MULTI-AGENT STRATEGO

University of Rotterdam
for Professional Education
Faculty of Computer Science
Software Engineering

Delft University of Technology
Delft Faculty of ITS
Department of Mediamatica
Man-Machine Interaction

Place, date:	Rotterdam, 26 august 2004
Author:	Mohannad Ismail

MULTI-AGENT STRATEGO

By

Mohannad Ismail

A thesis submitted in partial
fulfillment of the requirements for the
degree of

Bachelor of Computer Science

University of Rotterdam

2004

Approved by

ir. M.M.M. Abdelghany University of Rotterdam
Chairperson of Supervisory Committee

Drs.dr. L.J.M. Rothkrantz T.U. Delft University of Technology

Program Authorized
to Offer Degree Bachelor of Computer Science

Date 26 august 2004

TABLE OF CONTENTS

ACKNOWLEDGMENTS

This report describes the bachelor end report that I have done in the past 5 months. This is the final stage in the bachelor program to become an engineer. This project was headed by drs.dr. L.J.M. Rothkrantz. And I also would like to thank him for helping me and guide me during this project. Special thank to Ir. M.M.M Abdelghany for the support and help with the solutions of problems. Also special thanks to my colleagues in the lab for sharing the knowledge and helping me with the all kinds of technical problems and offering solutions, and last but not least my family for providing me with all the support I needed.

Abstract

The field of multi-agent systems is an active area of research. One of the possible applications of a multi-agent system is the use of distributed techniques for problem solving. Instead of approaching the problem from a central point of view, a multi-agent system can impose a new mode of reasoning by breaking the problem down in a totally different way.

In this report we investigate a distributed approach to playing Stratego. The individual pieces of the Stratego army are represented by computational agents that each have their own field of perception, evaluation and behavior.

A first prototype of a framework has been built that consists of a simulation environment for the agents and an implementation of the agent's evaluation function. The agents have a rule engine that generates behavior that is a resultant of the environment in which they live. This report presents a result of playing the game using agents against a human player.

Chapter 1

Introduction

This report describes an attempt to play the Stratego game with multiple agents using decentralized decision making. The Stratego game is a board game where two players battle each other with their armies of pieces. The object of the game is to capture the enemy flag by moving pieces towards the enemy and try to capture enemy pieces. An interesting property of the game is that the information the players have is incomplete, because the identity of the opponent's pieces is concealed until exposed by battles between pieces.

1.1 Motivation

Our motivations for using the multiple agent approach are as follows. When we consider a human society from a central point of view we see that it is a very complex system. A possible attempt to understand the complex behavior of a human society may be considering it as a system that is made up of individuals that each has their own characteristics, behavior patterns and interactions with each other's. It is the sum of all the local actions and interactions that constitutes the overall behavior of the society. In other words, we can understand this complex system by considering it in a distributed fashion. We expect that the distributed approach of taking a local point of view in stead of a central point of view, can not only be used to understand complex systems but may also be used to solve complex problems. This investigation is an attempt to support this hypothesis by considering the Stratego game. Specially we want to investigate whether a distributed way of playing this game will provide us with a means to break down the complexity of playing it. We believe that the Stratego game can serve as an example for supporting our hypothesis, because of the characteristics of the game. The game brings about a high complexity when seen from a central point of view. This is a direct consequence of the fact that during the greater part of the game, both players have incomplete information of the board situation. Our approach will be an attempt to handle the game's complexity by using the distributed decision-making at the level of the Stratego pieces.

The complexity of the game can best be recognized by attempting to design a computer algorithm that approaches the game the same way human players do. When a human player plays Stratego, the human takes a central point of view of the game. Up to a certain level every human can learn to play the game. We assume that the human brain can somehow handle the complexity of the game by making up tactics and strategies, form hypothesis and go after their intuition.

Depending on the level of play, some of the abilities of the human players may be implicit knowledge.

1.2 Project Goals

What is the goal of this project? The goal is to create the board game Stratego on the computer. This will be an attempt to play the Stratego game with multiple agents and using an expert system for decentralized decision making. The rule engine is Jess (the Rule Engine for the Java™ Platform). The project goal can be split up in the following subgoals:

- Literature study.
- Design a model.
- Implementation of the model.
- Build a running prototype.
- Test of the prototype.

What is the scope of this project?

- To create a working version of the game which can be played against the computer.
- The computer must use agents.
- The use of CLIPS for the rule set of the agents.

What are high-level features you are sure to build?

- An easy to use graphical user interface.
- A rule set of the game Stratego.
- A working computer opponent.

What are the high-level assumptions or ground rules for the project?

- There is no central point of view.
- Our approach will be an attempt to handle the game's complexity by using the distributed decision-making at the level of the Stratego pieces.
- Every individual piece/agent takes its decision based on its perception of the world and reasoning mechanism.
- Individuals differ in their perception of the world and reasoning mechanism.
- The game will be implemented in the programming language Java
- The main developing platform will be Windows (though it should run on any Java enabled platform)

Chapter 2

Multi-agent systems

This chapter provides some background material of the investigation. The relatively new field of research called multi-agent systems is described. But first an introduction is given to the concept of an intelligent agent.

2.1 Intelligent agents

In the computer science literature a lot of papers, reports and books contain the word "agent". Apparently it has become a very popular word of describing systems and software. But there seem to be a lot of disagreements as to what the characteristics of these systems and software are that constitute an agent.

In this Section we will review some of the main interpretations of the agent-concept. Before we start comparing definitions and interpretations we need to have an understanding of what types of agents we will discuss. The Collins English dictionary gives a rather broad definition of the word agent:

1. A person who acts on behalf of another person, group, business, government etc.
2. A person or thing that acts or has the power to act.
3. A substance or organism that exterts some force or effect.
4. The means by which something occurs or is achieved.
5. A person representing a business concern.

According to this characterization virtually any system can be classified as an agent. Thus our agent definition needs to be more specific, to characterize the kind of agents in which we are interested. Our interpretation will be in the context of agents that are used in the computer science literature. These have two main characteristics, a level of intelligence and autonomy.

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. See Figure 2.1 for a schematic view of an agent interacting with its environment.

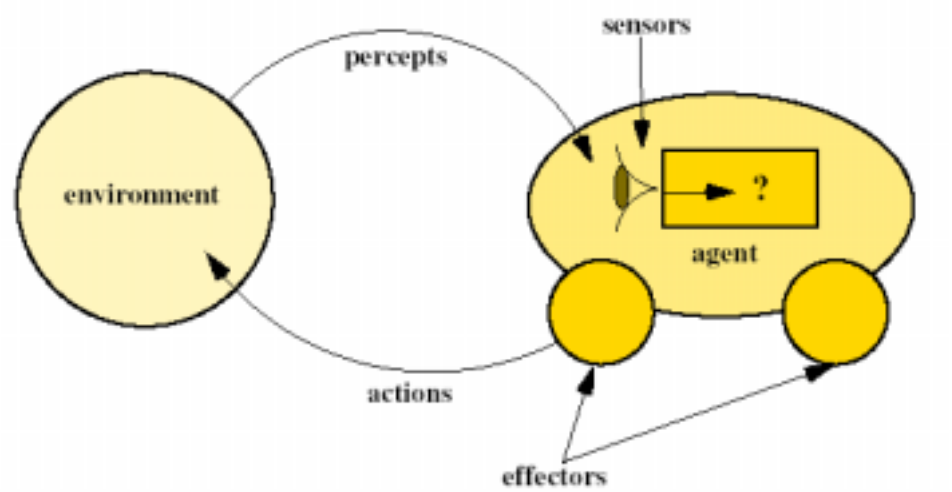


Figure 2.1: Agents interact with environments through sensors and effectors

2.2 Multiple cooperative agents

So what about multi-agent systems? A multi-agent system is a system in which multiple agents are working together, possibly in a distributed context. There are a number of reasons to distribute artificial intelligence.

A lot depends on the nature of the problems that are to be solved, or the topics that are to be investigated. Because of the possible use in multiple domains, the essentials of multi-agent systems can best be understood by considering the areas of application.

Ferber (1999) has made a classification of the possible areas of application as shown in Figure 2.2. In the following paragraphs an outline of each of these five categories of applications is given.

Problem solving. This concerns the use of software agents, brought into action to accomplish tasks that are of use to humans. These software agents are computing agents and have no real physical structure. Ferber (1999) discriminates between `distributed solving of problems' and `solving of distributed problems'. The first concerns a problem solving where the expertise to solve the problem is distributed among agents, i.e. the agent system comprises of a number of agent-specialists. The latter deals with problems that are themselves distributed. Typically, in these applications multiple agents of identical skills are used. Distributed techniques for problem solving are sometimes used for problems where the domain is not distributed nor is the expertise. Yet sometimes a multi-agent system can dictate a different point of view of the problem, which might make it possible to break down the problem to an easier way to solve.

Multi-agent simulation. Theoretical models of the surrounding world are sometimes used in simulations to explain or forecast natural phenomena. In contrast with the traditional analytical models that are used, the multi- agent approach to modelling is conceptually different. Instead of creating the model from a central viewpoint, individuals are directly represented along with their behavior and interactions. This way the modeller expects to see emergent behavior patterns arising during simulations.

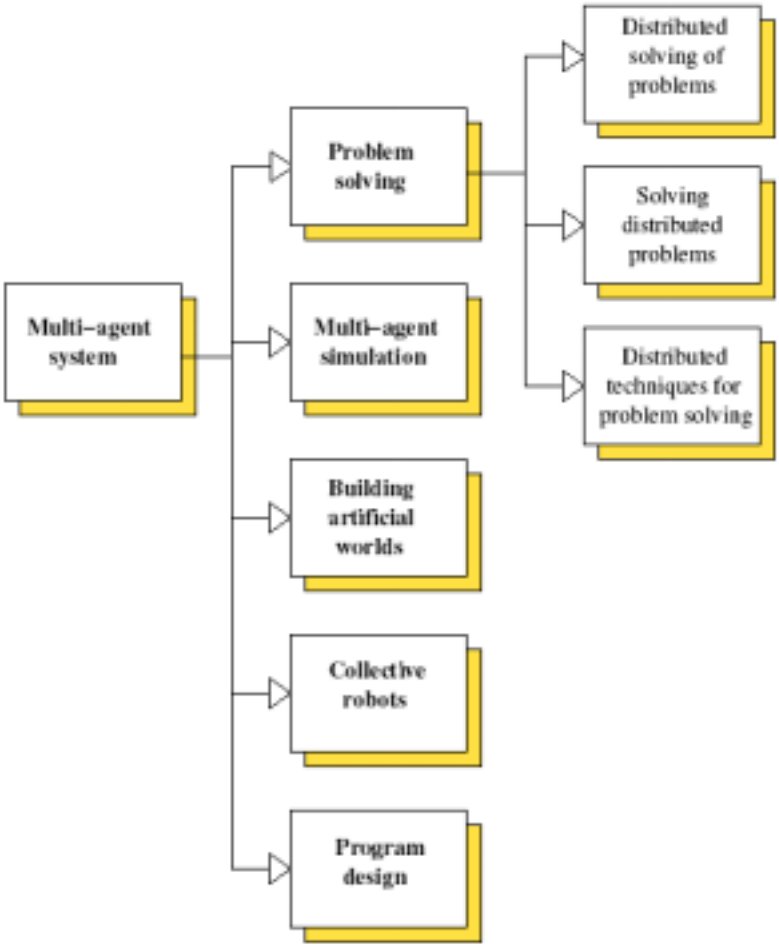


Figure 2.2: A classification of the various types of application for multi-agent systems.

Building artificial worlds. A big part of research efforts on agents is the construction of synthetic worlds. These are worlds made up by the designer for investigating the use of agents. In this case the agents are purely computational agents. These worlds are specifically constructed in order to investigate agent-concepts, interactions amongst agents or to gain an understanding of the in of behavior on the regulation of a society.

Collective robotics. An active area of research is the construction of multiple physical robots situated in a shared environment. Research interests can be to

build autonomous robots or a study of interaction and cooperation between robots.

Program design. According to Ferber multi-agent systems can be used for kenetic program design. The ambition of this concept of designing is to be able to create distributed systems and software that operate with great and ability to adapt to the environment.

Chapter 3

Playing Stratego

This chapter covers the basics of playing the Stratego game. First some general things about the game are described, the layout of the board, number of pieces and rules of the game.

3.1 The game

The game Stratego was designed by Milton Bradley Company in 1961. It is a military strategy game for two players fighting each other on a board of 10 x 10 squares. Each player has an army of pieces that each represent a military man or a military object (a bomb or a flag). Every piece has a certain rank that is used to determine the outcome of battles between pieces.

The goal of the game is to capture the enemy flag. At the start of the game all of the ranks of enemy pieces are unknown. The players have to come up with a strategy to find out the positions and identities of enemy pieces, including the flag.

Upon starting the game, the players position their armies. They are completely free to position their pieces within their side on the board, but every square can contain only one piece. The first thing that both players have to do is think carefully about the initial strategic positions of their pieces. The initial positioning is a very important part of the strategies of the game. The strength of the army is to a considerable extent result of the initial positioning.

The board consists of ten rows of ten columns of squares on which two armies begin playing with forty pieces each. Part of the squares is covered water and forbid for pieces. See Figure 3.1 for a screenshot of the board and pieces.

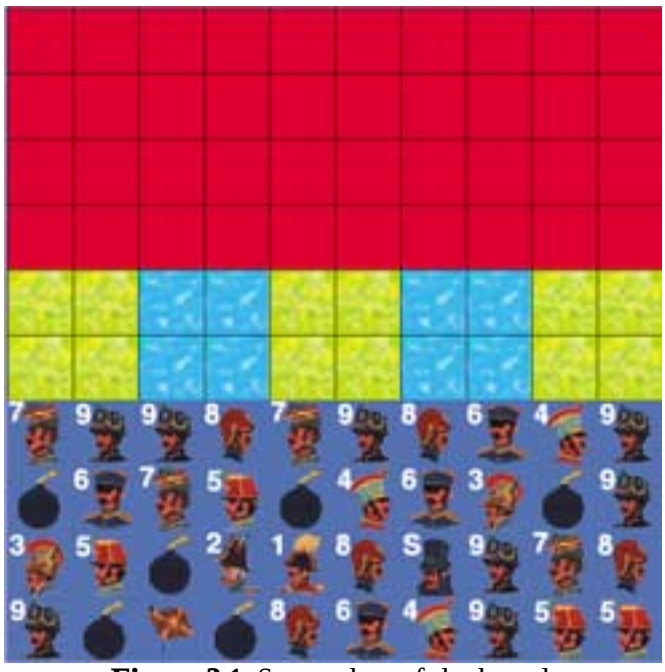


Figure 3.1: Screenshot of the board.

3.2 Rules of the game

As mentioned before the identity of any enemy piece is unknown at the start of the battle. When a piece attacks an enemy piece, the ranks of both pieces determine the outcome of the battle. The higher rank wins, which means that the looser is removed from the board. When the pieces share the same ranks both are removed. However there are two exceptions. Any piece attacking a bomb looses except for the miner who can dismantle bombs. The second exception concerns the spy. This piece can attack and defeat the marshal, but when the marshal attacks the spy the marshal wins.

A movable piece can move only one square at a time, either forward, backward, to the left or to the right. It attacks an enemy piece by trying to move to an already occupied square on the board. Two kinds of non-movable pieces exist which are bombs and the flag.

3.3 Description of the pieces

The Flag is the most important piece of the Stratego army, because once attacked the game is lost. Every enemy piece can beat it because it has the lowest rank.

The bomb is lethal for every enemy piece except for the miner, who can dismantle bombs. The bomb is the second non-movable piece. The bombs are very suitable for protecting important pieces in the army, such as the flag.

The spy is the lowest ranking moving. Every enemy piece attacking it wins, or when attacked by an enemy spy a draw occurs. However, the spy has an important quality. Upon attacking the enemy Marshall the spy wins (if the Marshall attacks the Spy, the spy loses).

The scout is piece with rank 3, which is mostly used to reveal enemy ranks at the cost of its own life. In the game there're 8 scouts for this purpose.

The miner is the only piece able to dismantle enemy bombs. It has rank 4, which means it can also capture enemy scouts & spies, but all other pieces with higher rank are lethal. 5 miners are available at the start of the game. It is important to position the miners somewhere where they can move easily. When for example an enemy bomb has been discovered, one of the miners has to come into action.

The sergeant typically belongs to the middle-ranks of the Stratego army. It has rank 5, all lower ranks except for the bomb ofcourse can be captured and all higher ranks are lethal. The initial Stratego army has 2 sergeants. The sergeants are probably at their best during the mid-game. They can capture the enemy spy and minors.

The lieutenant with rank 6, the lieutenant is the first higher rank in the army. The Stratego armies each have 4 lieutenants. Care should therefore be taken not to reveal its rank when unnecessary. However a role in the army may be to capture as much as enemy pieces as possible, because of its rank which is between low ranked pieces (that can be captured) and higher ranks (which are more important in the end game).

The captain has rank 7. With this rather strong rank the captain might want to conceal its rank for a while. It will probably come in handy in the end-game, if the captain is to be used for defences.

Also the captain can take offensive actions.

The major has rank 8. Each army has 3 majors. The major's starting position is preferably some where in the back, where it can defend lower ranks and wait for a possible attack when the risks can be estimated well. The major should think twice before attacking pieces that haven't moved yet, they may be bombs.

The colonel has rank 9. Each army has 2 colonels.

With rank 10, the General is the second strongest piece in the game, because of the Marshall's weakness against the spy, theoretically the General is the strongest piece in the game.

The Marshal is the most powerful piece in the Stratego army. It can only be captured by the enemy marshal (a draw), by the enemy bombs upon attacking them and by the spy attacking the marshal. It should never attack pieces that haven't moved yet, because they may be bombs. Preferably the marshal's rank is to be concealed as long as possible.

From our definition of agents we can elude that agents differ with respect to:

- **Perception**: we assume that agents have a different field of perception. Size of the higher the rank, the greater the field of percept.
- **Reasoning**: we assume that the agents have their own set of rules.
- **Actuator**: we assume that the agents have their own behaviour.

Chapter 4

Design

4.1 Requirements

This section discusses the requirements of our Stratego agent. These requirements will be discussed in terms of the agent's, its desired functionality and the way decisions are to be made.

4.1.1 The agent's environment

In this project almost every piece of the army is an agent (only computer played pieces). We consider 2 different types of agent's:

1. The lower rank agent.
2. The higher rank agent.

1- The lower ranked agents are: the scout, the sergeant, the lieutenant, the captain and the major. These soldiers can only look 1 square around them see figure 4.1.

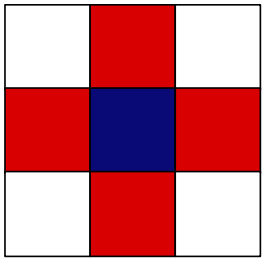


Figure 4.1: The blue agent only sees the red squares.

2- The high ranked agents are: the spy, the miner, the colonel, the general and the marshal. We considered the spy and the miner as high ranked view agents, because they have a second goal then capturing the enemy flag, which is capturing the enemy marshal for the spy and defusing the enemy bombs for the miner. Now is the percentage between low and high ranked agents is 50%. The view of the high ranked agents is 12 squares, figure 4.2

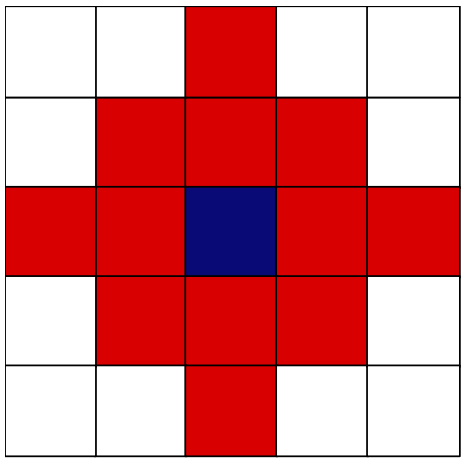


Figure 4.2: The blue agent sees the red squares.

Both kind of the agents work as figure 4.3

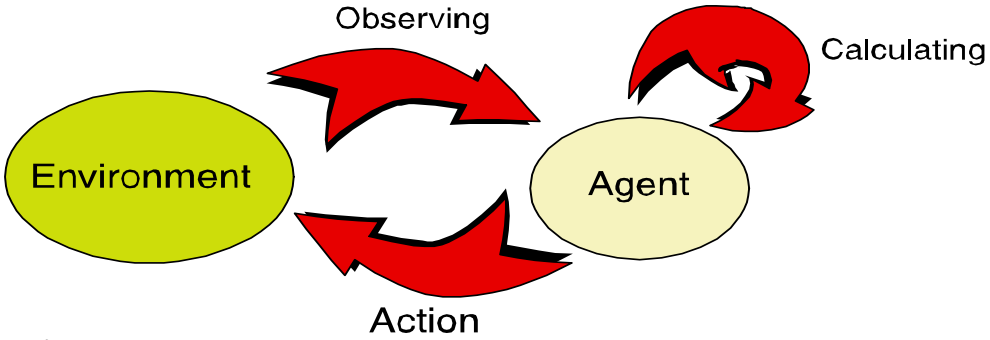


Figure 4.3: The agent observes the environment and calculates a best action.

The agent can see the environment. He calculates what's the best action[1] for him using the rule base engine that fires the rule-set with the Rete Algorithm[2]. Because we are working with more then one agent so we want the agent's to take the best action for the whole army. So now every agent communicates with the Central Decision Maker (CDM) and send his best action. The CDM decide what agent can take his calculated action. See figure 4.4.

[1] In chapter 6 we will discuss how the agents calculate their best score.

[2] In chapter 5 the Rete Algorithm will be explained

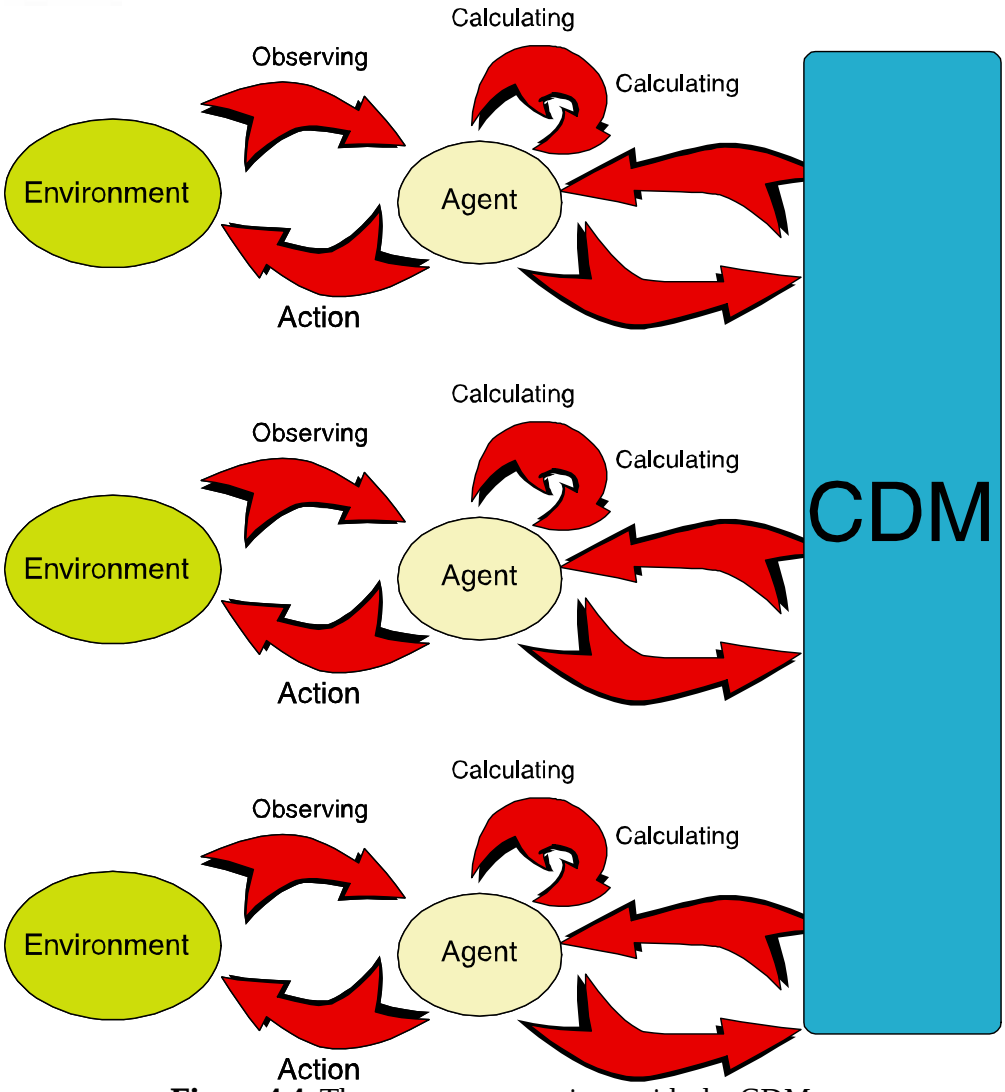


Figure 4.4: The agents communicate with the CDM

4.1.2 The agent's functionality

In designing the agents we want to make use of the fact that each piece in the Stratego army has a certain dedicated role. These roles originate from their specific ranks and the rules of the Stratego game. For some pieces this goal can be quite explicit. For example, the miner's primary goal is to dismantle enemy bombs. All pieces have secondary goals as well however, of which possibly the most important one is to stay alive. The scout however forms a clear exception to this general goal, as the scout's primary goal might be to discover the identity of enemy pieces-possibly at the cost of its own life.

We propose to define some degrees of freedom in our model of the agent that will allow us to experiment with different types of agents in the Stratego army. Specifically we define for each agent:

- The agent's perception range. Depending on the agent's role in the army the perception will be a diamond of range one or rang 2. Important pieces will have wider perceptions.
- The agent's "reactive" behavior. For every agent we define four elementary behaviors that are executed following a reaction in various situations. These behaviors are:
    1. **Attack**: attack an enemy piece that is situated within attacking range of the agent (at a distance of one square).
    2. **Flee**: move away from an enemy piece that is situated within attacking range of the agent. Ideally the agent moves to a square that is in the opposite direction of the enemy piece.
    3. **Wander**: random walk.
    4. **Stay**: do nothing.
- The agent's "cognitive" abilities, like for example evaluate situation, compute optimal next move, form hypotheses, make plans (marshal).

4.1.3 Decision making

Because of the fact that only one piece can move at a time, a mechanism has to be found that decides which agent is allowed to move. We propose three possible ways of implementing this mechanism:

1. Based on scores, where each agent evaluates its current situation and assigns scores to preferences of moving. A higher score will indicate a stronger desire to move and the agent with the highest score will be allowed to move.
2. Based on a random pick, where at each move a random choice will be made of the agent that is allowed to move. It is possible however that the chosen agent does not want to move. In this case another agent will be randomly chosen. When none of the agents wants to move, one of them will be forced to move.
3. Based on hierarchy in the Stratego army. Because we are speaking of an army, a way of organizing it may be a strict hierarchy based on ranks. In this scheme the highest rank in the army may decide which agent will be allowed to move.

We mainly use mechanism number 1 and if more then one agent highest score are equal then we use mechanism number two to pick up a random agent to move. If none of the agents wants to move e.g. because his highest score is staying then we force the agent to move.

4.2 UML

In this section some UML designs will be showed and explained. We will discuss the Use-Case diagram, the Class diagram and sequence diagram.

4.2.1 The Use-Case diagram

The human player doesn't have a lot of action to do in the game. The player can do the following actions with the program see figure 4.5.

The player can start, restart and exiting the game. And there is also an "about" button in the help menu of the game to show the game credits and version number. And for playing the game the user can set the pieces from the player-hand to the game board, and moving the pieces.

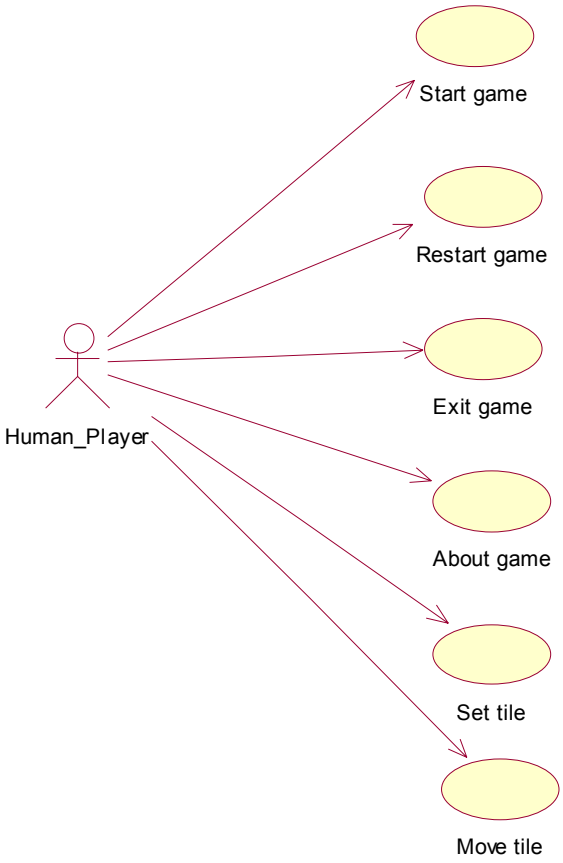


Figure 4.5: Use-case diagram

4.2.2 The Class diagram

The game uses packages see figure 4.6.

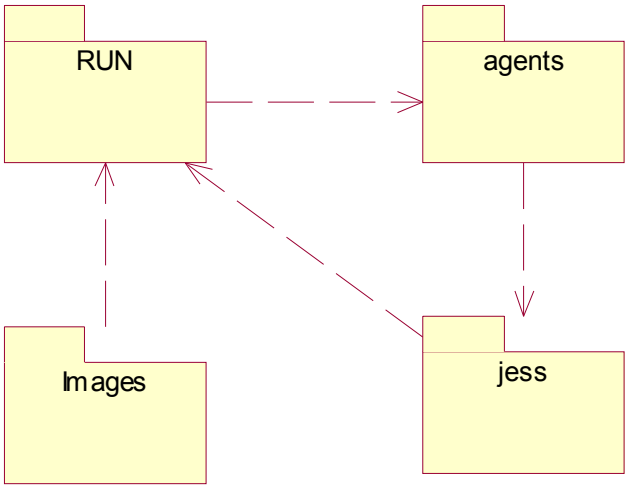


Figure 4.6: Packages

The "RUN" package is where the game is implemented and all the code is there. In the package "agents" is where the rule-set (CLIPS) of all the agents is located and also the files that make the connection between Jess and the code. "Jess" is the rule-base engine files. The package "Images" is where all the game images are stored.

If we zoom into RUN we see the following class diagram see figure 4.7

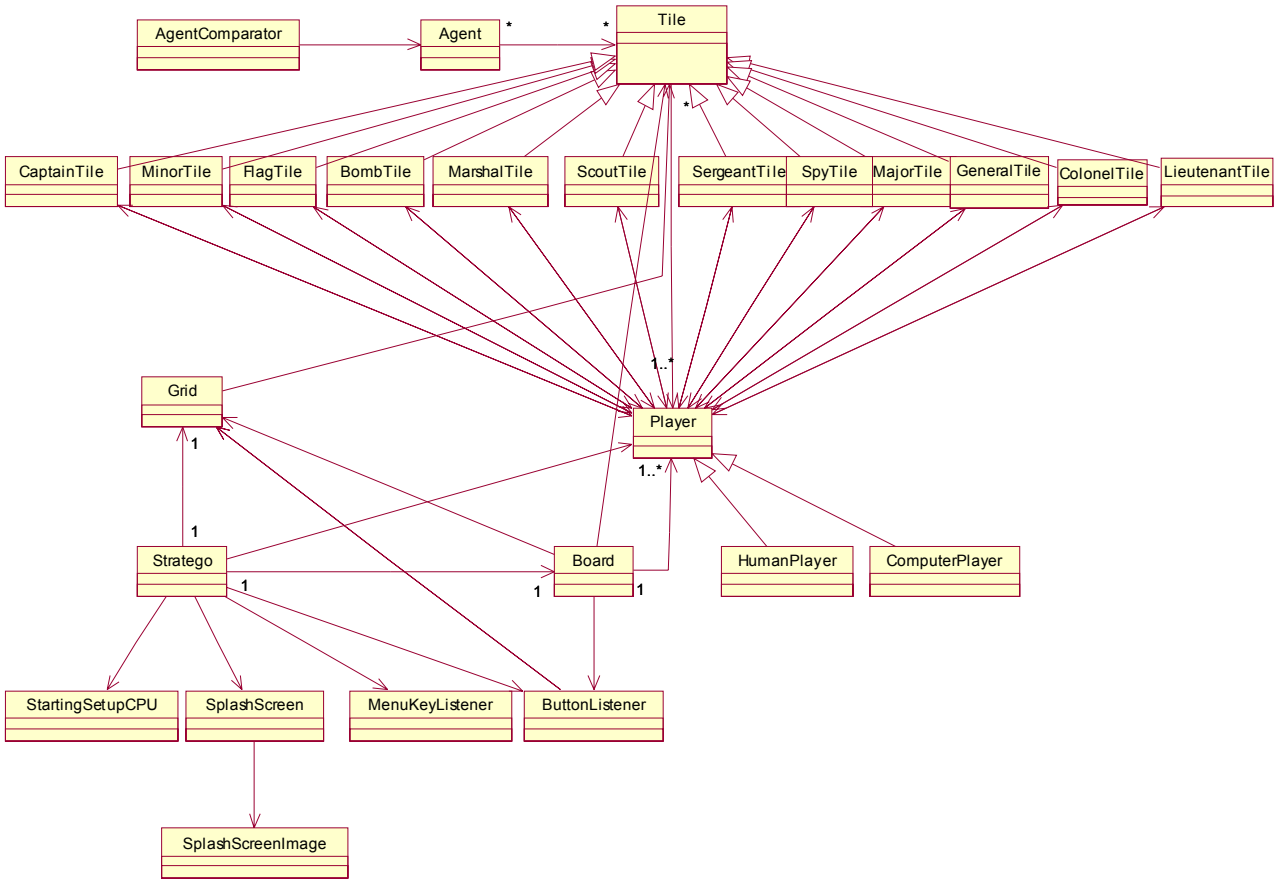
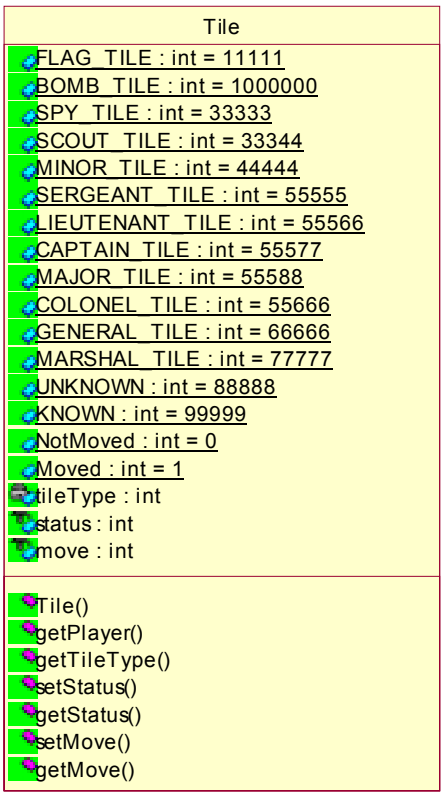


Figure 4.7: Stratego Class diagram.

See appendix for a bigger size of the class diagram.

As we can see that Tile is a super class. And the other subclasses are actually the Stratego army pieces. The agent is a Tile with the extra attribute, which are the x and y location, the agent best score and the move direction. See figure 4.8 for the Tile and the Agent classes.



From every agent we can get/set his location, score and move. The agent type is the agent rank.

Figure 4.8: Tile and Agent class.

The status of a tile is KNOWN or UNKNOWN. When the game starts all the agents are UNKNOWN if they go in a battle and win the battle, their status change to KNOWN. The move attribute shows if a Tile is moved from it place or not.

Check the appendix for the rest of the classes.

4.2.3 Sequence diagram

Here we will discuss what the system well do, when it's his turn to play. We will use a sequence diagram to summarize the method (figure 4.9).

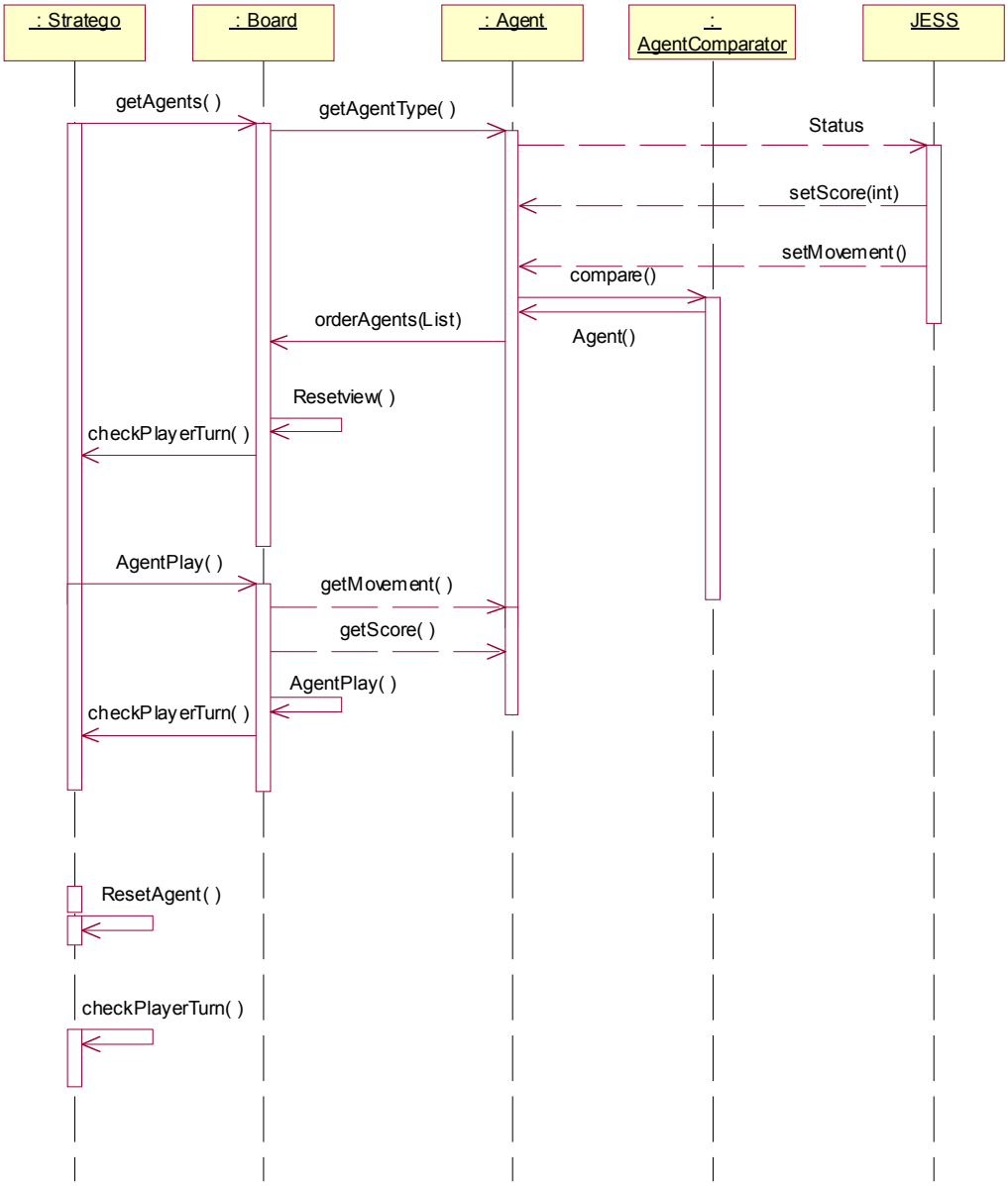


Figure 4.9: Computer turn sequence diagram.

The first step is to get all the agents on the board. After this is done every agent communicates with Jess to get his best action. Every agent sends his information to the AgentComparator to sort the agents and put the highest scoring agents at the top of the list. After that every agent percept is reset and the agent is ready to move.

The second step is the Agentplay() function. It picks a random choice from the set of the highest scoring agent. The agent tries to move, if the agent has been moved then the turn ends and everything will be reset with the function ResetAgent(). If the agent couldn't move for some reason like the end of the game board or there is water in his direction then he send an error message back. The AgentPlay function tries the second best high scoring agent to play if he couldn't move agent the third best etc… until an agent moves. It could happen that none of the agents can move, because he is blocked. An error message appears in the information area that none of the agents can't move. When this happen the game will stop.

4.3 The game board

The board consists of ten rows of ten columns of squares. In this Stratego design the decision was made to use double array. This way we can locate the agents very simple. See figure 4.10 for the schematic board view.

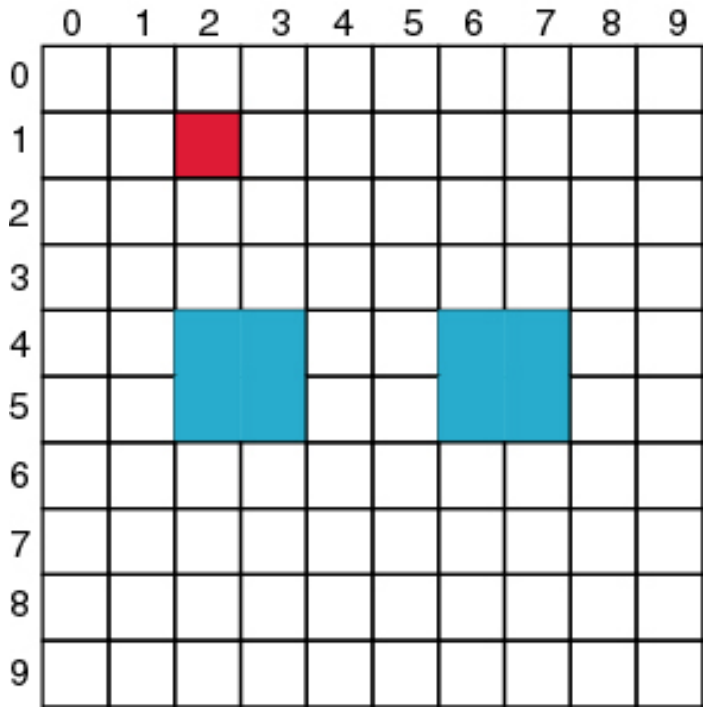


Figure 4.10: the Red agent has the location [1][2].

Also this way the agent can communicate with the other agents or check their environment with the following method:

If the agent is located at **[Y][X]**.
To check the environment from the north then the agent looks at **[Y - 1][X]**. From the south is **[Y + 1][X]**, East **[Y][X + 1]** and West **[Y][X – 1]**. See figure 4.11 for an overview.

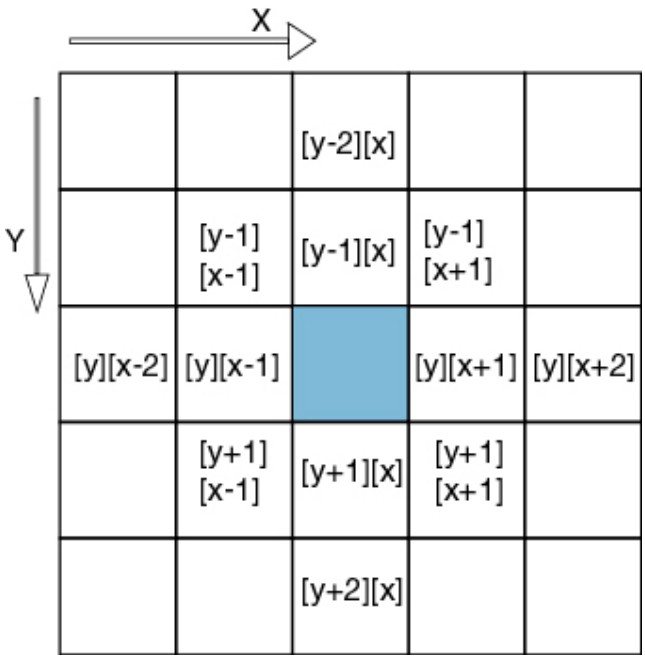


Figure 4.11: The blue agent percept in field

Chapter 5

Knowledge of the agents

This chapter focuses on the knowledge level of the individual agents. Since the agents represent pieces of the Stratego army, we want them to express behavior that can be seen as "rational" from their point of view. In other words, we want them to express behavior that will make the agents successful in achieving their goals. Our approach is based on a rule-set that explicitly defines what to do for a number of situations. Section 5.2 gives an elaborate discussion of a rule-set of one of the agents, the miner. But first Section 5.1 discusses the concept of rule-based systems in general.

5.1 Rule-based systems

Rule-based systems, also called production systems, form a well-known architecture for implementing systems based on artificial intelligence techniques. The heart of a rule-based system consists of a database, a rule-base and an inference engine. These components interact with the external environment through a perception of the environment and an execution to in the environment (see Figure 5.1).

The database contains a representation of the state of the environment in asserted facts. Upon perceiving the environment the agent asserts corresponding facts in the database. The rule base consists of a set of rules, each of which maps a specific state in the environment to one or more actions the agent performs. The rules take the following form:

if <list of conditions> then <list of actions>

Where <list of conditions> is associated with asserted facts in the database and <list of actions> are actions that may update other facts in the database or in the external environment. The connection between the facts in the database and the rules in the rule base is made by the inference engine. Upon assertion of

facts, the inference engine considers all rules in the rule base. When a state of the world matches a rule, the rule is said to be fired.

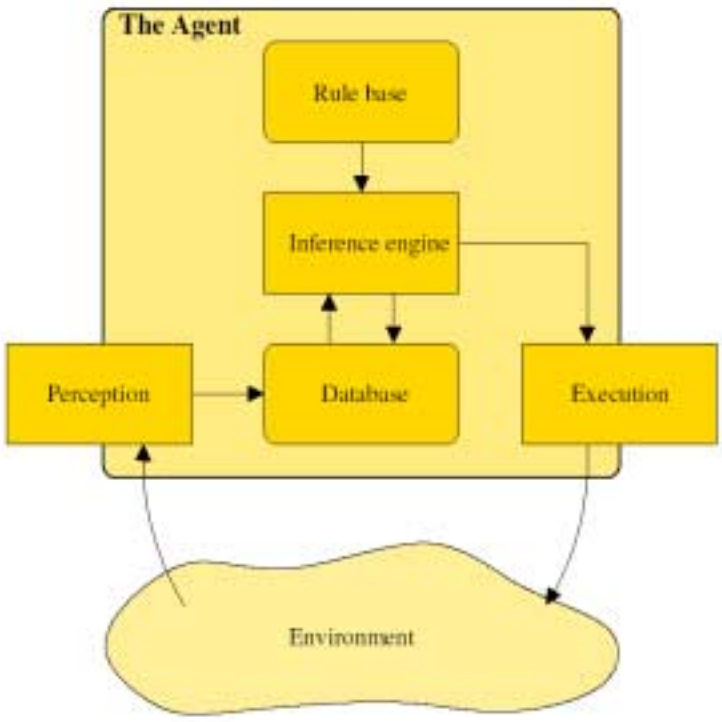


Figure 5.1: An agent based on a rule-based system

5.2 Rules for the agents' behavior

For each of the Stratego agents we have defined a set of rules that specify the behavior, according to the current situation of the agent. We call these rule-sets preference rules, since they indicate preferences to exhibit behavior rather than performing explicit actions.

In this section we will describe a preference rule-set for the miner agent in the Stratego army, check the appendix for the rest of the agents rule-set's. Every rule in the set defines several conditions to activate the rule and a preference that is expressed upon activation. The use of preferences instead of actions in

the rules arises from the desire to allow separate behaviors to be activated simultaneously. In Section 5.2.1 an example of a preference rule-set will be given.

5.2.1 Preference rules for the miner

Here we will give some example preference rules for the agent with rank 4, which is the miner. Its complete rule-set, along with the rule-sets of the other agents can be found in Appendix A. Since the miner has a dedicated role within the Stratego army, its behavior has to be somewhat cautious.

The miner has 29 preference rules, which take the following conditions into consideration:

- **Enemy bombs captured**: when all enemy bombs are captured there will be no more bombs to be dismantled and the miner will become less cautious.
- **I have moved**: when this condition is not met, the miner will be less eager to move since it does not want the enemy to know it is a movable piece.
- **My rank revealed**: when this condition is not met, the miner will be less eager to attack pieces because it will try to conceal its rank as long as possible.
- **Enemy at distance 1 or 2**: an enemy piece is spotted within range 1 or 2. Possible types of enemies are:
    1. An enemy piece with unknown rank.
    2. An enemy piece with a higher rank.
    3. An enemy piece with a lower rank.
    4. An enemy bomb!

Preference rule 1
This rule will fire the preference "attack" when the following conditions are met:
→ enemy bombs captured,
→ I have moved,
→ my rank revealed and
→ enemy with unknown rank present at distance 1.

In this case the miner is not very cautious because it does not need to dismantle bombs anymore, the enemy already knows that it is not a movable piece and its rank is already known.

Preference rule 13
This rule will fire the preference "fleeing" when the following conditions are met:
→ I have moved,
→ my rank revealed,
→ not enemy bombs captured and
→ enemy with unknown rank present at distance 1.

Here the miner has a preference for fleeing, because it still has to dismantle bombs and the enemy knows the rank of the miner.

Preference rule 22
This rule will fire the preference "stay" when the following conditions are met:
→ not I have moved,
→ not my rank revealed,
→ not enemy bombs captured and
→ enemy with higher rank present at distance 1.

Here the miner dares to stay in spite of the fact that it can be attacked by the enemy. The reason for staying is the fact that the miner has not moved yet and its rank is unknown. The miner wants to prevent the enemy from knowing that it is movable and the enemy piece may be careful.

Preference rule 27
This rule will fire the preference "attack" if and only if an enemy bomb has been spotted at distance 1. It is the task of the miner to dismantle bombs, therefore it will attack.

5.3 The Rete Algorithm

Jess is a rule-based expert system shell. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements (*rules*) to a set of data (the *knowledge base*). You define the rules that make up your own particular expert system. Jess rules look something like this:

```
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)
```

Note that this syntax is identical to the syntax used by CLIPS. This rule might be translated into pseudo-English as follows:

Enemy higher rank #rule1:
If
My rank is NOT revealed
And
An enemy of a higher rank is a distance 1
Then
Stay

The rank and the enemy higher rank entities would be found on the knowledge base. The knowledge base is therefore a kind of database of bits of factual knowledge about the world. The attributes (called slots) that things like ranks and enemy distance are allowed to have are defined in statements called `deftemplates`.

The typical expert system has a fixed set of rules while the knowledge base changes continuously. However, it is an empirical fact that, in most expert

systems, much of the knowledge base is also fairly fixed from one rule operation to the next. Although new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the expert system shell is very inefficient. This obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the knowledge base and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the knowledge base is stable, most of the tests will be repeated. You might call this the *rules finding facts* approach and its computational complexity is of the order of O(RF^P), where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the knowledge base. This escalates dramatically as the number of patterns per rule increases.

Jess instead uses a very efficient method known as the Rete (Latin for net) algorithm. The classic paper on the Rete algorithm ("Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19 (1982), 17-37) became the basis for a whole generation of fast expert system shells: OPS5, its descendant ART, and CLIPS. In the Rete algorithm, the inefficiency described above is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(RFP), or linear in the size of the fact base. Our discussion of the Rete algorithm is necessarily brief. The interested reader is referred to the Forgy paper or to Giarratano and Riley, "Expert Systems: Principles and Programming", Second Edition, PWS Publishing (Boston, 1993) for a more detailed treatment.

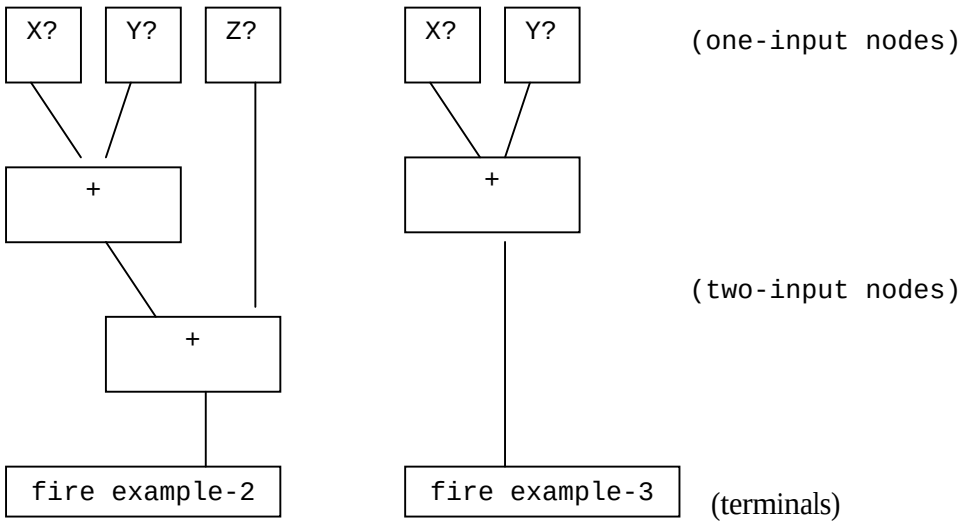
The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the knowledge base are processed by this network of nodes. At the bottom of the network are nodes representing individual rules. When a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an activation. The associated rule may have its RHS executed (fired) if the activation is not invalidated first by the removal of one or more facts from its activation set. Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used and there are also auxilliary types such as the terminal nodes mentioned above.

An example is often useful at this point. The following rules:

```
(defrule example-2
    (x)
    (y)
    (z)
    => )
```

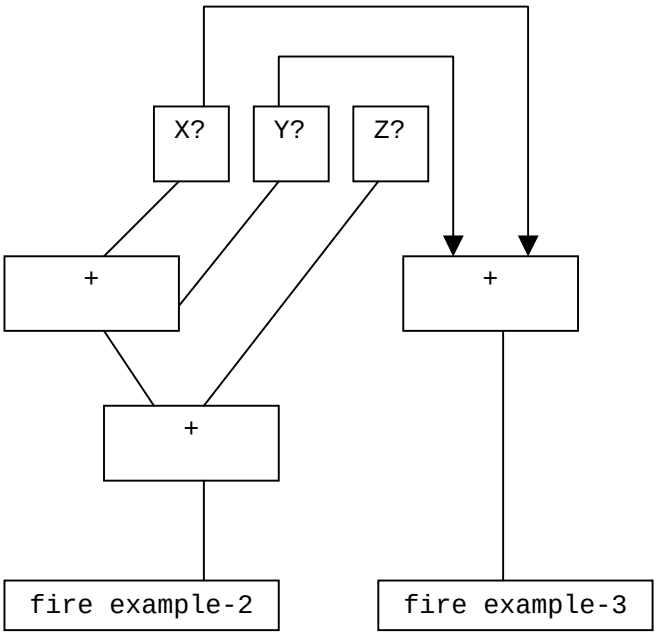
```
(defrule example-3
    (x)
    (y)
    => )
```

might be compiled into the following network:

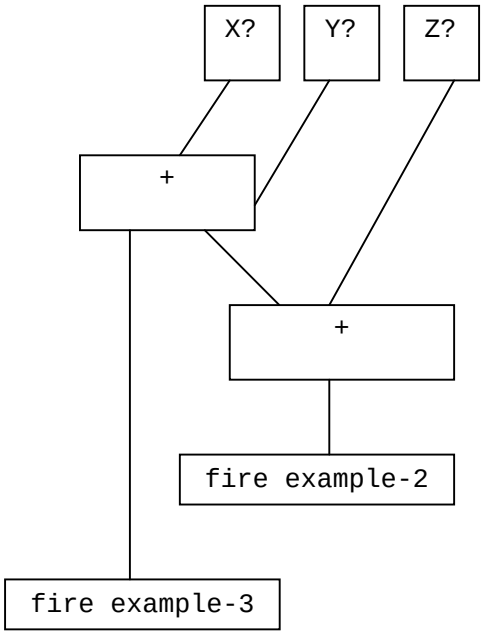


The nodes marked `x?`, etc., test if a fact contains the given data, while the nodes marked + remember all facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the knowledge base. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a *left memory* and a *right memory*. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the *pattern network*, while the two-input nodes make up the *join network*.

There are two simple optimizations that can make Rete even better. The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. The second is by modifying the network to share these nodes across the two rules (the arrows coming out of the top of the x? and y? nodes are outputs):



But that's not all the redundancy in the original network. Now we see that there is one joined node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:



The pattern and joined networks are collectively only half the size they were originally. This kind of sharing comes up very frequently in real systems and is a significant performance booster!

We can see the amount of sharing in a Jess network by using the `watch compilations` command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

```
example-2: +1+1+1+1+1+1+2+2+t
```

Each time `+1` appears in this string, a new one-input node is created. `+2` indicates a new two-input node. Now watch what happens when we compile example-3:

```
example-3: =1=1=1=1=2+t
```

Here we see that `=1` is printed whenever a pre-existing one-input node is shared; `=2` is printed when a two-input node is shared. `+t` represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected. Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3. Jess shares existing nodes very efficiently in this case.

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class jess.Node: Node1, Node2, NodeNot2, NodeJoin, and NodeTerm. The Node1 class is further specialized because it contains a command member which causes it to act differently depending on the tests or functions it needs to perform. For example, there are specializations of Node1 which test the first field (called the head) of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a

fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class jess.Jesp, while the actual network is assembled by code in the class jess.ReteCompiler. The execution of the network is handled by the class Rete. The jess.Main class itself is really just a small demonstration driver for the Jess package, in which all of the interesting work is done.

Chapter 6

Implementation

In this chapter we will describe the implementation for the Stratego expert system shell. The implementation has been done using the object-oriented programming language Java with the use of Jess the Java Expert System Shell (the Rule Engine for the Java™ Platform).

6.1 Jess - Java Expert System Shell

Since the game Stratego was developed using Java, a natural choice for an expert system shell is the Java expert system shell (Jess). Jess is a rule engine and scripting environment written entirely in Java. It was originally inspired by the CLIPS expert system shell, but has grown into a complete, distinct Java-in environment of its own. Because of its complete implementation in Java, the rule-engine can be easily embedded within the Java simulation environment. For detailed information about Jess see (Friedman-Hill 2000).

For every agent we have implemented the behaviors as described in Chapter 4, which are attack, wander and stay. Additionally, we have added some extra behaviors that apply to specific situations. These are:

1. **Attack-marshal**: a specific rule for the spy. When the spy sees the enemy marshal within attacking range, this behavior will make the spy attack it.
2. **Attack-bomb**: a specific rule for the miners. Upon seeing an enemy bomb within attacking range, the miner will be eager to attack it.
3. **Avoid**: a rule that is applied for all agents except the miners, because it is used to avoid enemy bombs.

In the prototype, 5 agents have a visual perception in the form of a diamond of five squares wide. See Figure 6.2 for a picture of an agent (miner) in its

environment. The miner sees an enemy piece with unknown rank (north square) and an enemy scout (north-east square). The agent also sees some

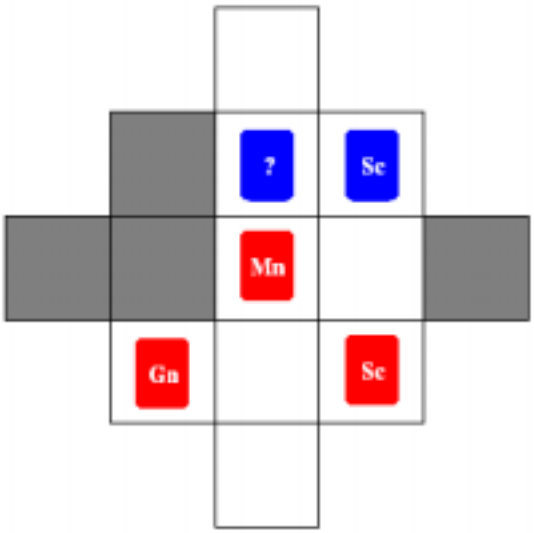


Figure 6.1: The miner in its environment

fellow agents, the general and a scout (south-west and south-east respectively). In the current implementation of the rule engines, the evaluation consists of a mapping from enemy locations to a desire to move (for each direction) or to stay, expressed in scores. In this specific example, the miner may want to flee from the unknown enemy. But it also sees an enemy scout which can be beaten. Therefore in this particular case the miner's behavior will be a mixture of the desire to and attack and wander:

```
      -150
75     0     60
      225
```

The scores indicated above express relative desires to go or stay. Negative scores mean that the agent does not want to go in the corresponding direction. In the example the scores are a resultant of the behaviors attack and wander. The behavior is due to the enemy with unknown rank. Since the miner is a somewhat cautious agent, the score to move backward is largest and the miner

will decide to go backward. The wander behavior is a less important behavior, which is used to express the general desire to move around the board. It has contributed to the scores with minor additions to each direction, the forward direction being the preferred wander-route.

The score above came from the "agent.clp" which is the general rule-engine of an agent. For every action and direction to move there is a set of score defined e.g.:

```
flee-scores
   (flee-score north -200 50 200 50)
   (flee-score west  50 -200 50 200)
   (flee-score south 200 50 -200 50)
   (flee-score east  50 200 50 -200)
   (flee-score north-north -100 50 100 50)
   (flee-score north-west  -75 -75 75 75)
   (flee-score west-west   50 -100 50 100)
   (flee-score south-west  75 -75 -75 75)
   (flee-score south-south 100 50 -100 50)
   (flee-score south-east  75 75 -75 -75)
   (flee-score east-east   50 100 50 -100)
   (flee-score north-east  -75 75 75 -75)

misc-scores
   (wander-score 50 25 10 25)
```

And because the decision was moving south, the wander score for south will be added to the flee south score **200** + **25** = 225. The wander score will be also added to the other direction.
North = -200 + 50 = -150
East = 50 + 10 = 60
West = 50 + 25 = 75

6.2 Simulating the agent's environment

When it's the computer player turn in the game the computer call all the agents on the board. Every agent tells where he is located and what he sees. And then every agent calculates his best action with communicating with the database (Jess). Jess run all the facts in the agent rule-set and send back the agent best action to do. The agent sends his location, type, status and score to the Central

Decision Maker (CDM). The CDM decide using the highest score of the agents which agent can make the move. See figure 6.2 for schematic simulating.

Figure 6.2: agent schematic simulating.

1. The agent observes the environment.
2. The agent sends his information to Jess.
3. Jess calculates the agent best score for moving to a specific direction (north, south, east or west) using the agent specific rule-set.
4. The agent send his score and the direction to move to the CDM
5. The CDM decide which agent move using the agent's scores and tell the agent to move.
6. The agent tries to makes his action. If the agent can't move because it's the end of the game board or water in his direction or a friendly agent blocks him, then he sends an error back to the CDM. The CDM give command to the second best scoring agent to move and so on until an agent move.

Note: If the agent best score is for staying. Then the agent doesn't make step 4, because it's not necessary for the agent to move or to take an action. This could happen when an agent is scared to move or when he is making a plan.

This cycle is cycles on every turn of the computer player. Also the agent score and the agent view reset on every turn. This way the agent checks his environment on every turn and updates his information of the board.

The agent's lifecycle can be viewed as a number of states and transitions. In Figure 6.3 an automaton is drawn with its states and transitions. The most important state in the automaton is the Evaluate state. Here, the Rete algorithm is applied using the percepts that have been received. If the evaluation leads to an action, it will cause a transition to the Sleep state. Currently the action that has been implemented is sending a move request to the CDM, waiting for an answer.

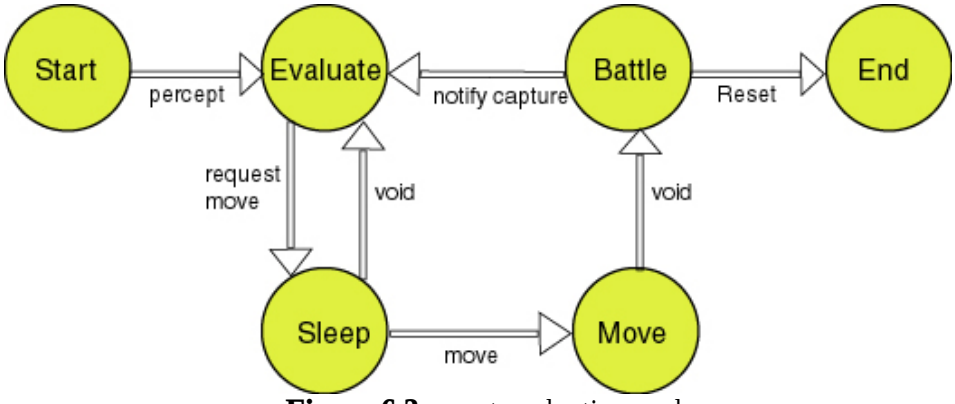


Figure 6.3: agent evaluating cycle

In the Move state a piece can do an actual move. From the move state there are two possible transitions to other states. When a move to an empty square was done the agent perceives some changes in its environment and evaluates them. The other possibility is a battle with an enemy piece. In the Battle state the agent either wins and notifies the capture or the agent looses and removed from the board.

6.3 Agent neighbours

As we mentioned before the board was designed 2 dimensional. And that the board have an X, Y-axis. One of the most important aspects in this project is the agent sensors. Our agents here use the board 2 dimensional characteristic to view their neighbours.

Every agent on the board has a unique location number like [2][3]. That agent is located at x =2 and y = 3. This makes his north neighbours location x = 2 and y = 3 –1. So we have a formula here:

Agent current location = ACL
Agent current X location = AXL
Agent current Y location = AYL

Neighbours North Agent = NNA

NNA = [AXL][AYL – 1]

And of course for the SNA (South)

SNA = [AXL][AYL +1]

This formula is implemented in every agent to view his neighbours. It also used for all other directions east, west, northeast, northwest, southeast etc….

Chapter 7

Testing the game

In this chapter we will test the Stratego game. Here we will test the game play, the agent view, the communications between, the agents, the agents and the rule base engine (Jess), the agents and the Central decision maker (CDM) and test if the high ranked view agents can think and make a plan and finally we will play the game and comment on the action taken by the system.

7.1 Game play

Because there isn't a running version available of the game that uses agents for the simulation. Is one of the main aspects of this project is to implement a running version of the Stratego game.

How are we going to test the game play? Simple by starting the game and play the game for a couple of times until we notice that the game don't have any errors who can stop the game playing or makes the game hangs. Also checking if there is some kind of information showed that the actual status is or who is won or lost an agent.

7.1.1 Game play test results

Our first notice is that the game doesn't have any Null pointers expectations. The following errors may appear in the information area:

If the error "Cannot be placed here" appear, then the user tried to place a tile or an agent in a wrong place. This could be: trying to place an agent in the water, or placing the agent on the same square where a friendly agent is placed.

If the error "ERROR: None of the agents can move" appears this mean that the agents can't move. This happened when for example all agent best score is for moving forward and there is water or another agent from the same colour is in front so the agent can't move.

A good thing about the information field is that you always can see who have the turn by the massages “Computer Turn” and “Human Turn”. Also when there is a battle between two agents the end result of the battle appears in the information field e.g. “Major Win From Scout”.

7.2 The agent view

Every agent has a limited amount of squares to view. Some have a low 4 squares view and other has high 12 squares. We will test only the high view agents, because we consider that if the 12 squares work then the low view of 4 also works.

We will place only one high view agent on the board and we are going to place enemy agents around him to check if the agent really sees the enemy.

7.2.1 Agent view test results

We placed a marshal in the middle of the game board. Then we placed other enemy agents around see figure 7.1

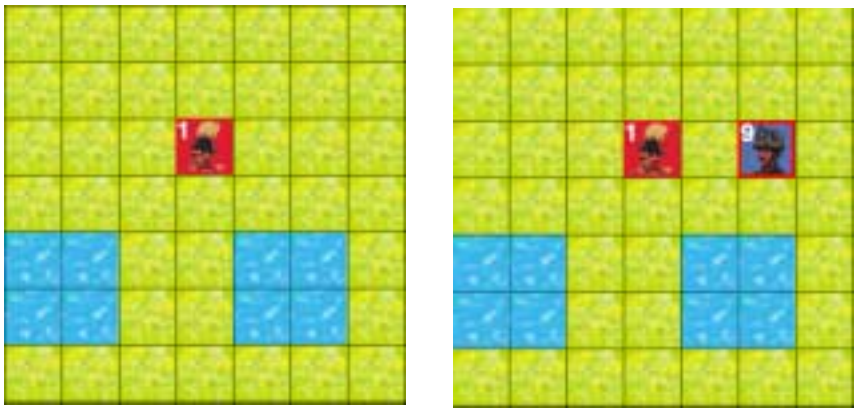


Figure 7.1: placing a marshal and a scout nearby.

Now the agent says:
RUN.Marshal[2][5] See from the West West Unknown enemy

Because the scout didn't go in a battle yet, his rank is not revealed. This mean the scout is still UNKNOWN. And as we see that our marshal saw the scout from the west west side of him.

Now we will test ALL the 12 squares at once see figure 7.2.

Figure 7.2: all 12 squares at once.

The Marshal says now:

RUN.Marshal[2][5] See from the South marshal
RUN.Marshal[2][5] See from the South South sergeant
RUN.Marshal[2][5] See from the East major
RUN.Marshal[2][5] See from the East East scout
RUN.Marshal[2][5] See from the North North captain
RUN.Marshal[2][5] See from the North colonel
RUN.Marshal[2][5] See from the West general
RUN.Marshal[2][5] See from the West West captain
RUN.Marshal[2][5] See from the North East spy
RUN.Marshal[2][5] See from the North West scout
RUN.Marshal[2][5] See from the South East minor
RUN.Marshal[2][5] See from the South West lieutenant

Now we made all agents rank known so we can check if the agent sees the enemy correctly. The test results show that the view of the agent is correct.

7.3 The communications between the agents

The communications between agents happened when an agent wins or loses a battle. The agent notifies the capture a bomb or a marshal. This is important because some of the agent's rule set is based on the information of the capture of the marshal or the bombs.

We will test this by capturing all the human bombs and the marshal and see if the agents get this information.
After capturing all the enemy bombs the agent's rule set now fire with:

f-51 (MAIN::enemy-bombs-captured)

And also in the output we notice:
enemybombs: 0, (total enemy bombs on board)

if the human player marshal is captured (it can only be captured by the spy when attacking or when the marshal lost from a bomb or a draw between the marshal's) then the rule set of the agent fire with:
f-52 (MAIN::enemy-marshal-captured)

The test result shows that the communication is correct.

7.4 The communication between the agents and Jess

Every kind of agent has his own rule set. And the communication between the rule-base engine and the agent must be correctly done. Otherwise the rule set wont fire all the rules or all the agent preferences.

The agents have to tell the rule base engine:

- The agent type.
- The agent status.
- The agent view.

7.4.1 Jess test results

When the rule-base engine fires a fact he logs it with f.

RUN.Colonel[4][8] Thats ME and here I am
RUN.Colonel[4][8] See from the North minor
RUN.Colonel[4][8] i-have-moved

RUN.Colonel[4][8] my-rank-revealed
f-0 (MAIN::initial-fact)
f-1 (MAIN::distance north 1)
f-2 (MAIN::distance west 1)
f-3 (MAIN::distance south 1)
f-4 (MAIN::distance east 1)
f-5 (MAIN::distance north-north 2)
f-6 (MAIN::distance north-west 2)
f-7 (MAIN::distance west-west 2)
f-8 (MAIN::distance south-west 2)
f-9 (MAIN::distance south-south 2)
f-10 (MAIN::distance south-east 2)
f-11 (MAIN::distance east-east 2)
f-12 (MAIN::distance north-east 2)
f-13 (MAIN::flee-score north -200 50 200 50)
f-14 (MAIN::flee-score west 50 -200 50 200)
f-15 (MAIN::flee-score south 200 50 -200 50)
f-16 (MAIN::flee-score east 50 200 50 -200)
f-17 (MAIN::flee-score north-north -100 50 100 50)
f-18 (MAIN::flee-score north-west -75 -75 75 75)
f-19 (MAIN::flee-score west-west 50 -100 50 100)
f-20 (MAIN::flee-score south-west 75 -75 -75 75)
f-21 (MAIN::flee-score south-south 100 50 -100 50)
f-22 (MAIN::flee-score south-east 75 75 -75 -75)
f-23 (MAIN::flee-score east-east 50 100 50 -100)
f-24 (MAIN::flee-score north-east -75 75 75 -75)
f-25 (MAIN::attack-score north 200 -50 -200 -50)
f-26 (MAIN::attack-score west -50 200 -50 -200)
f-27 (MAIN::attack-score south -200 -50 200 -50)
f-28 (MAIN::attack-score east -50 -200 -50 200)
f-29 (MAIN::attack-score north-north 100 -50 -100 -50)
f-30 (MAIN::attack-score north-west 75 75 -75 -75)
f-31 (MAIN::attack-score west-west -50 100 -50 -100)
f-32 (MAIN::attack-score south-west -75 75 75 -75)
f-33 (MAIN::attack-score south-south -100 -50 100 -50)
f-34 (MAIN::attack-score south-east -75 -75 75 75)
f-35 (MAIN::attack-score east-east -50 -100 -50 100)
f-36 (MAIN::attack-score north-east 75 -75 -75 75)
f-37 (MAIN::avoid-score north -1000 0 0 0)
f-38 (MAIN::avoid-score west 0 -1000 0 0)
f-39 (MAIN::avoid-score south 0 0 -1000 0)
f-40 (MAIN::avoid-score east 0 0 0 -1000)
f-41 (MAIN::stay-score 250)
f-42 (MAIN::wander-score 50 25 10 25)
f-43 (MAIN::attack-bomb-score north 1000 0 0 0)
f-44 (MAIN::attack-bomb-score west 0 1000 0 0)

f-45 (MAIN::attack-bomb-score south 0 0 1000 0)
f-46 (MAIN::attack-bomb-score east 0 0 0 1000)
f-47 (MAIN::attack-marshal-score north 10000 0 0 0)
f-48 (MAIN::attack-marshal-score west 0 10000 0 0)
f-49 (MAIN::attack-marshal-score south 0 0 10000 0)
f-50 (MAIN::attack-marshal-score east 0 0 0 10000)
f-51 (MAIN::i-have-moved)
f-52 (MAIN::my-rank-revealed)
f-53 (MAIN::enemy-lower-rank north)
f-54 (MAIN::enemy-marshal-captured)
f-55 (MAIN::enemy-bombs-captured)
f-56 (MAIN::attack north)
f-57 (MAIN::update-scores 0 200 -50 -200 -50)
f-58 (MAIN::wander)
f-59 (MAIN::update-scores 0 50 25 10 25)
For a total of 60 facts.
Score for staying: 0
Score for moving forward: 250
Score for moving left: -25
Score for moving backward: -190
Score for moving right: -25
My best score is = 250 and this is for moving Forward

The colonel first says what he is and what is his type and his view. And then he communicates with Jess

In f-51 and f-52 we see that the communication is successfully done. The agent told Jess his status, which is “i-have-moved” and “my-rank-revealed”.
f-53 is the agent view which is a lower-rank enemy agent from the north.
Also in f-54 and f-55 the agent tell Jess about what he received from other agents.

In f-56 Jess decides to attack north. This mean the agent best score is for attacking north (moving north).

At the end we see that the rule-base engine had fired 60 facts.

Here is the communication from Jess to the agent:
Score for staying: 0
Score for moving forward: 250
Score for moving left: -25
Score for moving backward: -190
Score for moving right: -25

And here is where the agent decides which direction is moving based on the best score:

My best score is = 250 and this is for moving Forward.

After checking those logs of the agent and Jess we can say that the communication is successfully done. And our agent can make a decision based on his own rule-set.

7.5 Testing the agents and the CDM communication

The communication between the agents and the Central Decision Maker (CDM) is done when all the agents compute their best score. After this is done the CDM use an ordering algorithm to put the agent with the best score above of the list. This is done by using the Java function Comparator.

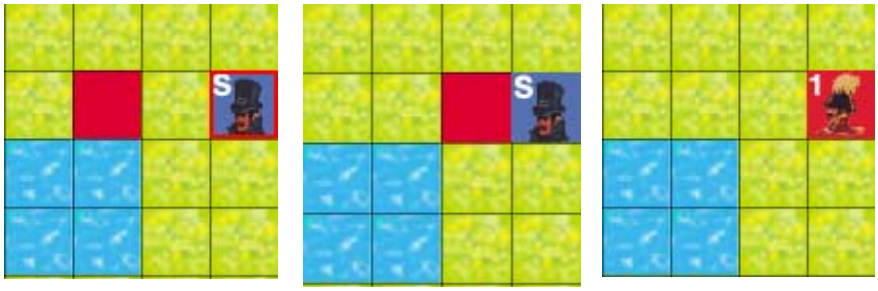
If the agent best score is for staying he doesn't communicate with the CDM, because he is not moving anywhere. Also if the agent best score is for moving forward and in the square in front of him is water, then he skips the communication with CDM.

When this is done the CDM tells the agent with the highest score to move. If he can't move e.g. there is a friendly agent in front. He communicates back to the CDM and sending the massage "I can't move". The CDM takes the second best scoring agent and tell him to move. See below for an example output.

Total Agents = 26
Best Scoring Agent = Colonel[4][8] with a score of 250 for moving Forward
Im moving Forward now
0=250
1=50
2=50
3=50
4=50
5=50
6=50
7=50
8=50
Etc…

7.6 Making a plan

Only the high view agents can make a plan, because they see more. We will again place a marshal and put an enemy two squares distance from marshal.



RUN.Marshal[3][7] See from the West West spy
RUN.Marshal[3][7] i-have-moved = false
RUN.Marshal[3][7] my-rank-revealed = false
My best score is = 125 and this is for moving Left

RUN.Marshal[3][8] Thats ME and here i am
RUN.Marshal[3][8] See from the West spy
RUN.Marshal[3][8] i-have-moved
RUN.Marshal[3][8] my-rank-revealed = false
My best score is = 225 and this is for moving Left

The marshal decided to move toward the spy and capture him.
Even if we put the Spy in a diagonal direction like north-east the marshal thinks 2 steps forward. He moves left and then he moves north to capture the spy.
This means that the high ranked agent's can think 2 steps forward using their ability of a 12 squares view.

7.7 Playing the game.

We started the game and choosed using the `Setup's` menu for the "Wheel Of Danger" starting setup. Figure 7.3

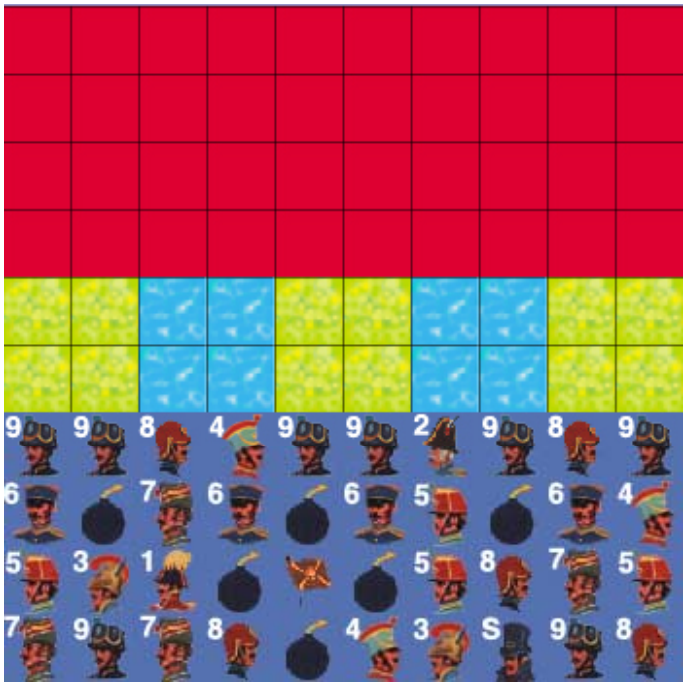


Figure 7.3: Wheel of Danger staring setup.

We started the game using the "Start" button. A die is thrown (figure 7.4) and the computer scored more then the us, therefore he have the turn to play first.

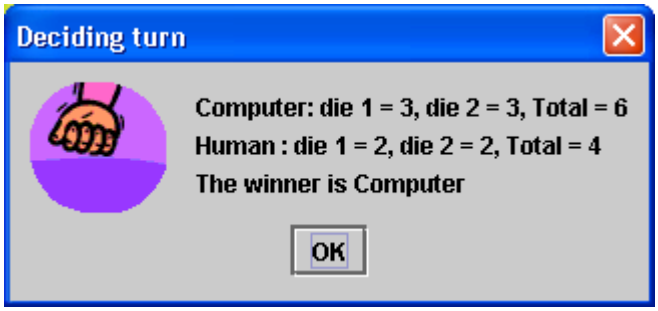


Figure 7.4: Deciding the turn with a die.

The first thing we notice while playing the game is that because of the random choice of the highest scoring agents, the agents in the back also move and therefore the agent in the front are slow with the attack (figure 7.5)

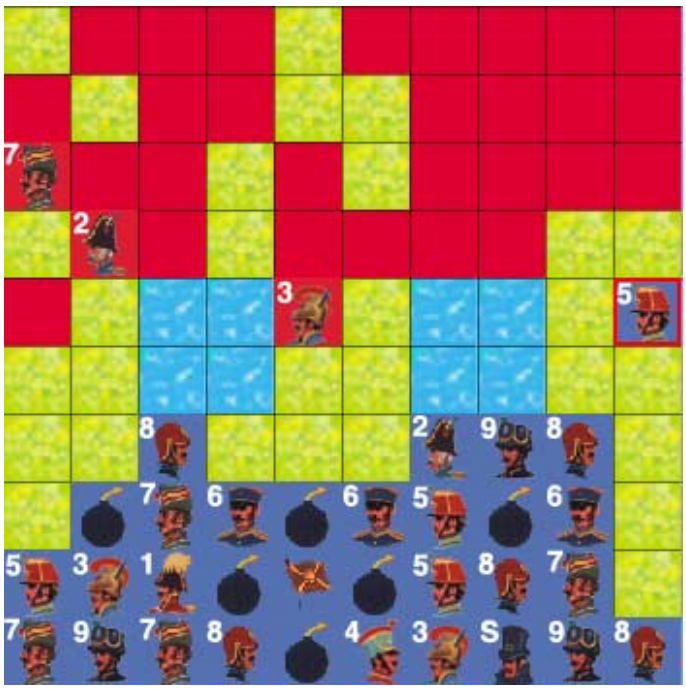


Figure 7.5: The agents at the back are moving and having a turn and slowing the front agents to move.

The general in figure 7.5 kept moving forward. And then he stopped when he reached the bomb (the bomb had a battle before and therefore its status is known). He is computing the following scores:

Score for staying: 250
Score for moving forward: 50
Score for moving left: 25
Score for moving backward: 10
Score for moving right: 25

We moved an unknown and a lower ranked enemy around the same general (figure 7.4).

Score for staying: 250
Score for moving forward: 0
Score for moving left: 225
Score for moving backward: -40
Score for moving right: -175

Figure 7.4: The general still deciding to stay, notice the score for moving forward!!

We decided to play on with the game and wait to see what the general is going to do when he is forced to play. While playing we noticed another interesting point. The known agent are avoiding our known general (figure 7.5)

Figure 7.5: The sergeant is fleeing from our general.

The computer marshal is very cautious if there is unknown enemy in his sight he never move even if he already moved and his status is known. But when that enemy is at an attacking distance he attacks him because it may be a spy.

A lieutenant agent reached the bottom of the game board. It's interesting to see what he is going to do now. We found out that he keeps attacking the enemies until someone stopped him.

The board situation is now as figure 7.6.

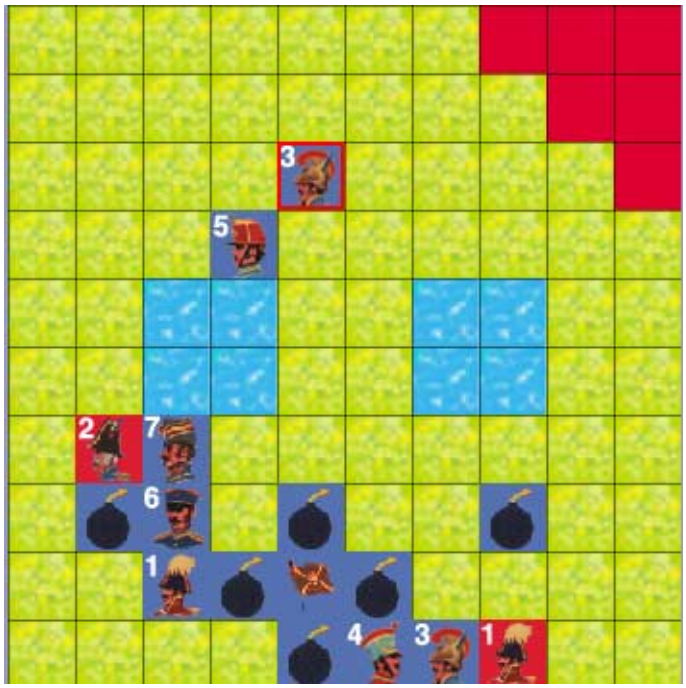


Figure 7.6: The board situation

Because both the red general and the red marshal have a score for staying with 250 and their best second score is 225 for attacking the system picked a random one to move. The marshal is assigned to move. He attacked the blue colonel and major. And then he went left avoiding the bomb and all the way to the left-end of the board. He kept moving left, right, left, right leaving the general waiting. This is because of the mechanism that the agent with a highest score of staying doesn't communicate with the CDM except when there aren't any more agents to move, then he is forced to pick his second highest score.

We decided to suicide the sergeant and the lieutenant by attacking the red general, to kill the general with the blue marshal. And then we attacked the red marshal with the blue marshal to make it a draw. The computer lost the game now because he doesn't have anymore pieces to move (figure 7.7).

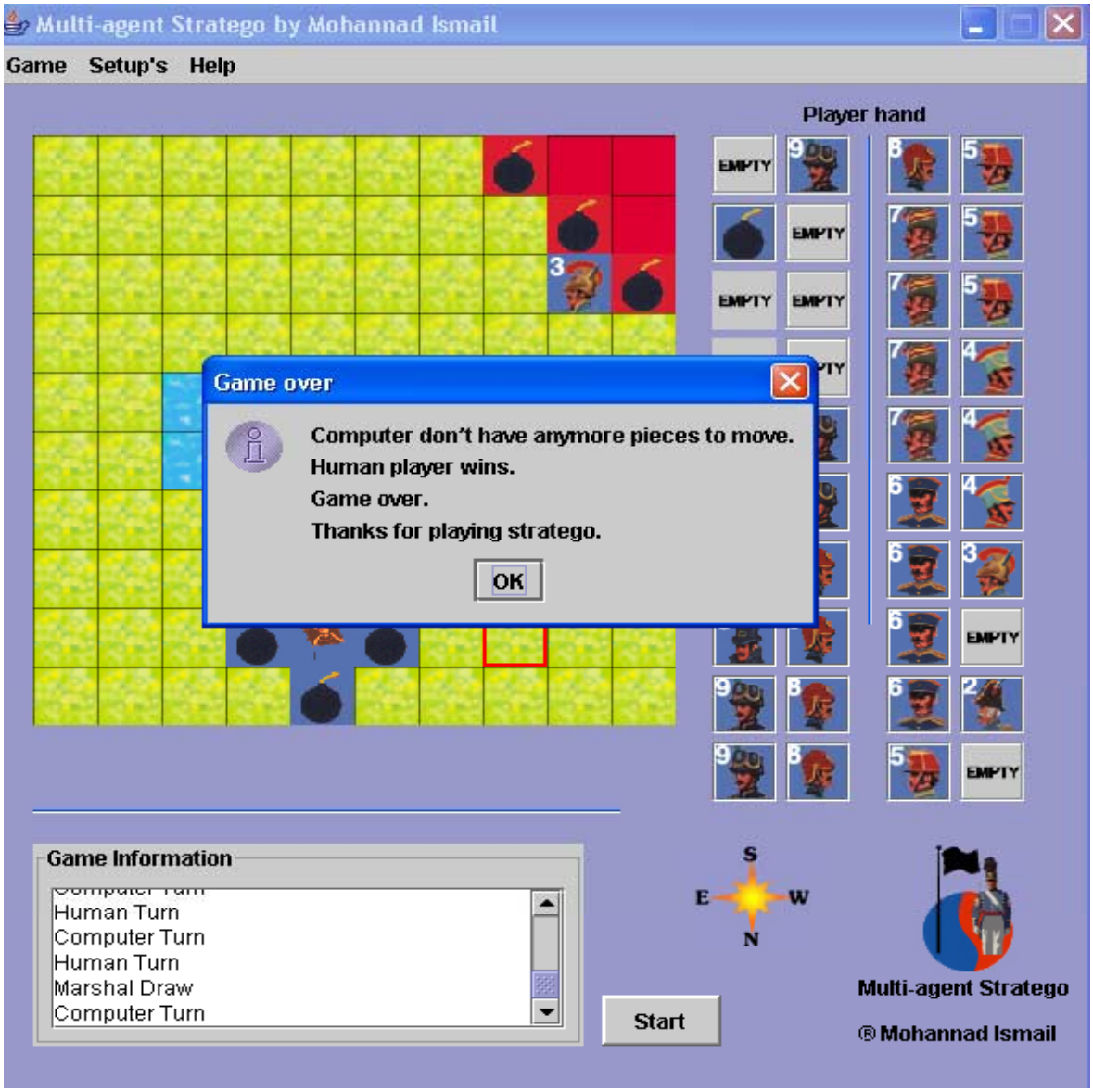


Figure 7.7: Computer losing the game.

After testing the game we can say that the prototype is successfully implemented.

Chapter 8

Manual

In this chapter we will describe how to use the game for the human player (the normal user or gamer) and finally a small manual about CLIPS for the programmers who wants to improve CLIPS.

8.1 User manual

The game needs only mouse clicks to be played.

Starting the game: Double click on Stratego.exe and the game will start.

Setting the pieces: One click on the desired piece to set on the board. And another click on an empty square on the board to set the piece. Do this until all the 40 pieces are placed on the board. *Note: it's not possible to move a piece after it's already set on the board, this will make the computer play.*

Moving the pieces: One click on the desired piece to move. Choose a square up, down, left or right from your piece and click it to move your piece. *Note: it's possible to move the piece more then 1 square. But this is only done for testing and trying to make better CLIPS for the agents.*

Restarting the game: The game can be restarted by clicking on the restart button in Game – Restart Game.

Exiting the game: The game can be quitted anytime by clicking the exit game in Game – Exit.

About the game: Information about the author and the game versioning will be showed by clicking the button Help – About.

8.2 CLIPS manual

The CLIPS are the brain of the agents. Every agent has his own rule-set. The CLIPS are case sensitive and therefore it asks extra attention when changing them.

As mention before every agent have his own rule-set. But there is one rule-set which is used for all agents which is the “agent.clp”. This file contains several general definitions that apply to all ranks. The code contains a `defclass` that is an external address, which makes exchange of data between Jess and Java possible.

The actions that can be used for the agents are:

- flee
- attack
- attack-marshal
- attack-bomb
- stay
- avoid
- wander

Upon asserting an action, the defclass (a Java-bean) will be updated by asserting the corresponding score.

```
(defclass scores agents.Scores)
```

In the Java source, the bean is passed to Jess as follows:

```
r.store("SCORES", new Scores());
r.executeCommand("(bind ?s (fetch SCORES))");
```

Following we can use the following jess commands:

```
(call ?s getCenter)
(call ?s setCenter value)
```

and as getters and setters for the bean, where in this case the Scores bean has a property `center'
Following will make sure that the globals will keep their values upon issuing a `reset'.

```
(set-reset-globals nil)
```

`deffacts` mean that this is a fact e.g.:

```
(deffacts distances
   (distance north 1)
)
```

Above define the fact distances. And that fact that can be fired is north.

Defrule defines a new rule for the agents. See appendix for the complete agent.clp file.

Agents CLIPS
We will take the CLIP of the miner as an example for our explanation.

Following makes use of the action available in agent.clp
`(batch agents/agent.clp)`

These are the specific globals for the miner upon change, call the assert-globals function to assert the new corresponding facts:

```
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-bombs-captured* = false)
```

```
# abstract rule 1
(defrule enemy-unknown-1
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)
```

If the enemy bombs are captured and I have moved and my rank is revealed and there is an unknown enemy with an attacking distance of 1 then flee.

Above defined the name of the rule by defrule enemy-unknown-1.
And then we give the rule the facts. `distance ?enemy 1` checks in what direction the enemy is located. The => means take the action which is flee in our situation.

It is also possible to use “not” like (not (my-rank-revealed)) for the facts.

Chapter 9

Conclusion

In this investigation we have examined a multi-agent approach for playing the game Stratego. This approach involves playing the Stratego game with multiple agents that each represents a piece in the Stratego army. The approach was based on the hypothesis that for some complex problems distributed techniques for solving them can result in more intuitive solutions.

We assumed that the Stratego game could serve as an excellent playground for testing in hypothesis. Typically, the game brings about a high complexity that makes an analysis of it rather difficult. This is due to the fact that both players have incomplete information about the board status. Furthermore, the game's characteristics lend their selves well for the distributed techniques that we proposed to use. Upon modelling the agents we can make use of the fact that the Stratego army consists of military men that each have their own dedicated role within the army. In other words, up to some level the agents can act as individuals where each of them has their own set of characteristics.

9.1 Evaluation

At the start of the investigation a literature survey has been done about the agent and multi-agent concepts. Firstly, this has led to insights about the level of rationality, autonomy and intelligence of agents. Additionally, studying the literature about the field of multi-agent systems gave us an understanding of different types of multi-agent systems and their uses, which allows us to fit in our approach with the field.

Unfortunately there have not been any studies or analyses about the Stratego game. We have written down some general considerations about the game's characteristics, but a thorough analysis is not a purpose of the investigation. In an attempt to acknowledge the complexity of game playing in general, we have

studied some papers on game playing theories. These theories apply to perfect information games. Theories about games with information however are sparse, and where available they are very complicated. This leads us to believe that an intuitive solution for finding a way to play Stratego is to be found using multiple agents, where we avoid reasoning about board positions seen from a central point of view.

One of the things that the literature survey has resulted in, is an inspiration for the agent's model. We have modelled the agent as a system that can exhibit behavior patterns which result from separate internal behaviors. These internal behaviors may inhibit each other, harmonize or even con with each other. This way of modeling the agent has some important advantages.

First of all, this way of modelling systems comes closer to explaining human intelligence than the traditional way of modelling intelligence. Secondly, a pragmatic advantage of this type of architecture is that the agent's behavior can be extended easily. By adding new internal behaviors, the agent expresses a new overall behavior.

We have chosen to use rule-based approach for specifying the agent's behavior. It turns out that for a Stratego agent we can specify a lot of rules which result from the characteristics of the game. We can easily define a number of goals for the agents such as staying alive, capture enemy pieces, etc.

Our design reflects the desire to build a generic tool for the investigation. The basic functionality of the design has been implemented successfully. The prototype currently allows for playing Stratego games locally. The most important part of Stratego is the implementation of a framework for the agents in which they have:

- Visual perception
- Evaluation of its environment
- Effectors

Note that every agent has its own characteristics. For each of the ranks we have implemented specific rule-sets. The behavior of the agent results from the specific rule-set that it uses according to its rank, its perception and its own status.

We have tested Stratego by letting the agents play against a human player. The experiments have resulted in some valuable ideas about our multi-agent approach. First of all, the results of our experiments support our hypothesis, which stated that for some complex problems distributed techniques have a more intuitive solution. We have shown that playing the game with multiple agents is an excellent approach to break down complexity of the game.

9.2 Future work

The future work that needs to be done to continue the investigation can be divided into several categories. First of all, several improvements to the framework can be made. As stated before, we have implemented the basic functionality of the design. An important part that still has to be done is adding extra functionality to the agents by specifying a communication scheme with which they can exchange information. Additionally, the communication should be a means for cooperation between the agents.

One of the aspects of the agents that can be extended is the rule-sets. First of all we want to have more functionality in the rule-sets. Extra rules are needed for communication and cooperation.

Using the rule-sets that we have implemented the agents do not know how to play the end game. This requires specific knowledge about how to play it. The rule-set that we have implemented is not enough for the end game, therefore extra rules have to be used. Filling in the message-passing mechanism will be an important demand for the extra rules. We can make specific rules for the opening, mid game and end game.

Bibliography

Multi-Agent Stratego
C. Treijtel's Master Thesis report
http://www.kbs.twi.tudelft.nl/Publications/MSc/2000-Treijtel-MSc.html

Sun Java Standard Edition Documentation
http://java.sun.com/docs/

Ed's Stratego Site
http://www.edcollins.com/stratego

Jess Java Expert System Shell
http://herzberg.ca.sandia.gov/jess/

Homepage of Multi-agent Stratego
http://stratego.tigris.org/

Jess-users mailing list
http://www.mail-archive.com/jess-users@sandia.gov/

Home of the original free and open source java IDE.
http://www.netbeans.org

Jshrink: Java Shrinker and Obfuscator
http://www.e-t.com/jshrink.html

Rational software, tool for UML design
http://www.rational.com

Books:

Artificial Intelligence: A Modern Approach
ISBN 0-13-080302-2

Data structures in Java
ISBN 0-201-30564-X

Praktisch UML 2de editie
ISBN 90-430-0494-4

Agent Engineering
ISBN 981-02-4558-0

AI Game Programming Wisdom
ISBN 1-58450-077-8

Multi-Agent Systems and Applications
ISBN 3-540-42312-5

Java How to Program Third Edition
ISBN 0-13-012507-5

APPENDIX A

The agent's source code

```
; agent.clp
; the general rule-engine of an agent
;
; This file contains several general definitions
; that apply to all ranks. The code contains a
; defclass that is an external address, which
; makes exchange of data between Jess and Java
; possible.
;
; The actions that can be used are:
;   flee
;   attack
;   attack-marshal
;   attack-bomb
;   stay
;   avoid
;   wander
;
; Upon asserting an action, the defclass (a Java-bean)
; will be updated by asserting the corresponding
; score.
(defclass scores agents.Scores)

; In the java source, the bean is passed to Jess as follows:
;
;       r.store("SCORES", new Scores());
;       r.executeCommand("(bind ?s (fetch SCORES))");
;
; Following we can use the following jess commands:
;       (call ?s getCenter)
;       (call ?s setCenter value)  and
; as getters and setters for the bean, where in this case
; the Scores bean has a property `center'


; This will make sure that the globals will keep their
; values upon issuing a `reset'
(set-reset-globals nil)

(deffacts distances
   (distance north 1)
   (distance west 1)
   (distance south 1)
   (distance east 1)
   (distance north-north 2)
   (distance north-west 2)
   (distance west-west 2)
   (distance south-west 2)
   (distance south-south 2)
   (distance south-east 2)
   (distance east-east 2)
   (distance north-east 2)
)
```

```
(deffacts flee-scores
   (flee-score north -200 50 200 50)
   (flee-score west  50 -200 50 200)
   (flee-score south 200 50 -200 50)
   (flee-score east  50 200 50 -200)
   (flee-score north-north -100 50 100 50)
   (flee-score north-west  -75 -75 75 75)
   (flee-score west-west   50 -100 50 100)
   (flee-score south-west  75 -75 -75 75)
   (flee-score south-south 100 50 -100 50)
   (flee-score south-east  75 75 -75 -75)
   (flee-score east-east   50 100 50 -100)
   (flee-score north-east  -75 75 75 -75)
)

(deffacts attack-scores
   (attack-score north 200 -50 -200 -50)
   (attack-score west  -50 200 -50 -200)
   (attack-score south -200 -50 200 -50)
   (attack-score east  -50 -200 -50 200)
   (attack-score north-north 100 -50 -100 -50)
   (attack-score north-west  75 75 -75 -75)
   (attack-score west-west   -50 100 -50 -100)
   (attack-score south-west  -75 75 75 -75)
   (attack-score south-south -100 -50 100 -50)
   (attack-score south-east  -75 -75 75 75)
   (attack-score east-east   -50 -100 -50 100)
   (attack-score north-east  75 -75 -75 75)
)

(deffacts avoid-scores
   (avoid-score north -1000 0 0 0)
   (avoid-score west  0 -1000 0 0)
   (avoid-score south 0 0 -1000 0)
   (avoid-score east  0 0 0 -1000)
)

(deffacts misc-scores
   (stay-score 250)
   (wander-score 50 25 10 25)
)

(deffacts attack-bomb-scores
   (attack-bomb-score north 1000 0 0 0)
   (attack-bomb-score west  0 1000 0 0)
   (attack-bomb-score south 0 0 1000 0)
   (attack-bomb-score east  0 0 0 1000)
)

(deffacts attack-marshal-scores
   (attack-marshal-score north 10000 0 0 0)
   (attack-marshal-score west  0 10000 0 0)
   (attack-marshal-score south 0 0 10000 0)
   (attack-marshal-score east  0 0 0 10000)
)

(defrule action-attack
   (attack ?enemy)
   (attack-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
```

```
)

;for the minors
(defrule action-attack-bomb
   (attack-bomb ?enemy)
   (attack-bomb-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

;for the spy
(defrule action-attack-marshal
   (attack-marshal ?enemy)
   (attack-marshal-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-flee
   (flee ?enemy)
   (flee-score ?enemy ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-stay
   (stay)
   (stay-score ?center)
   =>
   (assert (update-scores ?center 0 0 0 0))
)

(defrule action-avoid
   (avoid ?enemy)
   (avoid-score ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)

(defrule action-wander
   (wander)
   (wander-score ?forward ?left ?backward ?right)
   =>
   (assert (update-scores 0 ?forward ?left ?backward ?right))
)


(defrule update-scores
   (update-scores ?c ?f ?l ?b ?r)
   =>
   (call ?s setCenter (+ (call ?s getCenter) ?c))
   (call ?s setForward (+ (call ?s getForward) ?f))
   (call ?s setLeft (+ (call ?s getLeft) ?l))
   (call ?s setBackward (+ (call ?s getBackward) ?b))
   (call ?s setRight (+ (call ?s getRight) ?r))
)
```

```
; spy.clp
; the rule-engine of the spy
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the spy
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*enemy-marshal-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*enemy-marshal-captured* true) then
        (assert (enemy-marshal-captured))
    )
)

;these are the specific rules for the spy
(defrule enemy-unknown-1
   (enemy-unknown ?enemy)
   (i-have-moved)
   =>
   (assert (flee))
)

; abstract rule 6 (stay)
(defrule enemy-unknown-2
   (enemy-unknown ?enemy)
   (not (i-have-moved))
   =>
   (assert (stay)
   )
)

;hey the general is at distance 1-->attack him!
(defrule enemy-marshal-1
   (enemy-marshal ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack ?enemy))
)

;we haven't moved yet and the enemy general is at distance 2,
;the spy definately wants to stay!
(defrule enemy-marshal-2
   (not (i-have-moved))
   (enemy-marshal ?enemy)
   (distance ?enemy 2)
```

```
   =>
   (assert (stay))
)

;the general is at distance 2, the spy may want to stay foot
(defrule enemy-marshal-2
   (i-have-moved)
   (enemy-marshal ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;always watch out for the bomb!
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (avoid ?enemy))
)
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; scout.clp
; the rule-engine of the scout
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the spy
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the scout

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-unknown-2
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 9 and 10
(defrule enemy-unknown-3
   (not (my-rank-revealed))
   (not (i-have-moved))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 3 and 4
(defrule enemy-higher-rank-1
   (my-rank-revealed)
```

```
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rule 7
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (i-have-moved)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;abstract rule 8
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (i-have-moved)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;abstract rule 9 and 10
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (not (i-have-moved))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rule 13:always watch out for the bomb!
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 14: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; minor.clp
; the rule-engine of the miner
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the miner
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-bombs-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
    (if (eq ?*enemy-bombs-captured* true) then
        (assert (enemy-bombs-captured))
    )
)

;these are the specific rules for the miner

;abstract rule 1
(defrule enemy-unknown-1
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;abstract rule 2
(defrule enemy-unknown-2
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;abstract rules 5 and 6
(defrule enemy-unknown-3
   (enemy-bombs-captured)
```

```
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 9 and 10
(defrule enemy-unknown-4
   (enemy-bombs-captured)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)

;abstract rule 13
(defrule enemy-unknown-5
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;abstract rule 14
(defrule enemy-unknown-6
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;abstract rule 17
(defrule enemy-unknown-7
   (i-have-moved)
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;abstract rule 18
(defrule enemy-unknown-8
   (i-have-moved)
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)
```

```
;abstract rules 19 and 20
(defrule enemy-unknown-9
   (enemy-unknown ?enemy)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   =>
   (assert (stay))
)

;abstract rules 3 and 4
(defrule enemy-higher-rank-1
   (enemy-bombs-captured)
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rules 7 and 8
(defrule enemy-higher-rank-2
   (enemy-bombs-captured)
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rules 11 and 12
(defrule enemy-higher-rank-3
   (enemy-bombs-captured)
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rules 15 and 16
(defrule enemy-higher-rank-4
   (i-have-moved)
   (my-rank-revealed)
   (not (enemy-bombs-captured))
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rules 21 and 22
(defrule enemy-higher-rank-5
   (not (i-have-moved))
   (not (my-rank-revealed))
   (not (enemy-bombs-captured))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rule 23 and 24
```

```
(defrule enemy-lower-rank-1
   (enemy-lower-rank ?enemy)
   (not (my-rank-revealed))
   =>
   (assert (stay))
)

;abstract rule 25 and 26
(defrule enemy-lower-rank-2
   (enemy-lower-rank ?enemy)
   (my-rank-revealed)
   =>
   (assert (attack ?enemy))
)

;abstract rules 27: hey a bomb at distance 1!
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack-bomb ?enemy))
)

;abstract rule 28: hey a bomb at distance 2!
(defrule enemy-bomb-2
   (enemy-bomb ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rule 29: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)


;abstract rules 5 and 6
(defrule enemy-minor-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-minor ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 11 and 17
(defrule enemy-minor-2
   (not (my-rank-revealed))
   (enemy-minor ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)
```

```
; sergeant.clp
; the rule-engine of the sergeant
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the sergeant
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-bombs-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the sergeant

;abstract rule 1 and 2
(defrule enemy-unknown-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 9 and 10
(defrule enemy-unknown-2
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 23 and 24
(defrule enemy-unknown-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 3
```

```
(defrule enemy-higher-rank-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (flee ?enemy))
)

;abstract rule 4
(defrule enemy-higher-rank-2
   (i-have-moved)
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;abstract rules 9 and 10
(defrule enemy-higher-rank-3
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rules 15 and 16
(defrule enemy-higher-rank-4
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   =>
   (assert (stay))
)

;abstract rules 5 and 6
(defrule enemy-sergeant-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-sergeant ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 11 and 17
(defrule enemy-sergeant-2
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 12
(defrule enemy-sergeant-3
   (my-rank-revealed)
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 2)
```

```
   =>
   (assert (attack ?enemy))
)

;abstract rule 18
(defrule enemy-sergeant-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-sergeant ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)

;abstract rule 7 and 8
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 19 and 20: avoid the bomb
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 21: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; lieutenant.clp
; the rule-engine of the lieutenant
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the lieutenant
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the lieutenant

;abstract rules 1 and 2
(defrule enemy-unknown
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)


;abstract rules 3 and 4
(defrule enemy-higher-rank-1
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rule 9
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 10
(defrule enemy-higher-rank-3
```

```
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-lieutenant-1
   (my-rank-revealed)
   (enemy-lieutenant ?enemy)
   =>
   (assert (stay))
)

;abstract rule 15
(defrule enemy-lieutenant-2
   (not (my-rank-revealed))
   (enemy-lieutenant ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 16
(defrule enemy-lieutenant-3
   (not (my-rank-revealed))
   (enemy-lieutenant ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack))
)

;abstract rules 13 and 14: avoid the bomb
(defrule enemy-bomb-1
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 15: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; captain.clp
; the rule-engine of the captain
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the captain
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the captain

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (i-have-moved)
   (my-rank-revealed)
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)

;abstract rules 9 and 10
(defrule enemy-unknown-2
   (i-have-moved)
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 13 and 14
(defrule enemy-unknown-3
   (not (i-have-moved))
   (not (my-rank-revealed))
   (enemy-unknown ?enemy)
   =>
   (assert (stay))
)
```

```
;abstract rules 3 and 4
(defrule enemy-higher-rank-1
   (my-rank-revealed)
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rule 11
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 11
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)


;abstract rules 5 and 6
(defrule enemy-captain
   (enemy-captain ?enemy)
   =>
   (assert (stay))
)

;abstract rules 7 and 8
(defrule enemy-lower-rank-3
   (not (my-rank-revealed))
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)


;abstract rules 15 and 16: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 17: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)

;abstract rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
```

```
   (assert (attack ?enemy))
)
```

```
; major.clp
; the rule-engine of the major
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the major
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the major

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 7 and 8
(defrule enemy-higher-rank-1
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rule 9
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 10
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
```

```
   (assert (attack ?enemy))
)


;abstract rules 3 and 4
(defrule enemy-major
   (my-rank-revealed)
   (enemy-major ?enemy)
   =>
   (assert (stay))
)

;abstract rule 11
(defrule enemy-major
   (not (my-rank-revealed))
   (enemy-major ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 12
(defrule enemy-major
   (not (my-rank-revealed))
   (enemy-major ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 17: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; colonel.clp
; the rule-engine of the major
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the major
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the major

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 7 and 8
(defrule enemy-higher-rank-1
   (enemy-higher-rank ?enemy)
   =>
   (assert (flee ?enemy))
)

;abstract rule 9
(defrule enemy-higher-rank-2
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 10
(defrule enemy-higher-rank-3
   (not (my-rank-revealed))
   (enemy-higher-rank ?enemy)
   (distance ?enemy 2)
   =>
```

```
   (assert (attack ?enemy))
)


;abstract rules 3 and 4
(defrule enemy-colonel
   (my-rank-revealed)
   (enemy-colonel ?enemy)
   =>
   (assert (stay))
)

;abstract rule 11
(defrule enemy-colonel
   (not (my-rank-revealed))
   (enemy-colonel ?enemy)
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 12
(defrule enemy-colonel
   (not (my-rank-revealed))
   (enemy-major ?enemy)
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 17: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

```
; general.clp
; the rule-engine of the general
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the general
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-spy-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
    (if (eq ?*enemy-spy-captured* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the general

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (enemy-spy-captured)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 11
(defrule enemy-unknown-2
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack ?enemy))
)

;abstract rule 12-> beware of the spy
(defrule enemy-unknown-3
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)
```

```
;abstract rules 5 and 6
(defrule enemy-general-1
   (enemy-general ?enemy)
   (my-rank-revealed)
   =>
   (assert (attack ?enemy))
)

;abstract rule 9
(defrule enemy-general-2
   (enemy-general ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 10
(defrule enemy-general-2
   (enemy-general ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 15: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

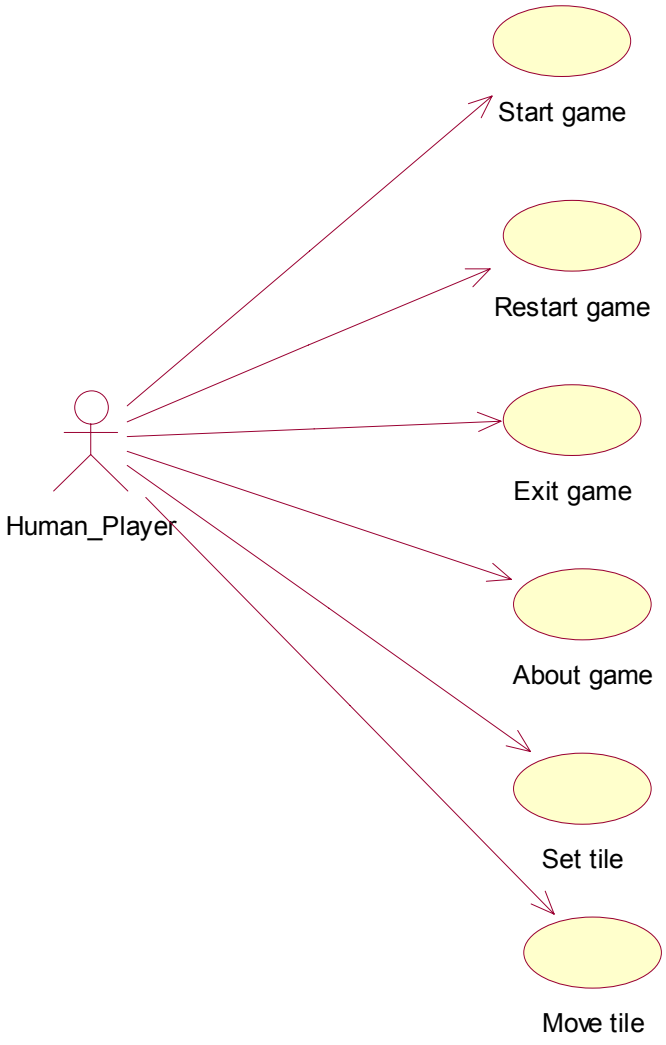
```
; marshal.clp
; the rule-engine of the general
;
; makes use of the action available in agent.clp
; available actions are:
;   flee
;   attack
;   attack-general
;   attack-bomb
;   stay
;   avoid
;   wander
;
(batch agents/agent.clp)

;these are the specific globals for the general
;upon change, call the assert-globals function
;to assert the new corresponding facts
(defglobal ?*i-have-moved* = false)
(defglobal ?*my-rank-revealed* = false)
(defglobal ?*enemy-spy-captured* = false)

(deffunction assert-globals()
    (if (eq ?*i-have-moved* true) then
        (assert (i-have-moved))
    )
    (if (eq ?*my-rank-revealed* true) then
        (assert (my-rank-revealed))
    )
    (if (eq ?*enemy-spy-captured* true) then
        (assert (my-rank-revealed))
    )
)

;these are the specific rules for the general

;abstract rules 1 and 2
(defrule enemy-unknown-1
   (enemy-spy-captured)
   (enemy-unknown ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rule 11
(defrule enemy-unknown-2
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 1)
   =>
   (assert (attack ?enemy))
)

;abstract rule 12-> beware of the spy
(defrule enemy-unknown-3
   (not (enemy-spy-captured))
   (enemy-unknown ?enemy)
   (distance ?enemy 2)
   =>
   (assert (stay))
)
```

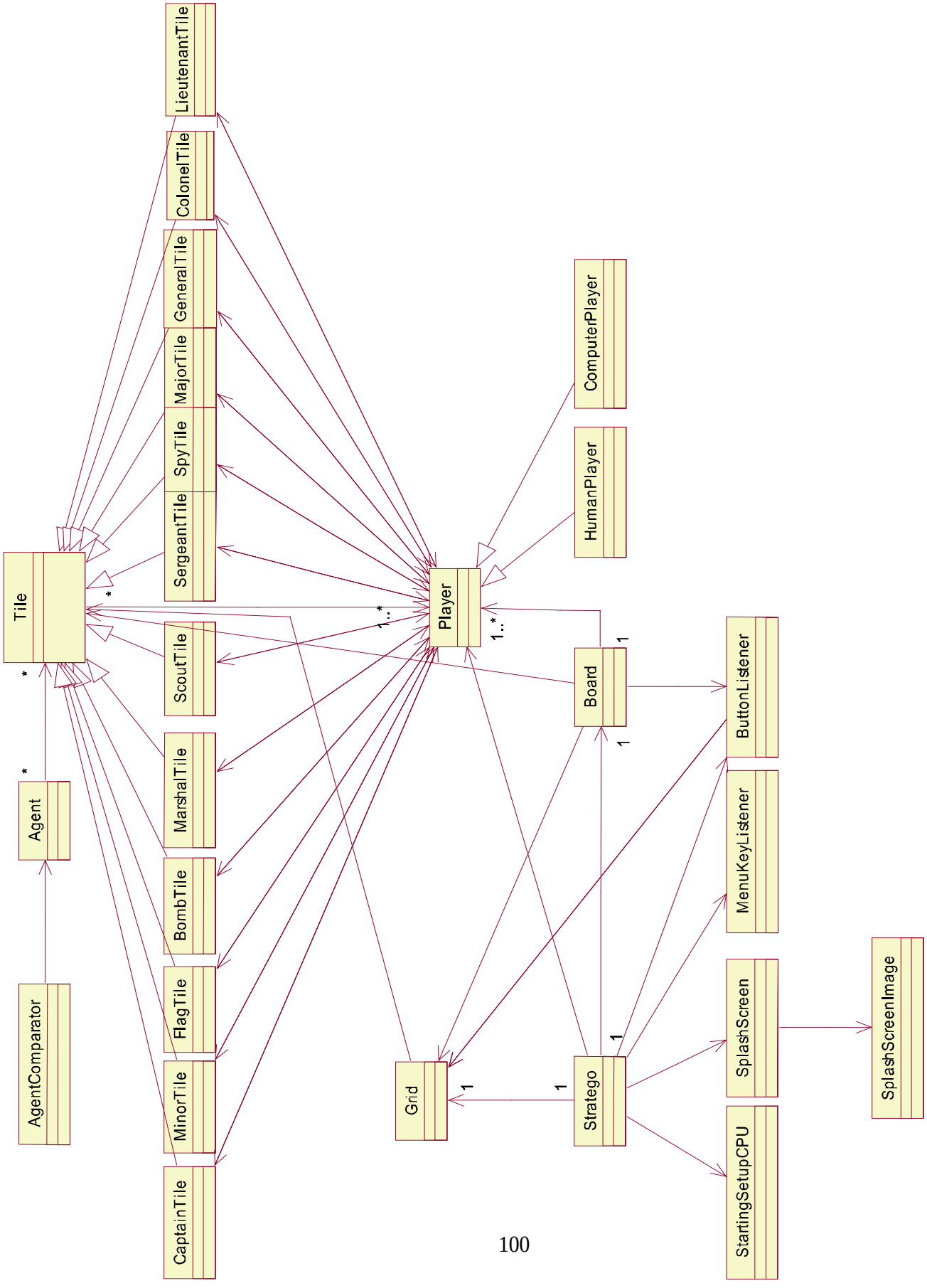
```
;abstract rules 5 and 6
(defrule enemy-marshal-1
   (enemy-marshal ?enemy)
   (my-rank-revealed)
   =>
   (assert (attack ?enemy))
)

;abstract rule 9
(defrule enemy-marshal-2
   (enemy-marshal ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 1)
   =>
   (assert (stay))
)

;abstract rule 10
(defrule enemy-marshal-2
   (enemy-marshal ?enemy)
   (not (my-rank-revealed))
   (distance ?enemy 2)
   =>
   (assert (attack ?enemy))
)

;abstract rules 5 and 6
(defrule enemy-lower-rank
   (enemy-lower-rank ?enemy)
   =>
   (assert (attack ?enemy))
)

;abstract rules 13 and 14: avoid the bomb
(defrule enemy-bomb
   (enemy-bomb ?enemy)
   =>
   (assert (avoid ?enemy))
)

;abstract rule 15: wander behavior
(defrule wander
   (initial-fact)
   =>
   (assert (wander))
)
```

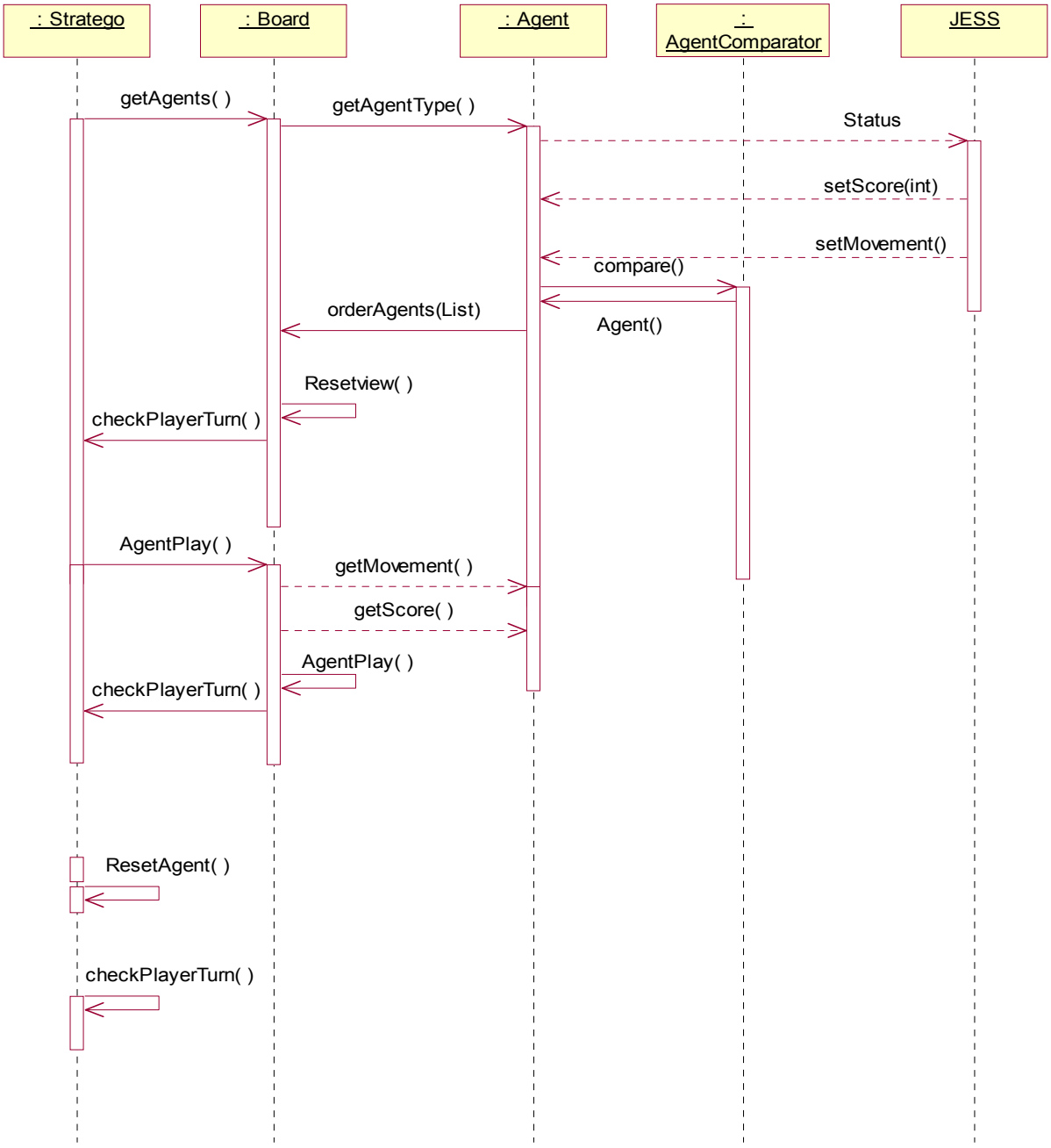
APPENDIX B

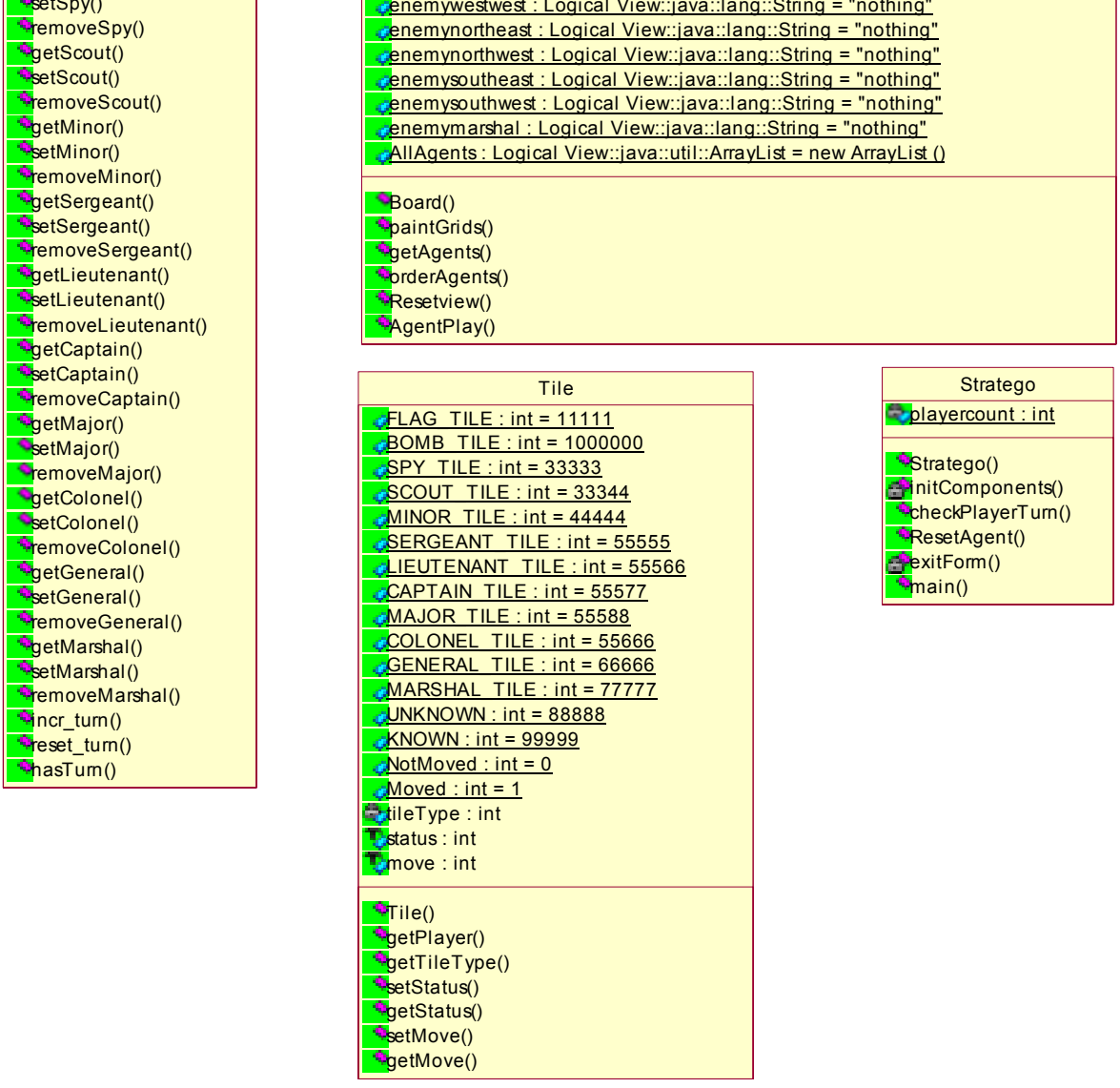
UML diagrams

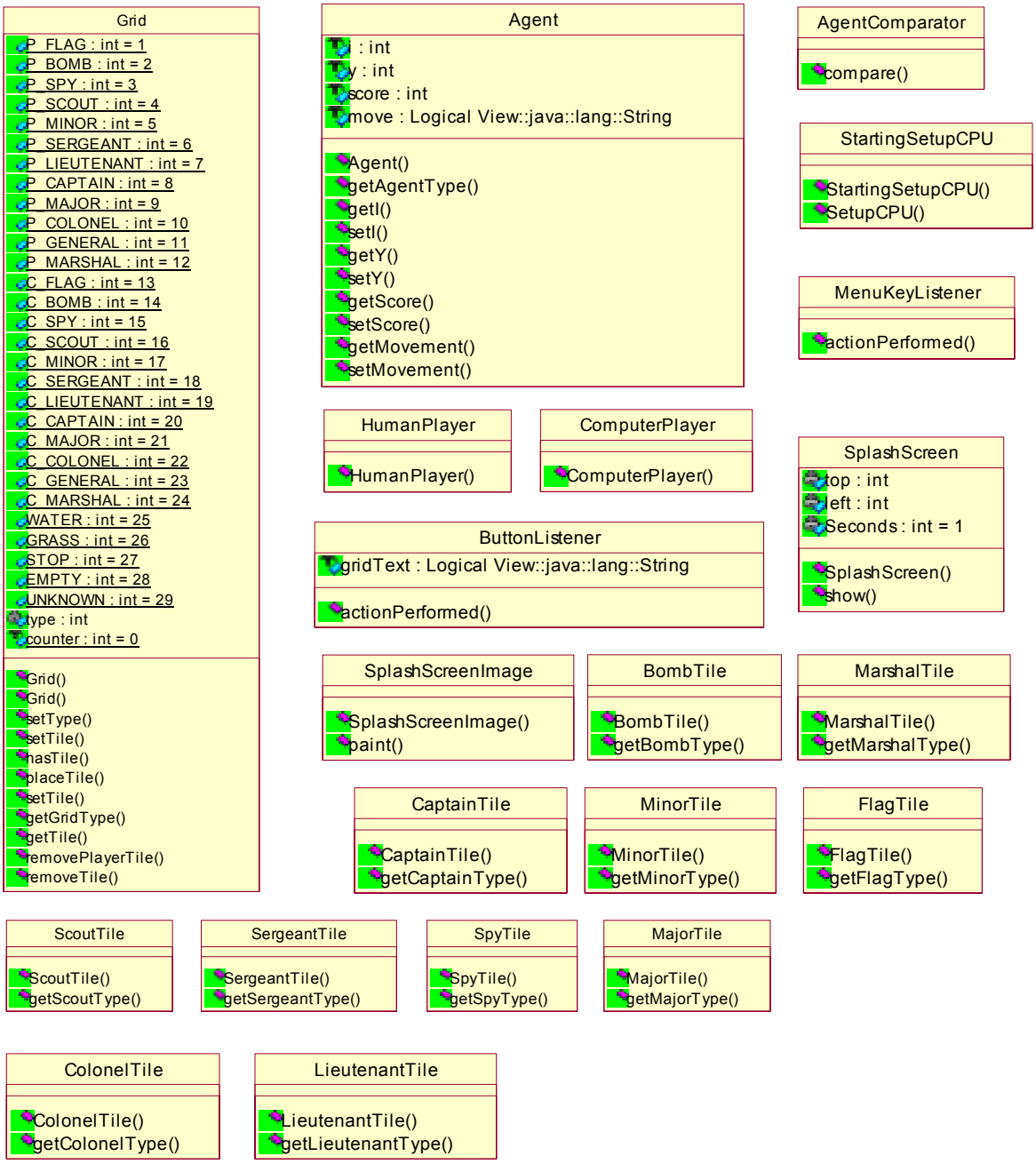
Use-case diagram




LieutenantTile
ColonelTile
GeneralTile
MajorTile
SpyTile
SergeantTile
ScoutTile
MarshalTile
BombTile
FlagTile
MinorTile
CaptainTile
Tile
Agent
AgentComparator
Player
ComputerPlayer
HumanPlayer
Board
Grid
Stratego
ButtonListener
MenuKeyListener
SplashScreen
StartingSetupCPU
SplashScreenImage
*
1..*
1


: Stratego
: Board
: Agent
: AgentComparator
JESS
getAgents( )
getAgentType( )
Status
setScore(int)
setMovement()
compare()
Agent()
orderAgents(List)
Resetview( )
checkPlayerTurn( )
AgentPlay( )
getMovement( )
getScore( )
AgentPlay( )
checkPlayerTurn( )
ResetAgent( )
checkPlayerTurn( )


Player
CPU : int = 1122
HUMAN : int = 2211
turn : int
score[] : int
playerType : int
Player()
isCPU()
getPlayerType()
getFlag()
setFlag()
removeFlag()
getBomb()
setBomb()
removeBomb()
getSpy()
setSpy()
removeSpy()
getScout()
setScout()
removeScout()
getMinor()
setMinor()
removeMinor()
getSergeant()
setSergeant()
removeSergeant()
getLieutenant()
setLieutenant()
removeLieutenant()
getCaptain()
setCaptain()
removeCaptain()
getMajor()
setMajor()
removeMajor()
getColonel()
setColonel()
removeColonel()
getGeneral()
setGeneral()
removeGeneral()
getMarshal()
setMarshal()
removeMarshal()
incr_turn()
reset_turn()
hasTurn()
Board
foundtile[][] : Logical View::java::lang::String = new String [10][10]
Tilename[][] : Logical View::java::lang::String = new String [10][10]
pressed : Logical View::java::lang::String = null
num : int = 0
havemoved : int = 0
mknown : int = 0
movment : Logical View::java::lang::String = ""
enemysouth : Logical View::java::lang::String = "nothing"
enemyeast : Logical View::java::lang::String = "nothing"
enemynorth : Logical View::java::lang::String = "nothing"
enemywest : Logical View::java::lang::String = "nothing"
agentname : Logical View::java::lang::String = "nothing"
enemybombs : int = 0
enemysouthsouth : Logical View::java::lang::String = "nothing"
enemyeasteast : Logical View::java::lang::String = "nothing"
enemynorthnorth : Logical View::java::lang::String = "nothing"
enemywestwest : Logical View::java::lang::String = "nothing"
enemynortheast : Logical View::java::lang::String = "nothing"
enemynorthwest : Logical View::java::lang::String = "nothing"
enemysoutheast : Logical View::java::lang::String = "nothing"
enemysouthwest : Logical View::java::lang::String = "nothing"
enemymarshal : Logical View::java::lang::String = "nothing"
AllAgents : Logical View::java::util::ArrayList = new ArrayList ()
Board()
paintGrids()
getAgents()
orderAgents()
Resetview()
AgentPlay()
Tile
FLAG_TILE : int = 11111
BOMB_TILE : int = 1000000
SPY_TILE : int = 33333
SCOUT_TILE : int = 33344
MINOR_TILE : int = 44444
SERGEANT_TILE : int = 55555
LIEUTENANT_TILE : int = 55566
CAPTAIN_TILE : int = 55577
MAJOR_TILE : int = 55588
COLONEL_TILE : int = 55666
GENERAL_TILE : int = 66666
MARSHAL_TILE : int = 77777
UNKNOWN : int = 88888
KNOWN : int = 99999
NotMoved : int = 0
Moved : int = 1
tileType : int
status : int
move : int
Tile()
getPlayer()
getTileType()
setStatus()
getStatus()
setMove()
getMove()
Stratego
playercount : int
Stratego()
initComponents()
checkPlayerTurn()
ResetAgent()
exitForm()
main()


Grid
P_FLAG : int = 1
P_BOMB : int = 2
P_SPY : int = 3
P_SCOUT : int = 4
P_MINOR : int = 5
P_SERGEANT : int = 6
P_LIEUTENANT : int = 7
P_CAPTAIN : int = 8
P_MAJOR : int = 9
P_COLONEL : int = 10
P_GENERAL : int = 11
P_MARSHAL : int = 12
C_FLAG : int = 13
C_BOMB : int = 14
C_SPY : int = 15
C_SCOUT : int = 16
C_MINOR : int = 17
C_SERGEANT : int = 18
C_LIEUTENANT : int = 19
C_CAPTAIN : int = 20
C_MAJOR : int = 21
C_COLONEL : int = 22
C_GENERAL : int = 23
C_MARSHAL : int = 24
WATER : int = 25
GRASS : int = 26
STOP : int = 27
EMPTY : int = 28
UNKNOWN : int = 29
type : int
counter : int = 0
Grid()
Grid()
setType()
setTile()
hasTile()
placeTile()
setTile()
getGridType()
getTile()
removePlayerTile()
removeTile()
Agent
i : int
y : int
score : int
move : Logical View::java::lang::String
Agent()
getAgentType()
getI()
setI()
getY()
setY()
getScore()
setScore()
getMovement()
setMovement()
AgentComparator
compare()
StartingSetupCPU
StartingSetupCPU()
SetupCPU()
MenuKeyListener
actionPerformed()
HumanPlayer
HumanPlayer()
ComputerPlayer
ComputerPlayer()
SplashScreen
top : int
left : int
Seconds : int = 1
SplashScreen()
show()
ButtonListener
gridText : Logical View::java::lang::String
actionPerformed()
SplashScreenImage
SplashScreenImage()
paint()
BombTile
BombTile()
getBombType()
MarshalTile
MarshalTile()
getMarshalType()
CaptainTile
CaptainTile()
getCaptainType()
MinorTile
MinorTile()
getMinorType()
FlagTile
FlagTile()
getFlagType()
ScoutTile
ScoutTile()
getScoutType()
SergeantTile
SergeantTile()
getSergeantType()
SpyTile
SpyTile()
getSpyType()
MajorTile
MajorTile()
getMajorType()
ColonelTile
ColonelTile()
getColonelType()
LieutenantTile
LieutenantTile()
getLieutenantType()