Wargame

Title: Wargame

Subject: Language engineering

Semester:
SW4, Spring Semester 2011

Project group:
sw402a

Participants:
Henrik Klarup
Kasper Møller Andersen
Kristian Kolding Foged-Ladefoged
Lasse Rørbæk
Rasmus Aaen
Simon Frandsen

Supervisor:
Jorge Pablo Cordero Hernandez

Number of copies:

Number of pages:

Number of appendices:

Completed: 27. May 2011

Synopsis:

In this project we will develop a small language to control the logics of a multi agent system.

I

Preface

This report is written in the fourth semester of the software engineering study at Aalborg University in the spring 2011.

The goal of this project is to acquire knowledge about fundamental principles of programming languages and techniques for description and translation of languages in general. Also a goal is to get a basic knowledge of central computer science and software technical subjects with a focus on language processing theories and techniques **ref. to study regulation**.
We are going to achieve these goal by designing and implementing a small language for controlling a multi agent system in the form of a wargame. We are going to use Visual Studio and C#, because we have used these tools in earlier semesters and are used to the C# syntax.

The report is written i LaTeX, and we have used Google Docs and TortoiseSVN for revision control.

Source code examples in the report is represented as follows:

```
1  if (spelling.ToLower().Equals(spellings[i]))
2    {
3       this.kind = i;
4       break;
5    }
```

Source Code 1: Example of representation of source code in the report.

Indhold

Kapitel 1

Introduction

header - in this chapter we will introduce...

1.1 Motivation

1.2 Tools

tail - we have introduced...

Kapitel 2

The Wargame

header - in this chapter we will outline...

2.1   Wargame Scenario

The war game is initialized and the number of agents on the teams is chosen by the user. The first user types the first command, and clicks the button *Execute* to execute the command. When the user is done making his draws, he ends his turn by pressing the *End Turn* button. The moves available for the user to make is up, down, left and right (one coordinate at a time), and it is also possible to make several moves with an agent, if you select the agent and type the coordinates you want the agent to move to. When a collision between agents from opposing teams occur, a random function is called, which decides which agent wins the fight, favoring the unit with the highest rank.

tail - in this chapter we outlined...

Kapitel 3

Implementation

header - in this chapter..

3.1 Making the Scanner

The scanner is converting the string of text, the input, to a compilation of
tokens and keywords. The first method of the scanner is a big switch created
to sort the current word according to the token starters **??**. If the first char-
acter of a word is a letter, the word is automaticly assigned as an identifier
and a string with the word is created.
When the word is saved as a Token, the Token class searches for any key-
word, that would be able to match the exact string, e.g. if the string spells
the word „for", the Token class changes the string to a **for** token.

```
1  public Token(int kind, string spelling, int row, int col)
2      {
3          this.kind = kind;
4          this.spelling = spelling;
5          this.row = row;
6          this.col = col;
7
8          if (kind == (int)keywords.IDENTIFIER)
9          {
10             for (int i = (int)keywords.IF_LOOP; i <= psy191
                   (int)keywords.FALSE; i++)
11             {
12                 if (spelling.ToLower().Equals(spellings[i]))
13                 {
```

```
14                          this.kind = i;
15                          break;
16                      }
17                  }
18              }
19          }
```

Source Code 3.1: The token method with overloads.

In the token overload method, IF_LOOP and FALSE is a part of an enum and then casted as an int, kind is an int identifier and spellings is a string array of the kinds of keywords and tokens availible, as seen below.

```
1   public static string[] spellings =
2       {
3           "<identifier>", "<number>", "<operator>", psy191
                "<string>", ";", ":", "(", ")", "=", "{", "}",
4           "if", "else", "for", "while", "bool", "new", psy191
                "main", "team", "agent", "squad", "coord", "void",
5           "actionpattern", "num", "string", "true", psy191
                "false", ",", ".", "<EOL>", "<EOT>", "<ERROR>"
6       };
```

Source Code 3.2: The string array spellings.

3.2   Making the Parser

The parser is what takes the compilation of tokens and keywords generated by the scanner and builds an abstract syntax tree (AST) from it (while also checking for grammatical correctness). To accomodate all the different tokens, each token has a unique parsing method, that is called whenever a corresponding token is checked. Each of these methods then generate their own subtree of the AST, and returns that subtree, so it can be added to the AST.
For details on how the AST works, see section 3.3.

```
1   public AST parse()
2       {
3           return parseMainblock();
4       }
```

Source Code 3.3: This is the main parsing method

```
1  private AST parseMainblock()
2      {
3          Mainblock main;
4          switch(currentToken.kind)
5          {
6              case (int)Token.keywords.MAIN:
7                  acceptIt();
8                  accept(Token.keywords.LPAREN);
9                  accept(Token.keywords.RPAREN);
10                 main = new Mainblock(parseBlock());
11                 accept(Token.keywords.EOT);
12                 return main;
13             default:
14                 // Error message
15                 accept(Token.keywords.ERROR);
16                 return null;
17         }
18     }
```

Source Code 3.4: This method parses a mainblock and returns a mainblock object

In the parseMainblock example, we see that it returns a Mainblock object (which inherits from the AST class) called main. The constructor for the Mainblock takes a Block object as its input, so main is instantiated with a parseBlock call.

The parser checks for grammatical correctness, by checking if each token is of the expected kind. For example, a command should always end with a semicolon, so the parser checks for a semicolon after each command. If there isn't one, the parser returns an error saying what line the error was on, and which token did not match an expected token.

3.3 The Abstract Syntax Tree

3.4 Code Generation

3.5 The Graphical User Interface

The user interface is made as a windows form application. With Visual Studios designer tools it is simple to make a nice design with buttons, panels, and windows just the way you want.

The main idea of the design of the user interface is that there should be only a few buttons, sat the user should not spend a lot of time figuring out what all the buttons do. Furthermore we have designed the interface so that the main structure looks like other popular strategy computer games (see A.1 and A.2 in appendix). We have done this to make the application easy to learn how to use.

3.5.1 Game Start Settings

When the game is started, a dialog box is shown where one can choose the size of the war zone. We have chosen to have three fixed grid sizes, because of the way we draw the grid 3.6.

The functions of the dialog box is:

1. *Small, Medium, Large* radio buttons - select one to choose the grid size.

2. *Start* button - starts the game.

3.5.2 Interface Functions

The functions of the game interface is:

1. *War zone* - contains the grid on which the war game unfolds.

2. *Command center* - here the user types the commands to navigate the agents around the grid.

3. *Stats field* - shows the stats of a selected agent.

4. *Combat log* - contains a combat log on who killed who in fights between agents.

5. *Command list* - contains the list of available commands the user can type in the *command center*.

6. *MousePos grid* - shows the grid point of the mouse position.

7. *Execute* button - executes the typed in command in the *command center*.

8. *End turn* button - ends the turn and gives the turn to the next player.

9. *Reset game* button - sets up a new game.

10. *Quit game* button - closes the game.

3.6   Drawing the Grid and Agents

sub conclusion - in this chapter we have made...

Kapitel 4

Discussion

header...

4.1   Usability

tail...

Kapitel 5

Epilogue

header...

5.1   Conclusion

5.2   Future Work

tail...

Bilag A

Appendix

A.1   Other Games

Figur A.1: Screen shot of the game user interface in Red Alert 2.

Figur A.2: Screen shot of the game user interface in Command and Conquer 3.

Litteratur