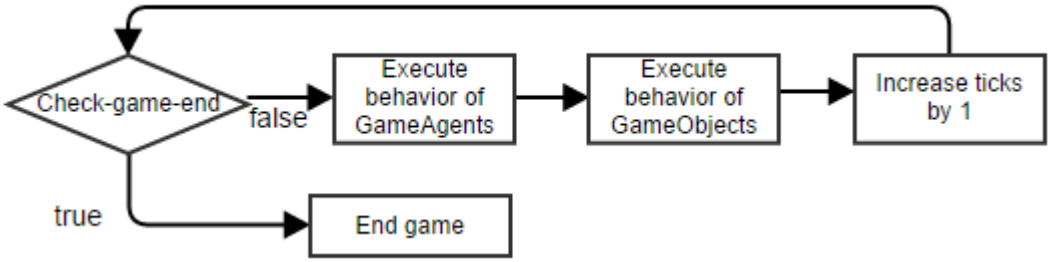

Figure 25: Run-game Procedure

4.4.6 NetLogo Procedure Calls

The following diagram displays the flow of the various procedure calls.

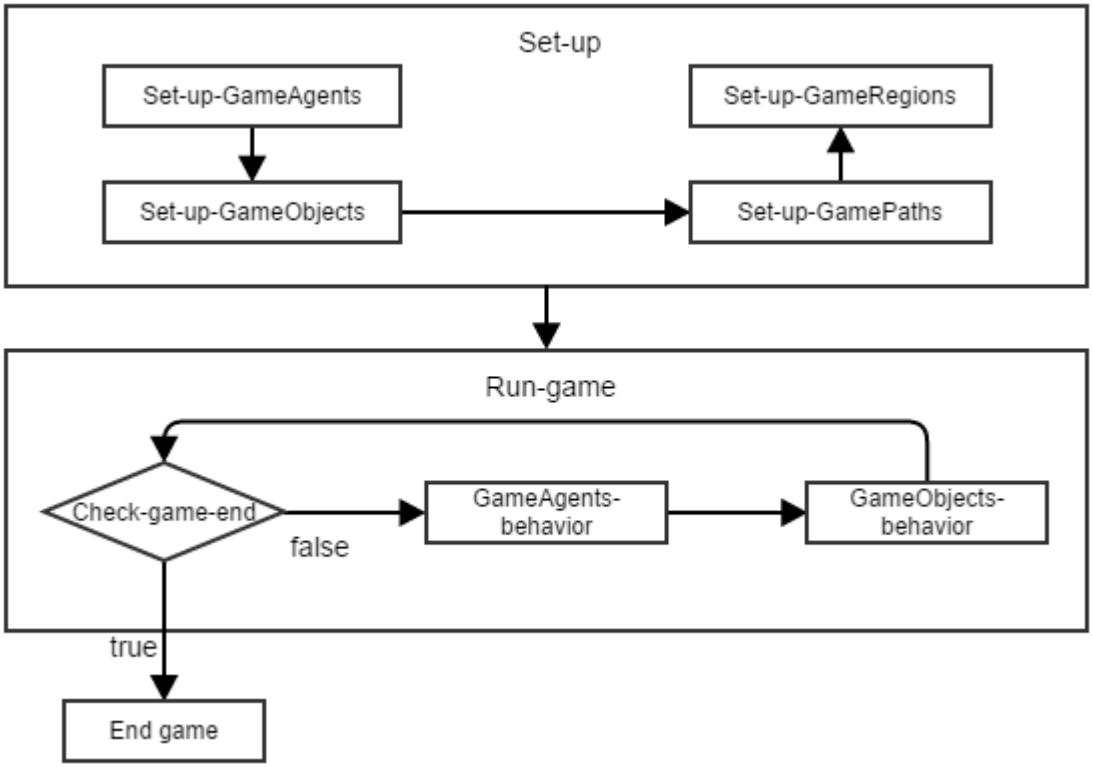

Figure 26: NetLogo Project Procedure Calls

4.5 Assisting Code Development

While the user types the code required for the game, a way is provided to refer to any of the properties of the GameEntities in NetLogo code as a reference. This is useful in case the user wants to change the name of a property.

For example: There is a Belief with Name "MyBelief" and this Belief is used in the code of Interaction "MyInteraction". To change the Belief's Name property, the user should also visit the Interaction's code and replace the Belief's Name with the new Name. Instead, if the Belief is set into the Interaction's code as a reference, the user should not do anything. The correct Belief's Name will set by itself through translation. Code translation is taking place right before the project is exported to NetLogo project.

4.5.1 Errors in Code

At the moment, there is no way to check if there are NetLogo syntax errors, but the user can be warned if a reference (see 4.5) to a property is error (for example, the property was deleted). Logical errors and warnings are also captured. Those errors and warnings can be the following:

Errors

Update code of a Belief is missing.

Code of an Interaction is missing.

Code of a Situation is missing.

A Goal does not have any Plans.

A Plan does not have any Actions.

A GameAgent does not have a "Top Level Goal".

GameAgents do not exist.

Warnings

A GameAgent does not have Win Situation.

A GameObject does not have Die Situation.

A GameObject does not have Die Situation.

A GameEntity does not have an instance on the GameBoard.

4.6 Custom Procedures

The user is provided with procedures that help in coding, making use of the present architecture. These are the most common procedures that a user will have to use in order to code for the game. For example the custom procedure Approach is used to move a GameAgent in a straight line towards another GameAgent. The custom procedure Avoid is used to move a GameAgent in a straight line away from another GameAgent. The custom procedure FollowPath is used to make a GameAgent follow a GamePath etc.. For a complete list of all the available custom procedures see Appendix A.

4.7 Messaging

GameAgents can send and receive messages, based on the KQML protocol. The Knowledge Query and Manipulation Language, or KQML, is a language and protocol for communication among software agents and knowledge-based systems. KQML messaging closely follows that of Jason (Jason [2014]). Messages can be of type-performative Inform, Achieve, Query.

Inform messages inform a GameAgent about something. In order to standardize the communication protocol, the receiver should own a Belief of UpdateType Message with a Label the same Label used in the message. An example is shown in Figure 27.

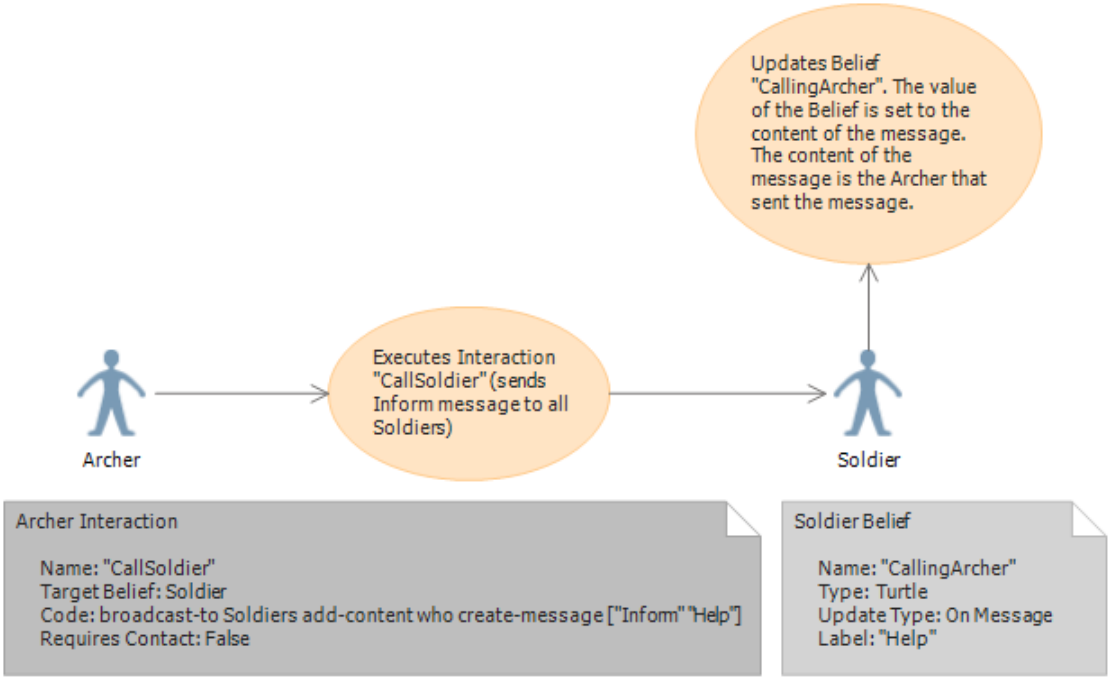

Figure 27: Message Inform

Achieve messages ask the receiver to achieve a Goal. The content of the message is a Goal Name of the receiver. An example is shown in Figure 28.

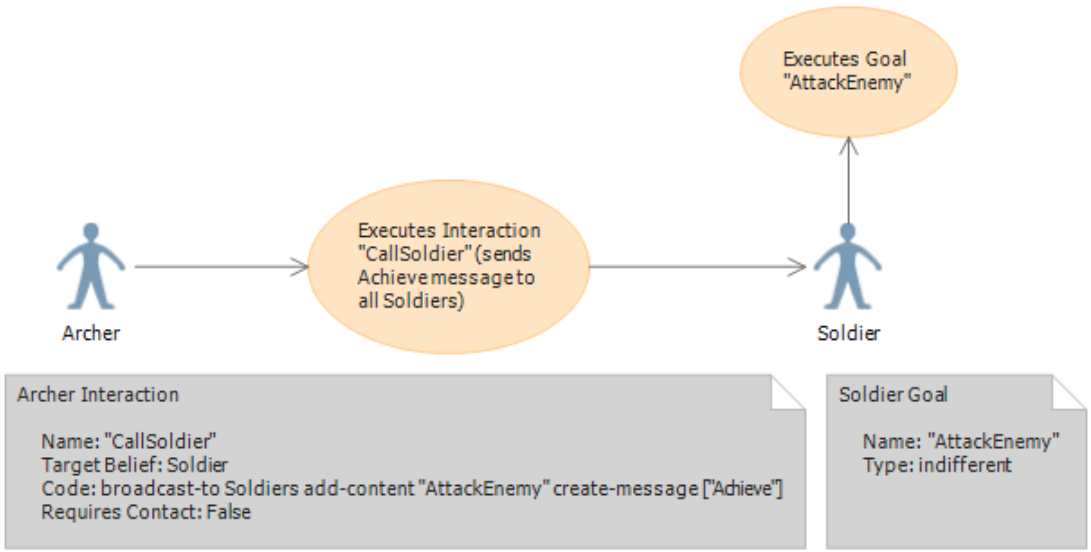

Figure 28: Message Achieve

Query messages ask the receiver about the value of a Belief that he has. The content of the message is a Belief Name of the receiver. The receiver automatically responds the value if his Belief with an Inform message. An example is shown in Figure 29.

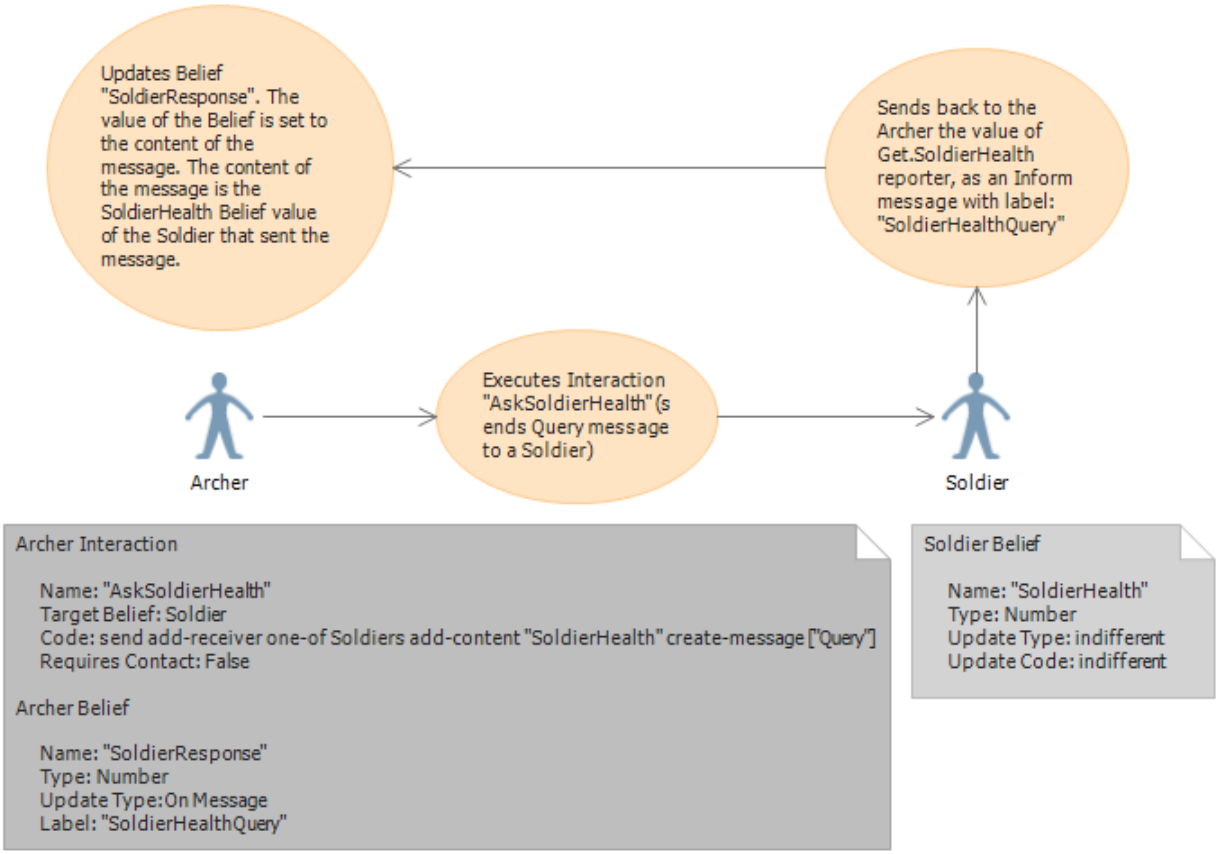

Figure 29: Message Query

4.8 Design

This chapter explains the design of N.G.B. web application, in order to export the created world (the game bots and the environment they live in) as NetLogo project. Following the structure of this chapter, we will discuss about:

- The main classes and their most significant properties and functions. Main classes are called the classes that hold data for GameEntities.
- The peripheral classes and their most significant properties and functions. Peripheral classes are called the classes that implement any necessary functionality other than those of the GameEntities.
- The user actions and layout of the required web pages.

4.8.1 Main Classes

The first step is to design the main classes that will hold the required data for the GameEntities. The application will consist of the classes shown in Table 2. Each class is described in details in the following sections.

Table 2: Main Classes

Class	**Description**
Project (4.8.1.1)	The name & description of the Project-game.
GameEntity (4.8.1.2)	The GameEntities of the Project-game.
Belief (4.8.1.3)	The Beliefs of the GameAgents.
Interaction (4.8.1.4)	The Interactions of the GameAgents.
Situation (4.8.1.5)	The Situations of the GameAgents.
Goal (4.8.1.6)	The Goals of the GameAgents.
Plan (4.8.1.7)	The Plans of the Goals.
PlanSituation (4.8.1.8)	The Situations of the Plans.
PlanAction (4.8.1.9)	The Actions of the Plans.
ObjectProperty (4.8.1.10)	The ObjectProperties of the GameObjects.
EntityOnBoard (4.8.1.11)	The instances of the GameEntities on the GameBoard.

4.8.1.1 Project

The top level concept of the application is the Project. A Project is the environment the GameEntities live in. Every Project has a Name, a Description and it's GameEntities, as shown in Table 3.

Table 3: Project Class

Property	**Description**
Name	A string holding the name of the Project.
Description	A string holding the description of the Project.
GameEntities	A list of GameEntity. An association with the class GameEntity (4.8.1.2).
IsCurrent	A boolean value.
IsPublic	A boolean value.

- **Description** A short description of the purpose of the Project.
- **GameEntities** A list containing all of the Project's GameEntities.
- **IsCurrent** A boolean value indicating if this Project is the current Project the user is editing. At any time, a user can edit only one Project.
- **IsPublic** A boolean value indicating if this Project is marked as public or private. If the Project is marked as private, then the Project is visible only to the user that owns the Project. If the Project is marked as public, any user will be able to create a copy of the Project in his personal Projects.

4.8.1.2 GameEntity

The main concept of the application is the GameEntity. GameEntity is an abstract class which GameAgent, GameObject, GameRegion and GamePath inherit from. GameEntity class provides common properties for all those classes as shown in Table 4. Apart from those properties, GameAgents and GameObjects are described with some specific properties. Those specific properties are shown in Table 5 for GameAgent and Table 6 for GameObject.

Table 4: GameEntity Class

Property	**Description**
Name	A string holding the name of the GameEntity.
Color	A string holding the color of the GameEntity.

- **Color** The color the GameEntity is going to be displayed in NetLogo.

Table 5: GameAgent Specific Properties

Property	**Description**
Shape	A string holding the shape of the GameAgent.
DieSituation	A string holding the corresponding NetLogo code (4.4).
WinSituation	A string holding the corresponding NetLogo code (4.4).
Beliefs	A list of Belief.
	An association with the class Belief (4.8.1.3).
Interactions	A list of Interaction.
	An association with the class Interaction (4.8.1.4).
Situations	A list of Situation.
	An association with the class Situation (4.8.1.5).
Goals	A list of Goal.
	An association with the class Goal (4.8.1.6).

- **Shape** The shape the GameAgent is going to be displayed in NetLogo.
- **DieSituation** The NetLogo code that defines when the GameAgent dies.
- **WinSituation** The NetLogo code that defines when the GameAgent wins.

Table 6: GameObject Specific Properties

Property	**Description**
Shape	A string holding the shape of the GameObject.
DieSituation	A string holding the corresponding NetLogo code (4.4).
ObjectProperties	A list of ObjectProperty. An association with the class ObjectProperty (4.8.1.10).

- **Shape** The shape the GameObject is going to be displayed in NetLogo.
- **DieSituation** The NetLogo code that defines when the GameObject is dead.

4.8.1.3 Belief

Belief is a key concept for a GameAgent. They are the GameAgent's knowledge about the environment and his internal state. Belief properties are described in Table 7.

Table 7: Belief Class

Property	**Description**
Name	A string holding the name of the Belief.
Type	A string holding the type of the Belief.
StartingValue	A string holding the corresponding NetLogo code (4.4).
UpdateCode	A string holding the corresponding NetLogo code (4.4).
UpdateType	A string holding the Belief's update type.

- **Type** The Type of the Belief accepts the values {Number, Turtle}.

  Number: The Belief is a numeric value.

  Turtle: The Belief points to a turtle.

- **StartingValue** NetLogo code used to initialize the value of the Belief when the game starts.
- **UpdateCode** NetLogo code used to update the value of the Belief, every time asked.
- **UpdateType** The UpdateType property of the Belief accepts the values {Static, Every time, Only if nobody, On every tick, On message}.
  - Static: The value of the Belief is set once when the game starts, using the StartingValue. The value of the Belief can also be set manually when necessary during the game.
  - Every time: The value of the Belief is updated every time asked using *UpdateCode*. The value of the Belief cannot be set manually during the game.
  - Only if nobody: The value of the Belief is updated every time asked using UpdateCode, but only if the Belief value is equal to nobody. Used with Beliefs of Type Turtle. The value of the Belief can also be set manually when necessary during the game.
  - On every tick: The value of the Belief is updated on every NetLogo tick. It can be set manually when necessary during the game.
  - On message: The value of the Belief is updated when an Inform message is received (see Messaging 4.7). To distinguish between multiple Inform messages, UpdateCode is used as a Label for the message. The value of the Belief can also be set manually when necessary during the game.

4.8.1.4 Interaction

An Interaction describes a GameAgent's interaction with the environment. Interactions' properties, as shown in Table 8.

Table 8: Interaction Class

Property	**Description**
Name	A string holding the name of the Interaction.
Code	A string holding the corresponding NetLogo code (4.4).
TargetBelief	An integer holding the ID of a Belief (4.8.1.3)
RequiresContact	A boolean value.

- **Code** The NetLogo code that is executed when the Interaction is executed.
- **TargetBelief** This property is not a reference to the class Belief, because that would cause cyclical reference. It just stores the ID of the Belief of Type Turtle, that this Interaction targets. If the Interaction targets no Belief, it's value is set to -1.
- **RequiresContact** A boolean value defining if the Code requires contact with the TargetBelief before execution.

4.8.1.5 Situation

A Situation reports if a condition is true or false. Situations have a Name, and a Code property, as shown in Table 9.

Table 9: Situation Class

Property	**Description**
Name	A string holding the name of the Situation.
Code	A string holding the corresponding NetLogo code (4.4).

- **Code** The NetLogo code that is executed when the Situation is called. This code reports true or false.

4.8.1.6 Goal

A Goal describes the GameAgents' goals. Goals have a Name and a Type property, as shown in Table 10.

Table 10: Goal Class

Property	**Description**
Name	A string holding the name of the Goal.
Order	An integer holding the order of the Goal.
Type	A string holding the type of the Goal.

- **Order** The order that this Goal is checked. Order is important only for Maintenance Goals, as described below.
- **Type** The Type of a Goal accepts the values {Top Level Goal, Maintenance Goal, Simple Goal}.
  - Top Level Goal: A Goal of this Type is the Goal that the GameAgent executes forever. It can be seen as the main obligation of the GameAgent. A GameAgent can have only one Top Level Goal. Order is not of importance in this type of Goal.
  - Maintenance Goal: A GameAgent checks if a Maintenance Goal should be achieved, on every tick of the game. Maintenance Goals describe the reactive behavior of the GameAgent. A GameAgent can have multiple Maintenance Goals. Order is important in this type of Goal. Maintenance Goals will be checked in order.
  - Simple Goal: A Goal of this Type is the Goal that the GameAgent can choose to achieve while executing another Goal. A GameAgent can have multiple Simple Goals. Order is not of importance in this type of Goal.

4.8.1.7 Plan

A Plan makes use of Interaction and Situation classes, as shown in Table 11.

Table 11: Plan Class

Property	**Description**
Order	An integer holding the order of the Plan.
PlanSituations	A list holding the PlanSituations of the Plan. An association with the class PlanSituation (4.8.1.8)
PlanActions	A list holding the PlanActions of the Plan. An association with the class PlanAction (4.8.1.9)

- **Order** The order that this Plan is checked-executed.
- **PlanSituations** The Situations that must report true, so that the Plan's PlanActions get executed.
- **PlanActions** The actions (Interactions and/or Goals) that this Plan must execute in order to be accomplished.

4.8.1.8 PlanSituation

PlanSituation class is like a bridge class connecting a Plan with one or more required Situations, so it's only properties are the ID of a Plan along with the ID of a Situation, as shown in Table 12.

Table 12: PlanSituation Class

Property	**Description**
PlanID	An integer holding the ID of a Plan (4.8.1.7).
SituationID	An integer holding the ID of a Situation (4.8.1.5).

4.8.1.9 PlanAction

PlanAction class is like a bridge class connecting a Plan with an action (Interaction and/or Goal) and Situation, so it's only properties are the ID of an action along with the ID of the Situation and the ActionType, as shown in Table 13.

Table 13: PlanAction Class

Property	**Description**
ActionType	A string holding the type of the action.
ActionID	An integer holding the ID of the action (4.8.1.4) (4.8.1.6).
SituationID	An integer holding the ID of the Situation (4.8.1.5).

- **ActionType** The ActionType property of a PlanAction accepts the values {Interaction, Goal}
  - Interaction: Means that the action of the PlanAction is an Interaction.
  - Goal: Means that the Action of the PlanAction is a Goal.
- **ActionID** If ActionType is set to "Interaction", ActionID holds the ID of an Interaction. If ActionType is set to "Goal", ActionID holds the ID of a Goal.
- **SituationID** This property holds the ID of a Situation. When this Situation reports true, the PlanAction stops executing.

4.8.1.10 ObjectProperty

An ObjectProperty inherits Belief class, as shown in Table 14.

Table 14: ObjectProperty Class

Property	**Description**
Name	A string holding the name of the ObjectProperty.
Type	A string holding the type of the ObjectProperty.
StartingValue	A string holding the corresponding NetLogo code (4.4).
UpdateCode	A string holding the corresponding NetLogo code (4.4).
UpdateType	A string holding the ObjectProperty's update type.

- **Type** The Type of the ObjectProperty accepts the values {Number, Turtle}.
  - Number: The ObjectProperty is a numeric value.
  - Turtle: The ObjectProperty points to a turtle.
- **StartingValue** NetLogo code used to initialize the value of the ObjectProperty when the game starts.
- **UpdateCode** NetLogo code used to update the value of the ObjectProperty, every time asked.
- **UpdateType** The UpdateType property of the ObjectProperty accepts the values {Static, Every time, Only if nobody, On every tick, On message}.
  - Static: The value of the ObjectProperty is set once when the game starts, using the StartingValue. The value of the ObjectProperty can also be set manually when necessary during the game.
  - Every time: The value of the ObjectProperty is updated every time asked using UpdateCode. The value of the ObjectProperty cannot be set manually during the game.
  - Only if nobody: The value of the ObjectProperty is updated every time asked using *UpdateCode*, but only if the ObjectProperty value is equal to nobody. Used with ObjectProperty of Type Turtle. The value of the ObjectProperty can

also be set manually when necessary during the game.

- On every tick: The value of the ObjectProperty is updated on every NetLogo tick. It can be set manually when necessary during the game.
- On message: The value of the ObjectProperty is updated when an Inform message is received (see Messaging 4.7). To distinguish between multiple Inform messages, UpdateCode is used as a Label for the message. The value of the ObjectProperty can also be set manually when necessary during the game.

4.8.1.11 EntityOnBoard

The class EntityOnBoard holds the instances of each GameEntity on the GameBoard. This class holds the X position, Y position, width and height of the instance, as shown in Table 15.

Table 15: EntityOnBoard Class

Property	**Description**
PosX	A double holding the X position of the GameEntity.
PosY	A double holding the Y position of the GameEntity.
Width	A double holding the Width of the GameEntity.
Height	A double holding the Height of the GameEntity.

4.8.1.12 Class Diagram

Figure 30 shows the class diagram of the main classes described above.

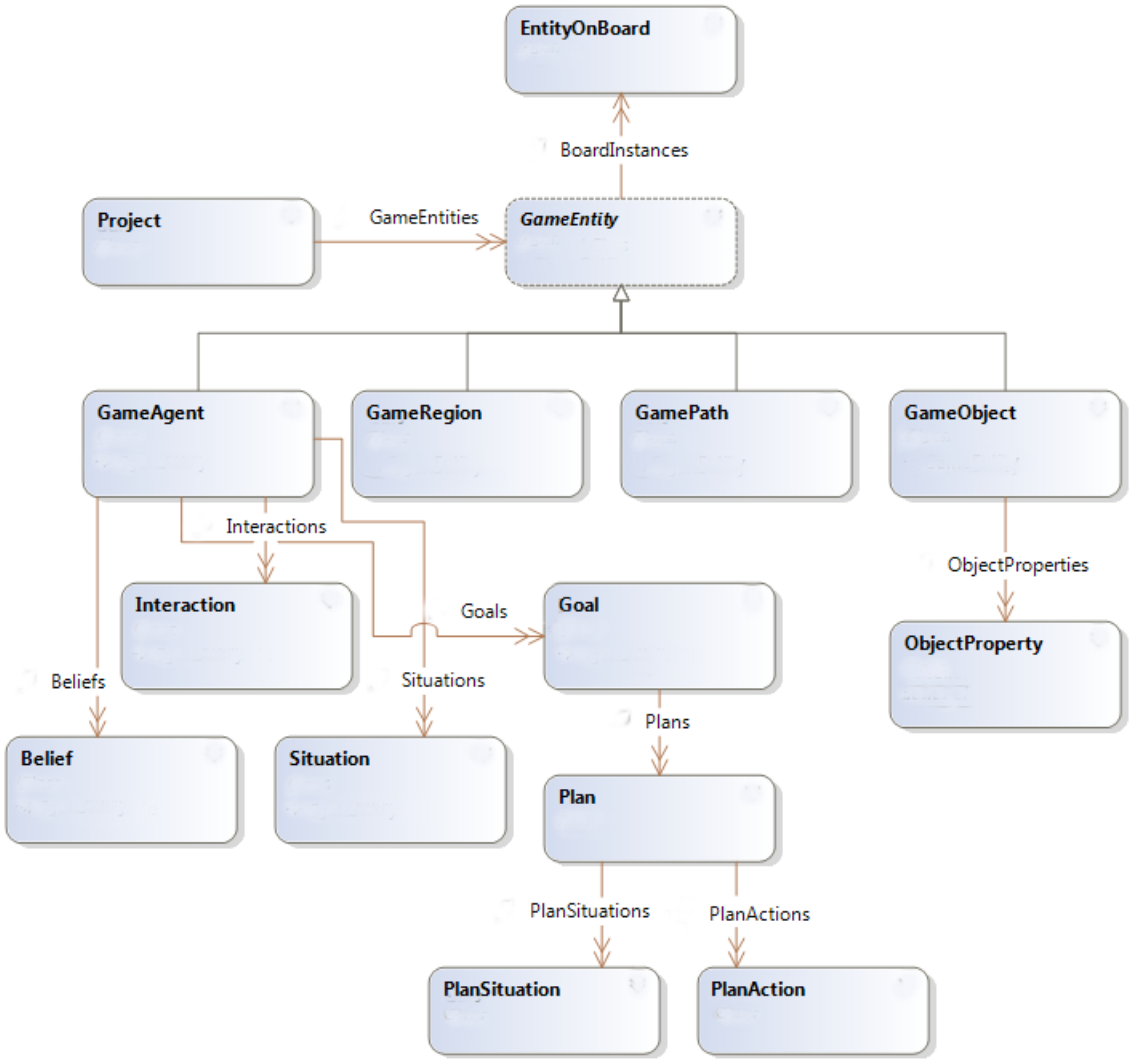

Figure 30: Class Diagram of Main Classes

4.8.2 Peripheral Classes

The next step is to design the classes that will hold the users' data and implement any other necessary functionality. Table 16 shows those classes.

Table 16: Application Peripheral Classes

Class	**Description**
ApplicationUser (4.8.2.1)	The application's user.
Contact (4.8.2.2)	The application's comments made by users.
FileSystem (4.8.2.3)	The functionality concerning the file system.
ErrorsWarnings (4.8.2.4)	The application's functionality for errors and warnings.
ExportCode (4.8.2.5)	The functionality for exporting a Project as NetLogo project.
Translate (4.8.2.6)	The functionality concerning code translation.
Log (4.8.2.7)	The application's logging functionality.

4.8.2.1 ApplicationUser

An ApplicationUser is a user of the application. This class holds the user name, password hash and user projects as shown in Table 17.

Table 17: ApplicationUser Class

Property	**Description**
UserName	A string holding the name of the ApplicationUser.
Password	A string holding the password hash of the ApplicationUser.
Projects	A list of Project. An association with the class Project (4.8.1.1)

4.8.2.2 Contact

Contact class holds the user email and comment, as shown in Table 18.

Table 18: Contact Class

Property	Description
Email	A string holding the email of the user.
Comment	A string holding the comment of the user.

4.8.2.3 FileSystem

This class implements the functionality concerning communication with the server's file system. It's main functions are shown in Table 19.

Table 19: FileSystem Class

Function	Description
GetAvailableShapes( )	Returns a list containing the available NetLogo shapes.
GetAvailableFunctions( )	Returns a list containing the available NetLogo functions.

- **GetAvailableShapes( )** Reads from the file "Prototype.nlogo" and returns the available shapes in a list.
- **GetAvailableFunctions( )** Reads from the file "Prototype.nlogo" and returns the available custom procedures and custom reporters in a list.

4.8.2.4 ErrorsWarnings

This class implements the functionality concerning errors and warnings in any Project. It's main functions are shown in Table 20.

Table 20: ErrorsWarnings Class

Function	Description
GetErrorsWarnings( )	Returns a list of errors/warnings in the given Project.
CheckIfNameExists( )	Returns true if a name already exists in a given Project.

- **GetErrorsWarnings( )** Scans the given Project (4.8.1.1) and returns a list of errors and/or warnings, if any (see 4.5.1). This list is used to display those errors/warnings to the user.
- **CheckIfNameExists( )** Scans the given Project (4.8.1.1) and returns true if a given Name is already used by another property (Belied, Interaction etc.) in the given Project. This function is used because in NetLogo it is forbidden for two or more functions to use the same signature name.

4.8.2.5 ExportCode

This class implements the functionality for exporting any Project as NetLogo project. It's main function is shown in Table 21.

Table 21: ExportCode Class

Function	**Description**
ExportNetLogo( )	Returns a string containing the NetLogo code for the given Project.

4.8.2.6 Translate

This class implements the functionality for translating any block of NetLogo code (see NetLogo Code 4.4). It's main functions are shown in Table 22.

Table 22: Translate Class

Function	**Description**
TranslateCode( )	Returns a string containing the translated code block.
HasErrors()	Returns true if the given Project (4.8.1.1) has any translation errors.

- **TranslateCode( )** Scans the given code block and translates any references back to their Name property. Returns the translated code (see 4.8.2.6).
- **HasErrors( )** Scans the given Project (4.8.1.1) and returns true if the Project has any translation errors (see 4.5.1).

4.8.2.7 Log

This class implements the necessary functionality for the application's logging system. It's main functions are shown in Table 23.

Table 23: Log Class

Function	Description
InstertLog( )	Insterts a log record to the database.
InstertErrorLog( )	Log the error-exception to a file.

4.8.2.8 Class Diagram

Figure 31 shows the class diagram as described above.

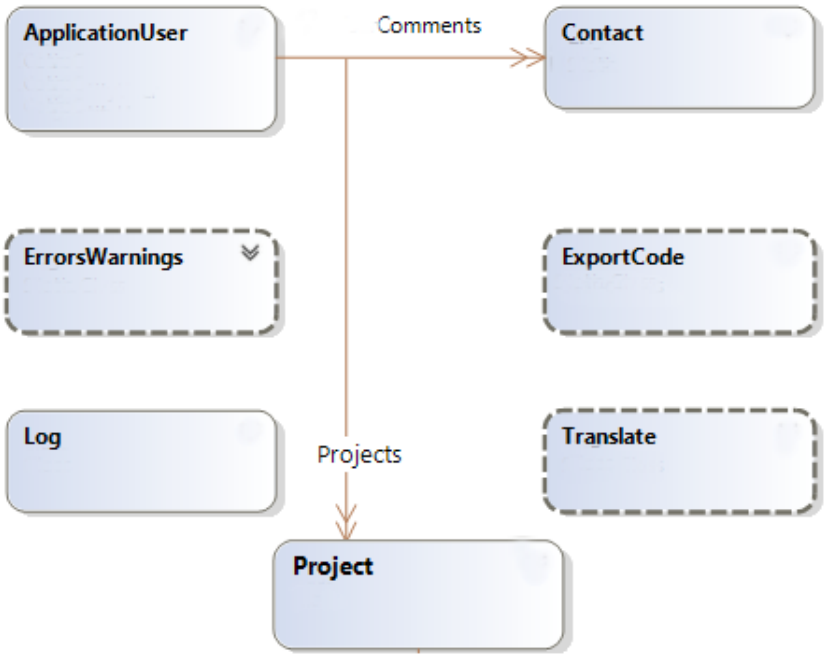

Figure 31: Class Diagram of Peripheral Classes

4.8.3 Web Application

The final step is to design the web application. We will discuss about what a user will be able to do in every page, and present a layout for every page.

4.8.3.1 User

Once an user logs in to the web application, the available actions are shown in Figure 32, while Figure 33 shows the layout.

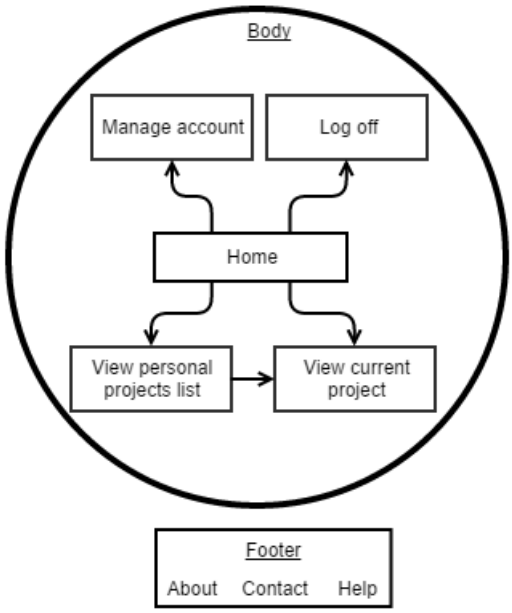

Figure 32: Website: User Actions

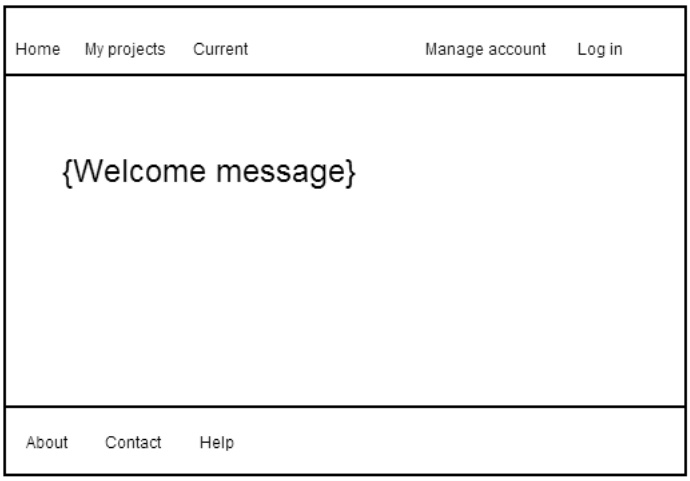

Figure 33: Website: User Layout

4.8.3.2 User & Projects

Then, the user is able to perform the actions concerning private or public projects, as shown in Figure 34. The corresponding layouts follow in Figures 35, 36 and 37.

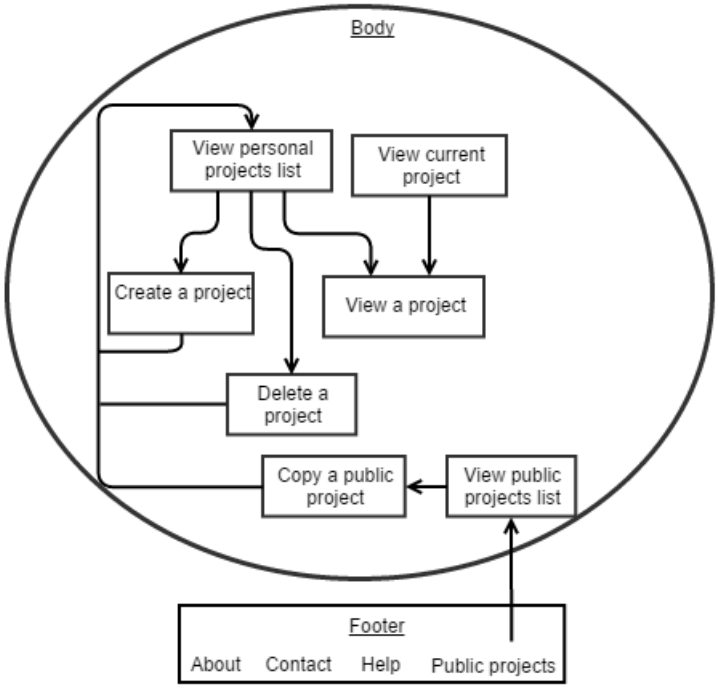

Figure 34: Website: Project Actions

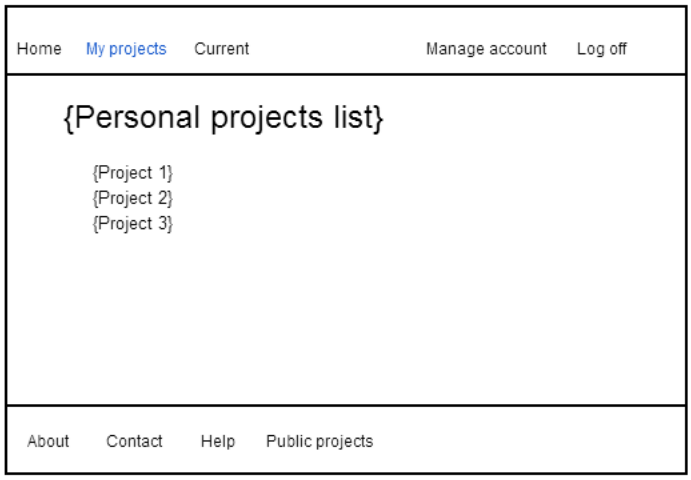

Figure 35: Website: Private Projects Layout

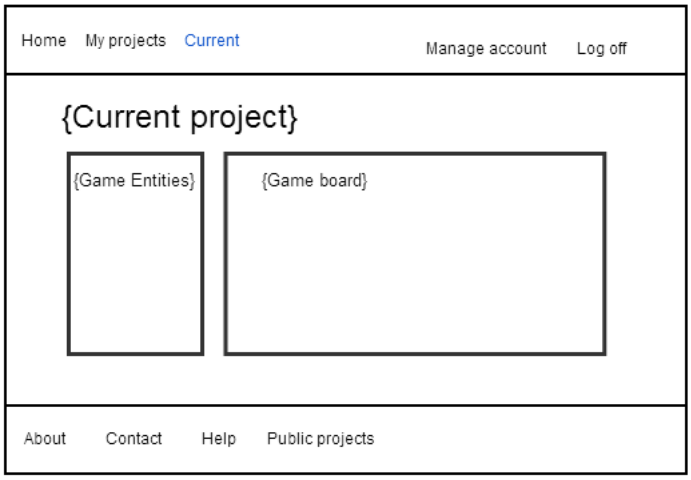

Figure 36: Website: Current Project Layout

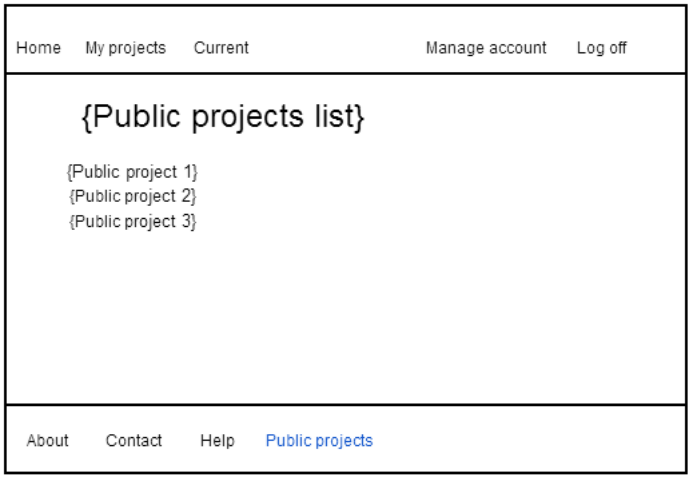

Figure 37: Website: Public Projects Layout

5 Implementation

This Chapter explains the implementation of the design of the web application, based on the design described in the previous Chapter 4.8. In this Chapter we will see:

- The tools, patterns and technologies used for the implementation.
- A description on how the tools and technologies were used.
- An overview of the look and feel and basic navigation in the web application.

5.1 Technologies

The tools, technologies and patterns used are listed in Table 24.

Table 24: Implementation Technologies

Tool/Technology/Pattern	**Used**
IDE	Microsoft Visual Studio 2013 Express Edition
Programming Language	C#, HTML, JavaScript
ORM [6]	Entity Framework 6 [7]
Design Pattern	Repository Pattern
UI Pattern	MVC [8]
Web Application Framework	ASP.NET MVC [9]
Database	Microsoft SQL Server 2012 Express Edition

5.2 MVC

The project created with Microsoft Visual Studio is an ASP.NET Web Application project that uses MVC template (Figure 38). This template generates various project folders, with the most important being *Model*, *View* and *Controller* folders.

Model Contains all the classes as described in Chapter 4.8.

View Contains all the HTML, CSS and JavaScript code used for the UI.

Controller Contains all the necessary controller classes [10] that update the Model and return the updated View to the user.

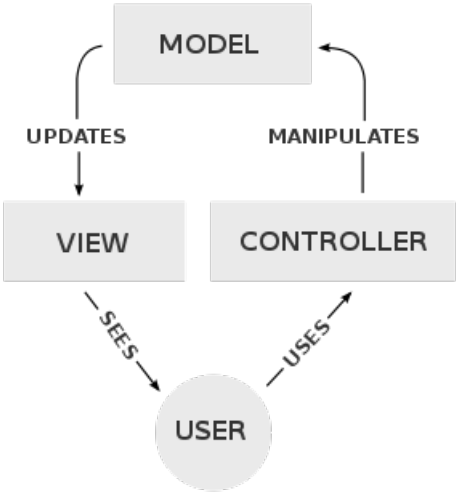

Figure 38: MVC Process
Source: wikipedia

[6]Object-Relational Mapping. A programming technique for converting data between incompatible type systems in object-oriented programming languages.

[7]Entity Framework (EF) is an open source object-relational mapping (ORM) framework for ADO.NET, part of .NET Framework.

[8]Model View Controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user

[9]ASP.NET MVC is an open source web application framework that implements the MVC pattern.

[10]A controller class provides methods that respond to HTTP requests that are made to an ASP.NET MVC Web application.

5.3 Entity Framework & Database Generation

Once the Model folder (the folder containing the classes) is filled with the required classes, EF takes over to create the database. The first time the project is run, EF uses the given connection string, connects to SQL Server and generates the database.

C# provides multiple ways to let the programmer define table names, column names, column types etc.. The way we used is called data annotations (MSDN). For example, Figure 39 displays data annotations used in class GameAgent.

```
public class GameAgent : GameEntity
{

    [Key]
    public int ID { get; set; }

    [Required]
    [StringLength(30, MinimumLength = 3, ErrorMessage = "Length must be between 3-30")]
    [NameSymbols]
    [NameExists]
    public String Name { get; set; }

    [Required]
    [StringLength(100)]
    public String Color { get; set; }

    [Required]
    [StringLength(100)]
    public String Shape { get; set; }
```

Figure 39: GameAgent Class: Data Annotations

The generated database schema of the application is shown in Figures 41, 42, while Figure 40 displays the layers used in an application that makes use of EF.

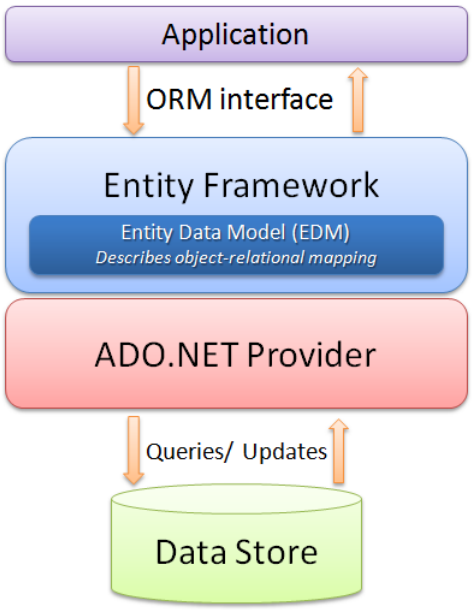

Figure 40: Entity Framework
Source: MSDN

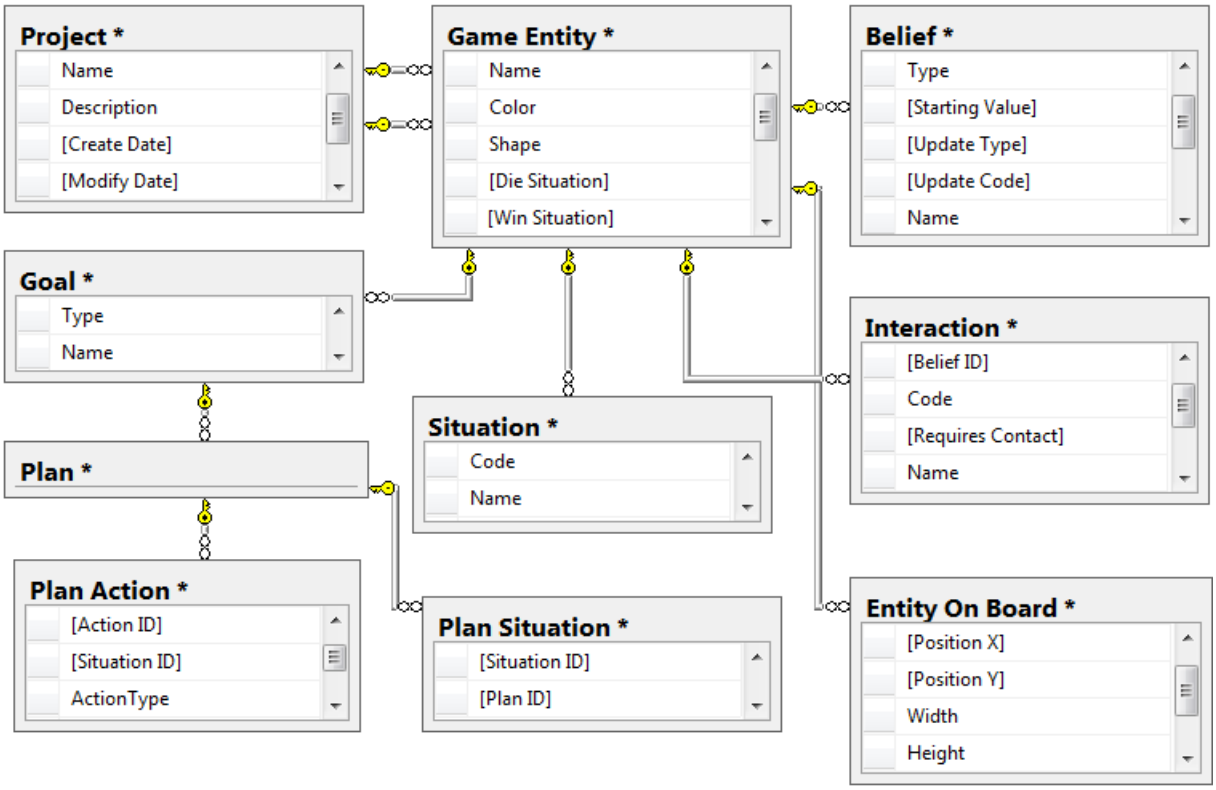

Figure 41: Database Schema: Main Tables

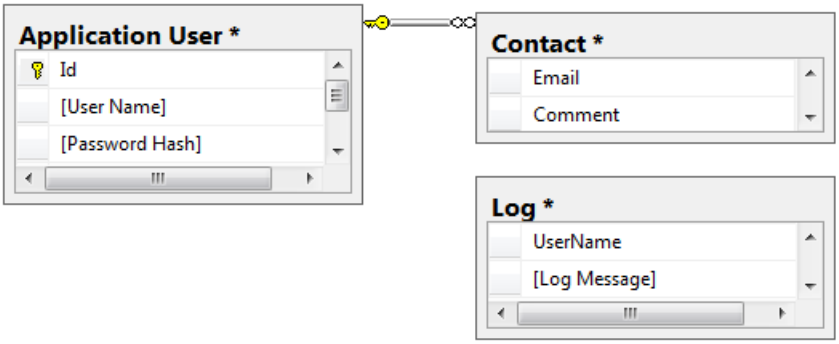

Figure 42: Database Schema: Peripheral Tables

5.4 The Web Application

5.4.1 Home

Once the web application is run, *Home* page loads as shown in Figure 43.

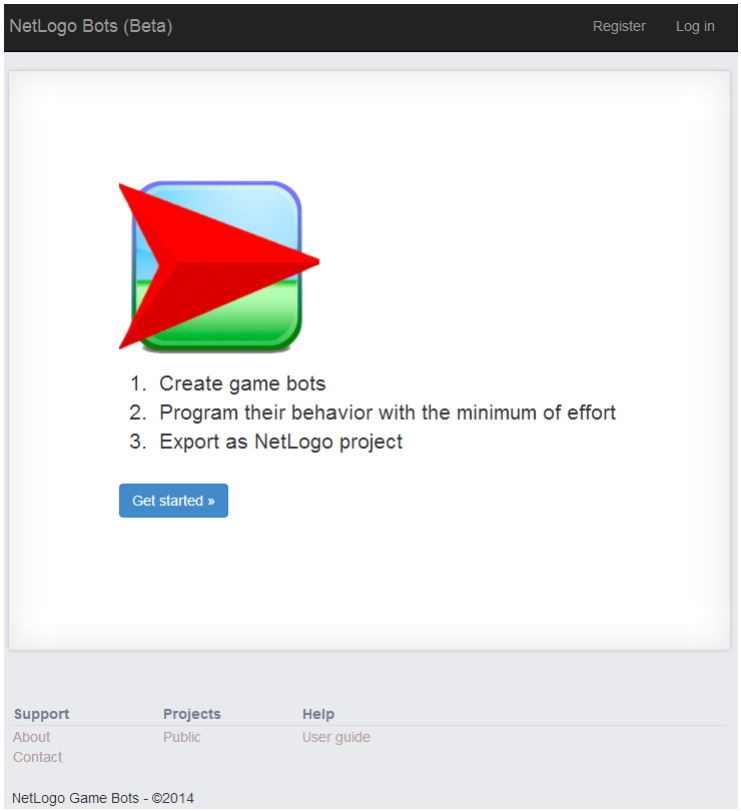

Figure 43: Home Page

5.4.2 Register

A user should register to create an account as follows on Figure 44. In this page the user has to provide a user name, a password to be able to log in, and a CAPTCHA [11].

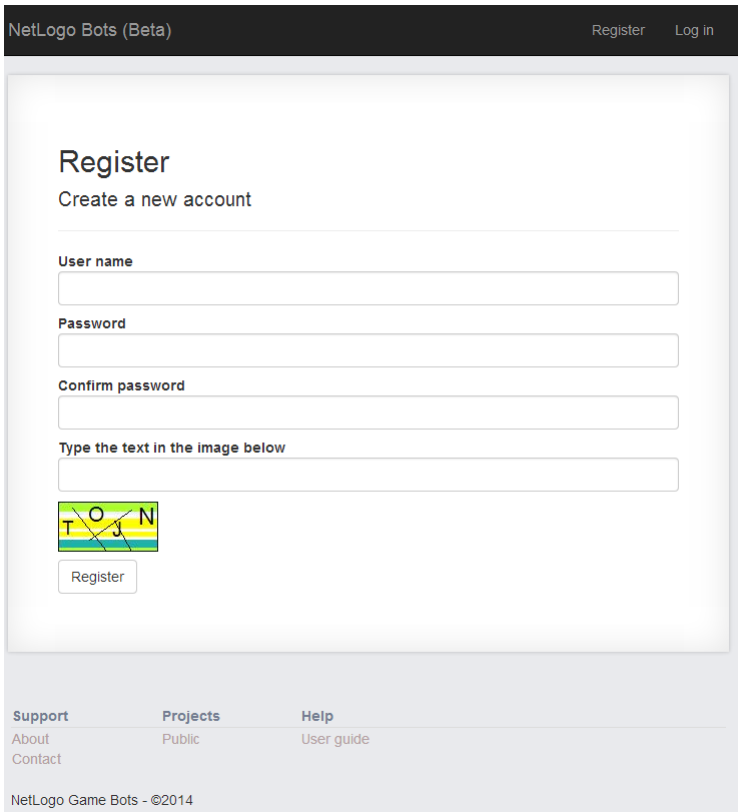

Figure 44: Register Page

5.4.3 My Projects

Once a user registers, the application gets redirected to the *Home* page. The user is now able to access *My projects* page as shown in Figure 45.

[11] CAPTCHA (an acronym for "Completely Automated Public Turing test to tell Computers and Humans Apart") is a type of challenge-response test used in computing to determine whether or not the user is human.

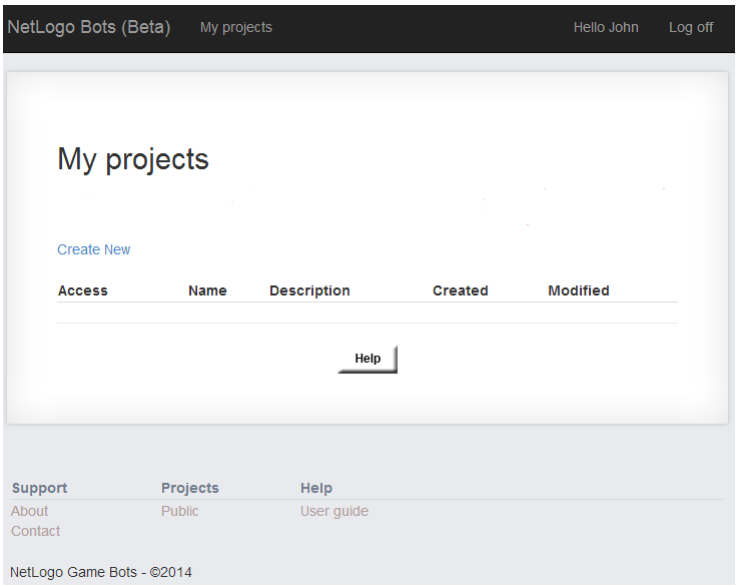

Figure 45: My projects Page

5.4.4 Create Project

The user is in position to create a new project selecting the link *Create new* in page *My projects*. To create a project the user has to provide a project name and optionally a description as shown in Figure 46. After creating the project, the user gets redirected to page *My projects* (Figure 47) where the new project is listed. In this page the user is able to rename/delete/edit the project and mark the project as public.

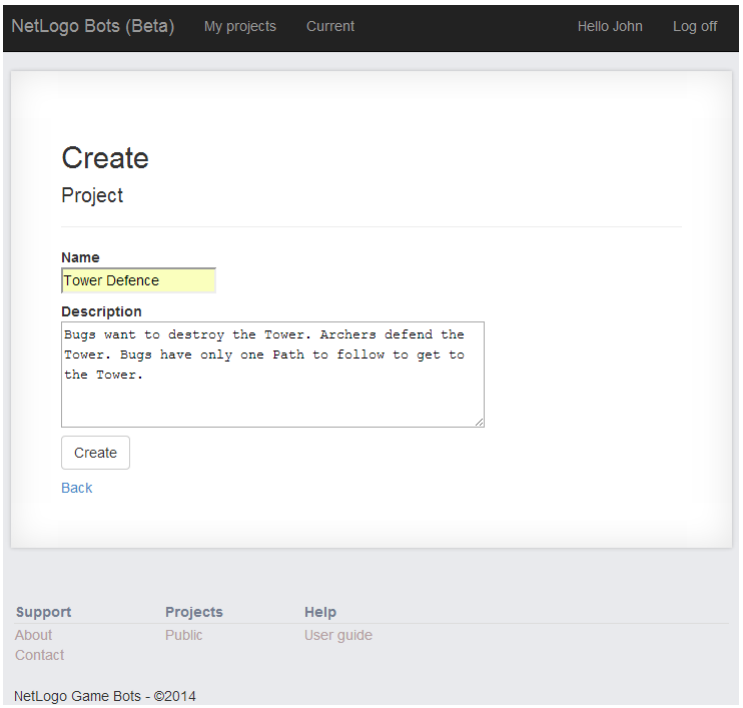

Figure 46: Create New Project Page

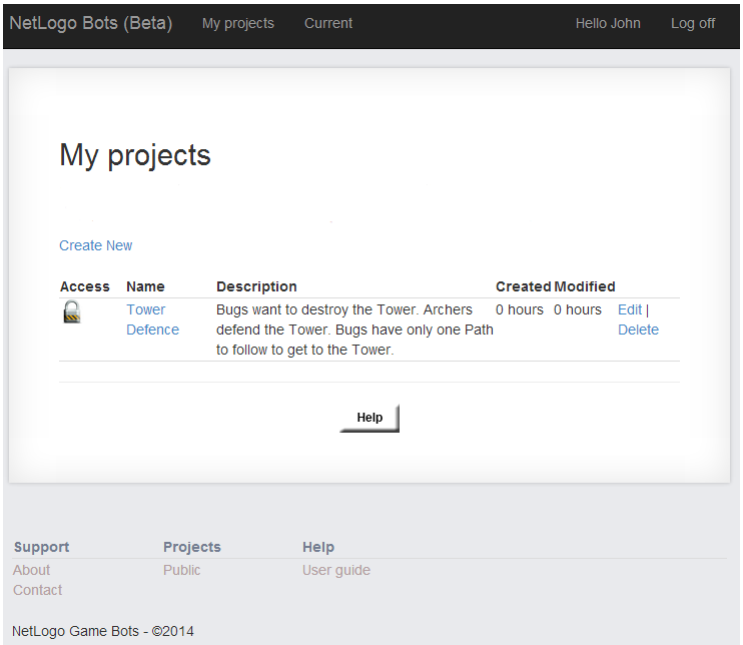

Figure 47: My Projects Page

5.4.5 Edit Project

Selecting the link of the project (the project name) in page *My projects* the user enters inside this project and he is ready to start adding GameEntities and editing their properties, as shown in Figure 48.

At the bottom of this page (and every page in the web application) there is a *Help* button. The user can select the button to get more information about what he is able to do on the current page. What a user can do in this page is shown in Figure 49.

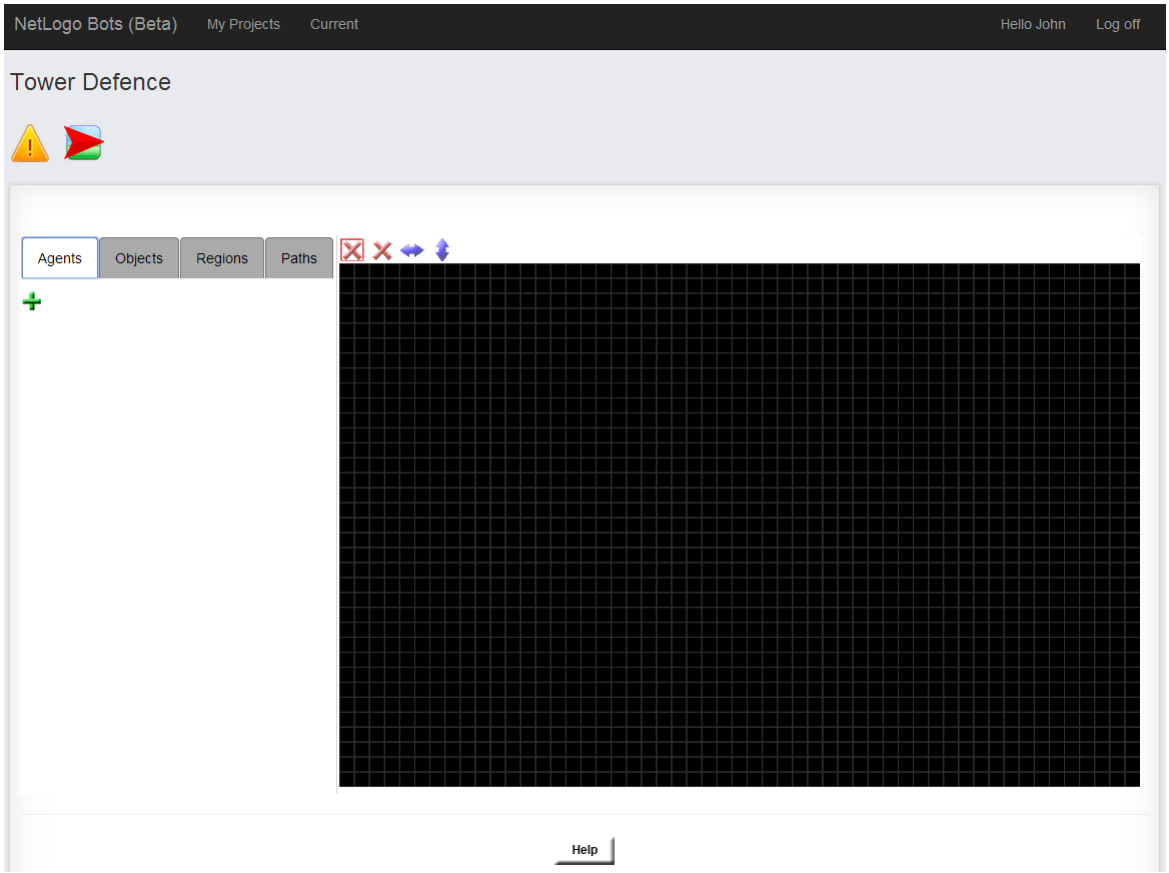

Figure 48: Edit Project Page

- **This page displays the Agents, Objects, Regions and Paths of the game.**
  The term "Entities" refers to the Agents, Objects, Regions and Paths.
  - Entities
    - Select to add an Entity.
    - Select to delete an Entity.
    - & Drag n drop to the board (black canvas) to add the Entity to the board.
  - Board
    - Drag the Entities to move them around on the board.
    - Resize Regions by dragging those from the bottom right corner.
    - Select to delete all Entities from the board.
    - Select to delete the selected Entitiy from the board.
    - Select to spread the width of the selected Region to cover the entire board.
    - Select to spread the height of the selected Region to cover the entire board.
  - Project
    - Select to download the project as NetLogo project.
    - Select to get a list of logical errors and/or warnings in the project.

Figure 49: Edit Project Page - Help

5.4.6 Detect Errors in Project

Selecting the icon shown in Figure 50 opens a new tab in the browser that displays any errors and/or warnings (see 4.5.1). Figure 51 shows the case that there are not any errors/warnings.

Figure 50: Detect Errors/Warnings Icon

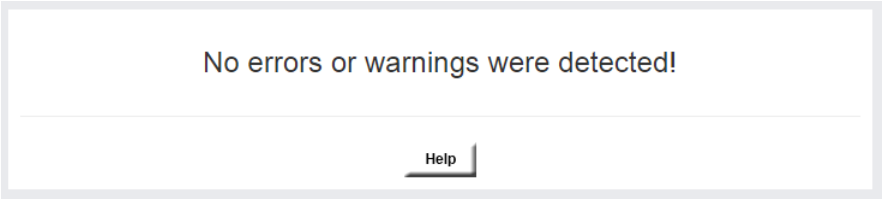

Figure 51: Error/Warning Page

5.4.7 Export to NetLogo Project

Selecting the icon shown in Figure 52 will generate and download the Project as NetLogo project (see 4.4). The downloaded file will be named after the Project Name

property. For example, if the Project Name is "MyProject" the downloaded file will be named as "MyProject.nlogo".

Figure 52: Export to NetLogo Icon

6 Case Study

In this Chapter, we will use the web application described in Chapters 4, 5, to create a project and export as NetLogo project. The project will be a Tower Defense game. Tower Defense is a sub-genre of real-time strategy video games. The goal of Tower Defense games is to try to stop enemies from crossing a map by building traps to slow them down and towers which shoot at them as they pass. In our case, there will be no traps and towers but archers and soldiers that attack the enemies. The process is described in steps followed by the corresponding figures and tables.

6.1 Game description

The GameAgents of the game will be Bugs, Archers and Soldiers. Bugs have to choose between two different GamePaths to approach the GameObject Tower. As soon as Bugs are next to the Tower they attack the Tower. Archers shoot the Bugs on sight. Archers also send a message to inform the Soldiers that Bugs are on their way to the Tower. Soldiers, as soon as they get the message, they approach the calling Archer to offer help in killing Bugs. A high level description of the GameEntities behavior follows. Figure 53 will help in better understanding the game scenario.

6.1.1 Bug

- Bugs wait for some time before approaching the Tower. During that time, Bugs move randomly.
- After that time, Bugs randomly select one of the GamePaths (BugPath1 or BugPath2) and follow it until the end.
- When Bugs reach the end of the selected GamePath, they attack the Tower.
- Bugs own a health property. When health drops below zero, Bugs die.

6.1.2 Archer

- Archers wait in their position. They own a view range property. If a Bug enters their view range, they shoot the Bug.
- Archers also send a message to Soldiers calling for help.

6.1.3 Soldier

- Soldiers patrol their area following SoldierPath.
- If an Archer calls them, they leave the GamePath and approach the calling Archer offering help in killing Bugs.
- Soldiers attack Bugs on sight and follow them until someone wins the game.

6.1.4 Tower

- Tower owns a health property. When health drops below zero, the Tower is destroyed.

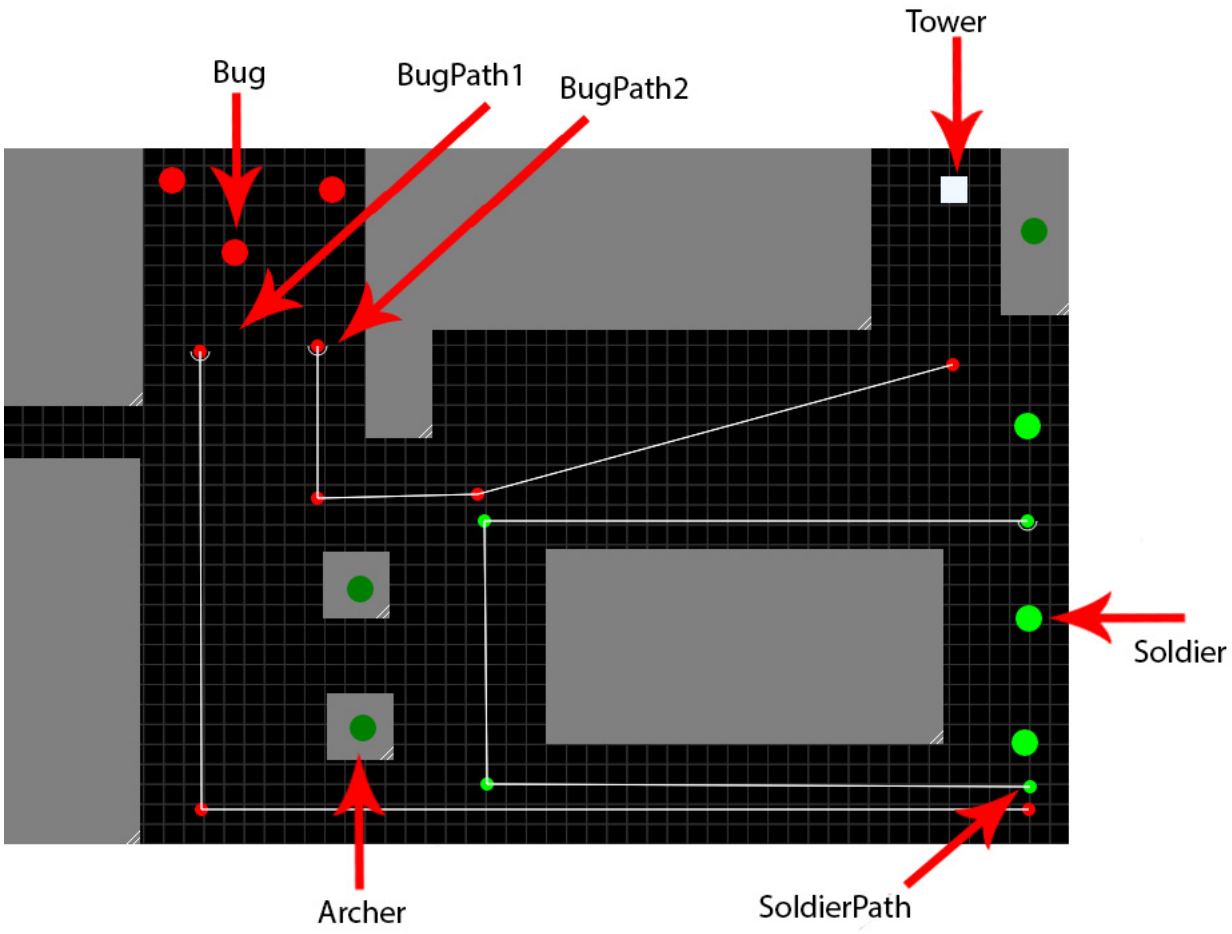

Figure 53: Tower Defense - GameBoard - GameEntities

6.2 GameEntities and their Properties

The first step is to create the GameAgents, GameObjects, GameRegions, GamePaths and their properties.

GameAgents Bug, Archer, Soldier

GameObjects Tower

GameRegions Wall

GamePaths BugPath1, BugPath2, SoldierPath

6.2.1 Bug Properties

Table 25: GameAgent Bug Properties

Property	**Code**
Win Situation	BugWin
Die Situation	BugDead

6.2.2 Bug Beliefs

Table 26: GameAgent Bug Beliefs

Belief	**Type**	**Starting Value**	**Update Type**	**Update Code**
BugHealth	Number	100	Static	(empty)
TargetTower	Turtle	one-of Towers	Static	(empty)
RandomSelectedPath	Number	random 10	Static	(empty)
BugDamage	Number	(empty)	Every time	random 3 + 1
WaitTime	Number	random 20 + 10	Static	(empty)

- **BugHealth** starts at 100 and is reduced every time Archers or Soldiers attack the Bug.
- **TargetTower** is one of the Towers (the only Tower because there will be only one in the GameBoard).
- **RandomSelectedPath** is a random number from 0 to 9. This Belief is Static meaning it is initialed once on set-up function and stays the same. If this value is less than or equal to 5, the Bug will follow BugPath1, else the Bug will follow BugPath2.
- **BugDamage** is the damage the Bug applies to the Tower. It's value is random and varies from 1 to 3.

- **WaitTime** is the ticks the Bug waits before following a GamePath. It's value is random and varies from 10 to 29.

6.2.3 Bug Interactions

Table 27: GameAgent Bug Interactions

Interaction	**Target Belief**	**Code**	**R. Contact**
FollowSelectedPath	nobody	ifelse Get.RandomSelectedPath < 5 [FollowPath BugPath1] [FollowPath BugPath2]	false
AttackTower	TargetTower	ask Get.TargetTower [Set.TowerHealth Get.TowerHealth - Get.BugDamage]	true
WaitForRandomTicks	nobody	MoveRandomlyAvoidingRegion Wall	false

- **FollowSelectedPath** The Bug follows the selected GamePath. FollowPath is a custom function which tells the Bug to approach the nearest checkpoint and follow the GamePath until the end of it.
- **AttackTower** Applies damage to the TargetTower every time called. This interaction requires contact with the TargetTower
- **WaitForRandomTicks** The Bug waits until count of ticks passes the value of WaitTime. MoveRandomlyAvoidingRegion is a custom function that tells the Bug to move randomly but not in GameRegion Wall.

6.2.4 Bug Situations

Table 28: GameAgent Bug Situations

Situation	**Code**
BugWin	count Towers = 0
BugDead	Get.BugHealth <= 0
WaitTimePassed	ticks >= Get.WaitTime

- **BugWin** Returns true if count of Towers is zero (if the TargetTower is destroyed).
- **BugDead** Returns true when BugHealth reaches 0 or below.
- **WaitTimePassed** Returns true when count of ticks passes the value of WaitTime.

6.2.5 Bug Goals

Table 29: GameAgent Bug Goals & Plans

Goal	**Plan**	**Action**	**Type**
DestroyTower	NoCondition!	WaitForRandomTicks...WaitTimePassed FollowSelectedPath...Once! AttackTower...Forever!	Top Level Goal

- **DestroyTower** The Top Level Goal of Bug is to destroy the TargetTower. So at first the Bug waits, then follows the selected path and finally attacks the TargetTower until someone wins the game.

6.2.6 Archer Properties

Table 30: GameAgent Archer Properties

Property	**Code**
Win Situation	ArcherWin
Die Situation	(empty)

6.2.7 Archer Beliefs

Table 31: GameAgent Archer Beliefs

Belief	**Type**	**Starting Value**	**Update Type**	**Update Code**
ArcherViewRange	Number	8	Static	(empty)
TargetBug	Turtle	(empty)	Every time	NearestOf Bugs
ArcherDamage	Number	(empty)	Every time	random 3

- **ArcherViewRange** is the range in which Bugs are visible from the Archer.
- **TargetBug** is the Bug closest to the Archer. NearestOf is a custom function that reports the nearest of the AgentSet Bugs.
- **ArcherDamage** is a random value indicating the damage applied to the TargetBug when the last is shot.

6.2.8 Archer Interactions

Table 32: GameAgent Archer Interactions

Interaction	Target Belief	Code	R. Contact
ShootTargetBug	TargetBug	ask Get.TargetBug [Set.BugHealth Get.BugHealth - Get.ArcherDamage]	false
CallSoldiers	nobody	broadcast-to Soldiers add-content who create-message ["Inform" "Help"]	false

- **ShootTargetBug** When called, ArcherDamage is applied as damage on TargetBug.
- **CallSoldiers** When called, the Archer sends an Inform message to all Soldiers letting them know that Bugs have passed over his watch.

6.2.9 Archer Situations

Table 33: GameAgent Archer Situations

Situation	Code
ArcherWin	count Bugs = 0
TargetBugVisible	DistanceFrom Get.TargetBug <= Get.ArcherViewRange

- **ArcherWin** Returns true when there are not any Bugs left on the GameBoard.
- **TargetBugVisible** Returns true if the TargetBug is in the range ArcherViewRange. DistanceFrom is a custom function that reports the distance from the given GameAgent.

6.2.10 Archer Goals

Table 34: GameAgent Archer Goals & Plans

Goal	Plan	Action	Type
AttackBugs	TargetBugVisible	CallSoldiers...Once! ShootTargetBug...Once!	Top Level

- **AttackBugs** The Top Level Goal for Archers is to shoot the Bugs on sight. So, if the TargetBug is visible, Archers inform Soldiers and then shoot the Bugs.

6.2.11 Soldier Properties

Table 35: GameAgent Soldier Properties

Property	Code
Win Situation	SoldierWin
Die Situation	(empty)

6.2.12 Soldier Beliefs

Table 36: GameAgent Soldier Beliefs

Belief	Type	Starting Value	Update Type	Update Code
SoldierDamage	Number	(empty)	Every time	random 3 + 1
CallingArcher	Turtle	nobody	On message	Help
NearestBug	Turtle	(empty)	Every time	NearestOf Bugs
SoldierViewRange	Number	7	Static	(empty)

- **SoldierDamage** is a random number indicating the damage applied to Bugs when attacked.
- **NearestBug** is the Bug which is nearest to the Soldier.
- **SoldierViewRange** is a number indicating the view range of the Soldier.
- **CallingArcher** is a Turtle Belief which is updated when an Inform message with Label "Help" is recieved. Used to approach the CallingArcher to help him kill the Bugs.

6.2.13 Soldier Interactions

Table 37: GameAgent Soldier Interactions

Interaction	**Target Belief**	**Code**	**R. Contact**
Patrol	nobody	CyclePath SoldierPath	false
ApproachCallingArcher	CallingArcher	ApproachAvoidingRegion Get.CallingArcher Wall	false
AttackNearestBug	NearestBug	ask Get.NearestBug [Set.BugHealth Get.BugHealth - Get.SoldierDamage]	false

- **Patrol** When called, the Soldier cycles through the SoldierPath.
- **ApproachCallingArcher** When called the Soldier approaches the CallingArcher avoiding GameRegion Wall.
- **AttackNearestBug** When called, Soldier applies damage to the NearestBug.

6.2.14 Soldier Situations

Table 38: GameAgent Soldier Situations

Situation	**Code**
AnyArcherCalling	Get.CallingArcher != nobody
SoldierWin	count Bugs = 0
NearestBugVisible	DistanceFrom Get.NearestBug <= Get.SoldierViewRange

- **AnyArcherCalling** Returns true if Belief CallingArcher is not equal to nobody This means that an Archer has called for help.
- **SoldierWin** Returns true if there are not any Bugs left on the GameBoard.
- **NearestBugVisible** Returns true if the NearestBug is in the range SoldierViewRange.

6.2.15 Soldier Goals

Table 39: GameAgent Soldier Goals & Plans

Goal	**Plan**	**Action**	**Type**
PatrolArea	NoCondition!	Patrol...Forever!	Top Level
WatchForBugs	NearestBugVisible	AttackNearestBug...Default!	Maintenance
HelpArchers	AnyArcherCalling	ApproachCallingArcher...Default!	Maintenance

- **PatrolArea** The Top Level Goal for Soldiers is to follow SoldierPath.
- **WatchForBugs** While Soldiers follow SodierPath they might see any Bug. At that time Maintenance Goal WatchForBugs is executed to damage the Bugs.
- **HelpArchers** While Soldiers follow SodierPath they might be called by any Archer for help. So they leave SoldierPath to assist the CallingArcher.

6.2.16 Tower Properties

Table 40: GameObject Tower Properties

Property	**Code**
Die Situation	Get.TowerHealth <= 0

6.2.17 Tower ObjectProperties

Table 41: GameObject Tower ObjectProperties

ObjectProperty	**Type**	**Starting Value**	**Update Type**	**Update Code**
TowerHealth	Number	100	Static	empty

- **TowerHealth** starts at 100 and is reduced every time a Bug applies damage to the Tower.

6.3 The GameBoard

Following step is to drag and drop the previously created GameEntities on the GameBoard and create a scene similar to the following.

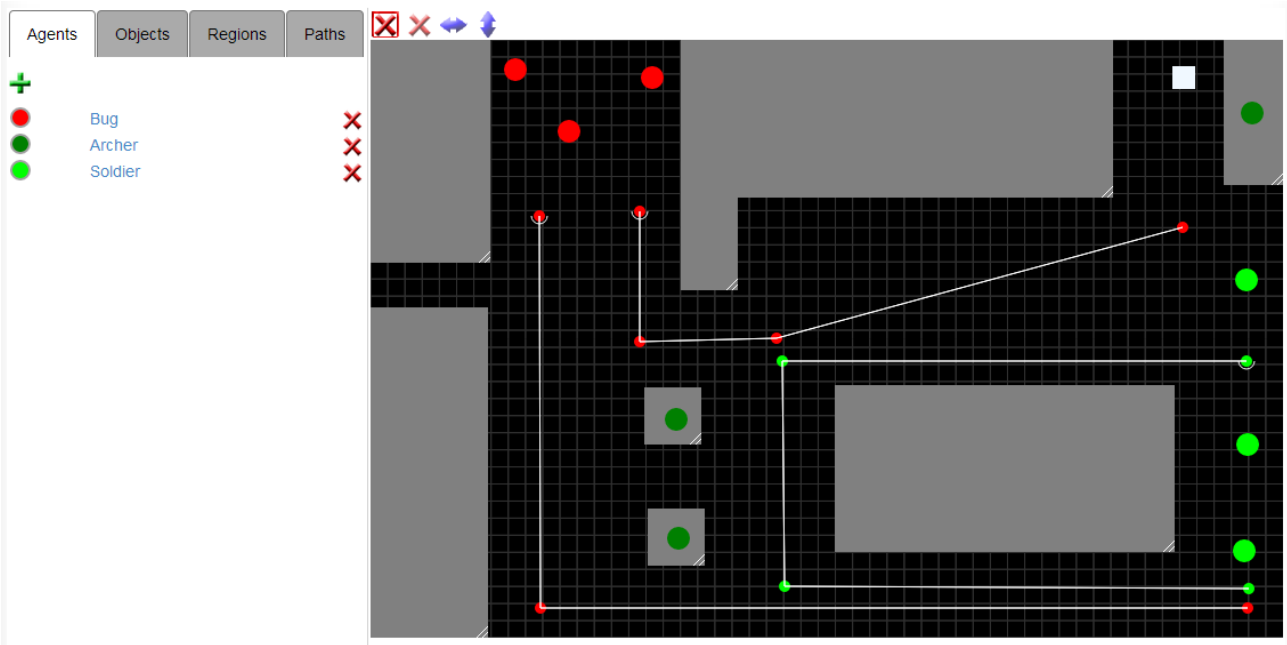

Figure 54: Tower Defense - GameBoard

6.4 Export as NetLogo Project

The last step is to export the project as NetLogo project as shown in Figure 55.

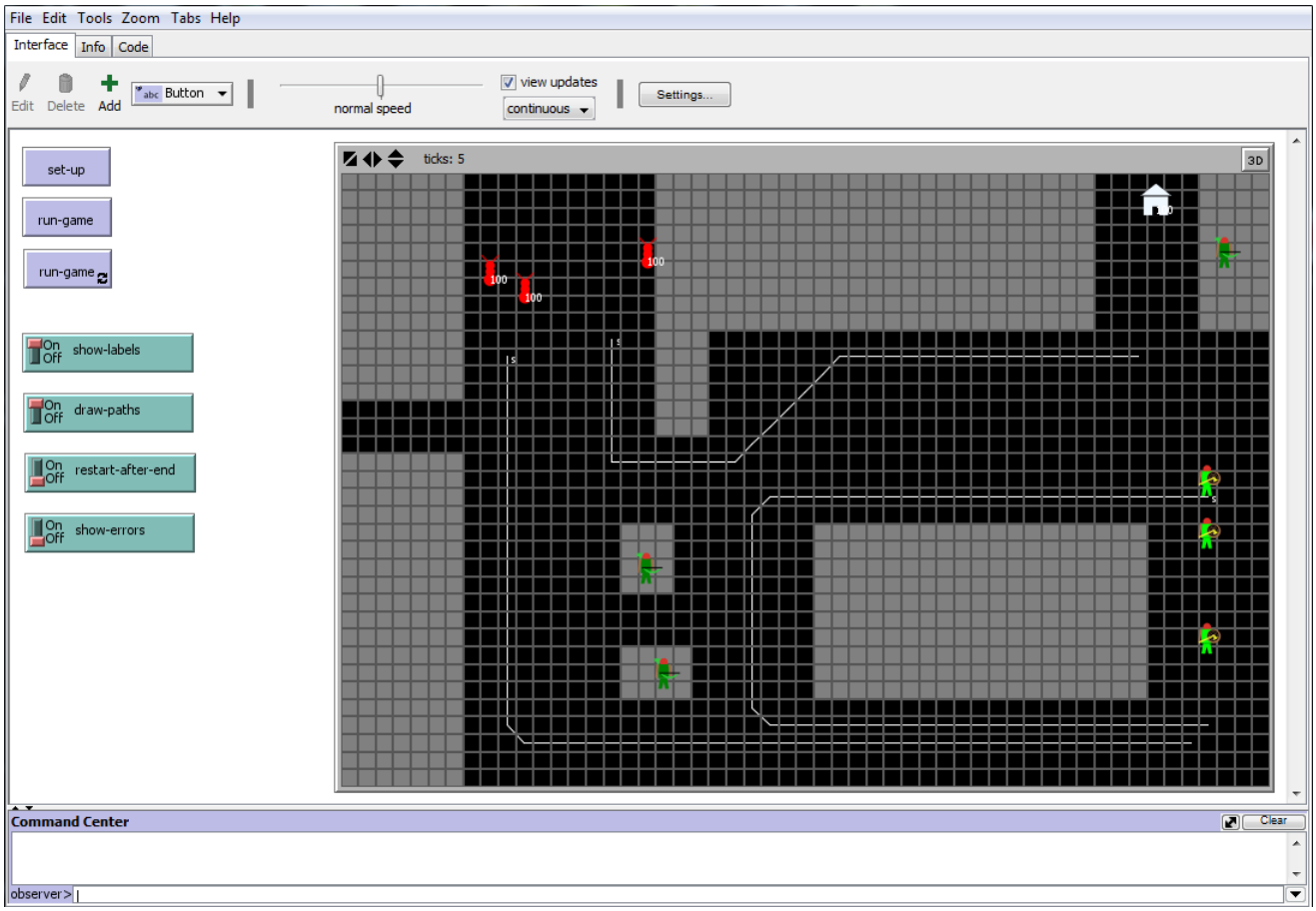

Figure 55: Tower Defense - NetLogo Project

7 Conclusions

In this dissertation we have talked about intelligent agents, we presented various architectures and we referred on related work on the field. Mainly, we have focused on the design and implementation of a web application (N.G.B.) that helps users create and test agents and their behavior, using BDI techniques.

Given the fast growing complexity of modern computer games, approaches used for traditional games can no longer be directly applied to the modern games. Techniques from fields like autonomous agents and planning would appear to be more appropriate. However, even these fields have to be extended by mechanisms designed to satisfy the sophisticated requirements of modern computer games such as real time, dynamics, resources and incomplete knowledge (Nareyek [2002]).

An attempt to standardize a development process is never easy, especially when it comes to a very complex subject such as intelligent agents. Yet, BDI architecture proves itself well structured and it's offered to overtake some of the obstacles.

7.1 Limitations of Research

The main limitations of this research were:

Validation/Verification tools By the word tools we mean utilities to observe and report if the agents behave as they are programmed to. Such tools are by definition almost impossible to report as expected, but an earnest effort would yield.

7.2 Future Extensions

The results of this dissertation point to several interesting directions for future work:

Player Character The current implementation does not support an active player in the game. In a future release the developer should be able to include the player character in the game.

Break up project in scenes A famous concept of game engines is the concept of a scene. A scene is like an isolated environment in the game. For example, in a game scenario that takes place in a building, the first floor could be a scene, while the second floor could be another scene. Future plans include breaking up the project into scenes. A checkpoint in one scene will transfer the characters to another scene and so on, until the final scene.

Alternative export for game engines Our results indicated that it will be worthwhile to implement exporting projects, ready for import in a game engine such as Unity3D [12]. To be more specific, the user will be able to create the whole game in the web application and export the project in some specific format. This format will be imported from the game engine, which will generate the game. The user then will be able to work on the rest (models, animations, graphics, menus etc.).

WebGl instead of NetLogo WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 3D graphics and 2D graphics within any compatible web browser without the use of plug-ins. In order to avoid downloading NetLogo projects, this relatively new technology, will allow the user to run the game in the browser, while working on it.

Game statistics Future plans include the implementation of tools aimed to provide statistics for the games created. For example, if we run the game 1000 times, which agent wins most of the times, or, how much damage should an agent cause so that all agents have the same chances to win.

[12] Unity3D is a cross-platform game creation system developed by Unity Technologies, including a game engine and integrated development environment (IDE). It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices.

Bibliography

B.W. Boehm. *A Spiral Model of Software Development and Enhancement*. ACM SIGSOFT Software Engineering Notes, 1986.

ChessProgramming. Arthur samuel. https://chessprogramming.wikispaces.com/Arthur+Samuel, 2014. Accessed: 2014-09-15.

ChessProgramming. Christopher strachey. https://chessprogramming.wikispaces.com/Christopher+Strachey, 2014. Accessed: 2014-09-15.

ChessProgramming. Dietrich prinz. https://chessprogramming.wikispaces.com/Dietrich+Prinz, 2014. Accessed: 2014-09-15.

ChessProgramming. Kasparov vs deep blue. https://chessprogramming.wikispaces.com/Kasparov+versus+Deep+Blue+1997, 2014. Accessed: 2014-09-15.

P.R. Cohen and H.J. Levesque. Intention is choice with commitment. 1990.

GameBots. Gamebots. http://gamebots.sourceforge.net/, 2014. Accessed: 2014-10-05.

M.R. Genesereth and S.P. Ketchpel. *Software Agents*. Communications of the ACM, 1994.

Haugeland J. *Artificial Intelligence: The Very Idea*. MIT press, 1985.

C. Janssen. Non-player character (npc). http://www.techopedia.com/definition/1920/non-player-character-np, 2014. Accessed: 2010-09-30.

Jason. A java-based interpreter, for an extended version of agentspeak. http://jason.sourceforge.net/wp/description/, 2014. Accessed: 2014-10-11.

D. Martin and P. Boyle. Intelligent agent parallel search and comparison engine, February 19 2004. WO Patent App. PCT/US2000/014,769.

A. Nareyek. Intelligent agents for computer games. http://www.ai-center.com/references/nareyek-02-gameagents.html, 2002.

Radgametools. Oodle network and data compression. http://www.radgametools.com/oodle.htm, 2014. Accessed: 2014-09-15.

Radgametools. Telemetry 2 performance visualization system. http://www.radgametools.com/telemetry.htm, 2014. Accessed: 2014-09-15.

Robocode. Robocode. http://robocode.sourceforge.net/, 2014. Accessed: 2014-10-05.

S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2 edition, 2003.

Wikipedia. Game development. http://en.wikipedia.org/wiki/Video_game_development, 2014. Accessed: 2014-09-15.

Wikipedia. Video game genres. http://en.wikipedia.org/wiki/Video_game_genres, 2014. Accessed: 2014-09-15.

M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. 1995.

A Appendix - Custom Procedures

- **Approach**

  Syntax: Approach [1]{GameAgent/GameObject Name}

  Moves straight for the given GameAgent or GameObject. For GameRegions use ApproachRegion.

- **ApproachAvoidingRegion**

  Syntax: ApproachAvoidingRegion [1]{GameAgent/GamerObject Name} [2]{GameRegion Name}

  Approaches the given GameAgent or GameObject avoiding the given GameRegion.

- **ApproachAvoidingRegions**

  Syntax: ApproachAvoidingRegions [1]{GameAgent/GamerObject Name} [2](list "{GameRegion Name}" "{GameRegion Name}")

  Approaches the given GameAgent or GameObject avoiding the given GameRegions.

- **ApproachFollowingPath**

  Syntax: ApproachFollowingPath [1]{GameAgent/GamerObject Name} [2]{GamePath Name}

  Moves to the nearest checkpoint of the given GamePath and follows the GamePath. At the end of the GamePath, the Agent approaches the given GameAgent or GameObject.

- **ApproachFurthestOf**

  Syntax: ApproachFurthestOf [1]{AgentSet/ObjectSet Name}

  Moves straight for the furthest of the given AgentSet or ObjectSet.

- **ApproachNearestOf**

  Syntax: ApproachNearestOf [1]{AgentSet/ObjectSet Name}

  Moves straight for the nearest of the given AgentSet or ObjectSet.

- **ApproachRegion**

  Syntax: ApproachRegion [1]{GameRegion Name}

Moves straight for the given GameRegion. Stops when GameRegion reached.

- **Avoid**

  Syntax: Avoid [1]{GameAgent/GameObject Name}

  Moves to the neighbor patch with max distance from the given GameAgent or GameObject.

- **CyclePath**

  Syntax: CyclePath [1]{GamePath Name}

  Moves to the GamePath first checkpoint and follows the GamePath. At the end, the GameAgent restarts the GamePath.

- **FollowPath**

  Syntax: FollowPath [1]{GamePath Name}

  Moves to the nearest checkpoint of the given GamePath and follows the GamePath. At the end of the GamePath, the GameAgent stops.

- **MoveRandomly**

  Syntax: MoveRandomly

  Moves randomly anywhere in the game board.

- **MoveRandomlyAvoidingRegion**

  Syntax: MoveRandomlyAvoidingRegion [1]{GameRegion Name}

  Moves randomly anywhere in the GameBoard but not in the given GameRegion.

- **MoveRandomlyAvoidingRegions**

  Syntax: MoveRandomlyAvoidingRegions (list "{GameRegion Name}" "{GameRegion Name}")

  Moves randomly anywhere in the GameBoard but not in the given GameRegions

- **MoveRandomlyOnRegion**

  Syntax: MoveRandomlyOnRegion [1]{GameRegion Name}

  Approaches and moves randomly anywhere in the given GameRegion.

B Appendix - Custom Reporters

- **DistanceFrom**

  Syntax: DistanceFrom [1]{GameAgent/GameObject Name}

  Reports the distance from the given GameAgent or GameObject.

- **FreeNeighbors**

  Syntax: FreeNeighbors

  Reports the neighbor patches with not any turtle on it.

- **FurthestOf**

  Syntax: FurthestOf [1]{AgentSet/ObjectSet Name}

  Reports the furthest of the given AgentSet or ObjectSet.

- **NearestOf**

  Syntax: NearestOf [1]{AgentSet/ObjectSet Name}

  Reports the nearest of the given AgentSet or ObjectSet.