VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

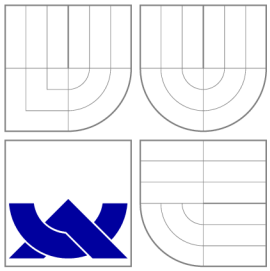
STRATEGIC GAME IN MULTI-AGENT SYSTEM JASON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

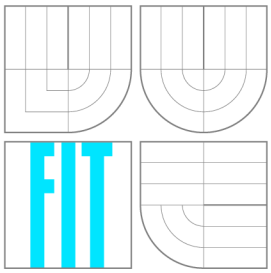
AUTOR PRÁCE
AUTHOR

MATEJ LEŠKO

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STRATEGICKÁ HRA V MULTI-AGENTNÍM PROSTŘEDÍ JASON

STRATEGIC GAME IN MULTI-AGENT SYSTEM JASON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MATEJ LEŠKO
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ KRÁL
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zaobírá problematikou koncepce a vývoje multi-agentní tahové strategické hry. Práce analyzuje teorii těchto her a agentních systémů a výsledky této analýzy následně zohledňuje v návrhu samotné hry. Ta implementuje kromě herních konceptů a ovládání, obecně užívaných, dvě úrovně kooperace umělé inteligence, mírnou a komplexní spolupráci. Agent je v této hře hráčem, ovládajícím různé jednotky. Hra je koncipovaná tak, že je jí možné rozšířit o nové druhy inteligence, případně o nové herní jednotky. Závěrečná část práce se soustředí na srovnání jednotlivých úrovní kooperace, na efektivitu jednotlivých umělých inteligencí a také na zhodnocení efektivity implementace hry. S tímto účelem byla vykonána série automatizovaných testů.

Abstract

This thesis describes challenges in design and development of a multi-agent turn-based strategy game. It discusses the necessary theoretical background of turn-based strategy games and agent based systems. These results were considerated into game concept. The resulting game implements, apart from concepts and game control which are in common use in nowadays turn-based strategy games, two different levels of cooperation of artificial intelligence, as moderate as complex too. In this application an agent commands each of its unit as a player. In addition, the game is designed in such way that it can be easily extended with new artificial intelligences or game units. Final part of this thesis compares these levels of cooperation and how effective artificial intelligence and application is. A number of automated tests were performed with this purpose.

Klíčová slova

Jason, AI, Multi-agentní systémy, Java, agent, strategická hra, BDI, A*, umělá inteligence

Keywords

Jason, AI, Multi-agent systems, Java, agent, strategic game, BDI, A*, artificial intelligence

Citace

Matej Leško: Strategic Game in Multi-Agent System Jason, bakalářská práce, Brno, FIT VUT v Brně, 2014

Strategic Game in Multi-Agent System Jason

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Krále

.....
Matej Leško
May 21, 2014

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce, panu Ing. Jiřímu Králi za odbornou pomoc a čas, který mně věnoval při tvorbě práce. Také bych rád poděkoval mé rodině za jejich trpělivost a podporu.

Contents

1 Introduction **3**
- 1.1 Goals . . . 4
- 1.2 Overall Structure . . . 4

2 Theoretical Background **5**
- 2.1 Reactive Systems . . . 5
- 2.2 Agent . . . 6
- 2.3 Multi-agent systems . . . 7
- 2.4 The BDI Agent Model . . . 8
- 2.5 Practical Reasoning . . . 9

3 Turn-Based Strategies **11**
- 3.1 What is TBS . . . 11
- 3.2 Examples of TBS . . . 11

4 Game Design **15**
- 4.1 Game Principles and Mechanics . . . 15
- 4.2 Game Objects . . . 16
- 4.3 Game Modes . . . 18
- 4.4 Game Interface . . . 19
- 4.5 Design of Artificial Intelligence . . . 21

5 Implementation **25**
- 5.1 Implementation in General . . . 25
- 5.2 Implementation of AI . . . 29
- 5.3 Implementation of Game Objects . . . 30
- 5.4 Implementation of Interface . . . 31
- 5.5 Known Bugs . . . 33

6 Experiments **34**
- 6.1 Experiments Conditions . . . 34
- 6.2 One vs One . . . 35
- 6.3 Team Play . . . 36
- 6.4 All vs All . . . 37

7 Conclusion **38**

A Java – Jason Interface **41**
A.1 Percepts 41
A.2 External Actions 42
A.3 Internal Actions 43

B CD Content **44**

Chapter 1

Introduction

Playing games is an important part of personality development, mainly in early age[1]. The purpose of the most games is to teach us something and train our perception or our communication skills. In fact some video games can be very helpful to people with some disease like cancer, diabetes or asthma[16]. By living in a technologically advanced environment these concepts, used in games, were integrated into software applications, or so called video games[13]. There are many types of video games in general and they are divided into groups by many factors, like if they are two dimensional or three dimensional, multiplayer or single-player, real time or turn based and alike. So there is a variety of games, from first person action shooters to logical puzzles. This thesis will be particularly interested in in so called turn based strategies.

A turn based strategy is a computer game, where the game environment is mostly done as a grid, or something that looks alike, and every player, living or non-living, does his actions only when he is on his turn. These actions can consist from everything imaginable, but generally, it is something like controlling armies, building defence structures or using diplomacy to achieve more peaceful solutions. However this thesis aims more for artificial intelligence (AI) of such games than in game-play itself. From the genre can be already determined some requirements for properties of the artificial intelligence. For example the AI does not have to consider so much perceptions from the environment, as it is in real time strategies on the one side, but on the other, its evaluation of actual situation is usually more sophisticated. Like trying to evaluate future action steps of enemy and decide its action policy according to the evaluated state. Next, units in these games are usually only game objects controlled by a player and do not posses AI at all.

This thesis describes very similar concept chosen in developed application. Every non-living player acts as an agent. The agent model is an event-driven execution model providing proactive and reactive behaviour[14]. Each agent receives percepts from some environment. In this case, the game environment is based on a grid. Cells in grid, or so called nodes, can posses game objects like units, resources or agents bases. These percepts are perceivable by all agents. So, agent can see all objects, nothing is hidden from him. In the implemented game, agents share a game environment with each other and affects in same way by every taken action. Agents are able to *co-operate* and act upon information from this environment. It is possible to deduce now, that game itself is actually a multi-agent system. The code of agents is written in and interpreted by Jason[7, 17], interpreter and programming language.

1.1 Goals

The purpose of this thesis is to introduce a development of turn based strategy with a focus on multi-agent artificial intelligence. The game is entitled Bachelor Wars and is released and available as open-source program, therefore anyone can modify it. There are implemented three levels of artificial intelligence in actual state. The purpose is to watch how agents with these different levels of artificial intelligence are able survive, fight and co-operate in the game environment. Below is list of goals of this thesis:

- Design a strategic game, based on the degree of cooperation of agents. The game will include 2 modes of cooperation: low and high cooperation. Moreover, the game will run in 2 modes of game-play: human vs. AI and AI vs. AI.
- Experiment with the game's modes and compare the modes according to their success rate and time complexity.
- Assess the results and discuss possible future development.

1.2 Overall Structure

The thesis is structured as follows: The first two chapters of this thesis provide essential theory background for understanding the game implementation. As the main focus of the game is the artificial intelligence, especially multi-agent systems, a brief theory can be found in the chapter 2. Next, more specific information about turn based strategies, and its game principles, with some brief history and examples of today games are given in the chapter 3.

Chapter 4 introduces a game design in general, used game mechanism and modes. Describes a game interface and a design of artificial intelligence. Focus on implementation details and a possible modification of game to fulfil player needs together with implementation of the game and differences from game design is discussed in chapter 5. Experiments, used testing environment and conditions are described in 6.

In final chapter 7 is a summarization of the development, achieved results and implemented game features. There is also discussion about possible optimizations and improvements for this project.

Chapter 2

Theoretical Background

In this chapter, in sections 2.1 and 2.2 are briefly introduced agents and reactive systems together with their capabilities and properties. There is a closer look at an agent oriented development, how it differs from widely used functional approach and what are main aspects and benefits of using agent oriented systems. In section 2.3 are discussed multi-agent systems, what are their capabilities and how do they work. The BDI agent model is desribed in section 2.4, together with *beliefs, desires* and *intentions*. The main focus of this chapter is on decision making process of BDI agents in section 2.5 and BDI architecture at general.

2.1 Reactive Systems

According to specification[4]:

> Reactive systems are systems that cannot adequately be described by the *relational* or *functional* view. The relational view regards programs as functions... from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behaviour... Every concurrent system... must be studied by behavioural means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules.

From this point reader is able to distinguish, that main difference between functional programming and reactive systems is in input-compute-output operational structure. Moreover for better understanding the difference, functional programs can be though off as mathematical functions:

$$f : I \rightarrow O \tag{2.1}$$

Where I is domain of possible of *inputs* and O is a range of possible *outputs*. Difference is that *reactive systems* are able to maintain a long-term, ongoing interaction with their environment. So they do not compute some function of input and terminate as would be done by functional system, but waits for another possible interaction. Examples of such programs include online banking systems, operating systems, webservers, process control systems and the like.

2.2 Agent

By *agents* reader can understand a more complex class of systems which is a subset of reactive systems. So agent is an reactive system that a programmer can delegate some task to it and system itself is able to determine, what is the best approach to achieve this task. The name 'Agent' was chosen, because in general, users of such systems can think of it as active entity, or purposeful producer of actions. They are sent into environment to achieve goals given by programmers. They want from these agents to actively pursue these goals delegated on them by figuring out, what is the best way of accomplish these goals. So, programmer itself does not have to tell them how to do these tasks in a low-level detail.

More precisely, *agents* are systems that are situated in some *environment* and are capable of *sensing* this environment via sensors(*e.g.* like camera, detectors...) and have a repertoire of possible *actions* they are able to perform(via *effectors* or *actuators*) with intention to modify their environment. Actual deciding what to do is achieved by manipulating *plans*. To better understand, in figure 2.1 is shown schematic example of the agent.

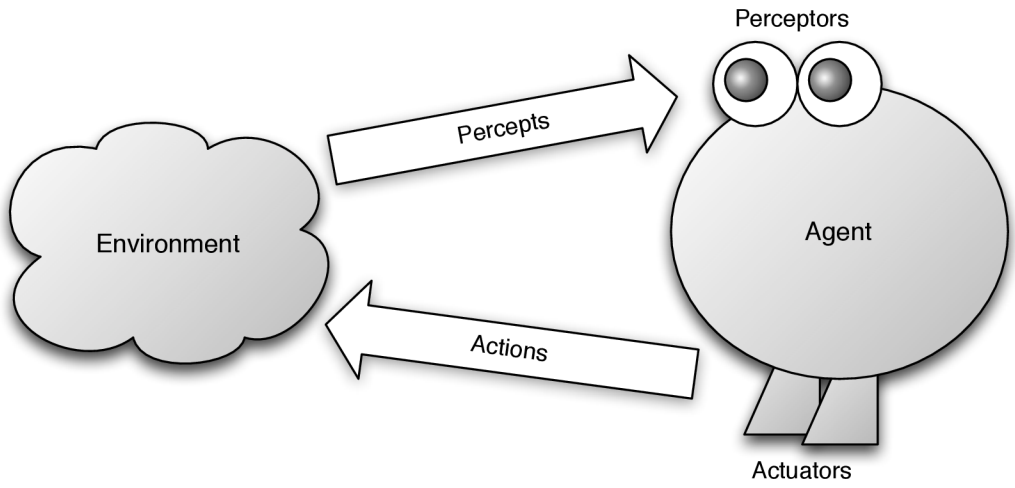


Figure 2.1: Scheme relation between agent and environment[7].

Characteristics of Agent

According to Wooldridge and Jennings[6] agents *should* have the following properties:

- *autonomy;*
- *proactiveness;*
- *reactivity;*
- *social ability.*

Autonomy

By *autonomy* is generally meant an ability of agent to decide how best to act to achieve delegated goals upon this agent. Therefore the ability of agent to construct goals is strongly

bounded by the goals that the programmer delegated. Moreover the way in which agents will act to accomplish their goals is bounded by the *plans* given by the programmer. These plans define the ways in which agent can act to achieve its goals and sub-goals.

This functionality allows agents to put together these plans on the fly, in order to construct more complex overall plans to achieve programmer goals. To simplify, autonomy allows agents to act independently to achieve goals, that were delegated on them. Thus autonomous agents are able to make independent decisions about how to achieve their delegated goals. To simplify even more, agent actions are under its own control and are not driven by other entities.

Proactiveness

Proactiveness means being able to exhibit *goal-directed* behaviour. In other words, agent will try to *achieve* a particular goal delegated upon him. As a contrary could serve a Java's objects. Which could be considered as absolutely passive agents. Such object is essentially passive and act only if some method is invoked on it.

Reactivity

Reactivity represents an agents ability to respond to changes in the environment. Agent should respond to these changes with effective balance between *goal-directed* and *reactive* behaviour. While *goal-directed* behaviour can be explained as execution of plan in order to achieve a goal, *reactive* behaviour is a series of actions to be executed upon environmental changes as a respond on them. For better understanding, moving to the enemy base with intention to seize it, is an example of goal-directed behaviour while responding on support request form ally is an example of reactive behaviour.

Social Ability

Here by *social ability* is meant a *cooperation* and *coordination* of activities with other agents. Agents of Jason[7, 17] are able to communicate not only in terms of exchanging bytes or invoking methods, but they are able to communicate on *knowledge level*. So they are able to communicate their *beliefs*, *goals* and *plans* with each other.

2.3 Multi-agent systems

Single agent systems are rare in practice. Most of the time are used so called *multi-agent systems*. In such systems each agent has its own *sphere of influence*. In other words, it is a part of shared environment, that the agent is able to have influence. These spheres of influence do not have to intersect and then this part is controlled only by one agent. If these spheres overlaps, the environment is *jointly* controlled. In such the case things are more complicated because to achieve desired outcome in the environment, agent have to take into account how other agents in the environment will act. As can reader deduce, this ability to delegate goals, pass percepts and alike to others is essential in multi-agent systems. An example of such system is shown in the figure 2.2.

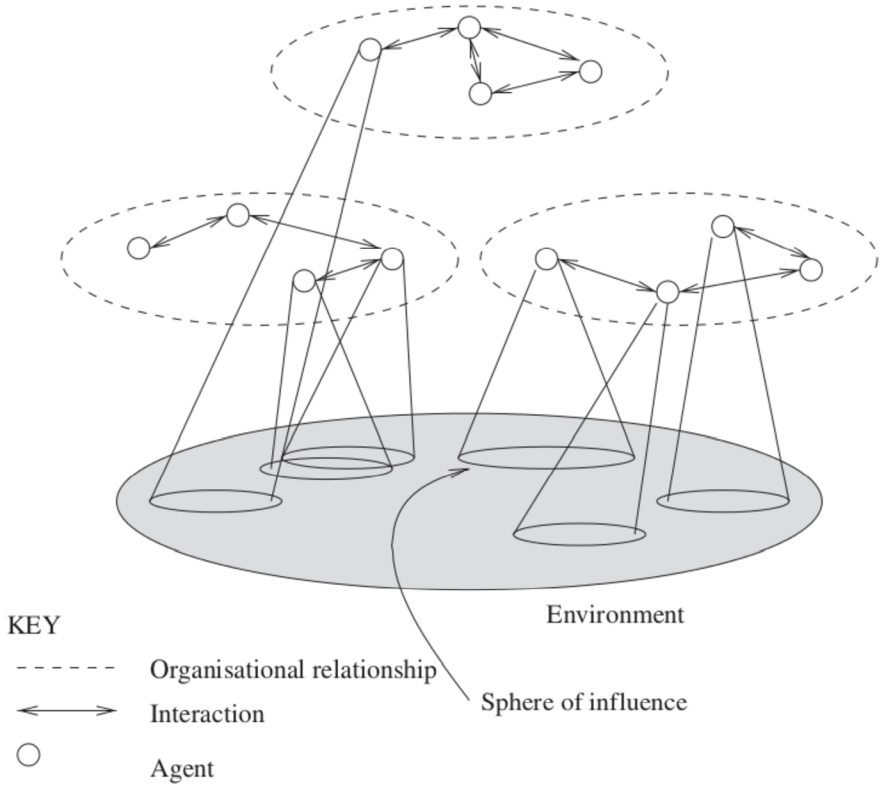


Figure 2.2: Typical structure of multi-agent system[7].

2.4 The BDI Agent Model

BDI or *belief-desire-intention* model was inspired by and based on model of human behaviour that was developed by philosophers. The idea was to model something which humans possess naturally – some mental states. So basically these systems are computer programs with computational analogues of beliefs, desires and intentions. Distinction between beliefs, desires and intentions[7]:

- **Beliefs** can be described as information that agents possess about the environment. Here is shown the similarity with human beliefs. These can be inaccurate, out of date or wrong. As an example we could be used `cpuLoad(32)` to represent agent belief about cpu load in the given environment.

- **Desires** represents all possible states of affairs that agent *might* like to accomplish. In other words they are *options* for an agent. They can be considered as *potential influencers* that agent may act upon. From this, it is possible to tell, that not all agent desires can be compatible to each other. It is perfectly reasonable for rational agent.

- **Intentions** can be understood as state of affairs that the agent has decided to work towards. They can be goals delegated to the agent or may result from considering available options for agent. Intentions itself are therefore the chosen options. After

selecting the intention, agent is *committed to* it. So intention can be dropped only if the goal is no longer achievable or the intention was fulfilled. From this can be deduced, that intentions have three main properties. They are *persistent*, constraining the future and *pro-attitudes* – they tend to lead to action.

2.5 Practical Reasoning

Practical reasoning is decision-making model, which underlays the BDI model. So practical reasoning models the process of figuring out what to do. Therefore it can be considered as *reasoning directed towards actions*[7]. As said earlier in 2.4, BDI model is based on human behaviour. According to Woolridge, human practical reasoning seems to consist of two distinct activities[14]:

- *deliberation* - thinking on what we want to achieve;
- *means-end reasoning* - how we want to achieve it.

Deliberation

Deliberation process leads agent into adopting *intentions*(2.4). More precisely it is the process of selecting between different possible plans. These so called applicable plans are plans, that have their context satisfied, according to current agent's belief base. To summarize and simplify, *deliberation process* means thinking on what we want to achieve.

Means-Ends Reasoning

Means-ends reasoning is the process where the question: *"how we want to achieve it"* is asked. In other words, it is the process of deciding how to achieve and end, using the available means[7]. This process can be better known as *planning*[2]. As input for this process serve:

- A *goal*, or *intention*: something that the agent wants to achieve.
- The agent's current *beliefs* about the *state of the environment.*
- The *actions* available to the agent.

As an output is generated a course of action, called as *plan.* It is possible to think about it as a "recipe" too. Nowadays is focus on one simple idea that has proven to be quite powerful in practice[7]. This idea consist of that a programmer develops a portion of partial plans for an agent at design time, and task for agent is to assemble these plans at runtime. Apart from original focus in AI to assembly of a complete course of action, in which atomic components are actions available to the agent, is this approach significantly less resource demanding.

The Procedural Reasoning System

The Procedural Reasoning System or *PRS*, see the figure 2.3, is a system where agent is equipped with a library of pre-compiled plans.

These are manually constructed by the agent programmer. Each plan in *PRS* have the following components:

- a *goal* – the post-condition of the plan;
- a *context* – the pre–condition of the plan;
- a *body* – the course of action to carry out.

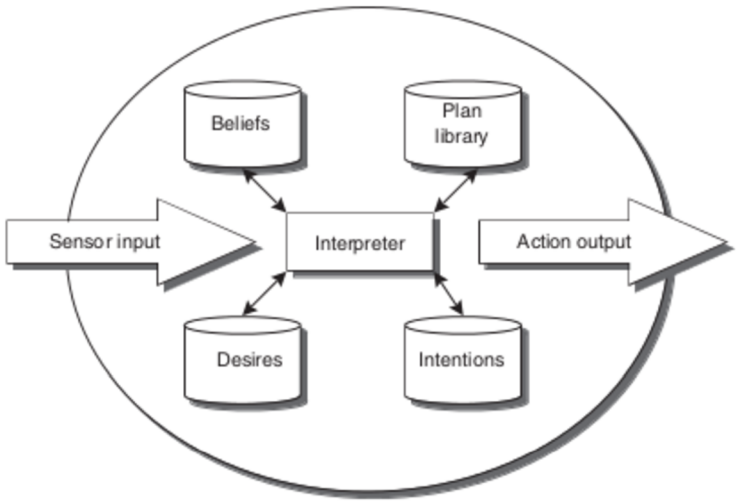


Figure 2.3: The Procedural Reasoning System (PRS)[7].

The Goal

The goal, or *the post-condition of PRS plan*, represents states of affairs, that stands as output of executing this plan. To simplify the goal of PRS plan defines *what the plan is good for*. In Jason, the goal can be represented as *e.g.* `!goto(X,Y)` – which indicates that after executing its plan, agent's position will be unified with X and Y variables.

The Context

The context defines *the pre-condition of PRS plan*. It defines all conditions of the environment that must be true in order for the plan to be successful. In Jason, the Context can be represented *e.g.* `!goto(X,Y) : noEnemyNear & canMove` – this goal and its plan will be executed only if the context is true. that means agent belief base contain beliefs *noEnemy* and *canMove*.

The Body

The body can be described as course of action to execute. It can be a simple list of actions to be executed ort hanks to PRS there is possibility for a lot of richer plans to exist. As an example can serve `!goto(X,Y) : true <- !moveLeftLeg; !moveRightLeg.`. Where achievement goals *moveLeftLeg* and *moveRightLeg* represents parts of the body.

Chapter 3

Turn-Based Strategies

In this chapter the reader will be apprised of turn-based strategies. What they are, what is the difference between turn-based and real-time strategies and why are so popular, is discussed in section 3.1. An examples of the past and actual famous TBS games are shown in section 3.2, together with a brief description of artificial intelligence used in one game from examples.

3.1 What is TBS

Turn-based strategy, or *TBS* is a strategy game, where players take turns when playing. Apart from real-time strategies (RTS), player has much more time to contemplate his strategy before acting. The main difference is in the time flow. While in real-time strategies all players share the same game time and have the same chance to react on game events, in turn-based strategies every player has his own time flow. This game time starts when the player takes his turn and ends with the event, saying that player already finished all his actions.

Another difference between TBS and RTS is in game environment and its possibilities. In TBS, the player is usually in some grid related environment. He, or his units, has exactly defined where and in which direction is possible to go. As an example could be used a picture from a game called *Battle for Wesnoth*, see figure 3.1. Due to these restrictions, players have to think harder to find out weakness in enemy's defence, attack or in strategy in general. As an examples can serve classic games as *chess*[11], which is widely considered for an ultimate turn-based strategy.

It is possible to deduce, that all these characteristic attributes makes TBS very popular. They are ideal for long-term matches, where every step can be the last, and careful strategic actions are prior to hotheaded ones.

3.2 Examples of TBS

After section 3.1, reader should be able to tell main characteristics of TBS and understand difference between RTS and TBS. Therefore here are, for completeness, introduced some examples of turn-based strategies. As representative element was chosen *Sid Meier's Civilization series*. It is the shining example of turn-based strategies. With its long history, an addictive gameplay and a its re-playable ability this series is one of the most successful

Figure 3.1: Screenshot showing a Battle for Wesnoth[1].

games in history. According B. Edwards this game changed the course of computer strategy games forever[5].

Civilization is a turn-based historical strategy game, where a player single-handedly guides the development of a civilization over the course of millennium, from the stone age to the space age. In this game each player represents the leader of a certain nation or ethnic group (civilization). Among the main strengths of *Civilization* is how its designer, Sid Meier, actually represents mysteriously accurate possible future, if the course of history was pushed *just a little bit.*

No wonder that many critics recognize Sid Meier as one of the greatest software designers in history. At that time, functions like random map generation, multiple ways to win, or up to 15 additional computer opponents was something unbelievable. Moreover this all fitted into only three megabytes.

His work is even more remarkable due to a fact that Meier handled most of the programming on *Civilization* himself, even doing all of the early artwork for the game. Interesting part is that the first version did not feature the turn-based gameplay, but a real-time model. But the gameplay was so dull and boring, he and his companion decided to implement it in TBS style. A figure 3.2 shows the first version of *Civilization.*

Nowadays the latest game from this series is *Sid Meier's Civilization V*, developed by *Firaxis Games*[15]. This game posess entirely new game engine, with hexagonal tiles instead of classical square one. Compared to the first version of this series, the difference is nothing

[1]Source: http://en.wikipedia.org/wiki/File:Battle_for_Wesnoth_0.8.5_chaotic_indexed.png

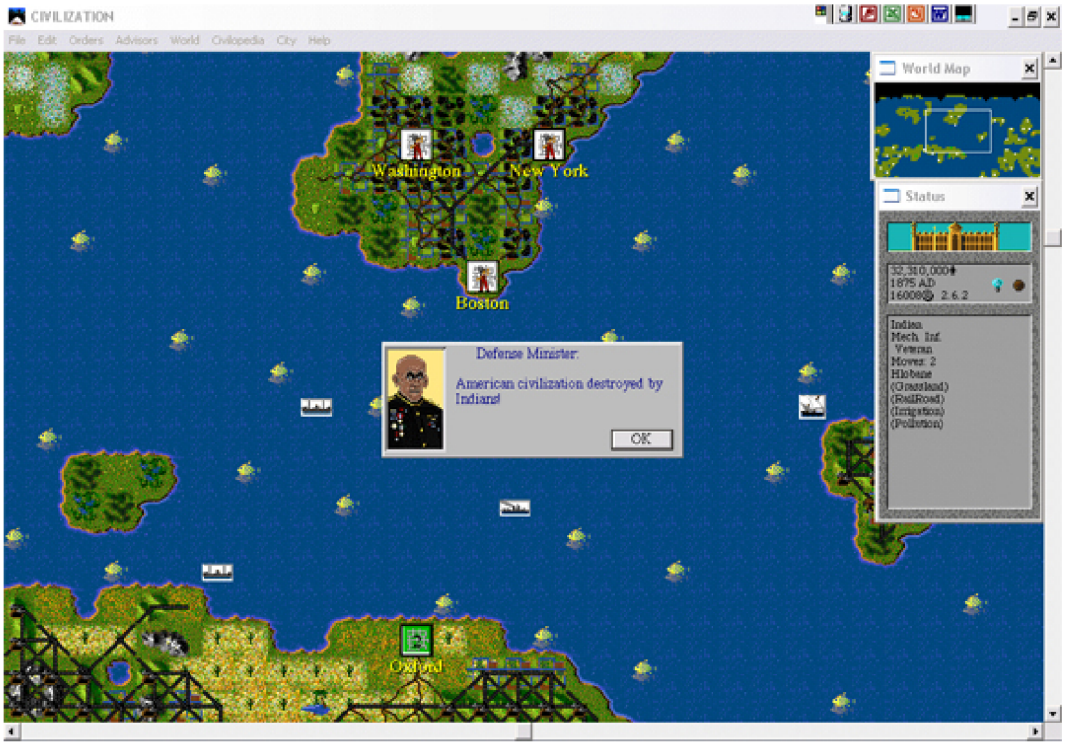


Figure 3.2: A world map screenshot from the Civilization 1[5].

Figure 3.3: A screenshot of the video game Civilization V. Starting location with one city and one warrior unit showing[3].

less than *drastic*. These changes are, besides much better graphics, *e.g.* game features like

[3]Source: http://en.wikipedia.org/wiki/File:Civilization_V_screenshot.png

community, modding and multiplayer. To compare with the first version see figure 3.3.

Civilization V - AI

Civilization V has one very interesting part. The artificial intelligence(AI) is designed to operate on four levels[12]:

- *tactical* – on this level are controlled individual units;
- *operational* – on this level AI oversees whole war front;
- *strategic* – AI manages whole empire;
- *grand strategic* – AI sets long-term goals and determines how to win a game.

Even more, every AI-controlled leader has his unique personality. It is determined by combination of *flavours*. Every flavour have value on a ten-point scale. These flavours are grouped into categories like growth, strategy, military and alike.

Chapter 4

Game Design

The idea at the beginning was that everything should be parametrized and therefore allowed to be modified. Whole concept of the game is based on it. Everything was designed to withstand unpredictable changes in the code, like new objects in the game, or new types of AI. This is the main *strength* of the *design* of the game Bachelor Wars, that it will not break down like a card-castle if the new element appears.

Very important fact is, that this thesis is focused on the artificial intelligence itself, so elements like graphics, interface and other visible components are on the second place. That is why the game look may appears very simple. The main purpose was a design of inner elements. Of course, there was an ambition to use some powerful frameworks, but due to odd behaviour and the high learning curve this idea was not realised.

As it already appears, designing is one of the most important processes during the development of the game. Good game design can solve many problems before the actual implementation of the game concept starts. This process is not only about animating units, colours and shapes. It covers other aspects, like thinking about game mechanics and rules, which are described in a section 4.1, game concepts like *e.g. modes*, which are in the section 4.3. There are other things that normal user does not even think about, like design of the *artificial intelligence*, which is in section 4.5. Of course, design of the game comprehends the look of the game interface too, like in section 4.4.

4.1 Game Principles and Mechanics

The game principles of Bachelor Wars are simple in its bases. The main purpose of the game is to fulfill winning conditions given by chosen *mode*, or seize enemy's base and therefore win the game. The game itself requires at least two active players, but no more than four. This upper limit was set after realizing, that more active agents in the game could do more harm than good to the gameplay, because of unfair positioning and chaotic appearance. Design itself on the other hand *is able* to comprehend more that four players without problems.

This game is *different* from the usual stereotype of this this genre, *e.g.* there are no such things like buildings. You do not have to build barracks to buy a unit. There is no need to build mines, or farms. You do not have to wait for unit's training to use them. Everything is done by this way to make the game more aggressive, faster. The main aim is that a game session lasts no longer than approximately 10 minutes. It could be understood as the chess[11] with more players, a possibility to buy a unit and to gain resources.

When the game starts all players have same starting conditions. It means that amount of knowledge and slots[1] is the same, and every base is in one of the corners. During gameplay each AI player perceives the same game environment input. It means that it can see knowledge resources, its allies and enemies, respectively their units and bases.

Game itself contain a game settings interface that *allows* user to set the mode, active players and their level of artificial intelligence. For more info see 4.4 and 5.4. This interface allows actually to generate a map for a game session. This map consist of an invisible grid environment. It can be described as an invisible layout, which contain game objects. So every object is operating within this grid, not in the classical coordinate system used in Java's Swing[9].

4.2 Game Objects

As already mentioned, the design of the game Bachelor Wars is done in the way, that the implementation is able to withstand new elements without problems such as obstacles (discussed later). This is thanks to an idea from object-oriented approach used in Java. The main idea of the this design is that everything on the game map is a game object. These approach allowed a much needed flexibility in the later implementation phases described in 5.3.

Bases

A base in the game has a special function. Not only it represents an agent on the battlefield, it is a spawning place for agent's units too. The design of the base, as a game object, is simple. It is represented by a coloured curved square. The colour is the colour which is chosen by user, or set by default. On the other hand it is used to represent the agent in the environment, so its inner functionality is quite sophisticated. This functionality is described in 5.3. Part of this functionality is number of variables and properties. But only few of them is visible to the player:

- **Player** – the name of player, who own this base.
- **Units to create** – this property is signalizing, how many units is this base able to own on the map at the same time. This property is parametrized, thus modifiable. In the thesis and in the game also called *slots*.
- **Killed enemies** – specifies how many enemy's units were killed by units of this base.
- **Income per Round** – this property signalizes how much *knowledge* will be added to the actual resources at the beginning of the round. This property is parametrized, thus modifiable.
- **Knowledge to use** – how much *knowledge resources* the player is able to spare for units.

To simplify, the base could be understood as an intelligent box, that contains all information about agent's units, allies, income from knowledge resources, free slots and more. To summarize – it represents a communication bridge between the agent and Jason[7, 17].

[1] By *slot* is meant the capacity of base

Units

To be able to fulfill winning conditions every player has a set of available units. These units represents a tool, which is available to act as agent commands. These units do not posses any level of artificial intelligence.

On the other hand these game objects posses an ability to store an intentions that the agent gave them. As already mentioned, agents have intentions (see 2.4). Units serve as an envelope for agents to store these intentions, *e.g.* seize knowledge, attack on base... Therefore agent, as a general, creates unit and deliver its intentions to this unit. When the agent is able to manipulate with this unit again, the agent simple asks which intentions this unit already posses and tries to identify the best choice in actual round. This is described with other processes closely in 4.5 and 5.2. To be able to define more complex strategies and behaviour, unit types differ between each other in their attributes, which are described in this section.

The design of units is made in relation to the title of the game. Every unit represents a student of the Bachelor study programme according to unit's name. This step was made to entertain the user and the creator of the game. No deeper meaning is hidden in this action. The unit posses animated graphical representation in a similar fashion. The idea was to make a feeling from the game look a little bit better. For an example of one of the units see figure 4.1.

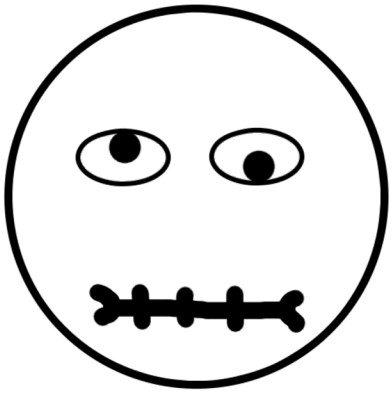


Figure 4.1: A Third Year Student - unit from Bachelor Wars.

A creation of characteristic attributes is part of the design of the unit too. Similarly to a *base*, the unit, as a class, has got huge number variables and properties, but only for few of them is a meaning to show. These attributes are:

- **ID** – or so called *identification number*. This property is defined by a unique value, which allows agents in the game environment determine to whom which unit belongs to. This value is part of the *NAME*.
- **NAME** – the name of the unit, this property is intended for living player only. It consist of type of unit and the *ID*.
- **HP** – or *hit points*. This property represents how much damage the unit is able to withstands. This unit's property is specified by finite value.

- **ATK** – *attack*, specify how much damage the unit can possibly cause.
- **MOV** – *movement*, specify the movement ability of the unit. To simplify this value express how much cells in game grid can be used for movement per round.
- **COST** – this property is expressing the value of the unit. How much *knowledge resources* the player needs to be able to buy this unit.

Unit is designed for two modes. According to goals (see 1.1), there are two main designed modes. For mode *Human* vs. *AI* there is an implemented interface, which is described at 4.4. Mode *AI* vs. *AI* has a similar interface, but without some elements, which are necessary for the other variant. The idea of two interfaces raised after a need to revert unit's movement, and a need to signalize that the intended action is finished. Technical matters are discussed in 5.3.

Knowledge

At the beginning of the development, a question raised about goals of this game. What will be considered as a source of in-game money? As many things were done having regards to the title of the game, naming "in-game money" as *knowledge* is no exception.

In classic games, players usually have to mine gold, or grow food to get resources to spare. Here the player has to seize a knowledge resource called *knowledge* too. So a *knowledge* is a source of regular income. Every knowledge has to be seized to become the resource. Seizing is done by units after an agent evaluates it is the best choice in actual round, according given game conditions and actual situation on the map. Knowledge is seized at the beginning of the following round. These objects are randomly generated on the map.

Design of the game provides interface to change values connected with the *knowledge* resources. These values are parametrized for better gameplay and testing. This interface is described in 4.4.

Obstacles

The primary motive for obstacles, as a game objects, was a better presentation of *pathfinding*. These game objects represent an inaccessible terrain, chunks of barriers that block the path and therefore make the game more *realistic*. As a secondarily motive serves to make game environment more interesting.

4.3 Game Modes

According to assigned specifications for this thesis (see 1.1), there are two main modes in the game. *AI* vs. *AI* and *Human* vs. *AI*.

But besides these, game design contains modes, that are not part of the official thesis goals. These modes were designed in order to demonstrate AI capabilities and its *opportunistic* behaviour. To be able even more distinguish differences between different AI levels, every mode has its own winning conditions. These conditions are along with the main constant wining condition → seize the enemies's bases.

These modes are:

- **Domination** – mode where the player need to dominate on the map. Rule is set to *posses* at least 80% of the knowledge resources for few rounds. This value (number of rounds) is parametrized and modifiable via interface described in 4.4. This mode is good from the strategic point of view. There are structures on the map, that need to be seized, hold and defended. While doing this, AI need to respond on the enemies, their attacks. This is why is this mode as default mode as well.

- **Annihilation** – mode where the player has to destroy all his opponents. This means to get to their base and survive 1 round there. This mode shows that even the weakest player can win, if he chooses his opponents wisely. It can be considered as a mix of two modes: *domination* and *madness* mode.

- **Madness** – mode where the winner is the one with X killed enemies. If the player destroys the enemy's base, every unit of this base is killed and marked as "killed" by the player. The X is modifiable,parametrized value. This mode aims mainly on how an AI or a team can divide their forces and analyse the game environment.

All of the modes above are described from a view of the living player, but their conditions are applied for AI or teams equally. All winning conditions is possible to combine with the *time limit*. For time limit is considered *number of rounds*. The game ends after reaching this limit. So to simplify, if the time limit is reached, the actual winning base (or team) is considered as a winner for actual game session.

4.4 Game Interface

Every game needs to have a communication bridge between its user and the environment. There is plenty approaches how to design an interface. Approach used for this game was very simple. As already mentioned, the graphics was not the main goal of this thesis. Therefore the interface is simple, with just necessary information to provide basic data about game elements and the environment. An overall scheme of game interface is shown in figure 4.2. The scheme consist of:

1. *Game Map* – a generated area, where all fight are in progress. Its creation is modifiable. For implementation details see 5.1

2. *Info Panel* – a panel where information about chosen base/unit is shown. This panel is also used for buying new units and confirming their actions.

3. *Shopping panel* – a panel, where is player able to chose which unit he wish to show in info panel. Used when buying new units.

4. *Statistical Panel* – a panel, where player can see actual winning base according to the mode with statistical properties for chosen mode. If living player is present in the game, there is a *button* to mark actual round as *finished*.

5. *Info area* – an text area, used for information about actual actions on the map.

For implementation details of used interfaces see 5.4.

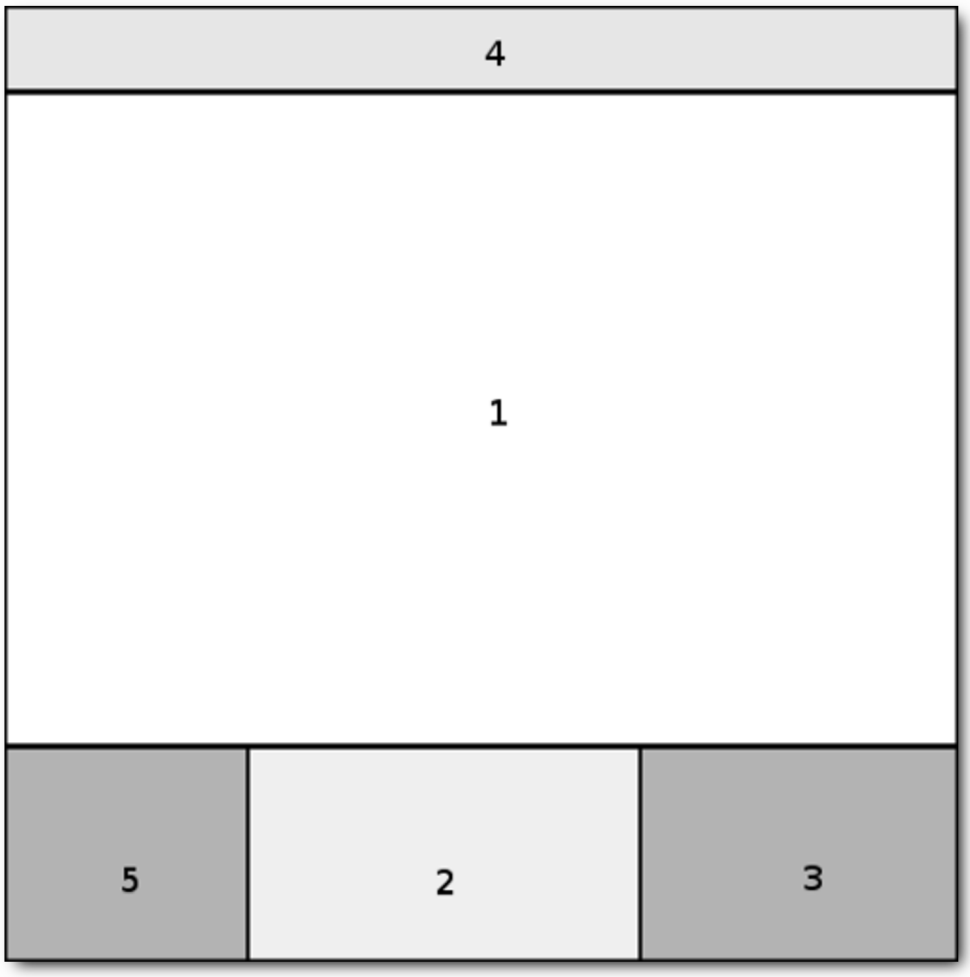


Figure 4.2: The overall scheme of game interface.

Settings Interface

As was mention in section 4.1 the game settings interface allows player set the mode, active players and their level of AI. Apart all of this, there are options to change values determining winning conditions, size of game grid, number of resources, called *"knowledge"* (see 4.2), number of obstacles, number of free slots for each base, income per seized knowledge and base income for every round, player's name, teams, colours and screen resolution.

All limitations were chosen similarly to the limitation of active players, mentioned in 4.1. These limits are high enough to allow interesting combinations of map generation and its size together with properties, connected with bonuses for seizing. For the example see figure 4.3.

As this information suggest, most of the game parameters are parametrized and therefore player is able to modify them. All of this effort was made with intention for easy gameplay modifications and environment testing. For more about experiments and testing see chapter 6.

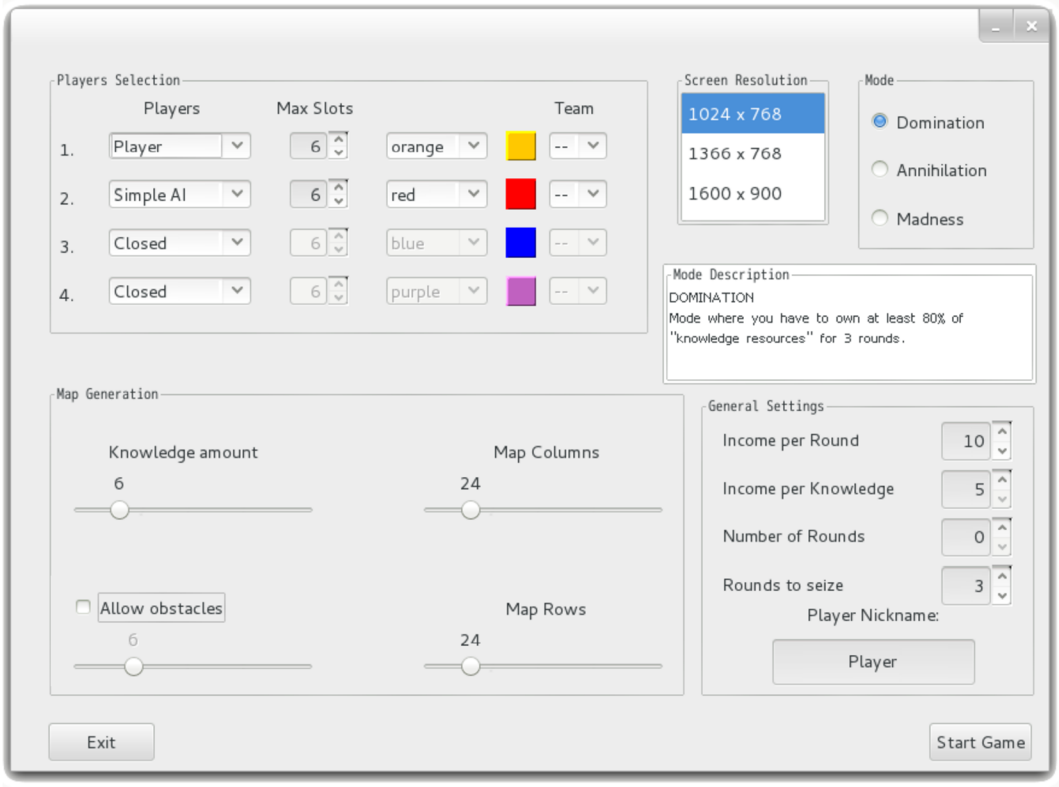


Figure 4.3: The game interface for settings.

Unit Interface

Bases and units share very similar information panel, but for units there is more options to show, following the mode AI vs. AI or Human vs. AI.

- *AI vs. AI* – in this mode, chosen unit shows only properties that are described in 4.2. No other options are available.
- *Human vs. AI* – here are more options to show according to actions (for implementation details see 5.4):
  - **buying** – Visible basic properties and a button *buy*.
  - **selecting unused living player's unit** – Visible basic properties and two buttons.
    1. *cancel button*;
    2. *done button*.
  - **selecting used living player's unit or enemy** – Same as for variant *AI vs. AI*.

4.5 Design of Artificial Intelligence

According to goals (see 1.1), the game should contain 2 levels of AI. *Low* and *high* cooperation level. In the game are integrated 3 levels of artificial intelligence:

- Simple AI;
- Medium AI;
- Advanced AI.

Simple AI can be considered as a root for design of Medium and Advanced AI. The main goal was to re-use as much behaviour as possible. Therefore Medium AI inherits from Simple AI and adds some characteristic behaviour of its own. An analogical technique was used for Advanced AI and Medium AI too.

Every AI posses basic behaviour patterns. These are used at the beginning of the unit creation. During this action agent perceives actual situation on the battlefield. According to this situation and chosen *mode* (see 4.3) stores basic intentions to the unit. These are:

- add *seize enemies's bases* to the intention map of the unit;
- according to the mode one of these intentions are stored into the intention map, in an order, given by specification of the chosen mode:
  - add nearest *free*[2] knowledge;
  - add nearest *free* enemy's unit;
  - add nearest enemy's unit.

These can be called as *persistent* intentions. Because they are stored in a particular unit during whole its existence, till the intention is not reached or the unit is dead. Intentions in general does not have to be accomplished by originally chosen unit. It is because of changing environment. Every level of AI has got a opportunistic behaviour. this means that if there is some game object in the area of influence of some unit, that was not originally in the intention map of this unit, it is added to the intention map as its new intention.

Another interesting part of AI design is that modules representing AI levels can be replaced for their modified versions, or the new one. Game itself does not contain any interface to add new modules representing another AI levels. But the game design and the implementation allows such things with only a few manual steps. At the beginning the idea was that user could be able to add new AI through some interface. Due to certain circumstances this interface was not implemented, but whole design and the implementation was lead to allow such a thing.

All AI levels can operate in two *states*. The first one could be called *standalone* state. In this state AI does not have any allies, thus it does not need to communicate with other agents. On the other hand, there is a state that could be called *co-operation* state. In this state is AI capable of team play, which plays very important part in the game. Except the Simple AI, other AI levels are capable of co-operation. With this purpose a simple communication API was created. Part of this communication is an agreement about a role. Currently there are two implemented team roles for agents:

- **seizer** – tries to seize 80% of nearest knowledge resources on the map and only a few units are used for defence or attacking. Seizer role is a preparation in order to achieve better economical status and therefore to be able buy better units after some time as an attacker.
- **attacker** – tries to attack the enemy's and get attention. Only about 20% of attack power is used to seize knowledge resources.

[2]By *free* is meant a knowledge that was not already assigned to another owned unit.

Mode play

These actions below are performed for units without any intention in their intention map. They are same whether AI is in the team or not. There are subtle nuances for medium and advanced AI level, but the core action is the same. A diversity in actions happens during gameplay thanks to the changing game environment.

- *Domination*
  1. seize nearest *free* knowledge;
  2. attack nearest *free* enemy's unit;
  3. attack nearest enemy's unit;
- *Annihilation*
  1. attack nearest *free* enemy's unit;
  2. seize nearest *free* knowledge;
  3. attack nearest enemy's unit;
- *Madness*
  1. attack nearest *free* enemy's unit;
  2. attack nearest enemy's unit;
  3. seize nearest *free* knowledge.

Simple AI

The simple AI posses basic behaviour for every action in the game. This means *e.g.* if the agent perceives a knowledge, it tries to seize it. It acts similarly in a matter of enemy's unit too. This behaviour could be described as "animal impulses", because this AI level does act as an unpredictable entity. Despite its randomized actions it is able to behave opportunistically. It means *e.g* that in actual round, thanks to changes in the environment, is able to seize some knowledge, with an unit, that was originally assigned somewhere else.

Unit creation

Whether in team or not, simple AI acts in same way during units creation. AI perceives actual affordable units and randomly chooses them for creation till there are no affordable units, or there are no available slots to fill.

Team Play

AI is capable to distinguish friendly and unfriendly agents – this means their units and bases. So it will not attack friendly units, just will ignore them. No real communication proceeds between allies. For example they steal a *knowledge* resources between each other. This level of AI just can not comprehend, that it could be better to leave it to its friendly owner.

Medium AI

At this level AI posses some interesting abilities. The main difference against *Simple AI* is that this level of AI *is able to communicate* with its allies. As already mentioned, for this purpose a simple communication API was created. Medium AI is able to use only some of this *"common language"*, therefore is able to co-operate with allies, which are able to use common language. This level differs from Simple AI in an ability to make an agreement on the role with other allies.

Unit creation

For **standalone** state, or state without allies, is behaviour of *Medium AI* same as behaviour of *Simple AI*.

The difference occurs when is Medium AI in *co-operation* state. Unit are created according to the role. Seizer is trying to buy a units which can travel to greater distance. Attacker is trying to buy a units that are able give better damage.

Team Play

Apart of *Simple AI*, there is the role deciding, its alternation among other allies and a resources agreement. This means seizer marks which resources are in its interest and attackers have the rest. The role is designed to change in regular intervals, to prevent one sided domination of ally.

Advanced AI

Main difference between *Medium AI* and *Advanced AI* is in an advanced battleground analysis. This level of AI is designed to try to predict an enemy's ambush and act. This is thanks to dividing a game environment into four sectors. AI is able to ask for a coordinated attack and the asked AI can give a negative answer.

Unit creation

Advanced AI is able to calculate best combination of affordable units for actual round. This differs according to actual chosen role, or situation. This designed approach solves some weaknesses inherited from *Medium AI*. Thanks to this designed approach, AI is much more adaptable to threats or opportunities.

Team Play

The main difference is in communication pattern, which is now more dynamic. The agent is capable of analysis of its environment and asks its allies for coordinated attack. The other agent according to its analysis is capable of negative or positive answer.

Chapter 5

Implementation

In this chapter will be briefly described some selected implementation problems of Bachelor Wars. This chapter is not meant as a detailed description of the whole game or its source code. It just briefly describe some interesting parts of implementation. At first, reader will be introduced into implementation in general in a section 5.1. In this section some interesting parts from whole game are discussed, like implementation of pathfinding and alike. In section 5.2 are briefly introduced interesting parts and problems connected with AI implementation. Later in section 5.3 are described implemented objects in the game and some of their interesting parts. Interface implementation and its functionality is described in section 5.4. Last part of this chapter, section 5.5, is devoted to known problems in the game.

5.1 Implementation in General

As already mentioned in chapter 4, the game is implemented in such fashion that its context is easily extensible and modifiable. Whole design and implementation was done with idea of interfaces that allow adding new units to the game, and new levels of artificial intelligence. Because some problems connected with implementation details and alike, these was not implemented. But it would be shame not to highlight the effort given to this idea. This *e.g* suggest that it is possible to create a new unit very easily. Actually implementing whole new unit is matter of few lines of code, therefore it could be proposed as future extension. The same applies for possible future extension of interface for adding new AI levels.

All implemented classes contains available API for their instances. During implementation a great effort was given into maximized usage of inheritance and object composition. Thanks to this approach any object is perceived similarly to *black box*. As great example can serve a class `Node`, which represents a cell of the game grid. This class has an API which allows adding and retrieving any game object, for implemented objects it has some user-friendly optimizations.

Implementation in general is thread-safe. This means that an access to the shared variables is synchronized. This implementation step was done after realizing, that Jason uses another threads and they call methods from commonly used `GameEnvironment` class. This and other important class dependencies are shown in figure 5.1.

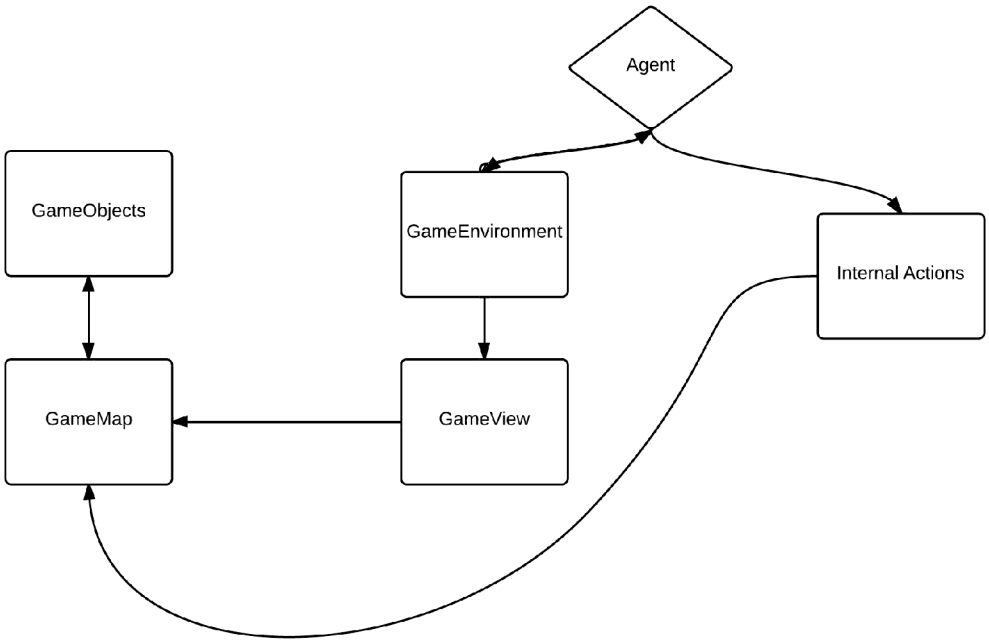


Figure 5.1: Higher abstraction example of the implementation scheme.

Node

This class is interesting not only for possibility of adding unknown game objects. Even if its implementation was not so hard, it can be considered together with a class `GameMap` as backbone of the game. Its functionality consists of generating game grid or implementing pathfinding too. Proper functionality of this class is necessary for every action in the game.

Pathfinding

The A* Algorithm[10] used for the evaluating of the shortest path is a little bit modified. It is because of instances of class `Unit`. Basically it work in two modes:

- do not ignore units;
- ignore units.

This is due to implementation issue, when the environment changes are on such level, that path, which normally exists, is not available due to actual unit's composition on the game map. In figure 5.2 is shown the problem, when the classic A* can not find a path to the *target*. When such problem appears the search algorithm switch itself from mode *"do not ignore units"*, into mode *"ignore units"*. In figure 5.3 is visible an ideal solution for this particular problem. This *"ideal"* solution is partially used, because of flag *ignore units*. The final solution in figure 5.4 hints, that as result is used ideal path *till* ignored units appears. This is possible thanks to *cohesion* of nodes in a particular found path and a path representation. For this representation is used Java's `LinkedList`. For better optimization, the Manhattan distance evaluation is used.

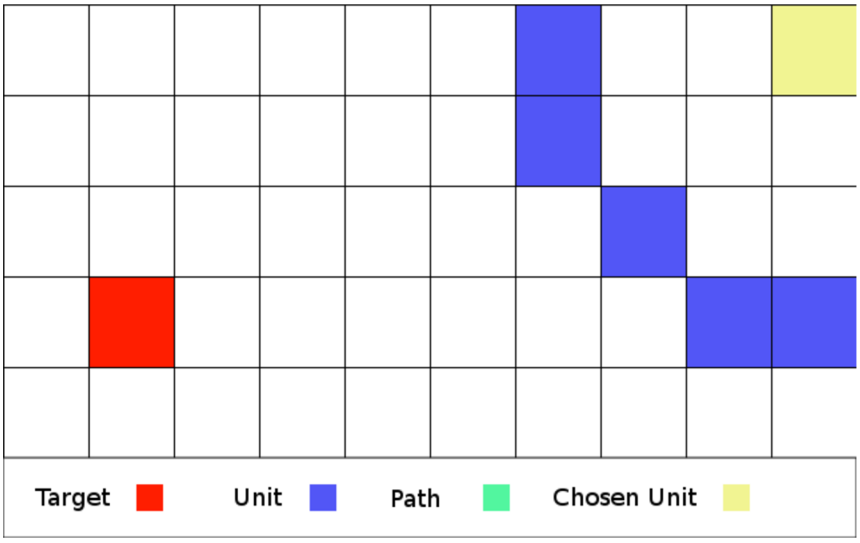


Figure 5.2: An example of problem, when A* can not find a path.

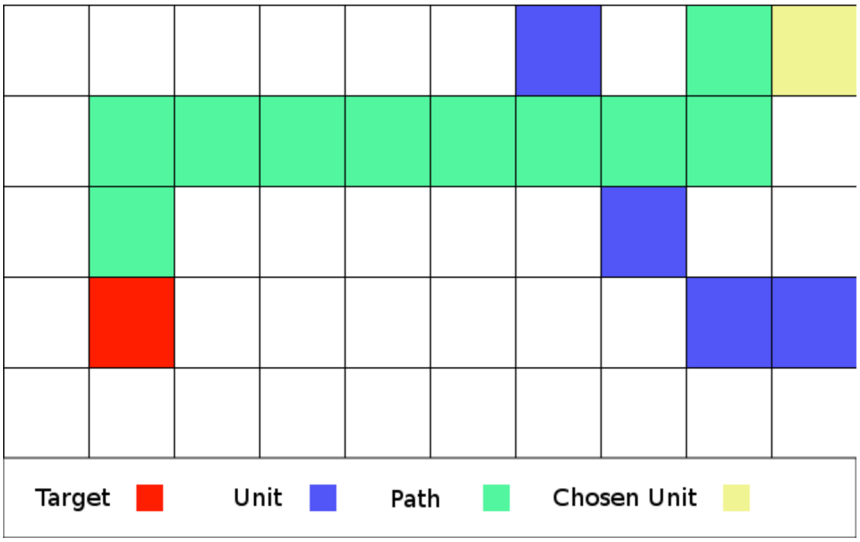


Figure 5.3: An example of ideal path solution.

Game Map

Game map is represented by the most important class in the implementation, `GameMap`. It is the game environment for players. It posses all informations about units, bases, knowledge resources and obstacles. It connects all informations so much needed for battleground analysis, statistics and gameplay itself. An example of the actual game is shown in figure 5.5.

`GameMap` is interesting due to another functionality too. It posses a method for base and unit creation. It interconnects data from class `GameSettings` into object representation together with connection of agent and its base. Apart from that, `GameMap` represents a battleground of the game Bachelor Wars, and it is a main entrance for the user of the

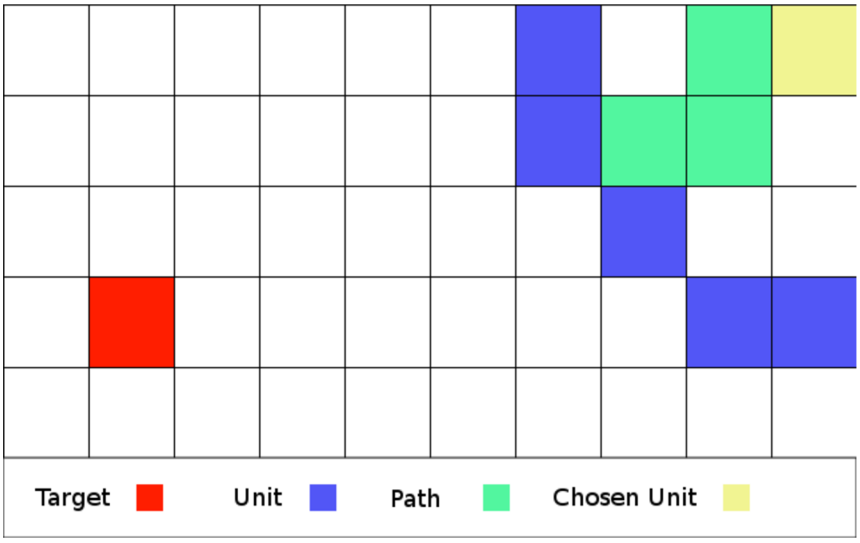


Figure 5.4: An example of the final solution, used for a particular unit.

Figure 5.5: Gameplay of domination mode.

game too. Therefore `GameMap` allows an interaction with every object in the game. This is thanks to already described nodes and an ability of each object to detect if user clicks on its shape.

This ability is used for object with meaning for it. Like bases and units. Knowledge resources and obstacles can react for player actions too, but for them this ability in final game has no meaning.

Another interesting part on the `GameMap` is an ability to draw unit's influence. This means, that for chosen unit is drawn its moving options and attack distance. An example is shown in figure 5.6.

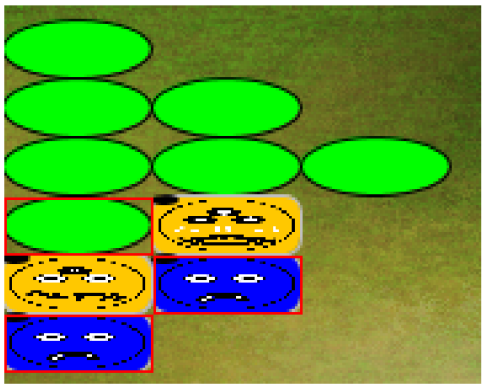


Figure 5.6: Example of *area of influence*.

5.2 Implementation of AI

Actually implemented AI levels (see 4.5) are stored in `./src/asl/`. The main effort during the implementation of AI was to inherit as much already implemented behaviour as possible. Therefore basic behaviour is common for every level.

Every action that agent can do in the environment needs to be done in its turn. As mentioned in 3.1, the main difference between RTS and TBS games is in time flow. Time flow between 2 players in TBS is relatively simple. Players are taking their turn after each other. More problematic is a game, where can be N players. This particular problem was solved by reacting on `+!can_act` belief addition. This belief is extremely important. Till this belief is not added, agent is in *"hibernated"* state.

The most important class for agent communication with environment is `GameEnv` class. In this class are methods, representing behaviour of external actions like `update_percepts` or `do_intention_if_possible`. `GameEnv` can be understood as the main entrance for communication with game environment and its objects. Another way how to communicate is trough internal actions, which are described in appendix A.3. From the point of communication there are three levels, in which an agent can operate:

- the agent understands no communication *(Simple AI, Medium AI, Advanced AI)*;
- the agent understands simple communication *(Medium AI, Advanced AI)*;
- the agent understands more advanced communication *(Advanced AI)*.

This means, that team members are operating between each other with their common language. While team where is Medium and Simple AI do not co-operate at all, because they do not understand each other, team composed from Medium and Advanced AI can co-operate with the simple communication.

Simple AI

This level of artificial intelligence differs from *Medium Advanced* one by an inability to communicate with allies. Main difference is visible *e.g.* on an achievement goal `!agree_with_allies`, which just does not exist in this level of AI. Without this goal, Simple AI just marks its start and continue with `!check_action(AgentID)`.

Medium AI

Medium AI differs from Advanced AI in a way of communication with its allies and an usage of some goals, like `!getMostMovableUnit` and alike. From the view of implementation is more interesting the communication between allies. Here is this communication used for and an agreement which one from the alliance will be seizer and attacker and for role alternation.

At the very beginning every agent asks its allies for an evaluation of their distance from the base to the knowledge resources. This value represents the sum of such distances. According this sum, agent can find out which role is best for him. Then this roles are alternating among allies in static way. This means that every 5 round roles are changed.

Changing the roles is is done between agent in that way, that actual seizer tells the next one *you will be seizer* during next change. The next seizer is randomly picked from attackers. This is done be adding a belief `imSeizer(Pos, Mark)`. The meaning of this is to prevent one sided strategy and buying stereotype. As already described, seizer role is weak against attacker, but attacker has slower units. It is good to change this roles to gather more flexible sort of units.

Medium AI without team is not different from the Simple AI. This is because the unpredictable behaviour of Simple AI has proven as very effective.

Advanced AI

The main difference from Medium AI without the team is the ability compute best combinations of affordable unit according to the situation on the map. This means if AI posses weak units, it will try to buy the best combination of units for available knowledge. This combinations are computed through internal actions.

Another main difference is that it tries predict an enemy's attack and tries defend itself from it. This is possible thanks to dividing a game map into sectors. In every sector is computed enemy's threat. According this computation is evaluated which enemy is probably trying to assault base and which enemy is not so secured against the attack.

In the matter of team play, Advanced AI is able to change buying pattern and be more adaptable. So this means that apart Medium AI seizer, Advanced AI evaluates the possibility of attack. If its result is *threat*, Advanced AI will produce combinations of affordable units according their attack ability.

5.3 Implementation of Game Objects

Every object in the game inherits its base properties from a class named as `GameObject`. This class purpose is, in a matter of abstraction, an encapsulation of bunch of nodes into a *shape*. This shape can react and be displayed if required. Because of huge flexibility of this game and modification possibilities, every new modelled class should inherits from `GameObject`. By this action will be secured backward compatibility with the game environment. For compatibility with Jason[7, 17], every GameObject shloud be able to be represented as a list.

Units

Class `Unit` is an extended representation of `GameObject` class. Its main role and design is explained in 4.2. Here are discussed some specific implementation details, which were

chosen as interesting.

Units are necessary elements for Bachelor Wars. They represent a tools used for interaction between an agent (see 2.2) and the environment. They can be perceived as intention boxes, because of intention map implemented into them. This is represented by `HashMap<GameObject, Intention>`, where `GameObject` is a target object on the map and `Intention` is an action intended to the object. Actually supported actions for game objects on the map are:

- KILL;
- SEIZE.

These actions represents basic actions on the map. It is possible to construct more complex intentions from them, *e.g.* action SUPPORT. *SUPPORT* could be composed as:

1. get target object;
2. get intention map of this target;
3. copy from this map into the intention map of chosen unit.

This actions would copy KILL and SEIZE intentions of target unit and therefore actual chosen unit could be tagged for same goals by agent. For actual game proportions these basic actions are enough and no more complex actions are used.

Another interesting part is connected with graphics. Every object in the game is graphically represented on the game map. And the sequence of objects painted on the map matters. So, *e.g.* if the base would be painted later than unit, unit would be hidden by newly painted base. This problem raised when a knowledge resources (see 4.2) were seized. Player could not see which knowledge is seized and where. The problem was solved by a *white little circle* in the bottom right corner of the unit. This signalize that actual unit is seizing a particular knowledge. Same problem appeared for units which were tagged as unused. Every *unused* unit is tagged by a *black little circle* in the upper left corner of the unit.

Another interesting thing is that units are comparable. They are compared by price, so the more valuable unit is the more expensive one. This ability is heavily used in some internal actions, which detailed description can be found in appendix A.3.

Bases

Base, as described in 4.2, is a bridge between Jason's agent and game environment. They are represented by a `Base` class. Every base has its *owner*, which is individual agent in Jason and its *type*. Type represents the level of AI used for this base. These types are described in 4.5.

5.4 Implementation of Interface

At the beginning the **libGDX**[3] was used. But due to some implementation problems connected with Jason[7, 17] and a lack of knowledge, this idea was not realized. Therefore all visual components are based on graphic capabilities of primary Java's GUI toolkit – Swing[8]. Their design is described in 4.4. As was already mentioned, the main purpose of thesis is *AI* and therefore graphical representation was not the primary goal.

Shopping panel

Shopping panel, or class `GamePanel`, represents a container comprised of objects that represents units available to buy. During the designing was not exactly set how many units will be available. According to the *flexibility* and the *modifiability* of the game, this panel is able to posses units that are not in actual portfolio of this game. Implementation resizes the visible units's avatars, this means that if the number of units is incremented or lowered the panel can adapt its visual representation.

Statistical Panel

Statistical panel, or class `StatisticalPanel`, purpose is to show the user actually winning agent by terms given by chosen mode, described in 4.3. So it context is changing according to the mode. User can see an actual best seizer, killer or conqueror. Except these statistic, the user can see actual round and time.

When the living user is present as player in the game, the button *end round* appears. Its function is to fire event, that player finished all his action for actual round and the next player can act. If player has some unit chosen to do action, this unit is set to it's original place.

Info Panel

This panel consist is represented by more classes. Each is shown with different information and after different event. Here is discussed `UnitInfoPanel` class. It has one interesting implemented ability. As it is shown in the figure 5.6, player chosen some unit. This actions fires event, where `UnitInfoPanel` shows, except unit properties, two buttons. These buttons are:

- *cancel* – Reverts the actual unit position to the original one for actual game round.
- *done* – Marks the actual unit as *already used*. This means a black mark is removed and unit is no longer selectable for actual round. Its functionality is done, by attacking an enemy too.

Another functionality appears, whe player wants to buy a new unit. During such event are shown basic atributes of such unit and a button *buy*. Its functionality is according to unit's *cost* and the base's available *knowledge*. So it will not be possible to buy the unit without enough resources.

5.5 Known Bugs

Game is relatively stable, but it can happen that during initialization of the game environment graphical interface will not show up and only small coloured square is visible in the middle of the screen. If this happens, restart of the application is necessary.

As was already described in 5.3 and in 4.2, game objects are selectable. To prevent concurrent modification, while a living player is playing, he is not able to click on game objects during AI actions. This functionality is disabled when he dies, or was not playing at all. During game session without living player, response from game map is not ideal. User has to click on desired object *more* times than once. This bug does not prevent any important functionality. It is more "cosmetic" bug than functional.

Another problem is connected with game screen positioning. Every time game starts, it is not centred, but aligned to the right corner. This bug is just cosmetic bug and do not prevent any functionality.

The last known problem is connected with map generation. There is a small chance to generate obstacles that will be hindering the path to a knowledge resource. The source of problem is the generator of positions, based on random numbers. With a higher number of obstacles and the smaller map, the probability of appearance is higher.

Chapter 6

Experiments

In this chapter are discussed experiments performed with purpose to compare each level of artificial intelligence. The conditions of experiments are discussed in section 6.1. Experiment with *one vs. one* scheme is descibed in section 6.2. In a section 6.3 is discussed a team play and its result. This result shows if the agent's role coordination through communication is more effective than an approach without communication or not. The all vs. all scheme is discussed in section 6.4, which demonstrates how effective is randomized unit's buying against computed one.

6.1 Experiments Conditions

All experiments were run with default settings characteristic for every mode. These common settings were:

Map:	24 x 24
Knowledge Amount:	6
Obstacles Amount	6
Income per Round	10
Income per Knowledge	5
Number of Rounds	INF

Apart from these, special settings for mode *Domination* and *Madness* were used:

- *Domination* – rounds to seize - 3;
- *Madness* – kills to win - 25.

This can raise questions about how valid could be such experiments, if there are no stable conditions for repetitions. Actually the described experiments *have* stable parameters like number of knowledge resources or obstacles on the map, but their position is totally random. By these random positions, the most realistic behaviour can be achieved. Tests with extreme combinations of resources and obstacles, such as all resources generated very close to the one particular player, were not counted into results. All tests were run on a machine with following specification:

Model:	ThinkPad t420
OS:	Fedora 19
CPU:	Intel Core i5 - 2540M CPU @ 2.60GHz x 4
RAM:	8 GB 1067 MHz DDR3
GPU:	Intel HD Graphics
HDD:	HitachiTravelstar Z7K320 320 GB
Java:	1.7.0_55
Java VM:	24.51b03 (mixed-mode)
Jason:	Jason 1.4.0 a

In a matter of time complexity the results are heavily dependable on the generated environment. With default settings the game session usuallylasts about 35 seconds in testing mode. In normal mode, the duration of game session is around 2-5 minutes. The maximum measured value was 3 minutes in testing mode. This was due to very evenly forces and knowledge income. The lowest measured value was 12 seconds.

6.2 One vs One

In this test Advanced level of artificial intelligence played against Medium one. Together AI players played 100 matches during this test. Simple AI was not included in this test due to already mentioned behaviour of Medium AI in 5.2. Test proves, that improved ability of Advanced AI to choose best affordable combination is working. The result is visible in figure 6.1.

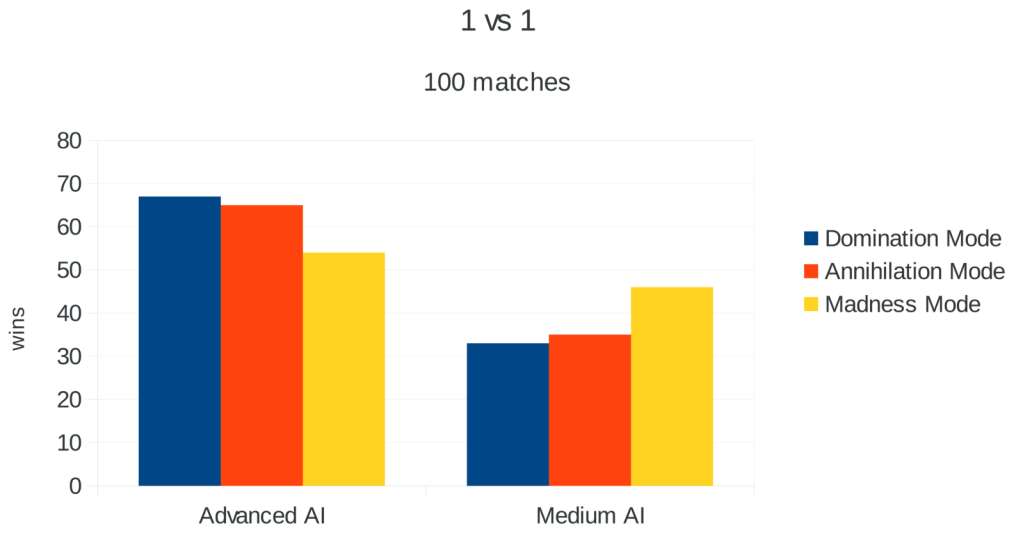


Figure 6.1: Results of *one vs. one* test.

6.3 Team Play

This experiment was performed with team vs. team scheme. According to game design 4.1, no more that 4 players are on the game map in the same time. To ensure same conditions for every team, *two vs. two* scheme was chosen. Together AI players played 100 matches during this test. Test shows interesting results. While Advanced AI is clear winner, the comparison between Medium AI and Simple AI shows that random unit picking is quite effective. This is due to predictable scheme of buying of Medium AI. It will choose always the best affordable *unit* (not combination), Simple AI has more mixed unit base, therefore is more flexible in some situations. Results are shown in the figure 6.2.

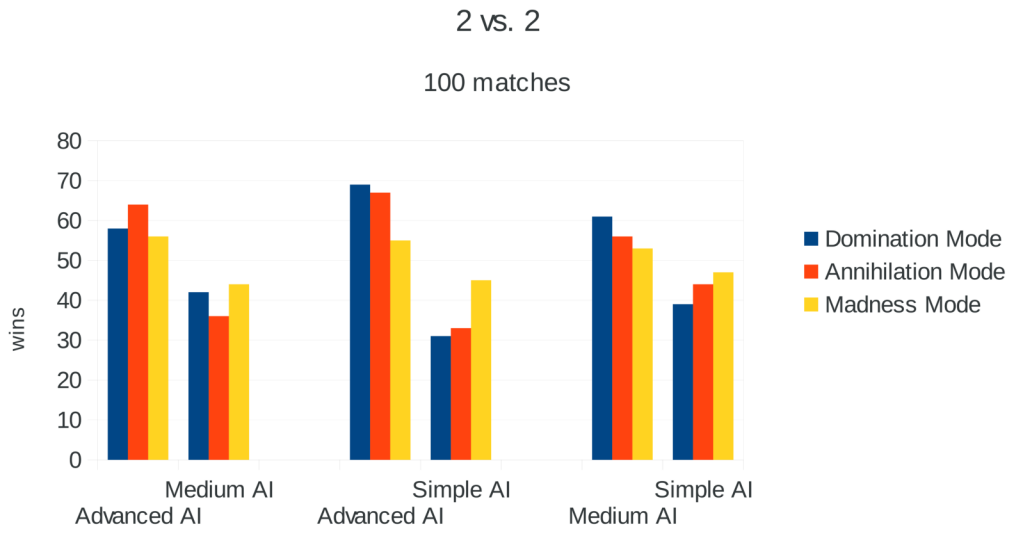


Figure 6.2: Results of *team play* test.

6.4 All vs All

The last experiment consist of *all vs. all* scheme. This experiment was chosen to compare behaviour of single Advanced AI in a game environment where is more opponents to fight, but no team player available. To ensure same test conditions, base position on the map was rotated. This was done due to position in the corner of the map. There is bigger probability defending itself against two opponents at the same time. The results shows clear domination of Advanced AI in *Domination mode* and *Annihilation mode*, which is thanks to intelligent unit picking. The *Madness* mode shows, that the random picking is almost as effective as the intelligent one in the matter of killing. The Medium AI and the Simple AI were put together, although they are on the same level without team player, because of the cosmetic matter of graph generation.

Results are shown in the figure 6.3. For better explanation of these results *Domination Mode* can be used. The blue bar represents results of each AI in *Domination Mode* with win ratio from 100 matches:

- Advanced AI – 54
- Medium AI – 34
- Simple AI – 12

Figure 6.3: Results of *all vs. all* test.

Chapter 7

Conclusion

The main goal of this thesis was a creation of a fully functional turn-based strategy game featuring two different levels of the artificial intelligence. The game should contain two main modes – *Human* vs. *AI* mode and *AI* vs. *AI* mode. As the reader could see through chapters, this goal was successfully accomplished. The game has fully functional interface with options to set players according these specifications, together with their level of artificial intelligence.

Even more, the game Bachelor Wars is capable to generate randomized maps with many options including position of bases, colours of players, income per resource and income per round and alike. If it would be not enough, there are implemented three modes for better demonstration of capabilities of AI levels. All of this is not combined with only two different levels of AI, but three, with different capabilities.

To test this artificial intelligence, series of experiments was performed. Their results clearly shows, that *Advanced AI* is more capable than others and its more adaptable approach is more effective. In the other hand, an ability of *Simple AI* to survive and win in battles *one vs. one* and *all vs. all*, mainly in the *Madness* mode is surprising. This shows, that randomized picking of unit is almost as effective as the best combination picking in the given round. Apart from this, as expected, the co-operating agents are more effective than agents without co-operation.

As possible future enhancement a creation of interfaces for unit and AI addition could be proposed. There is plenty of space for AI optimizations and enhancements too. Game itself could be extended by diplomacy and knowledge transactions between agents. Addition of new hexagonal grid environment could enhance game possibilities and bring new interesting results into account.

To summarize, this thesis achieved the original goals and proved, that it is possible to create a turn-based strategy game in Java and Jason using multi-agent approach. The game is an open-source project, thus it is free to use and available for modification.

Bibliography

[1] Cassie Landers. Early Childhood Development from Two to Six Years of Age. http://www.talkingpage.org/artic012.html#PLAY. *The Talking Pages* [online], 2013-04-05 [cit. 2014-05-17].

[2] Ghallab and Dana Nau and Paolo Traverso. *Automated Planning: Theory and Practice: The Morgan Kaufmann Series in Artificial Intelligence*. Morgan Kaufmann, CA, 2004. 978-1558608566.

[3] Zechner. LibGDX. http://libgdx.badlogicgames.com/, c 2013 -. [online], [cit. 2014-05-17].

[4] A. Pnueli. Specification and Development of Reactive Systems. In *Information Processing 86*, pages 845–858, North-Holland, 1986. IFIP.

[5] Benj Edwards. The History of Civilization. http://www.gamasutra.com/view/feature/129947/the_history_of_civilization.php, 2007 [cit. 2014-05-14].

[6] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[7] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems Using Jason*. John Wiley & Sons Ltd., 2007. ISBN 978-0-470-02900-8.

[8] Oracle Corporation. The Java Tutorials:About the JFC and Swing. http://docs.oracle.com/javase/tutorial/uiswing/start/about.html, c 1995-2014. [online], [cit. 2014-05-16].

[9] Oracle Corporation. The Java Tutorials:Getting Started with Graphics. http://docs.oracle.com/javase/tutorial/2d/basic2d/index.html, c 1995-2014. [online], [cit. 2014-05-15].

[10] Wikipedia. A* search algorithm — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=608822944, 2014. [online], [cit. 2014-05-17].

[11] Wikipedia. Chess — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Chess&oldid=607641486, 2014. [online], [cit. 2014-05-14].

[12] Wikipedia. Civilization v — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Civilization_V&oldid=608332404, 2014. [online], [cit. 2014-05-14].

[13] Wikipedia. Video game — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Video_game&oldid=607784112, 2014. [online], [cit. 2014-05-14].

[14] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd., 2002. ISBN 0-471-49691-X.

[15] WWW pages. Firaxis Games. http://www.firaxis.com/, 1997. [online], [cit. 2014-05-14].

[16] WWW pages. Video Games Help Patients and Health Care Providers. http://unews.utah.edu/news_releases/video-games-help-patients-and-health-care-providers/, 2012-09-19. [online], [cit. 2014-05-11].

[17] WWW pages. Jason | a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net/, [online], [cit. 2014-05-11].

Appendix A

Java – Jason Interface

Jason[7, 17] has two main approaches to be able communicate with game environment. The first one is through external actions and percepts and the other is through internal actions. In this chapter are briefly introduced methods and percepts, that agents perceive and can use.

A.1 Percepts

All percepts that agents perceive are obtained through external action `update_percepts`. Here are these percepts briefly described together with its functionality:

- `actualKnowledge(int)` – how much '*money*' the agent posses;
- `freeSlots(int)` – how many units can be actually created;
- `maximumSlots(int)` – the upper limit of number of units that agent can create;
- `agentID(int)` – an ID set to agent by the application;
- `mode(int)` – a representation of the actual game mode (Domination, ...);
- `fightingPower(int)` – a fighting power of the agent for the actual round;
- `movingCapability(int)` – a moving capability of the agent for the actual round;
- `round(int) -- the actual game round`;
- `team(list)` – list of team players with their names, *e.g.* `['Medium AI 1', 'Simple AI 2']`.

A.2 External Actions

External actions are actions that usually modificate an environment in some way. Internally they are represented by atoms. If an standalone atom is found in a body of agent's plan, environment tries to execute it as action. Here are listed all external actions actually supported by implemented `GameEnv` class:

- `update_percepts` – updates percepts for all agents in game environment;
- `create_unit(AgentID, Type` – creates unit according to Agent ID and type of the chosen unit;
- `mark_done` – remove actual active agent from list of active agents and adds belief `can_act` for next one;
- `mark_start` – by this action agent will seize occupied knowledge resources and bases;
- `move(UnitID, [X, Y]` – unit with given ID will perform movement on given coordinates;
- `do_intention_if_possible(UnitID, TargetID` – unit will try perform any action from *intention map* (see 4.5 and 5.3 on a particular target.

A.3 Internal Actions

Internal actions are actions that run internally within an agent rather than change the environment. These actions should not change an environment. They could be understand like actions, that helps to gather informations about environment. The during develomplemt these internal actions were created:

- `jason.addIntention(UnitID, TargetID, Type);`
- `jason.getAffordableUnits(AgentID, ListOfUnits);`
- `jason.getAffordableUnitGroups(AgentID, ListOfCombinations);`
- `jason.getAffordableUnitGroupsByAtkStrategy(AgentID, ListOfCombinations);`
- `jason.getAffordableUnitGroupsByDefenseStrategy(AgentID, ListOfCombinations);`
- `jason.getAffordableUnitGroupsByMovStrategy(AgentID, ListOfCombinations);`
- `jason.getEnemyBases(AgentID, ListOfBases);`
- `jason.getEnemyUnitInReach(UnitID, EnemyUnit);`
- `jason.getFriendlyUnitInReach(UnitID, FriendlyUnit);`
- `jason.getKnowledgeInReach(UnitID, Knowledge);`
- `jason.getKnowledgeDistance(AgentID, Distance);`
- `jason.getNearest(UnitID);`
- `jason.getNearestEnemy(UnitID, EnemyUnit);`
- `jason.getNearestFreeEnemy(UnitID, EnemyUnit);`
- `jason.getNearestFreeKnowledge(UnitID, Knowledge);`
- `jason.getNearestFriendlyUnit(UnitID, EnemyUnit);`
- `jason.getSortedIntentions(UnitID, ModeID, ListOfIntentions);`
- `jason.getSortedIntentionsByDistance(UnitID, ModeID, ListOfIntentions);`
- `jason.getSortedIntentionsByMode(UnitID, ModeID, ListOfIntentions);`
- `jason.getUsableUnits(AgentID, ListOfUnits);`
- `jason.hasIntention(UnitID);`
- `jason.isEmpty(x,y);`
- `jason.setRole(AgentID, Role).`

Appendix B

CD Content

Directory	Content
src/	Source code of the application
src/asl/	Source code of the environment
src/java/	Source code of agents
pics/	Images used in the application
lib/	Third–party libraries used in the application
bin/build.xml	Necessary for the compilation of the application
Jason-1.4.0a/	Version of Jason used during the development
tex/	Source code of this thesis
tex/fig	Figures used in this thesis