1. Introduction

1.1 Project proposal and aims

Logo is a programming language based on LISP and today known mostly for its turtle graphics software packages used in schools to help pupils explore basic concepts from geometry. It was created in 1967 by Wally Feurzeig and Seymour Papert at MIT, with specific educational uses in mind. It is Seymour Papert who pioneered its use in school classrooms as an educational system. It is an example of the Constructivist approach towards education:

> "In many schools today, the phrase 'Computer-Aided Instruction' means making the computer teach the child. One might say the computer is being used to program the child. In my vision the child programs the computer and, in so doing, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building." [35]

> "Traditional education codifies what it thinks citizens need to know and sets out to feed children this 'fish'. Constructionism is built on the assumption that children will do best by finding ('fishing') for themselves the specific knowledge that they need." [36]

The aim of my project is to build a web based application containing a Logo programming environment aimed at school students in which Logo files can be written, saved, loaded and executed, either on screen (for testing or if a robot is unavailable) or on a Lego Mindstorms™ robot.

In a group situation such as a classroom, a group of users will be able to share one robot and take turns executing their Logo programs on it. The robot will have wheels and will draw out the shape created by the Logo on the floor.

I will not be aiming to implement Logo fully, but rather I will be implementing a small manageable subset that I feel makes the application I develop of use to schools, as well as flexible enough to allow extra keywords/constructs to be easily added later. As a rough guide my software will be aimed at pupils in Keystages 3-4 (11-16 years of age).

1.2 The Logo language

In contrast to object oriented languages such as C++ and Java, Logo is not based upon 'objects' with attributes and states (with the exception that we could say that the turtle itself is an object) but instead is intended to support functional programming. Indeed, as we will see below, the function (called a

'procedure') is one of the most useful constructs in Logo.

The most basic commands in Logo that control the graphical turtle are the 'go forward' command (written variously as FORWARD, forward, FD or fd), and the 'turn right' command (RIGHT, RT, rt). So, one of the first Logo programs that a pupil would typically be exposed to would be to draw a square using "`FD 100 RT 90 FD 100 RT 90 FD 100 RT 90 FD 100`". In contrast to a Cartesian turtle with (x,y) coordinates, our Logo turtle has only a location on screen and an orientation (the direction it is facing). Indeed, "the Turtle is like a person – I am here and I am facing north...And from these similarities comes the Turtle's special ability to server as a representative of formal mathematics for a child". [35]

There are many dialects of Logo but typically variables are prefixed with a colon and are declared with a MAKE statement (See Fig 1.1i). Simple loops can be programmed by use of the 'repeat' command (REPEAT, RPT, rpt) which tells the turtle to execute the commands within a pair of square brackets a specific number of times. So, our 'draw square' program could be written as : `RPT 4[FD 100 RT 90]`. For children, a *function* is often thought of as telling the turtle the meaning of a new word. Logo typically uses the keywords TO and END to define a function. So, we could rewrite our 'draw square' program once more as in Fig 1.1(ii).

```
MAKE :x 1
MAKE :x :x+1
```

Fig 1.1 (i)- Logo make statements -

set the variable :x equal to 2

```
TO drawPoly :sideLength :N
RPT :N[
  FD :sideLength
  RT (360/:N)
]
END
drawPoly 100 4
```

Fig 1.1 (ii)- a Logo procedure

Recursion can also be used in Logo - for example Fig 1.2

```
TO SPIRAL :dist
  FORWARD :dist
  RIGHT 90
  SPIRAL :dist + 5
END

SPIRAL 10
```

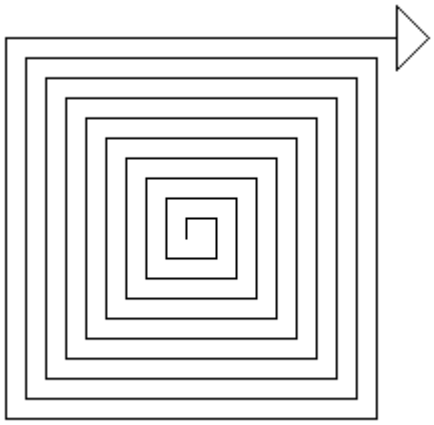


Fig 1.2 - Recursive Logo program and output

As Papert says, “Thus we have a trick called 'recursion' for setting up a never ending process...Of all ideas I have introduced to children, recursion stands out as the one idea that is particularly able to evoke an excited response.” [35].

There are many other keywords and programming constructs found in most dialects of Logo - for example the pen up/pen down commands. Loosely speaking the phrase 'turtle maths' relates to cut-down implementations of Logo used in school classrooms. Many of the fuller implementations of Logo include boolean expressions, lists/arrays, if/else commands, while loops, returning values from functions, random number generators, the ability to output strings to the command line or graphics window, and even IO facilities.

1.3 Lego Mindstorms and Lejos

Lego™ manufacture and sell a robot and microprocessor called the NXT Mindstorm. It comes with sensors and motors, and a set of Lego components. It is no coincidence that Papert's book [35] and the Lego robots share a name – in fact Dr. Papert today serves on the advisory board of the Lego Mindstorms project and Logo was accompanied by a physical robot as early as 1969. So, to some extent my entire project will be a reworking of this original project from forty years ago, albeit with different technologies.

Fig 1.3 - original Mindstorms robots (1969) (Image from [40])

The LEGO NXT 'brick' is about 1”x 2” x 4” in size. It comes with three motors and a range of sensors such as a colour sensor that can detect colours and brightness, touch sensors and an ultrasonic sensor to detect other objects and their proximity. Whilst opening up a world of fun possibilities, I do not intend to be using the sensors in this project.

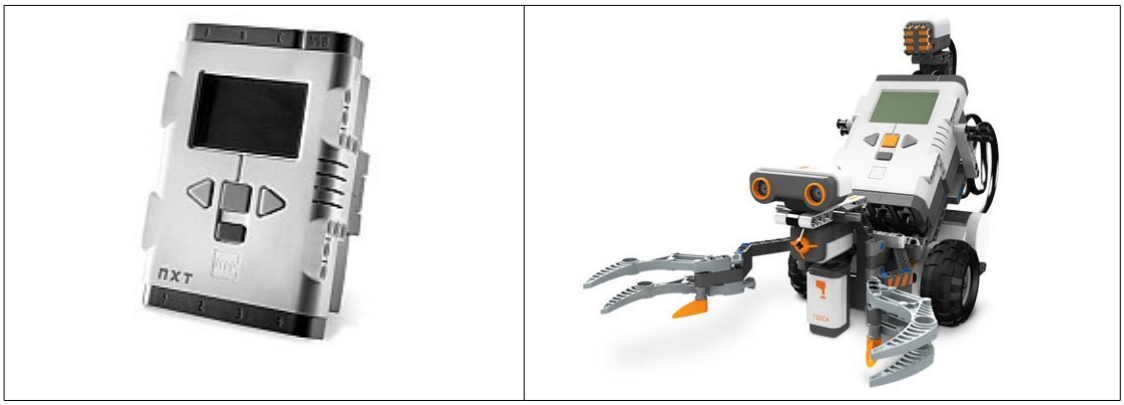


Fig 1.4 - Lego NXT

When purchased the NXT runs Lego's commercial software – called NXT-G (see section 2.2.5). What makes the NXT useful for my project is the fact that different software can be run on the microprocessor. By far the most appealing, and the reason I felt this project was feasible is called Lejos (a play on the phrase “Lego Java Operating System” and the Spanish word for 'far', to be pronounced as in the Spanish). A more detailed explanation of Lejos is found in section 5.

2 Research

This chapter contains a summary of my research into various Logo interpreters available on the web, on sale to schools and some relevant technical literature about the use of robots and Java in education.

2.1 Available Turtle Maths programs / Logo interpreters

There are numerous free Logo interpreters on the web or downloadable, a brief summary of the functionality and usability of some of the more notable ones follows, split between web based and those that need download / install.

Web-based:

http://logo.twentygototen.org - Written in JavaScript, with large text panel for entering Logo. Possibility to save output as image. Ability to run drawing at different speeds.

http://www.calormen.com/Logo/#examples - Written in JavaScript, no ability to export work done, or to run at different speeds. Very large subset of Logo implemented.

http://www.amberfrog.com/logo - Also Javascript, no ability to save/export work., or to run at different speeds. Very small and hard to use text entry box with very bad help functionality.

http://library.thinkquest.org/18446 - A Java applet. A simple subset of Logo is implemented, which can be executed line by line or continuously. Has a nice feature to upload a “procedure” for other people to use (which doesn't seem to be working).

http://homepage.mac.com/troy_stephens/TinyJavaLogo - Very old Java applet, supporting a very limited subset of Logo (for example you cannot go forward by a variable amount, only fd <Integer>).

Downloadable/Installable:

http://www.softronix.com/logo.html - MSWLogo for Windows. This seems to be a very popular choice for a simple free interpreter. Written in C++ with extremely fast performance and pop-up menus to define procedures, which I personally found confusing. Ability to save/load Logo files stored on the user's machine, to draw continuously or step-by-step, to print and to load bitmap images onto the screen

for fun. Implements a huge subset of Logo including many data structures, Windows networking, multimedia, 3D Logo, a pascal interpreter, as well as a huge help file and many demos. See Fig 2.1 for a screenshot.

http://www.eecs.berkeley.edu/~bh - A similarly huge implementation of Logo, from the University of California, Berkeley, written in C. Hangs when large numbers are used (UI becomes unresponsive when infinite loops are used). Features include save, load and print.

http://tortue.sourceforge.net - A nice looking Java Swing interpreter, with save/load functionality, a colour chooser for the pen colour, good error messages/error reporting and good help files. Full of bugs (even run/stop does not seem to work reliably)

http://activities.sugarlabs.org/en-US/sugar/addon/4027 - Implemented in Python - turtleArt users create Logo by snapping together visual blocks that represent commands. For example a “number” block and a “list of statements” block are plugged into a “repeat” block. It has a lot of available commands, in multiple languages (right, derecha, droit ...). In fact Logo is only part of what can be done with turtleArt - users can animate objects on screen, listen for keyboard and mouse interactions on objects and can include images and text as 'objects' as well as those drawn by the turtle. Files can be incorporated into slideshows and exported as html.

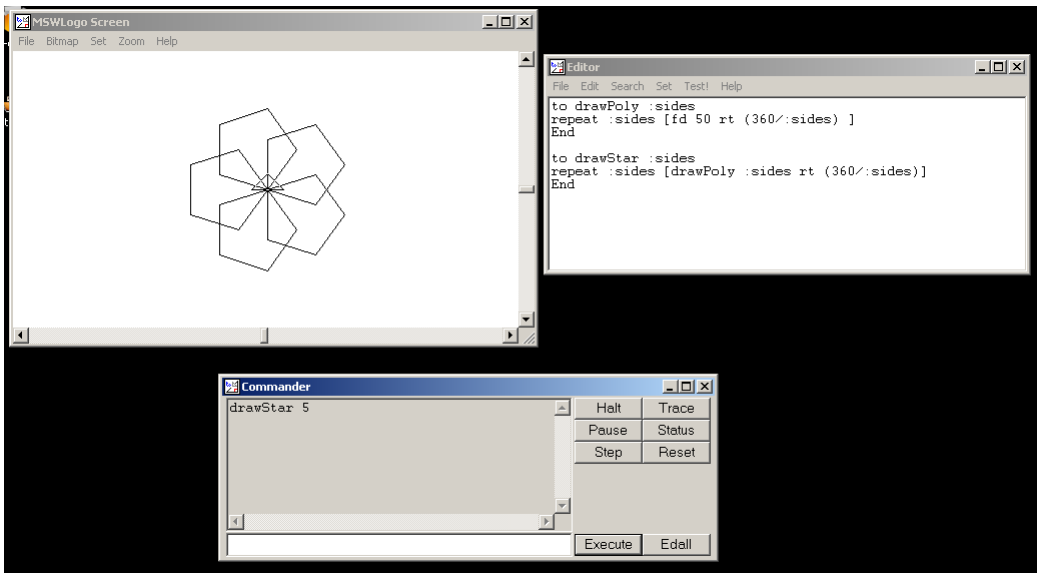


Fig 2.1 - MSWLogo screenshots

Another Logo-based product of interest to me is called StarLogo (See [39]). This takes Logo to a higher level, using unlimited numbers of turtles in parallel and the idea of *patches*. Patches form a grid on which the turtles move and are programmable to contain a number of *chemicals* at different levels that can be used to represent food, pheromones, fire, water, or actual chemicals. Together with code that control how turtles interact with the patches, StarLogo can be used to model complex decentralized systems. See [39] for more information.

2.2 Turtle maths products on sale to schools in the UK

The Primary National Strategy describes Logo as one way to support teaching primary school maths and ICT:

> "...by using a computer, pupils in Key Stages 1 and 2 can...estimate and compare measures of length or distance, angle, time, and so on...for example, by devising a sequence of instructions to move a floor robot or screen 'turtle' along a path, then modifying their instructions in the light of the robot's response." [7]

For this reason, a number of products aimed specifically at schools are on sale. Many of these are specially designed to be user friendly and to cater for the needs of younger learners. Some notes on the functionality of some of these follows.

2.2.1 ScreenTurtle2 [44]

Screen turtle 2 allows users to load images onto the screen (for example a maze that can then be programmatically navigated by the turtle). It implements a very large subset of Logo including pen colours and types of pen, a 'delete' pen that erases previously drawn lines, a 'fill' statement, a 'return to origin' statement, if/then/else statements, as well as printing words to the screen. Users' files can be saved and loaded. It aims to be user friendly by allowing many representations of words (FORWARD, forward, FD, fd all do the same thing). Procedures are defined in a separate pop-up panel which personally I do not find user friendly. It requires install on users computers.

2.2.2 2SimpleControl [1]

This product was actually the inspiration for this project - I played with it at an educational technology conference in London, January 2009. It is one of the few products that is designed to communicate with

an NXT robot. It consists of 6 modules (all based around the “Control” part of the ICT curriculum in the UK), one of which is Logo (see Fig 2.2)

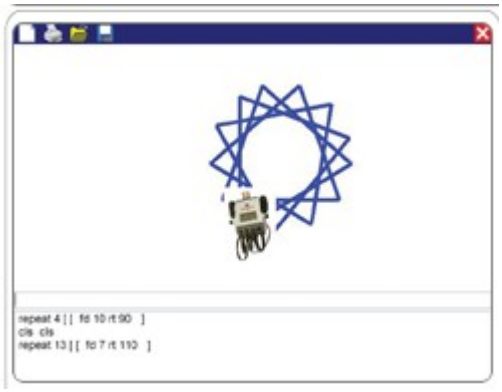


Fig 2.2 - 2SimpleControlNXT screenshot. Image from [1]

Users can load and save files (.logo files). I do not know what subset of Logo is implemented but the way it is entered is not user-friendly. A text box is provided into which commands are typed, but they execute on screen when the Enter key is pressed, so really only one line is available to type in. Programs are compiled and then uploaded to the NXT (connected by USB), and then the NXT's execute button must be pressed.

2.2.3 Terrapin Logo [42]

Terrapin Logo seems to be one of the oldest educational Logo packages and another that supports communication with an NXT. It supports a very large dialect of Logo including multiple turtles in one or multiple windows, lists/arrays, a lot of multimedia functionality and windows to view all currently defined turtles, variables and procedures. Very sophisticated graphics programs can be created including mouse and keyboard interaction and moving embedded images. It looks very sophisticated for younger users, but the project ideas for older users look impressive (See [43])

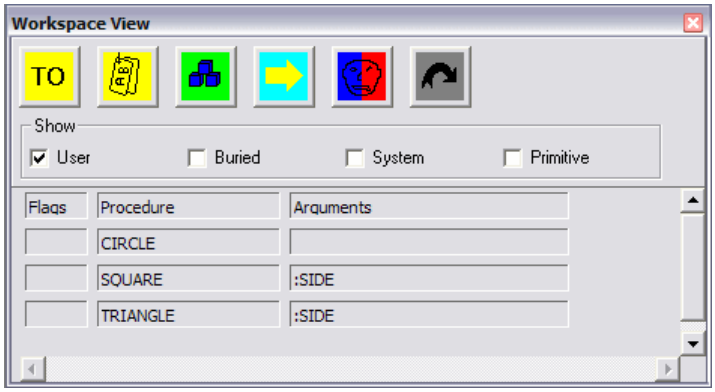


Fig 2.3 Terrapin Logo screenshot. Image from [42]

2.2.4 Logotron Imagine Logo [23]

This software can be used to create full multimedia activities, of which on-screen turtles are only a part. Activities can consist of multiple pages, and can contain sounds/music and animation. They "can be shared, printed, or published on the web or school intranet" [23]. Buttons and sliders can be dragged onto the screen and events associated with user interactions. Logo basics such as moving the turtle and defining procedures are edited using Windows-style dialog boxes, see Fig 2.4.

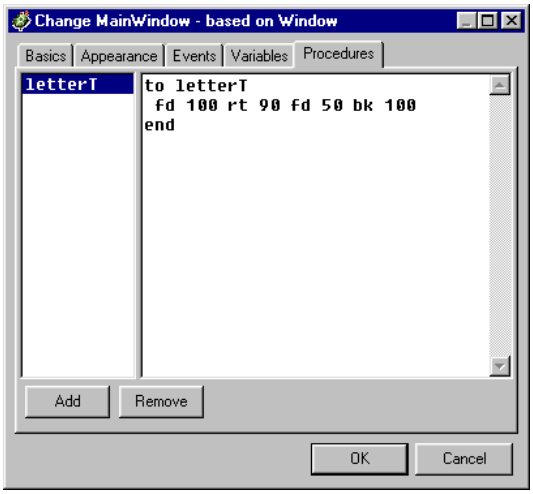


Fig. 2.4. Defining a procedure in Imagine Logo

2.2.5 Lego Mindstorms NXT-G

The software that comes with the NXT is a drag and drop environment in which pupils can write flow-chart style programs to compile and run on the firmware that comes ready-installed with the NXT. Snap together blocks create the flow of logic. For example, Fig 2.5 shows (reading from left to right) that the NXT runs one infinite loop (orange arrows) which executes the following: "wait for the touch sensor to be pressed then play a beep sound". Fig 2.6 shows a decision block (if statement). If the distance sensor registers true (an object is close to the NXT - the tolerance can be configured) then the top path is

followed and motors B and C are turned on in opposite directions. Otherwise C and B are both turned on forwards.

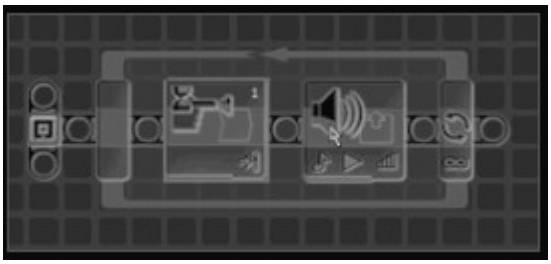


Fig 2.5 - NXT-G loop block

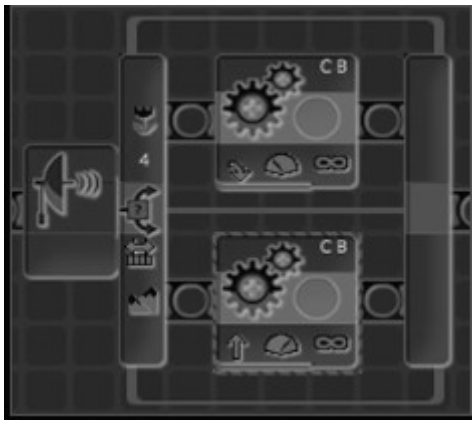


Fig 2.6 - NXT-G decision block

2.3 Relevant technical documents & research papers

There is considerable literature discussing the usefulness of robotics to the teaching and learning of programming at a far higher level than the school pupils this project is aimed at - namely the use of robots in learning aspects of object oriented programming - for example [6], [17].

A research paper dealing with a project very similar to mine is [11], in which the authors have created a system to allow undergraduate students to work on one Lego robot, write programs in a C-like language called NQC (Not Quite C) and compile and run these on one robot, feedback being obtained not by seeing the robot, but by a webcam showing the robot's movements:

> "The cost of the Mindstorms kit ($200) prohibits us from allowing each student to take a robot home with them. To address this problem, we have developed a web interface that allows students to remotely program the robot and watch the results via webcam....The telerobotic web interface we have developed uses a client-server architecture. This architecture allows transmission of NQC source code, webcam images, queue scheduling information, user information, and user chat messages. This system requires only that the user have a Java-enabled modern web browser." [11]

The authors explain the technical details of their system as follows:

> "The server for the system handles all requests for use of the robot. It maintains and manages a queue of users who are currently waiting to use the robot. In addition, the server must interface with a webcam in order to deliver constant visual feedback ...the server must interface to the NQC compiler and robot in order to compile, download, and run programs...The client is implemented as a Java applet that is embedded in a web page. The top of the applet contains a panel that represents the queue of users waiting to control the robot... By clicking on another user's icon, a user may initiate a chat message session with that user....The Test tab allows the active user to run and stop a current program on the robot. All other users may watch the robot's progress." [11]

An undergraduate project that was of particular interest to me is [24]. The author of this project has created software to control an NXT in two modes, interactive and macro. In the interactive mode the user controls the robot and it responds immediately, like a remote controlled car:

> "The Interactive role will control the robot directly using inputs and buttons through the web interface. Any action taken will influence the robot immediately i.e. real-time control. " [24]

The macro mode allows users to upload behaviour to the robot which it will then obey with no further instruction:

> "The Macro role will control the robot indirectly by choosing a list of macros (events, conditions and actions), uploading them to the robot, and allow the robot to interact with the world automatically. Any movement or action to be taken by the robot will be determined by the chosen macros i.e. current events taking place and conditions that are met." [24]
> "... e.g. Event - Echo Sensor : Condition - Distance < 10cm : Action - Stop. The macro list will be traversed until a match to the current condition sent by one of the robot's sensors is found. This triggers the action command to be sent to the robot." [25]

3 Software Requirements Specification

This chapter contains a description of the system I plan to develop, including functional requirements that describe a user's interaction with the system. The product is described in terms of its subsystems, and each is described.

3.1 Product perspective

3.1.1 Overview

The product is a multi-user web based Logo programming environment for LEGO Mindstorms robots.

Users have authenticated log in and are presented with a Logo programming environment consisting of text entry for Logo code, feedback (such as syntax errors) and a canvas on which the Logo will be drawn. Their files can be saved and loaded.

A further “back-end” component will maintain a list of users. Users can request and relinquish the control of one LEGO Mindstorms robot. The user in charge of the robot can send Logo to the robot to be executed (the robot will move and draw the Logo output).

It is intended to be used in schools, but is usable in any situation where a group of users wish to interact with one robot. The application is to be usable without a robot; Logo can still be drawn on screen and files can be saved and loaded. Unlike some of the products discussed in section 2, files are not saved locally - all files are saved on a web server.

3.1.2 Basic architecture

The finished system will need to run on a web server to provide access to the main application to all users as well as a database of users and the files they are working on. Users will need to have registered username and passwords. In a group (for example a classroom), the robot is connected directly to another computer (visible by all members of the group and labelled “Admin server” in Fig 3.1) by bluetooth.

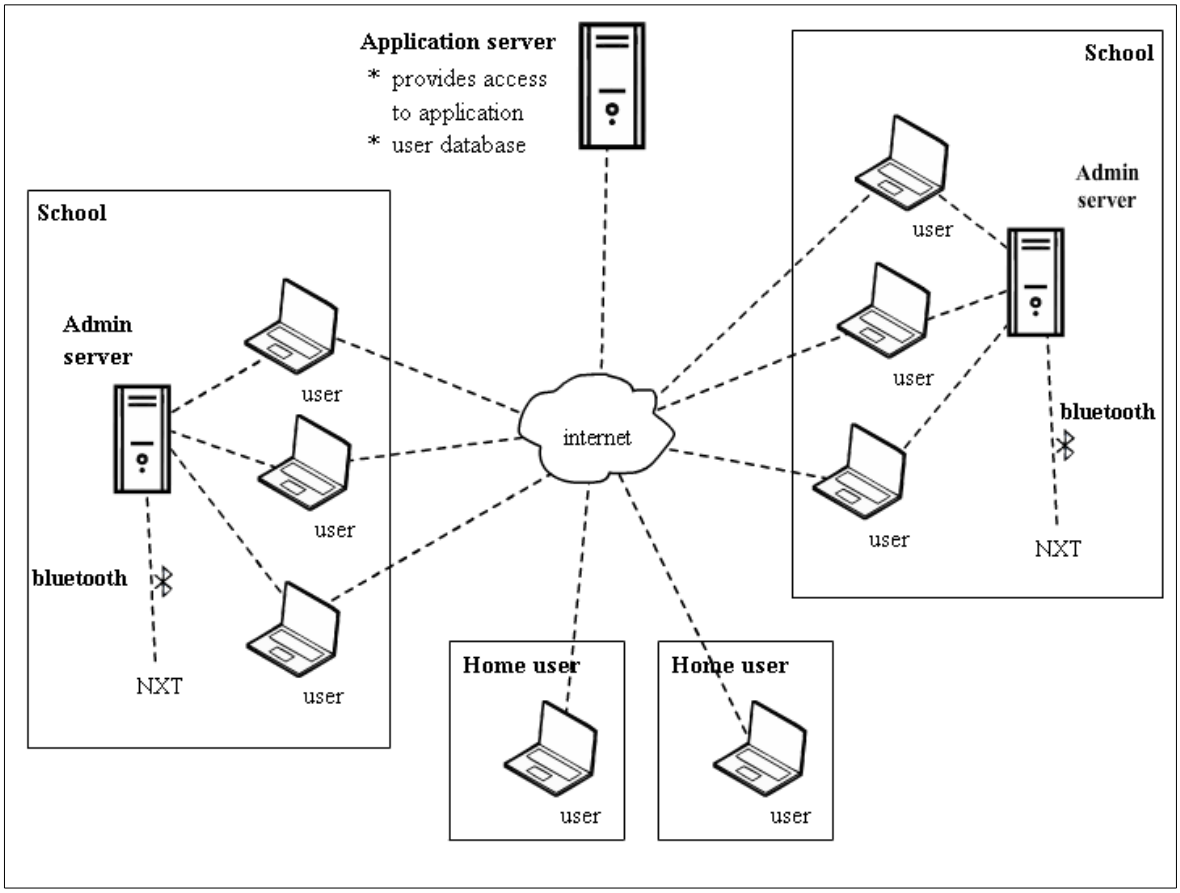


Fig 3.1 - basic architecture of the application(s) to be produced

3.1.3 Components and terminology

The system splits conceptually into the following six components, i) - vi) below:

i. **User authentication** - users need to be uniquely recognised by the system.

ii. **File management** - users can view and open files that they have been working on, delete files they don't want and create new files.

iii. **Logo editor** - a programming environment similar to other well known IDE's (but much simpler). This is where users execute Logo "locally" (i.e. on screen by a graphical 'turtle'), and obtain feedback regarding syntax errors.

iv. The **Logo interpreter** - the component that deals solely with syntax checking a Logo program, and interpreting it.

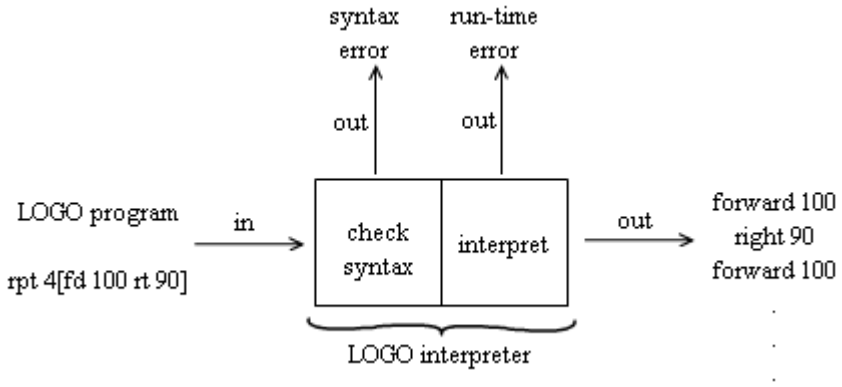


Fig 3.2 - Logo interpreter

A syntax error is essentially a badly written program (for example FORWARD instead of fd). A run time error cannot be caught before interpreting the program; it could be calling a procedure (or using a variable) that has not been defined yet, or perhaps a division by zero error.

v. **User management** - users can see a list of users logged in and can request/relinquish control of the robot. No more than one user can be in control of the robot at any time. If no-one is in control, the first person to request will be granted control, and control will remain with that user until relinquished (or on log out, or if the application is closed). When relinquished the next user in the waiting list will be granted control.

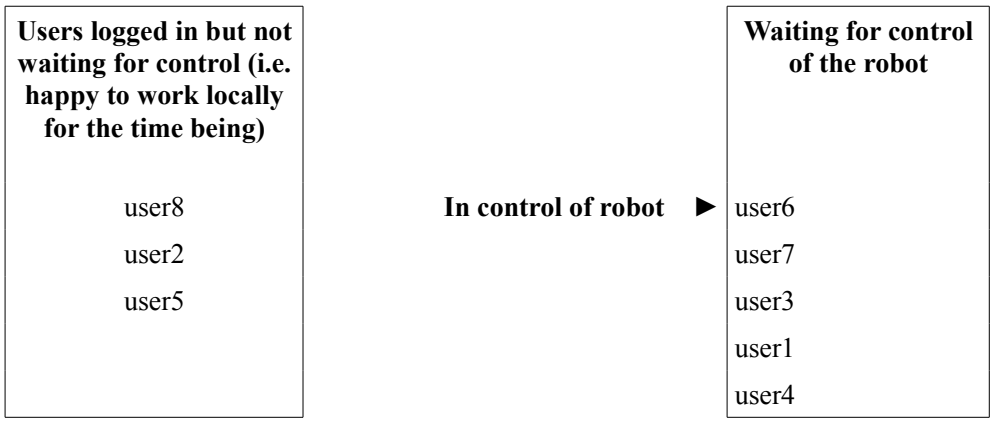


Fig 3.3 - User management

If user8 were to request control of the robot his/her name would be appended to the end of the 'waiting' queue and removed from the 'logged in but not waiting' queue. If user6 were to log out or relinquish control, user 7 would take control. A user should not be allowed to request control if they are already in the waiting queue or in control of the robot. A user can only relinquish when they are in control and their program is not running on the robot.

vi. **Robot Functionality** - details of the software running on the Lego Mindstorms robot itself. The robot is going to have two wheels to move around the ground, moving forward and rotating when instructed by the Logo.

3.2 Product Functions

3.2.1 Functional requirements

A complete list of functional requirements with explanations where necessary is contained in the use case diagrams below, divided into the six basic components listed above.

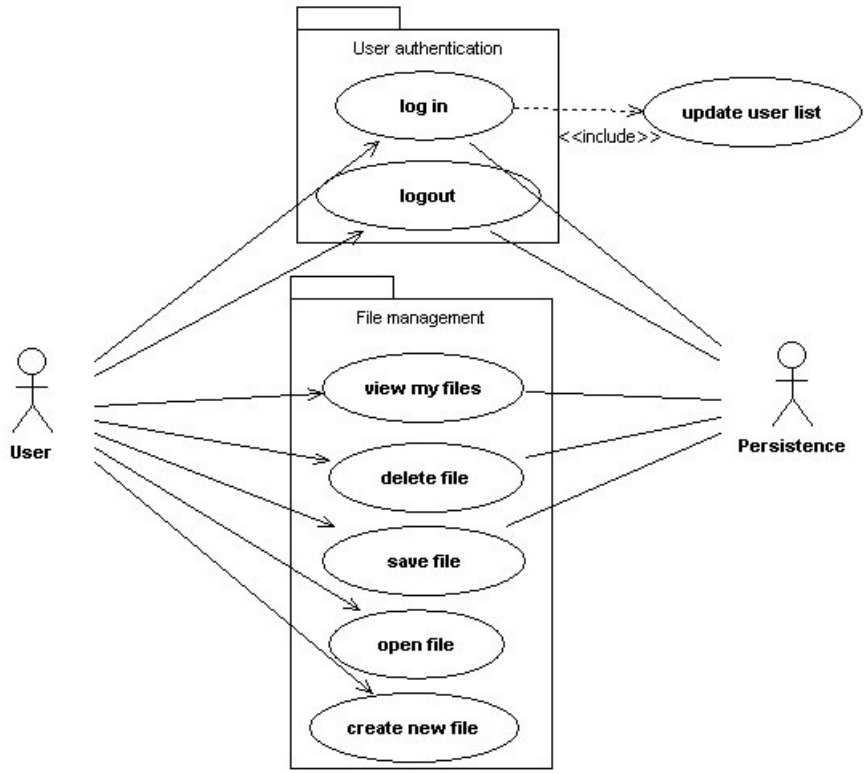


Fig 3.4 - File management & user authentication use cases

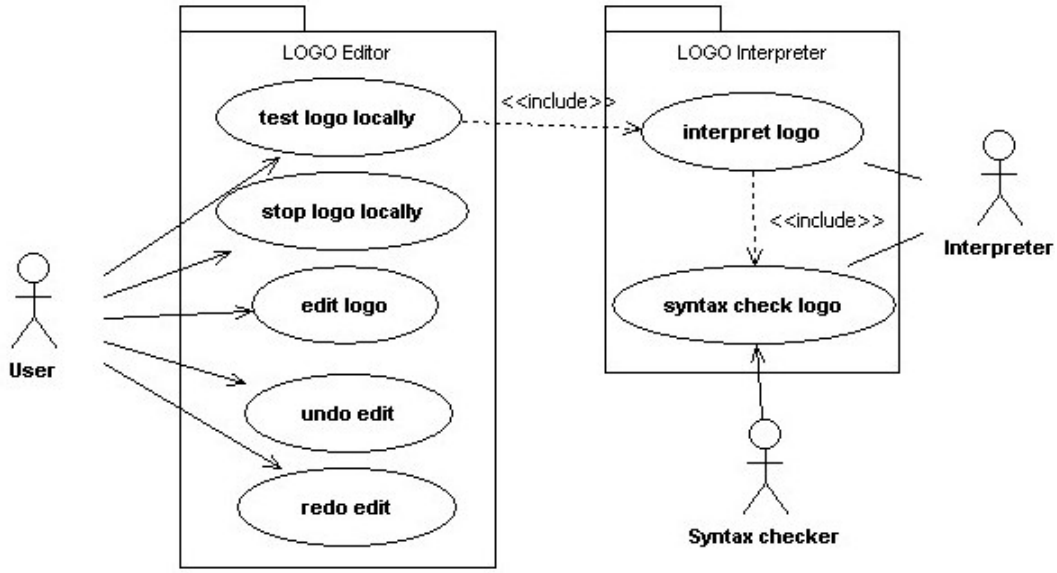


Fig 3.5 - Logo Editor and Logo interpreter use cases

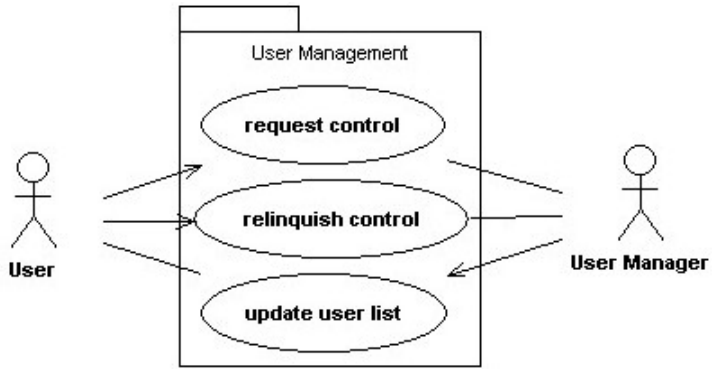


Fig 3.6 - User management use cases

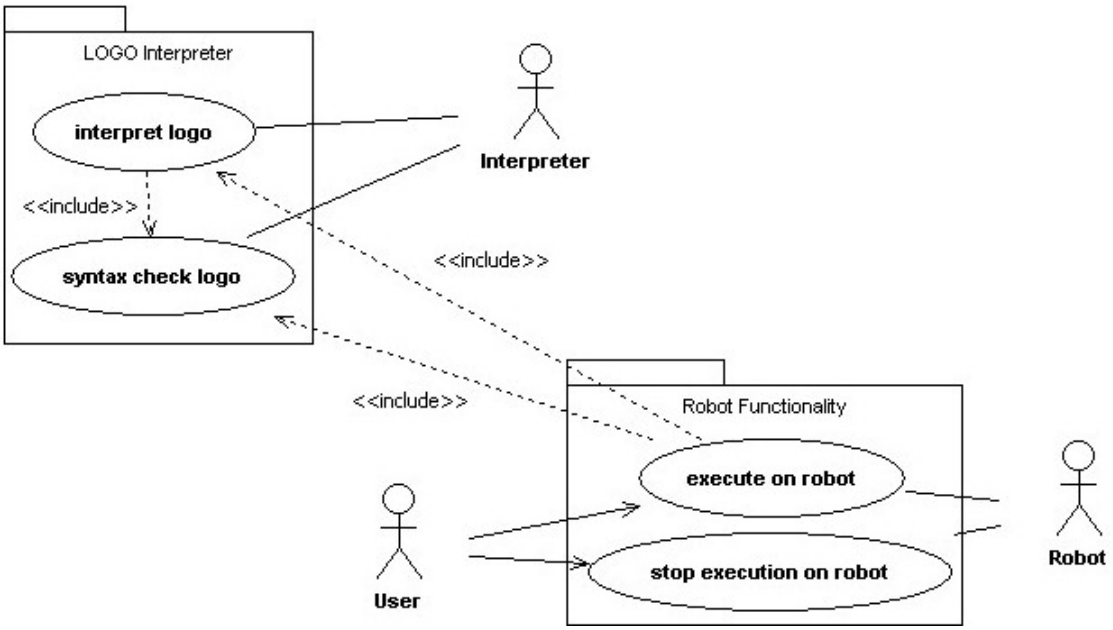


Fig 3.7 - Robot functionality and Logo interpreter use cases

Explanatory notes:

- The Logo Editor package contains all use cases regarding the 'local' editing and execution of Logo (on that user's screen rather than any interaction with the robot itself).
- The Persistence actor represents the storage of users' credentials and their files.
- The “edit logo” use case is a catch-all use case representing the fact that users can edit logo. In reality this use case will need to capture a number of features of the GUI that bear on exactly how and where (a textfield for example) that the user enters Logo. Undo and redo are represented an separate use cases.
- The Syntax Checker actor is an abstraction for the subsystem that checks the syntax of Logo. It will periodically and automatically check the syntax of the Logo entered, just like most modern IDEs, and will display errors if necessary, so that the user doesn't have to compile/run the Logo each time. It is also included in the 'interpret logo' use case since the Logo cannot be interpreted without first checking it has valid syntax.
- The Robot actor is a simplification. From the user's point of view there is one robot actor but in reality there will be complicated communication from the server to the robot.

3.2.2 Intended users and environment

The software is intended to be used primarily by a group of school students in a school. The robot must obviously be bought separately and in this project I do not concern myself with getting the robot built or running the code on it. Installation of the entire application will inevitably require a certain amount of technical ability but it should not be beyond the wit of an IT teacher for example.

In order for user names and passwords to be created, school students will need to register with an external body (imagine the company that sells the software). Users will then be able to access the application and their files from anywhere whether using a robot or not.

3.3 Requirements in detail

3.3.1 Detailed use cases

Each use case is given a name consisting of UC_ followed by the name from the use case diagram

above in camel case. These are found in Appendix B.

3.3.2 System attributes/non functional requirements

The application should support the following non-functional requirements:

- Error reporting
    - The application should display errors to school students that make some sense to younger learners. This includes syntax errors in their Logo as well as 'wrong username' errors and 'the robot does not seem to be connected' errors if for example the robot is switched off.
    - The back-end application should display more technical errors to ensure that bugs can be resolved/technical support given. These errors should be logged on the central application server as well as the 'admin server' that is connected to the robot.
- The speed/performance (including number of users) should be acceptable. The Logo should draw quickly on screen and the application should not hang while this is happening. Users should be able to test more Logo or send Logo to the robot while drawing is taking place on screen. Groups of up to 50 should not cause the application to crash or become slow.
- Security - user-names and passwords should be held securely and transmitted securely.
- Portability - the application should be accessible on common operating systems.
- Look and feel - the application should be user-friendly for younger users. Buttons should be large, icons should be colourful, the layout should be simple.

3.3.3 GUI Design

A rough design for the three main pages (login >> myFiles >> createLogo) is shown below in Figs 3.8, 3.9 and 3.10. The main application should be able to be delivered on a webpage or as a standalone application downloaded onto on the users' system.

Login screen

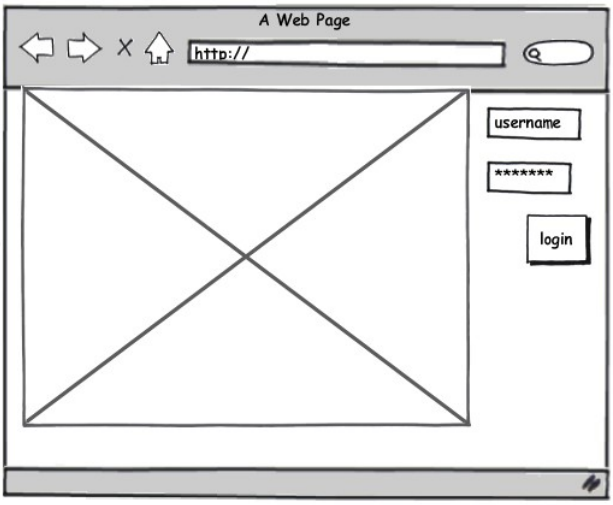


Fig 3.8 - GUI design for Log in screen

My Files screen

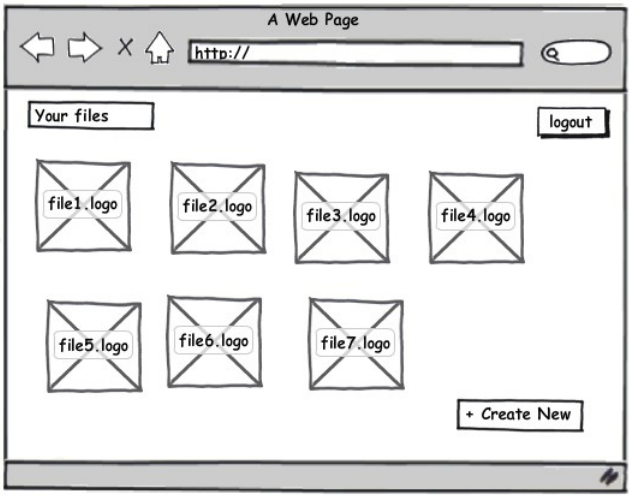


Fig 3.9 - GUI design for "My files" screen

CreateLogo screen

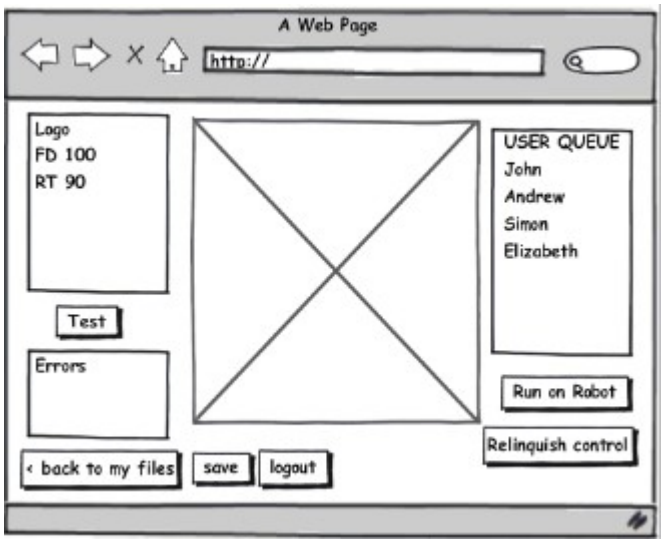


Fig 3.10 - GUI design for "Create Logo" screen

Passage between the screens is shown in Fig 3.11.

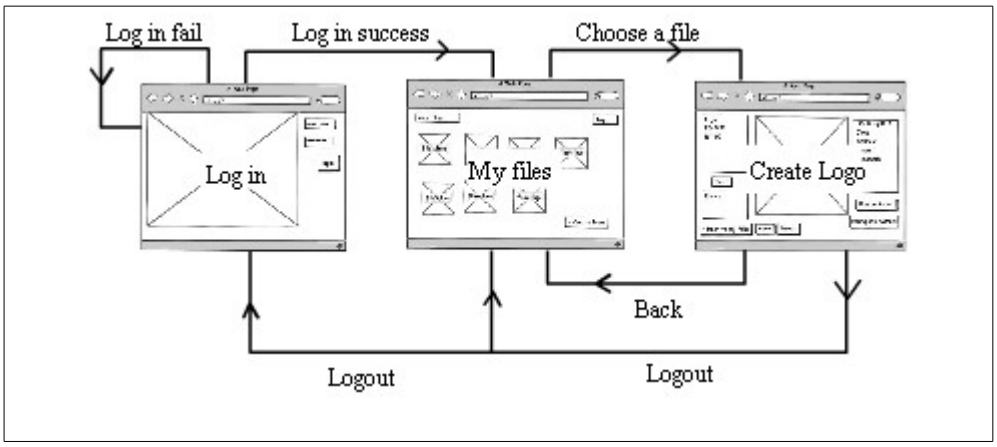


Fig 3.11 - Navigation between screens

3.3.4 Databases

It is my intention to allow the application to be used with an internet connection but without access to a robot, so that files can be loaded, edited and saved and Logo can be drawn on screen even if there is no robot to execute it. For this reason a database is needed to store login information and information regarding the files for to each user. Specifically, the minimal data to be stored is shown in Fig 3.12.

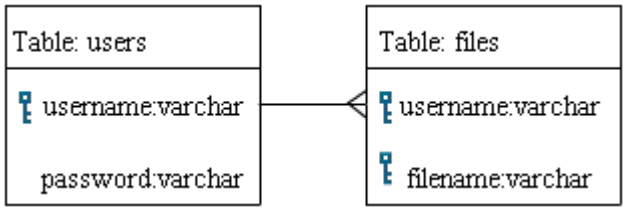


Fig 3.12 - Database tables

3.4 Iterations

To divide the project into manageable chunks the following iterations are to be used:

3.4.1 Iteration 1

Iteration 1 contains the core functionality for one unauthenticated user drawing Logo on screen and sending Logo to the robot to be executed. It does not include saving, or opening files and it relates only to one user. The functionality is shown in Fig 3.13.

A skeleton GUI	**From the Logo Editor package:**	**From the Robot functionality package:**
From the Logo interpreter package: UC_syntaxCheckLogo UC_interpretLogo	UC_testLogoLocally UC_stopLogoLocally UC_editLogo UC_undoEdit UC_redoEdit	UC_executeOnRobot UC_stopExecutionOnRobot

Fig 3.13 - Use cases for iteration 1

3.4.2 Iteration 2

Iteration 2 allows multiple users to communicate with the Robot. The users are still unauthenticated and they cannot save or load their files. It includes the use cases shown in Fig 3.14. In iterations 1 and 2 the 'my files' screen (Fig 3.9) is essentially redundant.

3.4.3 Iteration 3

Iteration 3 adds the external database for authenticating log-in and saving/loading files as well as the 'my files' screen GUI. The relevant use cases are in Fig 3.15.

From the User management package:	**From the File Management package: package:**	**From the User authentication**
UC_requestControl UC_relinquishControl UC_updateUserList	UC_viewMyFiles UC_deleteFile UC_saveFile UC_openFile UC_createNewFile	UC_logIn UC_logOut

Fig 3.14 - Use cases for iteration 2

Fig 3.15 - Use cases for iteration 3

3.4.4 Extensions

A number of extensions/variations on the project are to be kept in mind during design and development. The code and implementation choices should be chosen bearing in mind that the following should be feasible to implement at a later data:

i) The robot should be extensible to include a pen for drawing the logo on a sheet of paper on the floor (plus commands to raise and lower the pen).

ii) A chat room for all students working on one robot, for them to collaborate and swap ideas/help each other.

iii) Operation from the desktop - if a user *owns* a robot they could use the application alone on their own robot. This means that the physical location of the robot should not be overly restricted to an external machine.

iv) A number of the research papers mentioned in section 2 involve a purely web-based environment; the robot is viewed only through a webcam and live images are streamed to all users. It is desirable to keep this option open for later development.

v) Features by which Logo files can be communicated to others easily - perhaps printing or exported as jpg, perhaps they could be linked to directly as URLs somehow (like youtube's "copy this html to embed this video in your webpage" box), or perhaps easily incorporated in a school's website as some kind of applet/plugin. Perhaps even an 'export to facebook' facility or similar.

vi) Regarding deployment, I plan to develop a web-based system, but it would be good to keep the option open of being able to control the robot from a bluetooth enabled phone.

I do not intend to consider implementing any of the above in this project.

3.5 Development

I will be developing the software in NetBeans 6.8. I will be using StarUML [41] for all UML diagrams and Tortoise SVN [45] for version control of source code and documents. The repository is located at https://johngrindallproject.myversioncontrol.com/subversion/project.

4 The Logo interpreter

The Logo interpreter is a vital standalone component of the system. This separate chapter looks at the theory behind compilers/interpreters, usage of JavaCC and finally the technical details of my interpreter.

4.1 Interpreters and compilers – theory

4.1.1 Compilers and interpreters

A compiler is a program that translates code written in one language (called the source code) into another (the target file), typically a lower level (closer to machine code) language and often an executable binary file. The distinction between a compiler and an interpreter is that an interpreter does not produce a binary file, but rather analyses the source for correctness of syntax and then simply executes the commands one by one. [2] is quite an old reference book, and all examples are written in C, but it is still one of the best. Much of the rest of this chapter uses ideas, diagrams and code/pseudocode from the early chapters of this book, including our first summary of how compilers work:

> "...analysis consists of three phases:
>
> 1. Linear analysis, in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequences of characters having a collective meaning"
> 2. Hierarchical analysis, in which characters or tokens are grouped hierarchically into nested collections with a collective meaning.
> 3. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully" [2]

The next two subsections look at linear analysis (lexical analysis) and hierarchical analysis, also called syntax analysis or parsing. I use examples from Logo to make it more relevant.

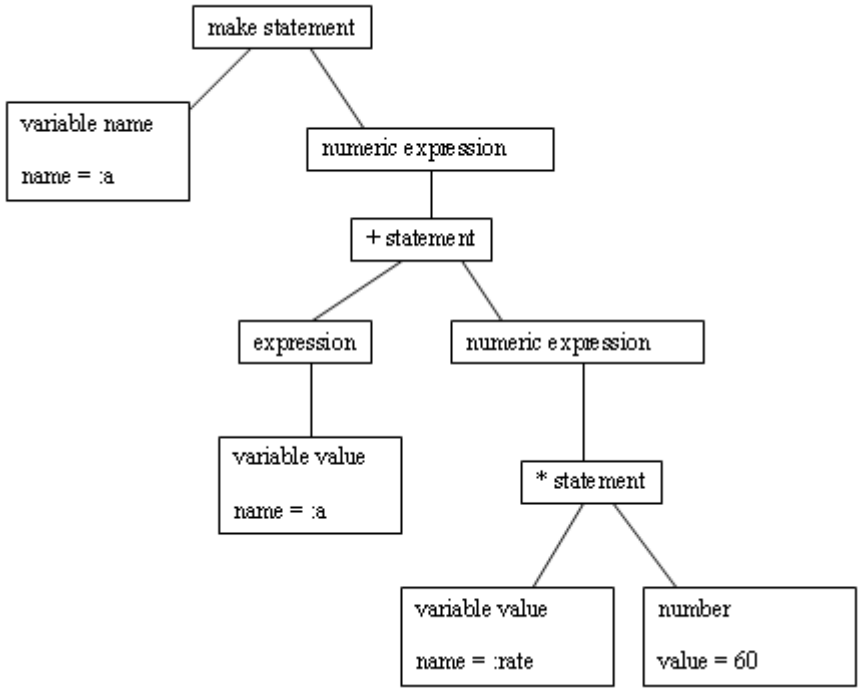
4.1.2 Lexical analysis

The first step in compiling or interpreting the program is to determine which characters join together into which words. For example 'm' 'a' 'k' 'e' must be recognised as the 'make' keyword, unless it is followed by a non-whitespace character in which case it could be a variable name, perhaps 'makeAmount'. This is achieved by placing the input characters into a stream and having the ability to read characters, store the partially recognised keywords and if necessary put characters back onto the

stream if reading them has confirmed that the word terminated *before* reading those characters. The lexical analyser also stores the type of each group of characters it recognises. A meaningful group of characters need not be a word in the English sense, so the more general word '*lexeme*' is used. The lexical analyser stores each lexeme and its type in a unit of code called a *token*. In Java, this will be a class. Ignoring characters like line returns, spaces and tabs, these tokens are passed to the next phase of the compiler.

4.1.3 Syntax analysis and the parse tree

To convert the list of tokens into something with meaning, a *tree* representation is used. Each token has a meaning only in the context of the tokens surrounding it, and it is the job of the syntax analyser to build this representation. For example, the program `make :a :a + :rate*60` should be converted to the more meaningful representation in Fig 4.1:



F*ig 4.1 - abstract syntax tree for the statement*
*make :a :a + :rate*60*

In this representation the two things that the 'make' statement needs are represented as its two children - the left represents the variable name into which the value of its right child (when evaluated) must be placed. The right child is the addition statement `:a + :rate*60` , and it also has children - namely the left hand child is evaluated (the current value of :a must be obtained) and added to the right

hand child. We can see that that storing the variable name token for :a in an object of type/*class* VariableName with one *attribute* called name will be useful, as will storing the number 60 in an object of *class* Number, with an *attribute* called value. Clearly the make statement's left child must be a particular kind of token - a variable name and not a number or another keyword. To describe these rules we use the idea of a grammar. To store the current values of the variables we need the idea of a symbol table and to store partially completed results of calculations such as `:rate*60` we need other storage space; we look at these next.

4.1.4 Context free grammars and EBNF

To describe the allowed structure of the parse tree a concept called a context-free grammar is used. A context-free grammar consists of a set of *tokens*, also called *terminals*, a set of *non-terminals* and a set of *productions*. The terminals are the tokens in our language - the keywords, numbers, the numeric operation signs and brackets for example. They cannot be further expanded and so are the terminating/leaf nodes in our trees. The non-terminals are composite statements such as 'MakeStatement' - it is non terminal because it cannot be a terminating leaf node in a parse tree since it has children. Productions designate the possible expansions of each non-terminal into other non-terminals and/or tokens.

A production rule (with tokens in bold) for the make statement in Logo is:

MakeStatement → **make** VariableName Expression

This means that a MakeStatement consists of the keyword/token **make** followed by a VariableName, followed by an expression. Variable names in Logo consist of a **colon** followed by an alphabetic name of one or more letter, so:

VariableName → **colon** (**<ALPHA>**)+

The **<ALPHA>** token is nothing but a letter of the alphabet a-z or A-Z. In my Logo I restrict variable names to containing just alphabetical characters rather than numbers - this seems typical.

The non-terminal 'Expression' is more complicated. Although they must all evaluate to numbers when the tree is walked, we can have different kinds of expressions containing variables, numbers, plus,

times, minus and divide and brackets. We can also have statements of the form "-7 " which is a negate statement rather than a minus statement with missing operand. We also have to take care of operator precedence for times and divide. Using [16] and [37] as very useful references, and a lot of testing and rewriting, I Fig 4.2 to be a correct grammar for such expressions:

Expression	→	MultiplicativeExpression (PlusExpression | MinusExpression)*
PlusExpression	→	**plus** MultiplicativeExpression
MinusExpression	→	**minus** MultiplicativeExpression
MultiplicativeExpression	→	UnaryExpression (TimesTerm | DivideTerm)*
TimesTerm	→	**times** UnaryExpression
DivideTerm	→	**divide** UnaryExpression
UnaryExpression	→	**minus** NumberExpression | NumberExpression
NumberExpression	→	VariableName | **number** | **leftbrace** Expression **rightbrace**

Fig 4.2 - BNF productions for expressions

Notice that an expression of the form **a + b * c** will be correctly identified as **a + (b*c)** because the "**a**" is matched to NumberExpression, and "**+ b*c** " is matched to PlusExpression.

This way of describing productions is called BNF (Backus-Naur form), and including the regular expression symbols +, ?, * we get Extended BNF (EBNF)

4.1.5 Creating the parse tree top down

Consider an example production rule: a → b | cd | e (in words "b; or c followed by d ; or e")

To encode this in a useful algorithm we associate a procedure to each of a,b,c, d and e. Recursively, imagine that we have reached a stage in the parsing at which we need to determine which (if any) out of b, cd or e we should expand 'a' into. The procedure associated to 'a' is thought of as an attempt to match the sequence of tokens so far unconsumed with possible expansions for 'a'. In pseudocode, this is shown in Fig 4.3.

```
procedure match_a(  ){
   //  try to match 'a'  with our token stream
   switch (first token out of the stream)  {
      case:  it matches the first token of 'b'
          match_b( )   // continue trying to match 'b'
      break;
      case:  it matches the first token of 'c'
          match_c( )   // continue trying to match 'c' and then 'd'
          match_d( )
      break;
      case:  it matches the first token of 'e'
          match_e( )   // continue trying to match 'e'
      break;
   }
}
```

Fig 4.3 - associating a procedure with each non terminal in a grammar

In this way the sequence of procedures used builds up the parse tree - reading from **left** to right and building the tree top-down, always trying to match the **left**-most possible expansion first. Such a parser is called LL (left left). The above looks **one** token ahead and tries to see what to match next. A parser that needs to look ahead '**k**' tokens before it can decide which to try and match is called an LL(**k**) parser.

4.1.6 The symbol table

We saw that when we walked the parse tree in Fig 4.1, we needed to keep track of the values of current variables. Logo is a simple language and variables are either global or local to a procedure, and procedures cannot be nested. For example, consider the program in Fig 4.4

```
1           make :a 1
2           make :b 1
3           to procName1()
4                 make :a 4
5                 make :a :a+1
6                 make :c 2
7           end
8           to procName2 (:d)
9                 make :a -1
10                make :b :d
11          end
12          procName1
13          procName2 7
```

Fig 4.4 - procedures in Logo

This has global variables :a and :b with values 1 and 1. Inside the first procedure call a new value of :a is declared to be 4, and then 5. The global one is now not accessible. :b is accessible, having a value of 1 and :c is a new variable that exists only inside procName1. procName2 also 'shadows' :a with its own value. *After* the first line is executed it also shadows :b too, with the value 7.

A symbol table needs some way of storing sub-tables for each procedure, keeping the global one too, and they also need to store function names, arguments and contents as well as variables. Visually, the above program's symbol table(s) could be represented as in Fig 4.5.

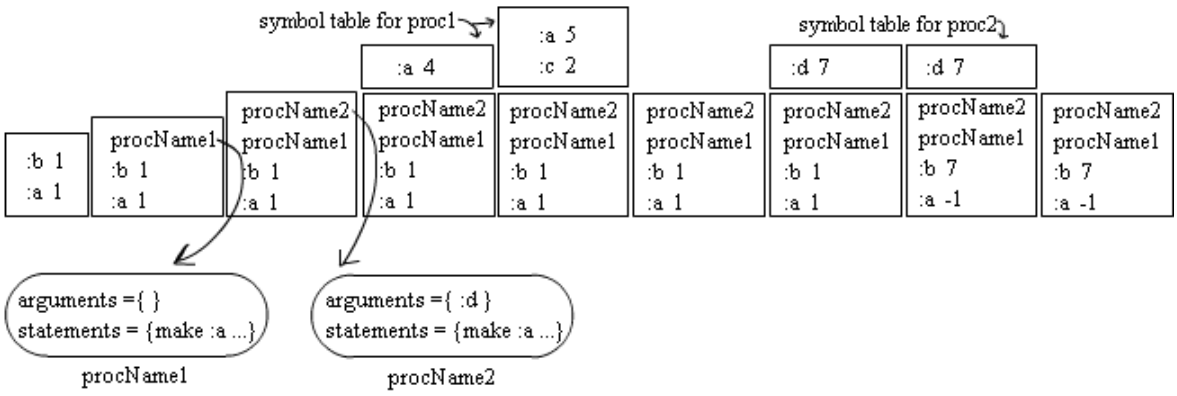


Fig 4.5 - symbol tables for Fig 4.4

In Logo all procedure definitions will essentially be in the main block.

4.1.7 The execution stack - walking the parse tree

Consider an extension of our example from Fig 4.1, namely a valid Logo program and its parse tree, in Figs 4.6 and 4.7:

```
make :a :a + :rate*60
fd :a
```

Fig 4.6 - example Logo program

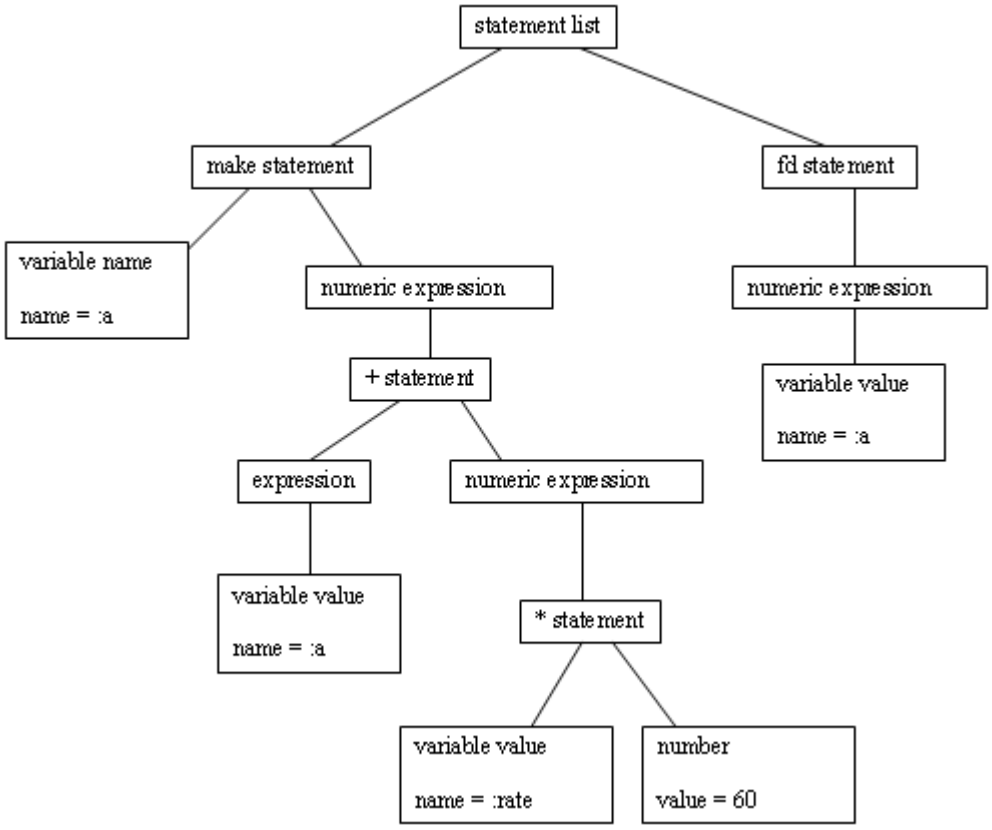


Fig 4.7 - parse tree for Fig 4.6

For simplicity let us assume that previously, :a and :rate are initialised to 2 and 0.5 respectively.

Walking the tree involves visiting each node in turn and doing whatever is necessary to execute the contents of that node. When we visit the make statement node we need to know two things, the variable name in question and the number to put into it. The way to get these pieces of information is to visit the

children first! So, the left child is visited and the name (note: not value) of the variable :a is pushed onto a FIFO stack. Then the right child is visited. To compute the value of the plus expression we need to know the values of the left and right operands again. So the tree is created in a top-down manner but must be visited in a bottom-up manner, depth first: {traverse the left subtree, traverse the right subtree, visit the node in question}.

In detail, the variable name :a is encountered and its value is looked up from the symbol table and pushed onto the stack, then when the times statement is visited, :rate's value from the symbol table and 60 are pushed onto the stack. The stack now contains {:a, 2, 0.5, 60} and we can visit the times statement. This will have to pop the two values of the stack, multiply them and push it back onto the stack, leaving {:a, 2, 30}. Next the plus node is visited, which pops 2 and 30, adds them and pushes the answer onto the stack, leaving {:a, 32}. And finally the make statement is visited which pops 32 and :a and assigns the value 32 to :a in the symbol table. Finally the forward statement is visited, which entails visiting its child. So, the variable name node if visited, looking up the value of :a, pushing that onto the stack (32). Finally the forward node is itself visited, and 32 is popped and the turtle moves forward 32 units. It is important to notice that sometimes ":a" represents a VariableName node into which the result of the right hand side must be placed, and at other times represents a VariableValue node, which when visited results in the current value of :a being looked up; these are two different types of node.

4.2 Java CC and JJTree

4.2.1 Introduction

Writing and testing extensible code that performs all of the functionality described above is clearly a difficult and long-winded task. It would be possible to write a lexical analyser and parser for a simple language like Logo, but from the point of view of extensibility (adding a new command) it is better to use a tool, and luckily tools exist to generate parsers for us. Lex and Yacc are the best known parser generators, outputting in C, and ANTLR [4] and JavaCC [38] seem to be the most popular for outputting in the Java language. ANTLR can output parsers in a variety of languages - Java, C, C++, C#, Python, Ruby, Perl, PHP, Actionscript to name some. After finding some nice tutorials ([3], [14], [27], [34], [37]]) and user guides I opted for JavaCC together with JJTree.

JavaCC and JJTree are BSD-licensed utilities for generating Java parsers. They generate all the Java

files (including the lexical analyser and the parser) needed to parse a program, given just the grammar file. An excellent introduction to JavaCC is found in [27]:

> "JavaCC itself is not a parser or a lexical anaylzer but a generator. This means that it outputs lexical analyzers and parser according to a specification that it reads in from a file. JavaCC produces lexical analysers and parsers written in Java. Parsers and lexical analysers tend to be long and complex components....specification files are easier to write, read, and modify compared with a hand-written Java programs. By using a parser generator like JavaCC, the software engineer can save a lot of time and produce software components of better quality." [27]

The JJTree utility is described on its homepage as :

> "JJTree is a preprocessor for JavaCC that inserts parse tree building actions at various places in the JavaCC source... By default JJTree generates code to construct parse tree nodes for each non-terminal in the language...JJTree defines a Java interface Node that all parse tree nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them... JJTree provides some basic support for the visitor design pattern. If the VISITOR option is set to true JJTree will insert an jjtAccept() method into all of the node classes it generates, and also generate a visitor interface that can be implemented and passed to the nodes to accept.... Another utility method is generated if the VISITOR options is set: { public void childrenAccept(MyParserVisitor visitor); } This walks over the node's children in turn, asking them to accept the visitor. This can be useful when implementing preorder and postorder traversals." [16]

Both JavaCC and JJTree are command line utilities - JJTree reads a .jjt file and generates nodes, classes for the visitor pattern (see below) and a .jj file. JavaCC takes the .jj file as an argument and generates the parser. For example:

```
C:\  jjtree fileName.jjt    // outputs fileName.jj
C:\  javacc  fileName.jj    // outputs entire parser
```

4.2.2 Grammar files

A .jjt file is provided to the JJTree utility, which lists the tokens and the productions required. Examples of two token definitions and two production procedures are shown in Fig 4.8, explained below.

```
TOKEN : { < MAKE    :    "make"                                    > }

TOKEN : { < DIGIT   :     "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"    > }


void fd_statement()#FdStatementNode:{}
{
    <FD>expression()
}


void vardefn()#VariableDefnNode : {Token t;}{
    t = <VARNAME>
    {
        jjtThis.addToData("name",t.image);
    }
}
```

Fig 4.8 - example token definitions in JavaCC

The #FdStatementNode statement means that we would like JavaCC to create a class of name FdStatementNode for us, in which we will store data and children of the 'fd' statement. As you can see, there will be one child - an *expression* node for the number of units we want the turtle to go forward, eg. 'fd 100' or 'fd (:x * 10)'.

The VariableDefn procedure illustrates how to pass in extra information into the procedure that is generated, namely that every VariableDefnNode consists of a 'variable name' token and that we want to store the actual name, as a string called “name” inside our node so that we can access the information later. For example, the name might be “:x”.

A class called SimpleNode is also generated for us by JavaCC. Each node generated extends this class. It deals with storing the children of each node as an array of nodes.

An extra base class called 'BaseNode.java' (written by the author) simply holds a hashmap into which I can store any relevant pieces of data such as the value of a number node, or the variable name in a VariableName node as in Fig 4.8. The `addToData()` method accomplishes this.

4.2.3 Walking the parse tree - the visitor design pattern

Using JJTree in addition to JavaCC generates code to use a *visitor pattern* to walk the tree. JJTree makes for us a visitor interface, as shown in Fig 4.9.

com.jgrindall.logojavacc.LogoParserVisitor

```
public interface LogoParserVisitor {
  public Object visit(SimpleNode node, Object data) throws
ParseException;
  public Object visit(ProgramNode node, Object data) throws
ParseException;
  public Object visit(ExpressionNode node, Object data) throws
ParseException;

  //  etc, for each kind of Node....
}
```

Fig 4.9 - LogoParserVisitor

The generic 'data' Object is used later to retrieve usable output from the Parser. JavaCC also puts 'visit' methods in each node, accepting a concrete implementation of LogoParserVisitor as the argument (see Fig 4.10)

```
public Object jjtAccept(LogoParserVisitor visitor, Object data) throws
ParseException{
    return visitor.visit(this, data);
}

public Object childrenAccept(LogoParserVisitor visitor,Object data) throws
ParseException{
    if (children != null) {
      for (int i = 0; i < children.length; ++i) {
        children[i].jjtAccept(visitor, data);
      }
    }
    return data;
}
```

Fig 4.10 - each node accepts the visitor

The visitor design pattern is used here. In [10], the intention of this pattern is described as representing "an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates...Use the Visitor pattern when...many distinct and unrelated operations need to be performed on objects in an object structure and you want to avoid 'polluting' their classes with these operations".

Since all the node classes are generated anew each time the grammar is changed it is possible to work on the visitor implementation and the grammar separately without worrying that they are too tightly coupled. The visitor design pattern is ideal for this. The previous two small methods are the only place where the visitor appears in the node classes - they simply send the visitor to their children and defer to the meat of the operations to the visitor class.

4.2.4 My implementation - The concrete visitor, the symbol table and the stack

My hand-written files are LogoParser.jjt, SymbolTable.java, FunctionWrapper.java, ILogoConsumer.java (see below) SymTableException.java, LogoParserVisitorImpl.java (the concrete implementation of the auto-generated interface) and BaseNode.java (containing just a Hashmap to store data). *All other* classes in the com.jgrindall.logojavacc package are auto-generated by JJTree and JavaCC and contain the `/* Generated By:JavaCC */` comment at the top. A full explanation of all methods is not possible here but points of interest are noted below. The grammar is found in Appendix C.

4.2.5 Class diagram

Class diagrams for the essentials of the JavaCC node structure and visitor pattern are shown in Figs 4.11 and 4.12.

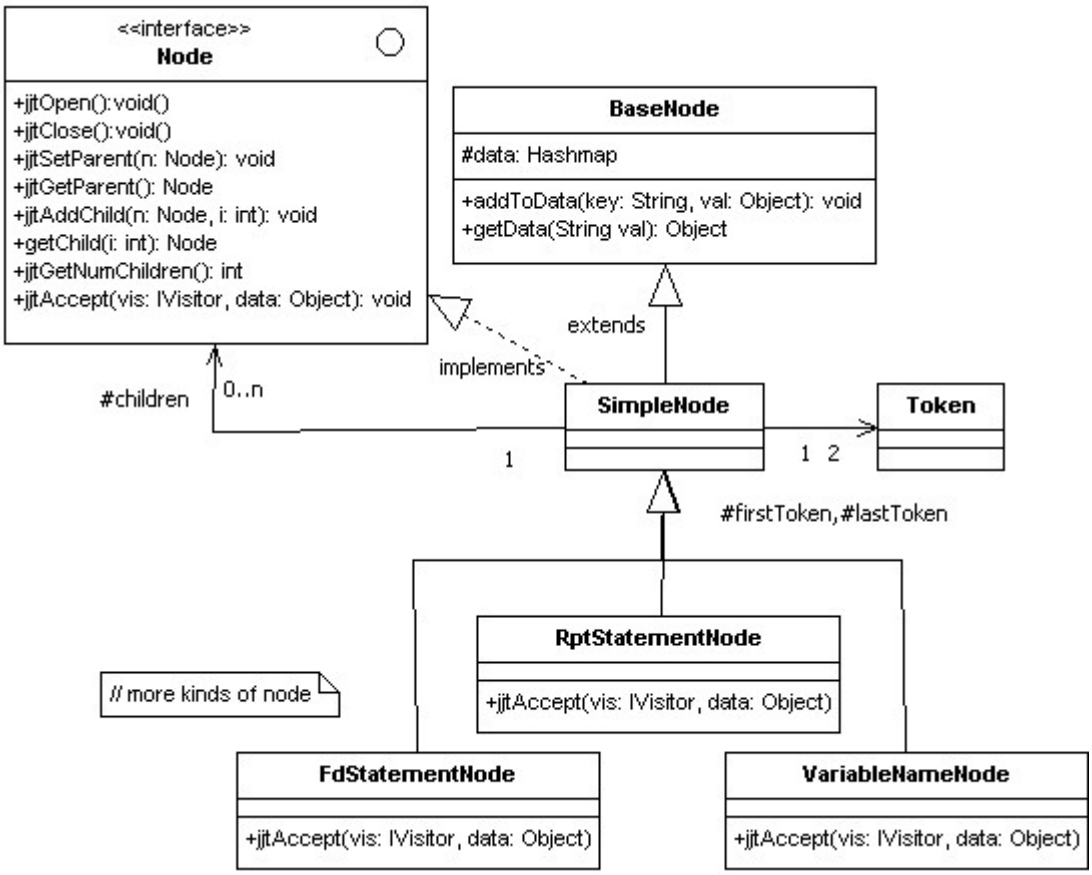


Fig 4.11 - basic JavaCC node structure (not all nodes shown). All classes apart from BaseNode are auto-generated by JavaCC

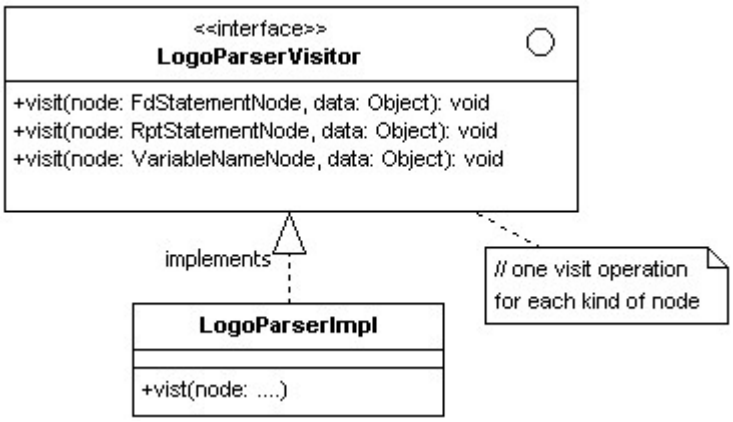


Fig 4.12 - JavaCC visitor interface and implementation. JavaCC generates the interface but the implementation must be provided.

4.2.6 Grammar file (LogoParser.jjt is found in Appendix C)

My syntax is slightly different to the standard Logo syntax - I have changed all keywords to lower case for ease of use and I have changed TO to 'proc' for procedure. I have also added brackets and commas to the signatures, like in most programming languages children are likely to meet in the future (Fig 4.13)

```
proc procName()                proc otherProcName(:a, :b, :c)
// statements                  // statements
end                            end
procName()                     otherProcName(:N+1, 10, -0.5)
```

Fig 4.13 - syntax for defining procedures

The input to a procedure is of type `(var_list())?` , where the `var_list` node is a comma separated list of variable names. Using brackets and colons makes my grammar LL(1) (For any A, p, q with A→ p and A→ q, p and q can never have the same first token).

Furthermore, I restrict what can be done inside a procedure and inside a 'rpt' statement node by using two different kinds of statement nodes (Fig 4.14)

inside_fn_statement_list → (inside_fn_statement)*

inside_fn_statement→fncall_statement | make_statement | fd_statement | rt_statement | rpt_statement

statement → inside_fn_statement | fndefine_statement

Fig 4.14 - restricting syntax for procedures - no nested procedures

It is maybe not necessary to do this - since I will make sure that procedures cannot be defined twice anyway, but I think it is useful from the point of view of having sensible error messages.

4.2.7 LogoParserVisitor Implementation

The following notes explain the salient points of interest in my implementation of the visitor pattern. See the source file com.jgrindall.logojavacc.LogoParserVisitorImpl.java for the methods discussed below. [37] was an excellent reference guide for learning the basics of visiting nodes.

- The stack is a java.util.Stack. We need only add and remove methods, and it is FIFO. It stores objects, and is used for variable names as well as numbers (stored as Double's).
- As explained above, visiting most nodes involves visiting its children first and so most methods

in the visitor class begin with visiting the node's children using :

```
node.childrenAccept(this, data);
```

- Visiting a VariableDefnNode (for example the left hand side of a '`make :a :a +1`' statement) involves visiting the children (actually there will be none) and simply putting the name on the stack to be used later (for example when the make node is visited and we need to know which variable to assign the value of the right-hand-side to).
- Visiting a VariableNameNode (for example the :a on the right hand side above) involves looking up the name of the variable (added in the jjt file `jjtThis.addToData("name",t.image)` - see Fig 4.8 for an example) and then retreiving the value from the Symbol table. It must have been added to the symbol table already if the code is semantically correct. Finally the value as a Double is pushed onto the stack for use later.
- When a make statement is visited we know that on the stack will be the variable name (from the left of the statement) and a Double (from the completion of visiting the right hand side; `:a + 1` in the above example). We pop those and assign the Double value to the variable name in the symbol table.
- Visiting plus, minus, times, divide nodes all work similarly. All of them are of the form **(expression) operator (expression)** and we know that the child expressions have been visited already, so the two values on top of the stack can be popped, combined (added, subtracted, multiplied or divided) and then that can be pushed onto the stack to be used later. Division by zero is caught and a ParseException is thrown. In fact ParseExceptions are thrown whenever anything goes wrong in parsing, the syntactic ones are written for us by JavaCC and it is up to us to deal with Logo specific errors.
- Repeat statements have two children - namely an expression representing the number of times to loop and a list of statements. In pseudocode, the handling of this is as in Fig 4.15:

com.jgrindall.logojavacc.LogoParserVisitorImpl.java:

```
public Object visit(RptStatementNode node, Object data) {
  loopNumNode = (ExpressionNode)node.children[0];
  statementListNode = (StatementListNode)node.children[1];
  visit(loopNumNode,data); //now the number of times to loop is on the stack.
  Object pop = stackPop();
  try to convert it to an integer > 0, else throw Logo specific error
  for(int i=1;i<=loopNumLong;i++){
          // visit them too, the right number of times.
          visit(statementListNode,data);
    }
}
```

Fig 4.15 - Visiting a rpt statement node.

- When a procedure *definition* is encountered the function name is the first child node. The other children provide the variable list node (which could be empty if the procedure has no arguments) and the list of statements to execute - a FnStatementListNode. These two objects (the variable list and statement list) are combined into one very simple object (with a getter and setter for each object) called a `FunctionWrapper` that is then added to the symbol table.

- When an 'execute function' node is visited the `FunctionWrapper` if retrieved from the symbol table using the name (the first child of the FnCallNode) and the number of arguments we have in the variableListNode from the `FunctionWrapper` is compared with the number of Doubles we pop off the stack. If they match then we enter a new block in the symbol table (see 4.5), add the local variables into the symbol table (they will shadow global ones with the same name), visit the StatementListNode from the `FunctionWrapper` and then exit the block in the symbol table.

- Fd and Rpt statements are the ones that we are really interested in! These give 'real' output (perhaps to the on screen turtle, perhaps to the robot, perhaps just to the console for testing). When the node is visited either a custom event could be dispatched and listened to by another class, or we could use the “data” Object that JavaCC generates for us (see Fig 4.10). Although the former might give less coupled code, I opted for the latter because the 'data' Object is already passed into the visitor for me by JavaCC and it doesn't seem good style for an algorithm

to include event dispatching. An object of interface type ILogoConsumer is used; any object implementing this interface has an `accept()` method that accepts an object of abstract type ALogoCommandObject. Subclassing this are LogoFdObject and LogoRtObject.

The ILogoConsumer objects will be responsible for doing whatever they need to with the commands (Fig 4.16)

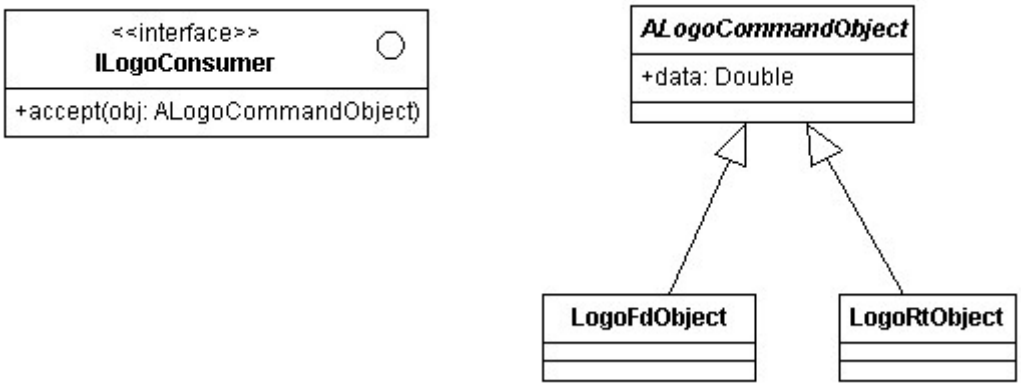


Fig 4.16 - The parser outputs ALogoCommandObjects to an ILogoConsumer

For example, a FdStatementNode is visited as in Fig 4.17:

com.jgrindall.logojavacc.LogoParserVisitorImpl.java:

```
public Object visit(FdStatementNode node,Object data) throws ParseException{
   node.childrenAccept(this, data);
   Double steps = (Double)this.stackPop();
   ((ILogoConsumer)data).accept(new LogoFdObject(steps));
}
```

Fig 4.17 - logic behind visiting a FdStatementNode

4.2.8 The symbol table

I adapted SymbolTable.java from [3] for my project. The SymbolTable class uses Hashmaps to map names of symbols to their meanings. 'Blocks' as shown in Fig 4.5 are defined in a helper class called StackedDictionary. The top block (the block in which code is currently executing) is referred to as 'top' and each block points to the next block (in Fig 4.5, the block below) using a reference called 'next'.

I adapted the files from [3], adding a completely separate Hashmap for functions (mapping procedure names to meanings) and also adding a maximum number of blocks (256). This is to prevent *Java* (not Logo) errors from Logo code such as that in Fig. 4.18:

```
proc recurse()
   recurse()
end
```

Fig 4.18 - infinitely nested recursion

When the stack depth goes above 256 I throw a SymTableException, which is caught by the LogoParserVisitor and a ParseException is thrown which is reported to the user. I also added an interface for the SymbolTable, ISymbolTable, so the LogoParserVisitor has the type as the interface not the implementation. I also added a *destroy()* method to StackedDictionary to clean up.

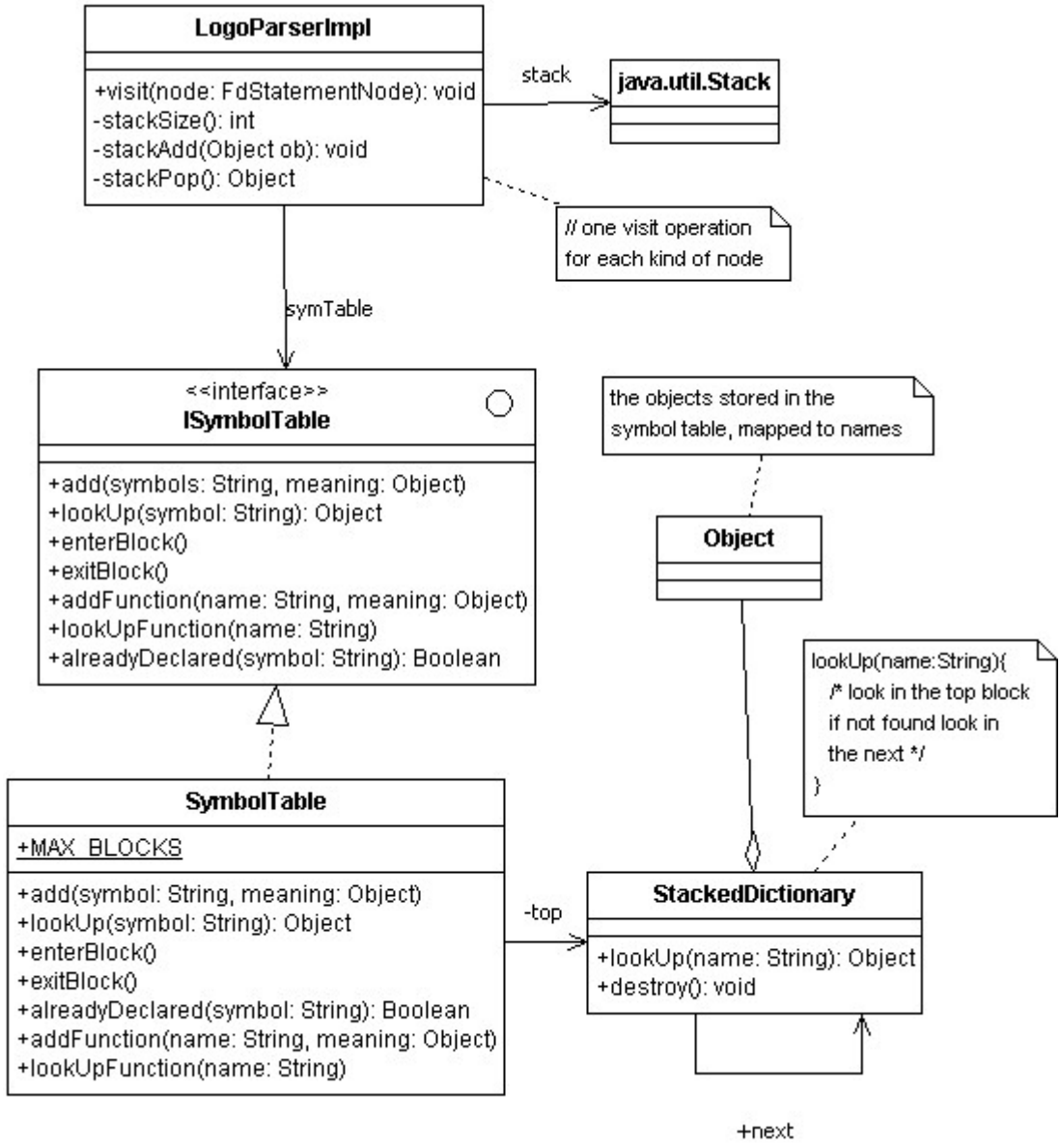


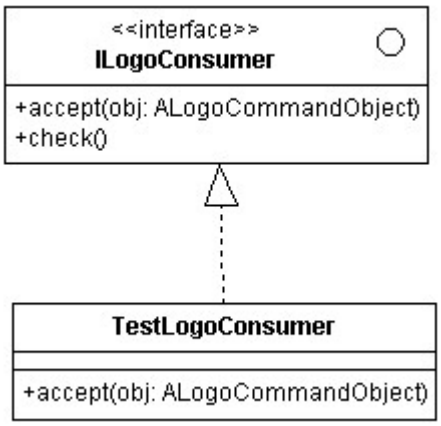
Fig 4.19 - implementation of the symbol table

A lookup from the symbol table consists of looking for the matching key in the topmost block and returning its value if found. The variable could be shadowing another variable of the same name in a block underneath. If not found, the block will call the same query on the 'next' block and so on.

4.3 Testing the Logo interpreter

This chapter contains details of the test programs I used to test the correctness of the JavaCC interpreter, covering the use cases UC_interpretLogo and UC_syntaxCheckLogo.

Since I don't want anything to happen to the output, I use a class called TestLogoConsumer (Fig 4.20) which implements the ILogoConsumer interface and implements the `accept()` method shown in Fig 4.16 by simply printing the first 'n' commands to the console. When 'n' commands are read an exception is thrown. This is so that I can check if `rpt 100000[fd 0.01]` programs execute successfully without having to restart the test.



4.20 - TestLogoConsumer outputs to the console when a Fd or Rt object is added.

I also include a 'dump' method in SimpleNode to output a node and its children as an XML string (see com.jgrindall.logojavacc.SimpleNode.java for details). In the early days of my grammar design/testing this was very useful for detecting exactly where errors occurred - XML being ideal for viewing tree structures with node names and attributes.

The Main class in the logojavacc package loads files test1.txt, test2.txt.... etc one by one and performs the following (Fig 4.21)

com.jgrindall.logojavacc.Main.java (pseudocode)

```
LogoParser parser = new LogoParser( fileInputStream for testi.txt ) ;
ProgramNode pn = parser.start();
// build tree

LogoParserVisitor v = new LogoParserVisitorImpl(parser);
pn.childrenAccept(v, new TestLogoConsumer());
// walk tree & output to console

System.out.println( pn.dump() );
// output XML
```

Fig 4.21 - testing the Logo interpreter

Test programs need to test for syntactic and semantic correctness, and they should include 'correct' Logo which should not give errors and 'incorrect' Logo which should give errors. My test cases, their purpose, expected and actual outcomes are contained in Appendix D. Ignore the 'Test on Robot' column until chapter 12. Test case test11.txt is the most interesting and test42.txt is also interesting. If spaces are missed out of Logo it still works - this is because my grammar (unlike most programming languages) restricts variables to starting with a colon. In fact, the program "make:a100fd:a" with no spaces at all works fine.

5 Lejos and the NXT robot

5.1 Introduction

Lejos [21] is a firmware replacement for the NXT brick, consisting of a virtual machine that resides on the robot, plus tools to compile Java code and to transfer the code to the robot to be executed and a user menu. The menu is found on a LCD screen on the front of the brick, containing menu items to view the compiled class files found on the robot, to delete one, to run one in the VM and to configure things such as bluetooth connectivity and the volume of sound.

Lejos is based on Java (only a subset is implemented) and supports the following:

- Threads & Synchronization
- Arrays, including multidimensional arrays
- Exception handling - try and catch
- Java types – float, double, long, int, byte, char
- Most of java.lang (for example Boolean, Integer, Double, String, StringBuffer, Math, Object, System, Thread, Throwable, Exceptions)
- Most of java.util (for example ArrayList, Hashtable, Queue, Stack, Vector)
- Most of java.io (for example some stream classes)
- Some of java.awt (for example Point, Rectangle)
- A Robotics API (See [20]) that contains useful methods for drawing Logo such as Motor.forward and Motor.stop

The memory available on the NXT is not as large as most mobile devices these days but is plenty for my purposes:

> "The NXT has 256kb of flash memory and 64kb of RAM. The first 32kb of flash memory is allocated to the leJOS NXJ firmware... The start-up menu occupies up to 48kb of memory that starts at address 32k (i.e. after the end of the firmware)... Objects are created in a heap that starts at the top of the RAM and grows downwards. The Java stack starts at the bottom of free RAM memory and grows up. A garbage collector frees memory used by unreferenced objects when the heap becomes full." [21]

5.2 Writing and executing Java programs on the NXT

The following steps are involved in writing and executing a Java program on the NXT brick:

- Download and install Lejos (http://lejos.sourceforge.net/nxj-downloads.php) and set the paths LEJOS_HOME and NXJ_HOME to point to the installation folder (for example on Windows,

C:\Program Files (x86)\leJOS NXJ). The installation folder contains a number of utilities to compile Java to .nxj bytecode, to upload from the client to the robot and a number of libraries such as bluetooth libraries (eg. bluecove.jar from bluecove.org)

- Replace Lego firmware on NXT brick with Lejos virtual machine. For windows systems see here: http://lejos.sourceforge.net/nxt/nxj/tutorial/Preliminaries/GettingStartedWindows.htm. A utility called nxjflash.bat replaces the NXT's ROM with the Lejos virtual machine.
- Write a .java file, using the Lejos API. I write the java code in NetBeans, with the Lejos plugin to get the correct syntax highlighting. This uses classes.jar in the Lejos install directory instead of the usual jar from Sun for performing the correct syntax highlighting.
- Compile the .java file into a .nxj class file. This is done by using the nxjc.bat file which calls javac, setting the classpath to classes.jar instead of the usual jar in the Java SDK install directory. This can be done easily from the command line but the sample project from Lejos provides an ANT file to compile and upload the .nxj file in one step (called “upload and run”).
- Upload the class file to the robot. Ensure that a bluetooth dongle has located the NXT and use the command line Lejos utility nxjupload.bat (or use the ANT file to compile and upload in one go)
- Execute the file on the VM (press the orange execute button on the NXT brick)

See [21] for more details/tutorials.

5.3 Communicating with the Robot

Only syntax-checked Logo should ever make it off the client's machine to the backend/robot, but once the Logo has been syntax checked there are still a number of possibilities:

i) Send pure logo from the client and interpret it on the robot. This seems a very bad idea because I suspect the final interpreter is going to be quite large, and contain many classes that are unsupported by Lejos. Also it is virtually impossible to debug programs once they are running on the brick; error messages are complex and the LCD has a very limited size.

ii) Send some representation of the parse tree and variable table to the robot (having checked that its syntax and semantics are ok) and then walk the tree. This suffers from pretty much the same problems as i).

iii) Fully expand the logo on the client, convert everything into a long list of fd, rt commands (potentially thousands of these if the user has written rpt 10000 [fd 1 rt 1], then strip off the first few hundred of them to ensure I don't run out of memory on the robot and execute them. This has the huge disadvantage that teachers might actually want infinitely executing programs to teach programming, for example rpt 10000 [fd 0.01 rt 0.01] makes a nice circle, and recursive programs are even more interesting.

iv) Send the commands in batches and have some kind of 'finished message' that tells the backend to send the next batch of messages. In this way the robot would store a fixed number of commands, execute them and then ask for more. The batch size could be 1 for simplicity, or a few hundred if the motion of the robot is discontinuous using a batch size of 1.

v) Convert the Logo into a java file (write a file using java.io), compile it using nxjc and upload it using nxjupload. I saw [1] at an education technology conference and I believe this is how their product works. This requires someone to press the orange button each time a program starts, and would presumably require some kind of licence from Lejos to distribute the utilities (although I have not checked this - it may be allowed). It is a nice solution because communication is 'one-time' and also the robot can be taken away from the computer and the same .nxj file used, but it may be very time-intensive to write the Java files correctly - especially if more commands such as if/else/function return statements etc are going to be added later.

My chosen solution is iv) - it will allow easy communication to the robot (from the users point of view), and translating from Logo to Java as in v) may be too time consuming for this project.

6 Implementation choices

The requirements for the Logo Editor are that it:

- has enough skin/styling support and nice components to make it user friendly for younger uses
- can interface to the Logo interpreter and the back end components
- It should support multi-threading so that the Logo can be drawn on screen and the user list can update without the application freezing.

Many technologies exist to create client-side web applications - for example Adobe Flex and Microsoft's Silverlight are capable of making very attractive GUIs, whilst Java and its recent addition JavaFX are more powerful. It is also a possibility to implement part of the front end in HTML (for example the log in screen and 'my files' screen and use another technology to create the application itself.

In the end I decided to implement the entire front end in Java. Flex is not multi-threaded and although the Java parser could be accessed by Remote Method Invocation from Flex, or - if embedded in a webpage - a 1×1 pixel Java applet I did a little testing and research and both of these were not satisfactory. I was able to get **Flex → Javascript → Java applet → Javascript → Flex** communication working, but it became very slow the larger the amount of Logo commands outputted. Also, if a possible extension is to allow desktop launch of the product (consider Spotify and Skype for example) neither of these will be satisfactory because no Javascript would be available.

Silverlight *is* multi-threaded and can be scripted in C#, and I'm sure the JavaCC parser *could* be translated into C# so this was an option. But given the work involved and the fact that the Robot is executing Java anyway, Java seemed the sensible choice - especially since serialised objects can be easily sent around the system if the same language is used for the front and back ends.

JavaFX was also an option - essentially JavaFX consists of easily scriptable user interface constructs above a Java foundation, and is intended for quick creation of modern looking user interfaces. But after some research it seemed as though I would end up having to mix in Swing anyway to get the full functionality I want. For these reasons I opted for Java running on the robot, Java Swing running the GUI and front end and so Java seemed a natural choice for the back end too, especially since serialized objects can be easily sent from Java to Java.

All users will need to maintain a real time connection to the host machine of the robot (called the admin

server in Fig 3.1) to facilitate the user list. It will not be acceptable for there to be a time lag between someone relinquishing control and other users seeing this. Given also that chat is one possible extension, some kind of socket connection is required. This admin application will also need to be multi-threaded, to deal with multiple users requesting/relinquishing control simultaneously. HTTP requests are sufficient for logging in and loading/saving files (in iteration 3) since these do not need to be in real time or persistent.

Communication from the front end to the back end can be achieved by Sockets, Remote Method Invocation/Remote Procedure Calls (RMI/RPC) and Servlets. As described above, HTTP requests through a short-lifetime servlet will be fine for logging in and retrieving lists of users' files, but RMI/RPC or sockets are the technologies of choice for maintaining the responsiveness of the application with regards to the user list. In the end I opted for a Socket connection.

7 Iteration 1 - Design

This section begins by looking at the MVC design pattern. This is followed by the designs for iteration 1, namely: the panel in which the user writes Logo, drawing of Logo on-screen and sending Logo to the robot.

7.1 The MVC Design Pattern & PureMVC

7.1.1 The Classic MVC framework

A common solution to the problems encountered when developing a user interface is to use a set of design patterns collectively known as MVC. According to [10],

> "The Model/View/Controller (MVC) triad of classes is used to build user interfaces ... MVC consists of three kinds of objects. The Model is the application object, the View is the screen presentation and the Controller defines the way the user interface reacts to user input...MVC decouples views and models by establishing a subscribe/notify protocol between them...Whenever the model's data changes, the model notifies views that depend on it...This approach lets you attach multiple views to a model to provide different presentations. You can also create new views for a model without rewriting it" [10].

Java Swing also uses the MVC idea (albeit with the view and controller collapsed into one class). A JButton contains a ButtonModel; a JList uses a ListModel to store the data regardless of the exact view component used to display it. A closer look at MVC (Fig 7.1) shows the flow of information and responsibilities:

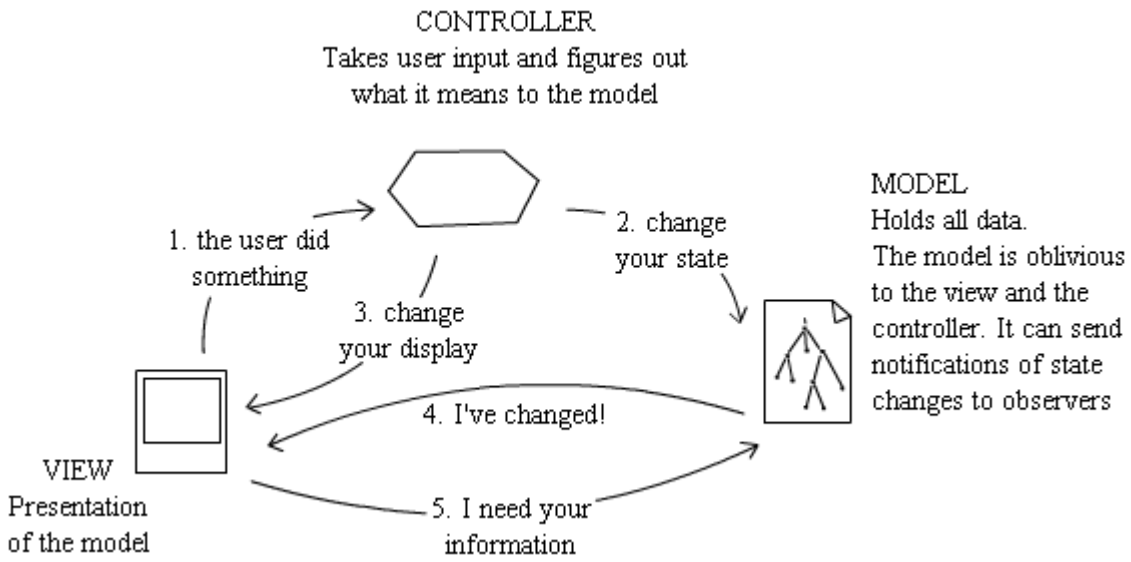


Fig 7.1 - MVC design pattern (based on image [9])

I was keen to use an MVC framework to build the GUI. One that is gaining popularity amongst web developers is called PureMVC ([12]), so I decided to investigate further and base my GUI upon this free, open source (Creative Commons) framework.

7.1.2 Introduction to PureMVC

As described by its author, "PureMVC is a lightweight framework for creating applications based upon the classic Model, View and Controller concept. The PureMVC framework has a very narrow goal. That is to help you separate your application's coding interests into three discrete tiers; Model, View and Controller. This separation of interests, and the tightness and direction of the couplings used to make them work together is of paramount importance in the building of scalable and maintainable applications ... With the right separation of interests, we may be able to reuse the Model tier in its entirety and simply fit new View and Controller tiers to it. " [13]

PureMVC ports exist in many languages including Actionscript 3, C#, Javascript, PHP, Objective C, Python, Ruby, C++ and of course Java.. The framework itself functions in a way that is independent of the specific language used and is explained in more detail below. This explanation borrows heavily from the PureMVC documentation ([13])

7.1.3 Proxies, Mediators and view components

The PureMVC framework contains a number of Singletons, namely the *Model*, *View*, *Controller* classes and the *Façade*. The Facade isn't very complicated, it simply passes any calls the programmer wishes to make to the Model, View or Controller singleton. A 'multi-core' version also exists which uses Multitons (a named map of Singletons) to allow multiple 'applications' to be run in the same virtual machine, but I will be using the single-core version.

The "models" in a PureMVC application consist of a number of named *Proxies*. A Proxy is nothing but a surrogate that has an underlying data object (this could be simply a string or integer, or it could encapsulate and provide access to a more complicated data or the result of a call to a database). The Model Singleton in a PureMVC application stores a list (in fact hashmap) of these proxies so that they can be easily retrieved. The diagram to have in mind is Fig 7.2 (from [13]):

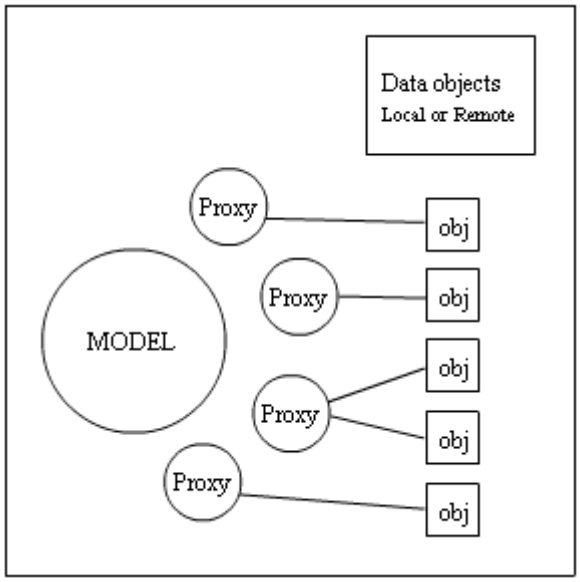


Fig 7.2 -PureMVC Model and Proxies - the Model singleton stores a list of Proxies that hold the real data objects.(From [13])

In a similar way to how a Proxy stores application data, the framework uses classes called *Mediators* to "steward" a UI view component (for example a Swing component). [10] gives the following definition of a Mediator: "Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently". The intention is that no PureMVC code exists in the component classes, so a Mediator might add an ActionListener to a button, listen for interactions and when the action is performed forward this fact into the PureMVC framework which will deal with it. Any changes to the UI are accomplished by the relevant Mediator hearing another message, and telling its stewarded component to update its visuals (see Fig 7.3)

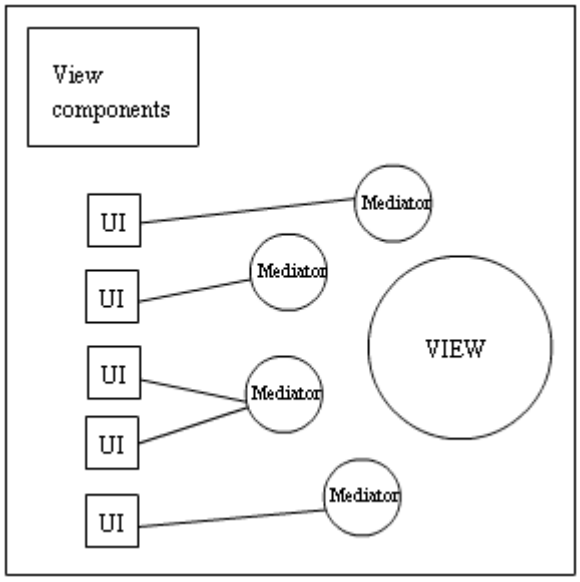


Fig 7.3 -PureMVC View, UI components and their Mediators. All communication to the UI components must be done via the Mediator.(From [13])

The View Singleton simply stores named references to Mediators, which can be 'retrieved' by name after they have been registered with PureMVC. Since a GUI will typically involve a composite tree structure of UI components (a panel inside another panel inside another for example), Mediators also register Mediators for their child components. Mediators are passed a reference to their view component (in my case all of these will be Java Swing components) in their constructor.

7.1.4 Notifications

Information is transported around the application using *Notifications*, which consist of a string for their name (for example 'DRAW_LOGO' or 'LOGIN_BUTTON_PRESSED') and a body (any Object) which is the information we wish to contain inside the Notification (for example the Logo we wish to draw or the username/password we entered). Notification names are just constant strings and are normally listed as static variables in the Facade so that they are easily accessible from anywhere. Notifications are rather like events apart from the fact that they are not tied in to actually being Java events. They function as follows:

i) Mediators can send Notifications. For example when a login textfield is filled in and a button pressed a Notification called LOGIN_BUTTON_PRESSED might be sent, containing the username and password.

ii) When they are registered with the PureMVC framework, a Mediator gives an array of Notifications that they are interested in, as the return of a method called `listNotificationInterests()`. When a Notification is sent from elsewhere in the system, the Mediator's `handleNotification()` method is called. It is passed the Notification as an argument and it responds (for example it might update its view component). Any data in a Proxy could be pulled in if needed to accomplish this (since the Model stores a list of Proxies - see above)

iii) Proxies can send Notifications too, for example to instruct any listening Mediators to update their visual display on the basis of some new model state.

iv) It is against the spirit of PureMVC, although perfectly possible, for a Mediator to retrieve another Mediator and mess around with it directly.

Following on from point ii) above, from [13], "PureMVC implements the Observer pattern so that the Core actors and their collaborators can communicate in a loosely-coupled way, and without platform dependency ...many Observers may receive and act upon the same Notification"

7.1.5 Commands

The more complex logic of the application is encapsulated in classes called *Commands*. Each Command object has an 'execute' method which is triggered when the Command is executed. In the Facade we can link Notifications together with Commands. For example, LOGIN_BUTTON_PRESSED might be tied up to a LoginButtonPressedCommand by using the line:

```
registerCommand(AppFacade.LOGIN_BUTTON_PRESSED, LoginButtonPressedCommand.class);
```

When the LOGIN_BUTTON_PRESSED Notification is sent (from anywhere in the application but perhaps from the Mediator of a log in panel) both of the following happen:

i) any Mediators that list LOGIN_BUTTON_PRESSED in their `listNotificationInterests()` method will handle the Notification as described above

ii) the execute method of the `LoginButtonPressedCommand` is called.

Commands can then perform the business logic - they can retrieve and manipulate data from Proxies, they can send other Notifications (which might be heard by some Mediators, or could result in other Commands being executed). They can also register new Mediators and Proxies (perhaps if a new on-screen object has been created) or remove them (if an object has been destroyed).

7.1.6 Initialization

A PureMVC application is initialized by building the bare GUI first (classes that contain swing components for example). There will be no business logic in these classes at all. Once the application is built we use the Facade once more to send a StartUpCommand. Inside the StartUpCommand we register proxies (our model) first, using lines such as

```
AppFacade.getInstance.registerProxy (new SomethingProxy());
```

Once the model is built we prepare the view, by a line such as:

```
AppFacade.getInstance.registerMediator (new
MainParentMediator(mainSwingComponent));
```

Mediators go through the children of their GUI components and create mediators for them as well. Each Mediator can use any Model state (Proxy) to put itself in order because these have already been registered.

7.1.7 Class diagram and example sequence diagram

To clarify the above, a class diagram showing the essential structure of PureMVC and an example sequence diagram for the LoginButtonPressed interaction described above is shown in Figs 7.4 and 7.5.

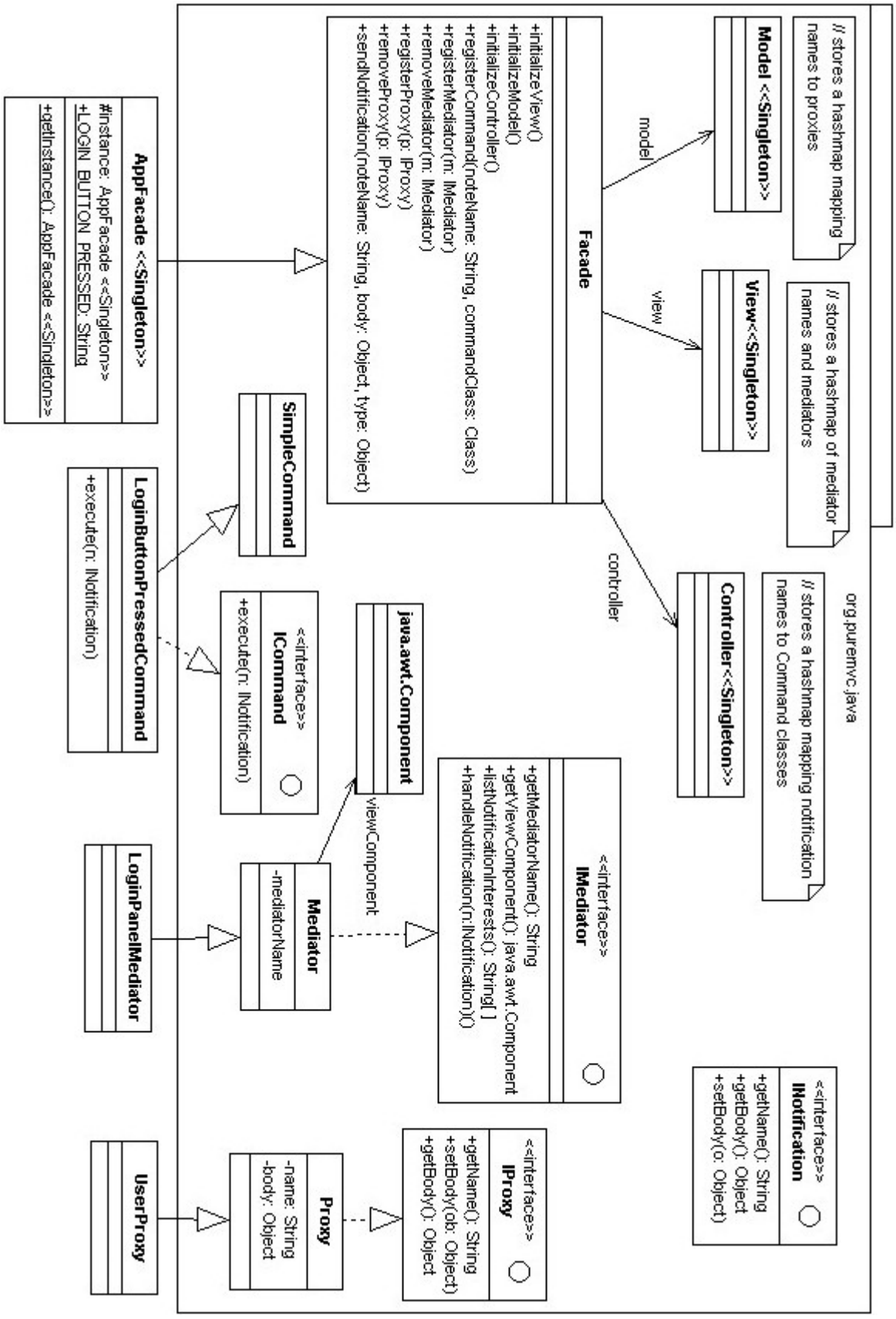


Fig 7.4 - Essentials of PureMVC

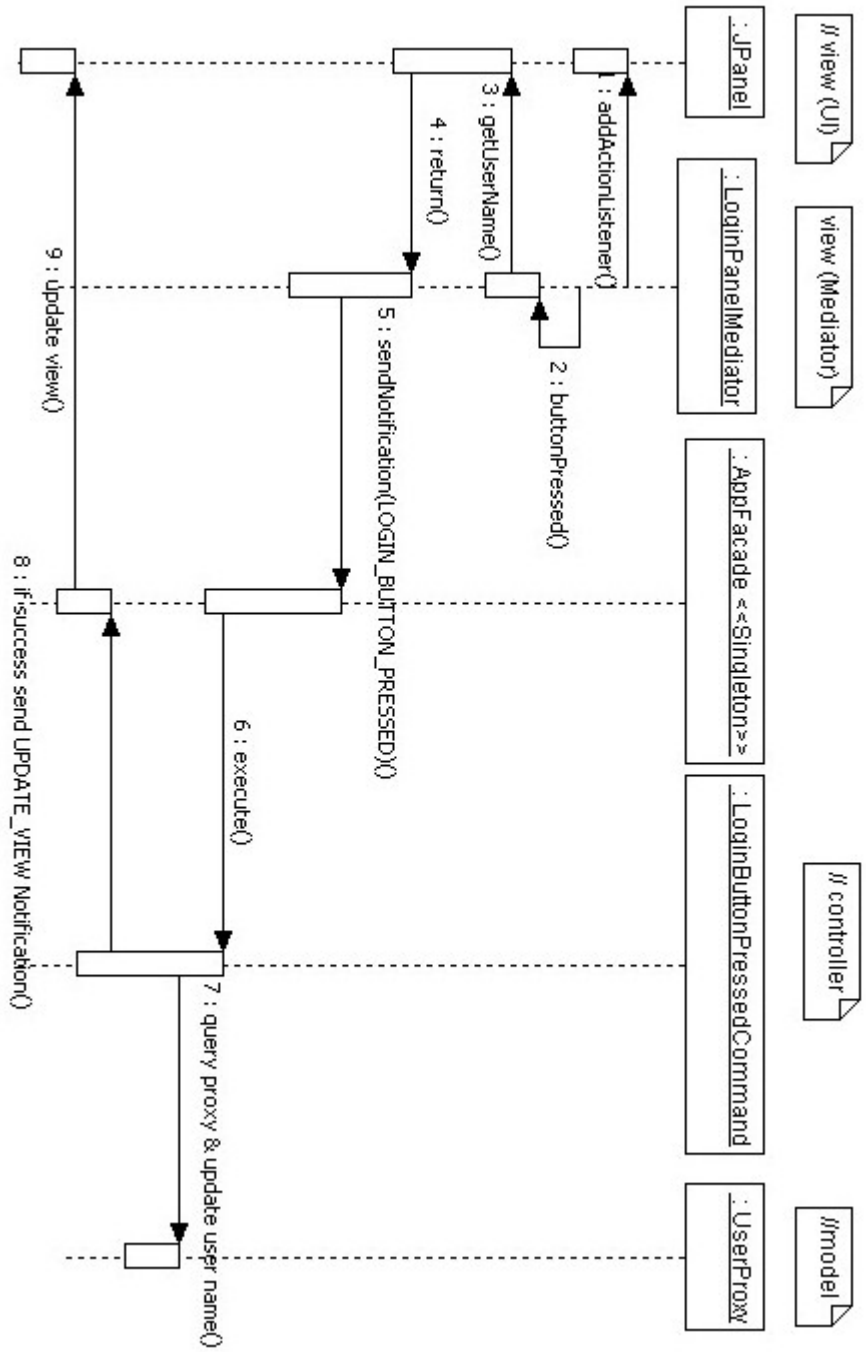


Fig 7.5 - Typical example sequence diagram for a PureMVC interaction

1. The Mediator of the Swing component attaches a listener to the login button.
2. Following a user interaction, the listener fires
3. The Mediator gets the text the user has entered
4. The UI component returns the text to the Mediator
5. The Mediator sends a Notification with name LOGIN_BUTTON_PRESSED, with the username (and presumably password etc) as the body. The AppFacade uses the PureMVC framework to map the Notification to a Command, and the command is instantiated.
6. The command is executed.

7. The Command can read/write any proxies to determine whether to let the user log in. The diagram above is simplified, perhaps a database call is made and some time later the Proxy sends another Notification. Perhaps another Command is used.
8. However the logic works, eventually come kind of UPDATE_VIEW Notification will be sent to update the view components (perhaps to forward the user to the next screen)

7.2 Basic Structure of the application

The three basic screens (Login, MyFiles and the Logo creation screen) will be contained in a Container with CardLayout. In iteration 1 the focus is on the "Create Logo screen", containing the components shown in Figs 7.6 and 7.7:

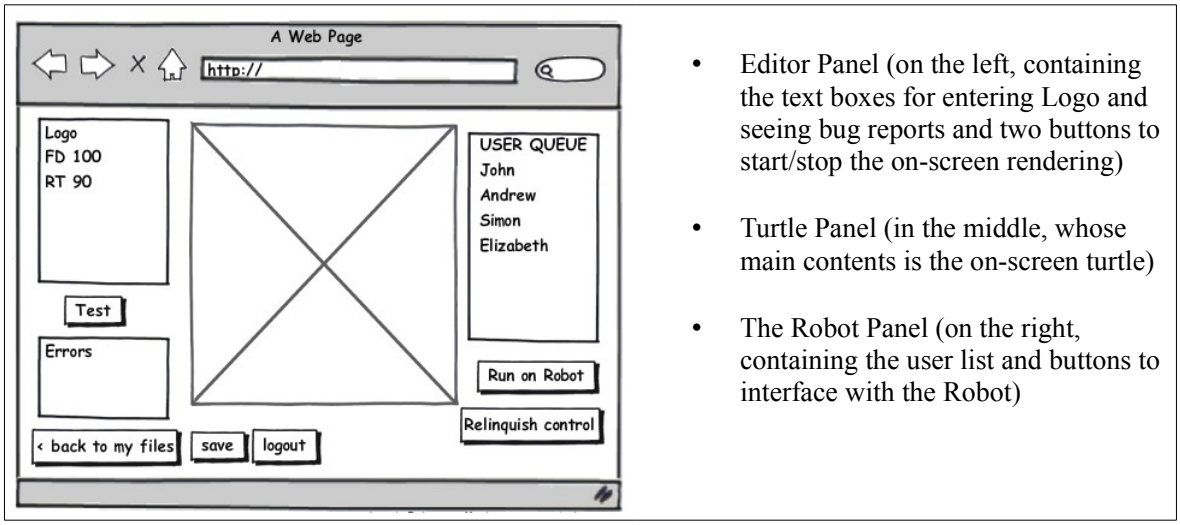


Fig 7.6 - layout of the "CreateLogo" screen

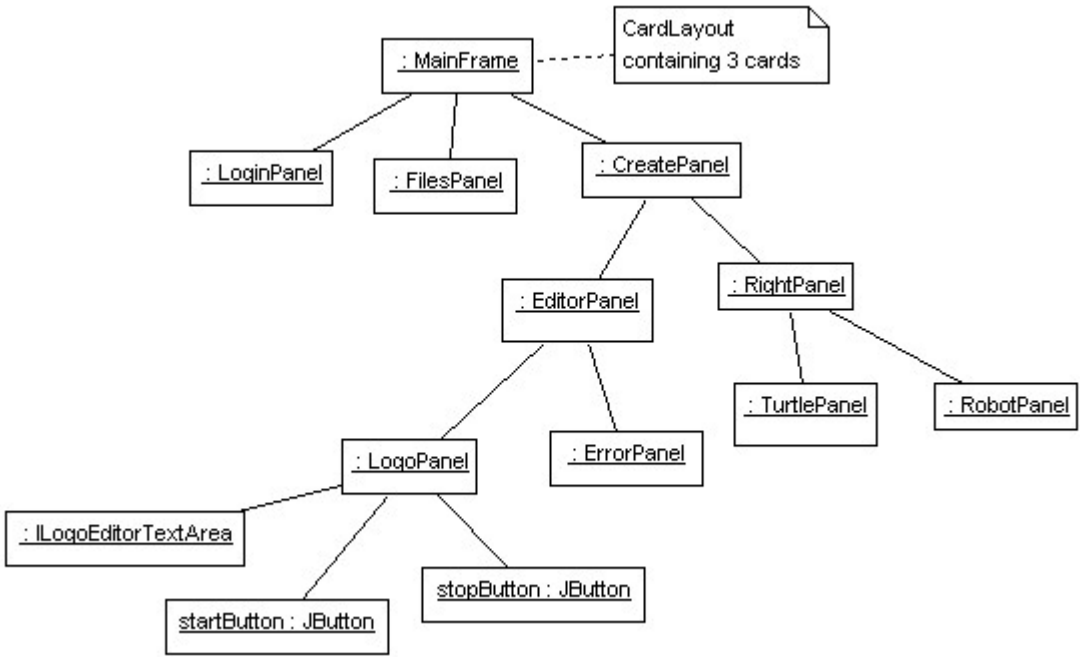


Fig 7.7 - hierarchy of UI components in the "CreateLogo" screen.
The LoginPanel and the FilesPanel functionality is contained in iterations 2 and 3.

7.3 The Logo Panel (UC_editLogo)

The Logo Panel represents the text area in which the user writes Logo and associated buttons. Is located at the top left in Fig 7.6.

7.3.1 The Logo text area

The text box into which the user writes Logo will implement an interface I will call ILogoEditorTextArea - this could enable different kinds of editors to be swapped around at a later date. Ultimately my concrete LogoEditorTextArea will be a subclass of JTextArea, containing getText() and setText() methods.

For user-friendliness I want to be able to highlight lines of text to show errors and also to show line numbers, so that the error messages from the Parser make more sense. The highlight() method is used for this. The relevant class diagram for such a design is shown in Fig 7.8:

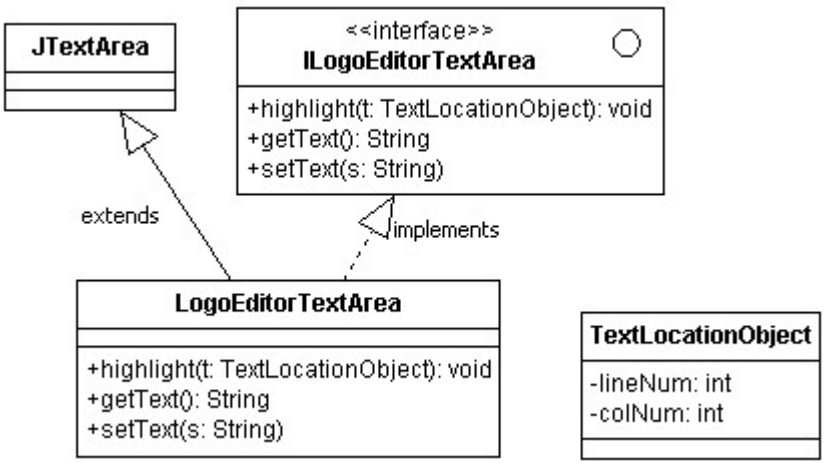


Fig 7.8 - basic design of the text area into which the user enters Logo

The TextLocationObject is very simple, and will just keeps a record of the position of the user's error so that bugs can be reported; an error has line number and column number.

7.3.2 The Logo text area Mediator

As we have seen, in the PureMVC framework, every visual component is 'stewarded' by a Mediator. The application will perform background syntax checking so that users don't have to click 'draw' every time they want to check their syntax. It also needs undo/redo functionality.

One huge mediator that handles both of these tasks is a possible (bad) design, as is using inheritance . The problem with inheritance is that the two functionalities are unrelated and having one extend the other doesn't make sense and is inflexible (the functionalities cannot be switched off or other functionality easily added). Java does not support multiple inheritance. Composition could be used to add an array of objects to a class, enabling them to be set (turned on or off) one by one.

It was finally decided to add the undoable behaviour to a subclass and to use the so-called *Decorator pattern* to add the background syntax checking. This makes sense because the undoable behaviour is an extension to the text functionality, whereas the background checking thread is more of an 'extra' decoration. It should be possible to add the decoration (and other future decorations) to any Mediator, allowing functionalities to be switched on and off simply.

The Decorator design pattern, is able to "Attach additional responsibilities to an object...a flexible alternative to subclassing for extending functionality." [10] The relevant class diagram is Fig 7.9:

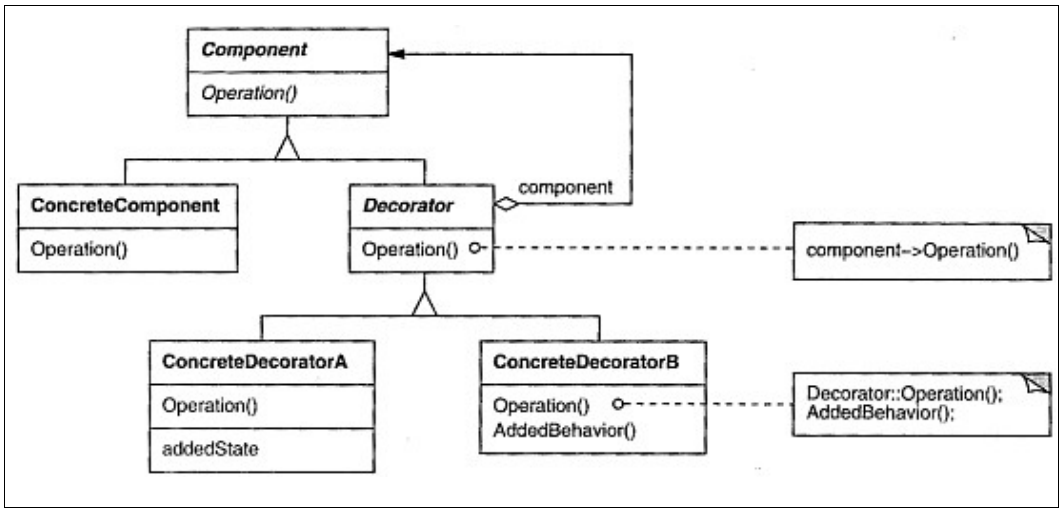


Fig 7.9 - the Decorator design pattern (diagram from [10])

The Decorator pattern works by wrapping a base object, for example we want to be able to write:

```
new BGCheckLogoPanelDecorator(new LogoPanelMediator()  );
```

or

```
new BGCheckLogoPanelDecorator(new UndoableLogoPanelMediator()  );
```

A class diagram and explanation follows, in Fig 7.10.

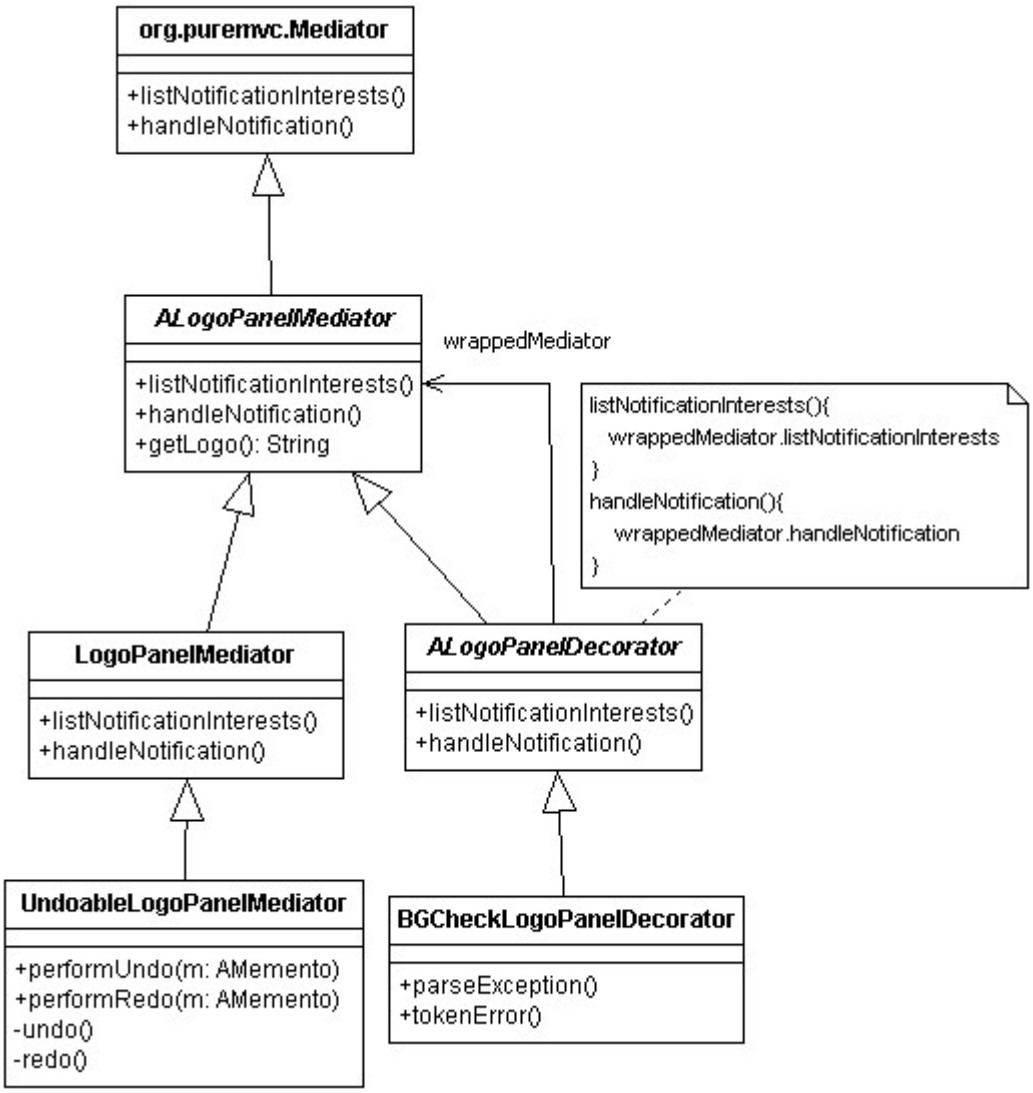


Fig 7.10 - using the Decorator pattern for the Mediator of the Logo text area

ALogoPanelMediator is an abstract base class for *all* our possible mediators. ALogoPanelDecorator is a base class for all the possible decorators. Its function is to act as a ALogoPanelMediator (since it extends it, it can be treated as such elsewhere in the code) and to also hold a reference to an ALogoPanelMediator. Decorators delegate all relevant method calls to their contained mediator, called wrappedMediator in Fig 7.10 and 'component' in Fig 7.9. For example, the important "listNotificationInterests" method (the method any Mediator uses to inform other classes of the messages it wishes to listen out for) will need to be implemented in ALogoPanelDecorator as:

```
public String[] listNotificationInterests(){
   return wrappedMediator.listNotificationInterests();
}
```

7.3.3 Background syntax checking

Background syntax checking must happen in a thread running off the main event dispatching thread since parsing a large Logo file can involve some heavy processing. It must interface with the LogoParser described in section 4. We do not need to actually visit the nodes of the tree, we just need to called LogoParser.start( ) and catch any errors - specifically ParseExceptions and TokenMgrErrors (both automatically generated for us by JavaCC).

So, the BGCheckLogoPanelDecorator will need to instantiate and start a thread which is responsible for checking the syntax. In pseudocode, shown in Fig 7.11.

```
public void run(){
   while(true){
   try {
       syntaxCheck();
   }
   catch(any exception){
       // report the error back
   }
   // wait until next time to syntax check
```

Fig 7.11 - inside the BGCheckLogoPanelDecorator

This thread will need to get hold of the logo the user has entered somehow, and also report back to the application any errors. To obtain the Logo an interface called ILogoSource is used; any class implementing this interface should be able to get hold of the Logo and pass it to the caller. For relaying error messages to the application another reference to some error handling object could be used, but to make the design less coupled dispatching error events to a listener (like a button event being heard by its parent) will be a better design. So parse exception events will be dispatched by the background checking thread and listened to elsewhere. The new classes in the design are shown below, Figs 7.12, 7.13 and 7.14:

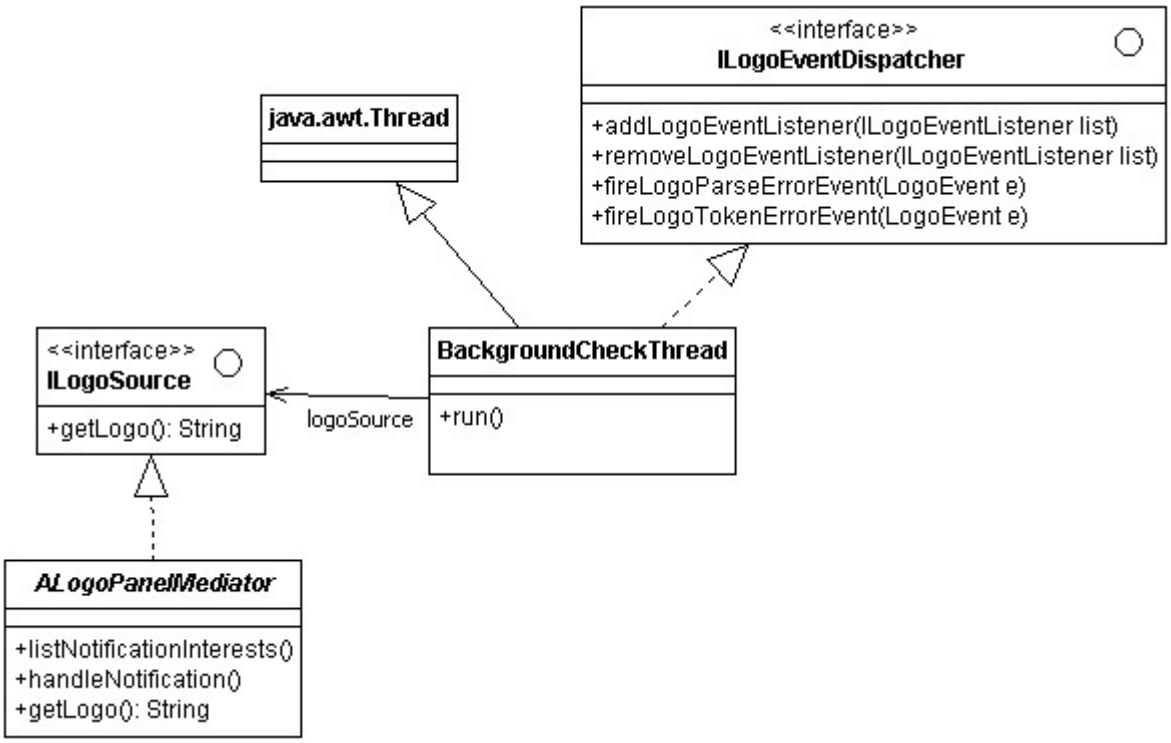


Fig 7.12 - classes relating to background syntax checking.
The background checking thread dispatches LogoEvents.

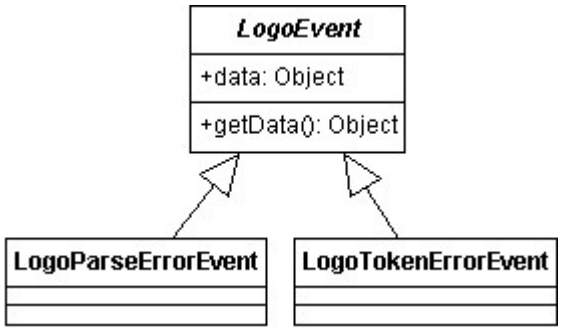


Fig 7.13 - custom events

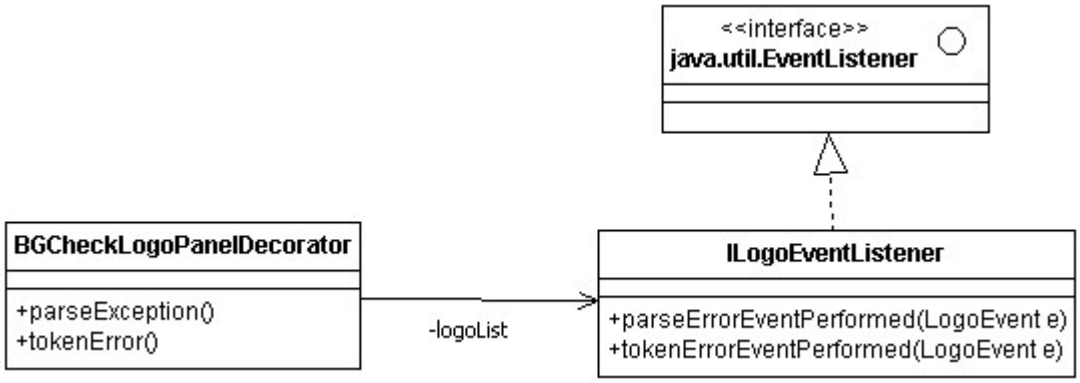


Fig 7.14 - the BGCheckLogoPanelDecorator listens to events
dispatched from the background checking thread.

By analogy with Swing events like button presses ("actionPerformed"), I call the methods that listen to these event somethingEventPerformed(). See [15] for a reference regarding handling custom events.

7.3.4 Undoable Behaviour (UC_undoEdit and UC_redoEdit)

The Undoable decorator must add functionality to listen for CTRL-Z and CTRL-Y key presses (these are used on Windows and Linux, I have not looked into Mac) and to store either the differences between consecutive edits (the inserted or removed text) or more simply the entire text. Since I am not expecting huge programs from primary school children I have opted for saving the entire text. I will also need a limit on the number of undos possible.

This functionality could be simply stored inside the undoable decorator, but after some research, I found a better way of implementing the undo list using a helper object called a *Memento* to encapsulate the Logo. The relevant class diagram from [10] is shown in Fig 7.15:

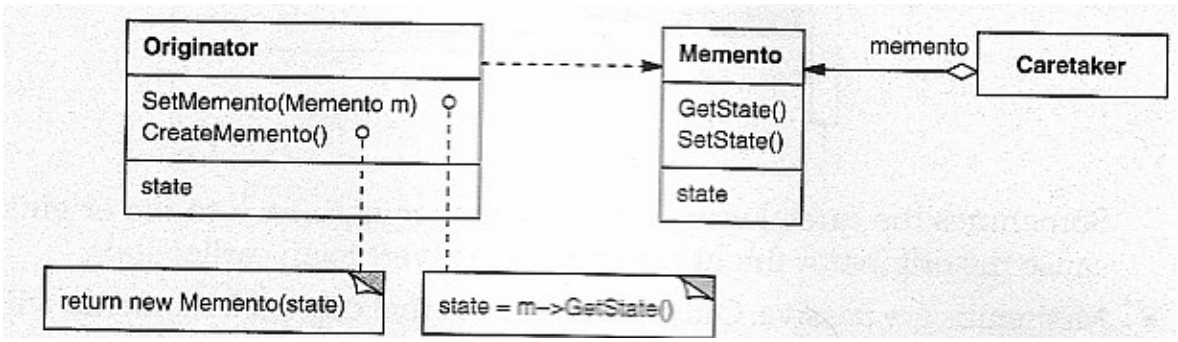


Fig 7.15 The Memento pattern (From [10])

The meaning of this diagram is that the Originator (in my code, the text area or its Mediator) is responsible for creating a Memento that contains its internal state. It also restores its internal state using a Memento passed back to it. The Memento hides this from the "Caretaker" object which holds Mementos but doesn't know or care exactly what is inside them. In my project this will be a String - but using Mementos allows this to be changed at a later date (perhaps into some kind of "Logo Object" which holds some extra data other than a String; perhaps if each procedure is written in a new window the Logo Object will be a number of Strings). Java's Swing framework actually has a class called an UndoManager [32] but I shall implement my own for more control and better understanding.

The UndoManager will need to store a list of Mementos. Undo and redo will retrieve the relevant

Memento.

A CTRL-Z keypress in the text area will result in:

```
if(undoManager.canUndo()){
      undoManager.undo();
}
```

and similarly for CTRL-Y and redo.

The UndoMananger will manage the undoing and then it will have to return to the originator to actually perform the undo (because only the client has a reference to the textField). The relevant classes, similar to Fig 7.15 are shown in Fig 7.16.

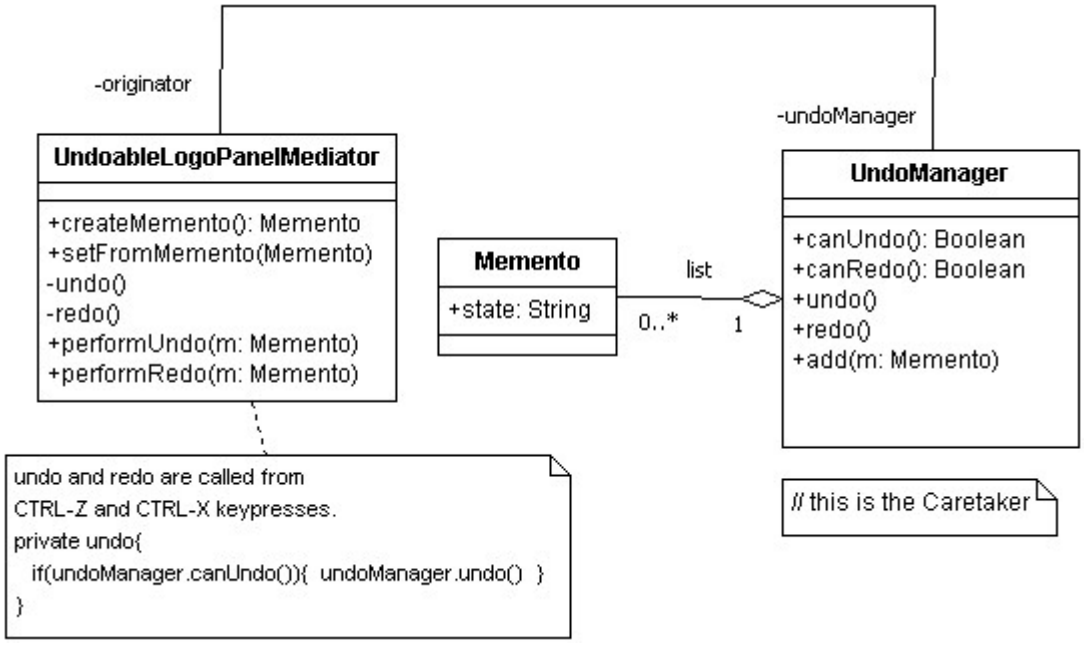


Fig 7.16 - classes relating to the UndoManager

The advantage of this design is that the UndoManager and the Memento objects can be changed separately from any changes in the Logo panel, or even the Mediator.

7.4 Processing and drawing Logo (UC_testLogoLocally)

Clicking on the draw button should trigger a PureMVC command to begin parsing the Logo and, if syntactically correct, to begin walking the parse tree. Processing will be done in a Proxy called LogoProcessingProxy. I'm not sure that a Proxy is the best place for the processing, but in PureMVC there are no permanent references to Commands, and Proxies in some sense represent a place for input/output of data (in my case to/from the Parser). In pseudocode this is shown in Fig. 7.17:

```
class LogoProcessingProxy{

   public void process(String logo){
      try{
         send to parser and  build tree  (  using JavaCC )
         if no errors, start new Thread in which to walk the tree.
      }
      catch(error){
         report error to user
      }
   }

}
```

Fig 7.17 - pseudocode for LogoProcessingProxy

As show in Fig 4.16, the Visitor of the tree is always passed an object of type ILogoConsumer which has an 'accept' method to store the FD and RT commands that are the actual (fd/ rt) output of a Logo program. These commands extend a common abstract base class, as shown in Fig 7.18.

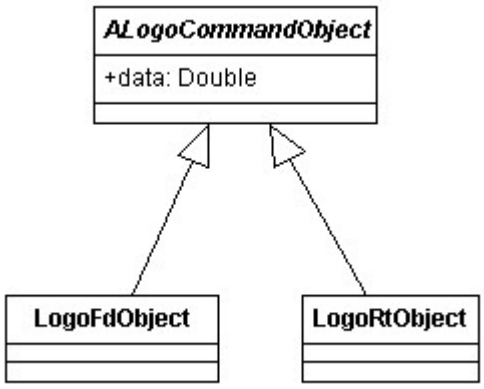


Fig 7.18 - Hierarchy of LogoCommandObjects

The LogoProcessingProxy will need to implement ILogoConsumer (see Fig 4.16), as shown in Fig 7.19. It takes Logo to be processed, accepts ALogoCommandObjects and does something with errors (sends Notifications to the rest of the application).

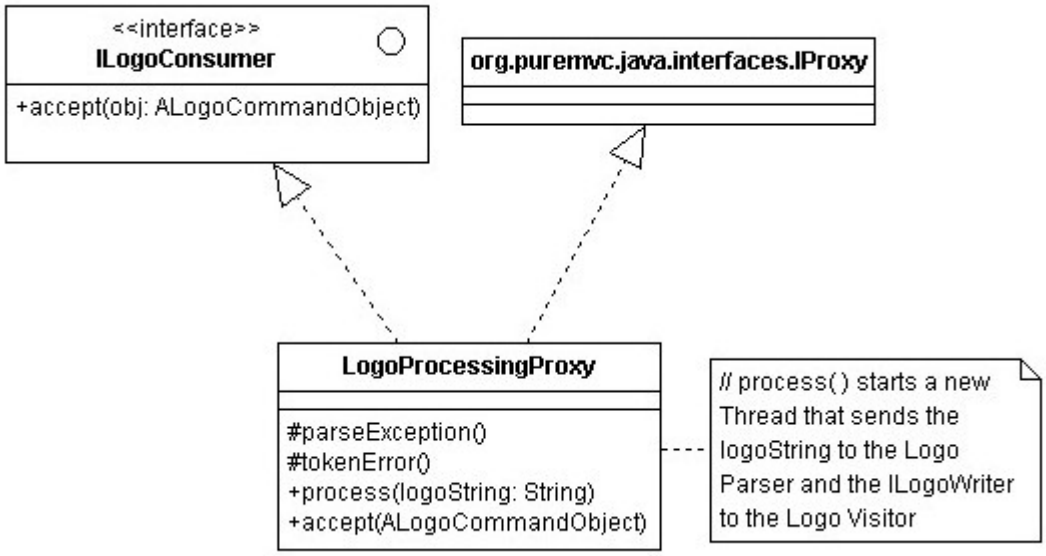


Fig 7.19 - class diagram for LogoProcessingProxy

Once the LogoProcessingProxy receives an ALogoCommandObject it will have to use the PureMVC framework to send a notification to the Turtle Canvas where the on-screen turtle will draw the relevant path (Fig 7.20):

TurtleMediator.java (pseudocode)

```
class TurtleMediator{
   public listNotificationInterests(){
      // draw logo and  clear panel
   }
   public handleNotification(notification n){
      if(draw logo){
         if(RT / FD){
           tell the UI component to rotate the turtle or move it forward
         }
      }
      else if(clear panel){
        tell the view component to clear the graphics.
      }
   }
}
```

Fig 7.20 - pseudocode for TurtleMediator

The turtle graphics Swing view component itself will consist of a JPanel which executes the drawing on its java.awt.Graphics, shown in Fig 7.21:

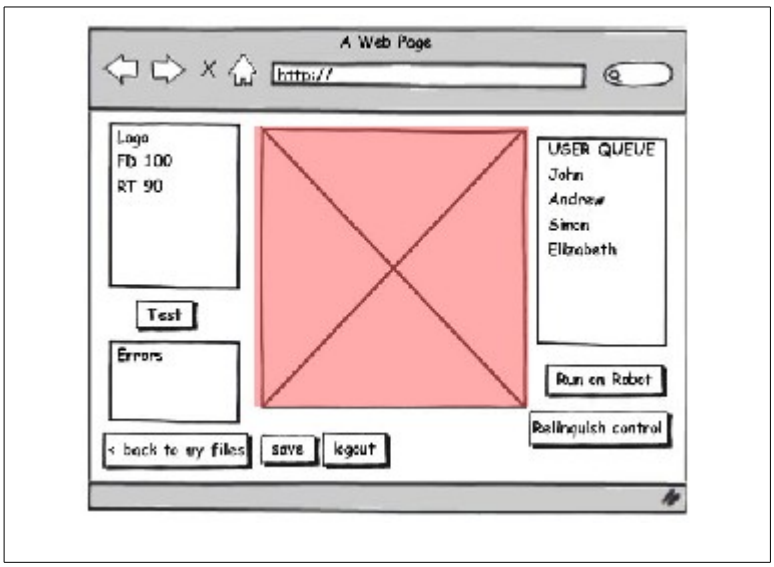


Fig 7.21 - the Turtle swing component on which Logo is drawn

It will need to store the x/y position of the turtle as well as its rotation, θ, and two methods to:

i) move forward 'D' units:

$x \rightarrow x + D*\text{Math.cos}(\theta)$

$y \rightarrow y + D*\text{Math.sin}(\theta)$

ii) rotate by α degrees/radians:

$\theta \rightarrow \theta + \alpha$

7.5 Sending Logo to the robot (UC_executeOnRobot)

A socket connection will be maintained to the machine that is connected to the robot (called the Admin server in Section 3). This must maintain a collection of threads, one for each connection, so that people can log in/log out and the user lists can be updated, while another thread is sending Logo to the robot for example. See Fig 7.22.

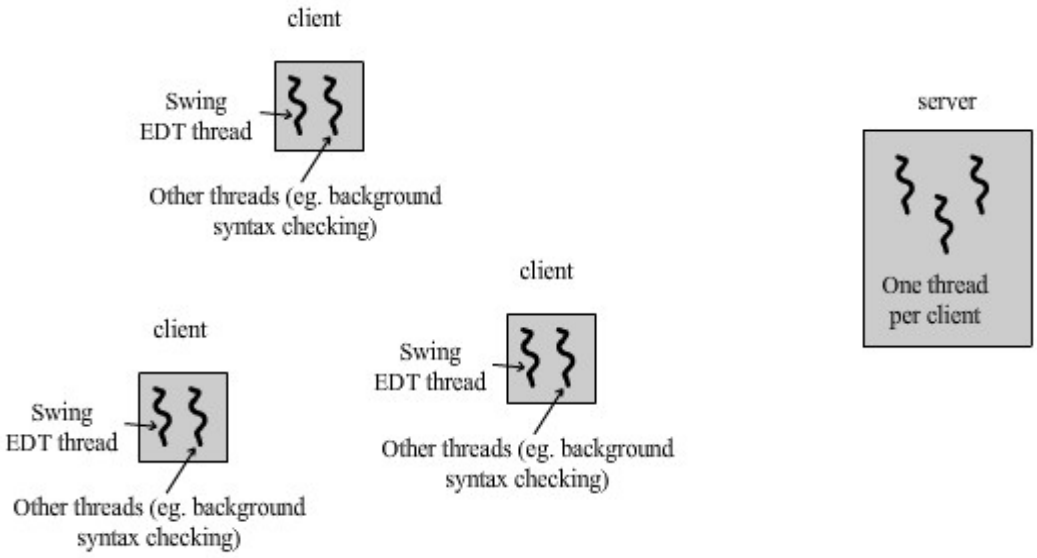


Fig 7.22 - multithreaded backend

To manage the multithreading on the server I plan to use the samples given in [29], but rather than using DataOutputStreams, I will use ObjectOutputStreams. This is because eventually the application will need to support a number of different kinds of object being sent back and forth between the front and back ends. Based heavily on [29], the pseudocode I need running on the server is shown in Fig 7.23:

```
class Server {
  Hashtable outputStreams = new Hashtable();
  ServerSocket serverSocket;
  Server() {
     serverSocket = new ServerSocket( // host and port go here  );
     // I use localhost to test locally, and port 5000
     while (true) {
        Socket s = serverSocket.accept();
        ObjectOutputStream dout = new
                                  ObjectOutputStream(s.outputStream());
        outputStreams.put(s,dout);
         new ServerThread( this, s );
      }
  }

  sendBackToClient(Socket s, message){
     ObjectOutputStream oos = outputStreams.get(s);
  }

  removeConnection( Socket s ) {
     //  as in [29]
  }
}
```

```
class ServerThread extends Thread {
  Server server;
  Socket socket;
  ServerThread( Server server, Socket socket ) {
    this.server = server;
    this.socket = socket;
    start();
  }
  run() {
     ObjectInputStream objIn = new ObjectInputStream(socket.inputStream());
     while (true) {
        Object message = objIn.readObject();
        // process the object
     }
   }
   receive(LogoSocketObject lObj){
      String logoString = lObj.getData();
      create parse tree, and catch any errors (eg. parse exceptions)
      if successful, begin walking the tree,
      and send logo command to the robot one by one
      to reply to the client use server.sendBackToClient(socket, message)
   }
}
```

Fig 7.23 - multithreaded backend, based on [29]

The code in the tutorial [29] contains a number of *synchronized* methods. Whenever multiple threads can be running simultaneously we must use some kind of concurrency locks to make sure that one thread doesn't interfere with the execution of another. For example, the if the list of users is going to iterated over using a simple 'for' loop, the list could be modified by one thread (a user logs in, or a user logs out) while another thread is actually half-way through iterating, causing errors (imagine if the user at index 'i' is deleted halfway through the loop processing item 'i').

The different kinds of object that can be sent over the socket will all be subclasses of an abstract class called ASocketObject (Fig 7.24)

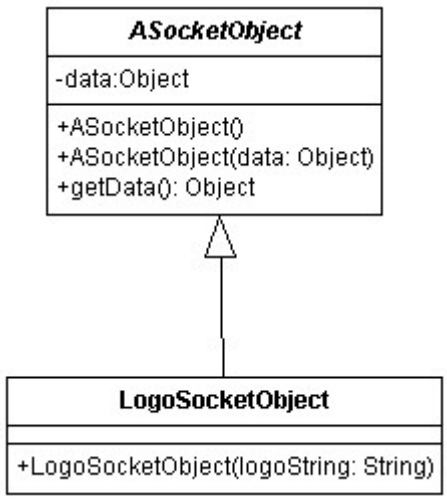


Fig 7.24 - objects sent over the socket. LogoSocketObject contains Logo to be sent to the robot.

7.6 Java running on the robot (UC_executeOnRobot)

Lejos itself comes bundled with examples of how to connect to the robot by bluetooth, in the files BTReceive and BTSend found in the samples directory of the Lejos install path. See [22] for details of these files - they demonstrate how to send data to the robot and how to receive data back. I need to adapt BTReceive.java to listen for messages from the backend and execute them. We cannot send a Logo string to the robot and we cannot just push objects into some kind of stream because they will arrive far quicker than the robot can perform them. I have no 'design' for this problem; I investigated a number of different solutions until I found one that worked. See Section 8.8 for implementation details.

To maintain flexibility I need an interface for my actual robot, then many different kinds of robot can (in theory) draw Logo. My interface is IRobot, and my actual robot will be represented by a class called TyredRobot, since my robot has two tyres. See Fig 7.25:

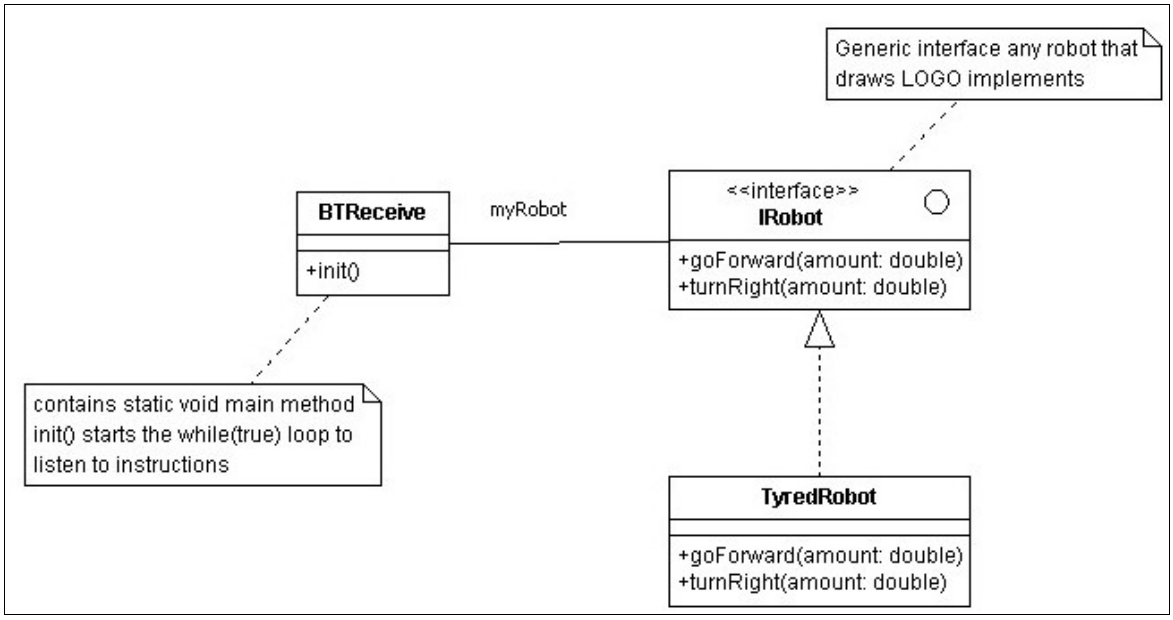


Fig 7.25 - code running on the robot

My robot will implement goForward as “turn both motors on forwards and wait the correct amount of time” and it will implement turnRight as “turn left motor on *forwards* and right motor on *backwards* and wait the correct amount of time”. Other robots might use completely different ways of drawing Logo (perhaps something like a plotter using Cartesian coordinates for example) and using IRobot allows this.

8 Iteration 1 - Development

This section is divided into subsections, each dealing with a different aspect of the application - the Logo text area and its undoable behaviour, starting and stopping Logo being drawn on-screen and starting and stopping Logo being executed by the robot.

8.1 Implementing the Logo text area and its line numbers

After a little research I found that displaying line numbers 1,2,3,4... on the left hand side of the text editor can be accomplished either by adding a new thin text area down the left, or by using a custom Border. I decided to use a LineNumberedBorder class, and I took the implementation almost wholesale from [8], with one adaptation to show a small exclamation mark on the line with an error (as in many IDEs). See com.jgrindall.logo.view.components.LineNumberedBorder.java for details.

8.2 Details of the UndoManager (UC_undoEdit, UC_redoEdit)

UndoManager.java (refer back to Fig 7.16) contains an ArrayList of Mementos and a pointer (an integer representing the current location of the pointer into the list). I created an abstract Memento class called AMemento and sub-classed it with mine, which contains the Logo String. To understand the logic of the UndoManager consider again typing in the word “grapefruit”. After getting as far as “gra” the list will look as in Fig 8.1.

0	1	2
g	gr	gra
		↑

Fig 8.1 - examples for the UndoManager

Undoing once will need to decrement the pointer and then restore the string “gr”. Undoing again will decrement the pointer to 0 and restore the string “g”. Redoing from “g” will need to advance the pointer once and restore from “gr”. So, our undo and redo code should be as shown in Fig 8.2:

```
function undo(){
    ptr--
    restore(  myList.getItemAt(ptr)   )
}
function redo(){
    ptr++;
    restore(  myList.getItemAt(ptr)  )
}
```

Fig 8.2 - undo and redo

We can see that the list is undoable when it has `ptr > 0` (ie. not at the start of the list) and that it is redoable when not at the end of the list. At the end of the list `ptr == list.length-1`, so we need `ptr < list.length -1`:

```
public Boolean canUndo(){
    return ptr > 0;
}
public Boolean canRedo(){
    return ptr < myList.size() - 1;
}
```

Fig 8.3 - canUndo and canRedo

Next, imagine that the list is full, for example if we set the maximum size to 6 and type “grapef”, as in Fig 8.4(i). If we now type “r” the list should look like as in Fig 8.4(ii); the pointer is not advanced and element 0 is removed. If instead we had undone a few times to this position shown in Fig 8.4(iii) and then typed 'r' the situation is different. This time we should delete from the end of the list and then insert (Fig 8.4(iv))

0	1	2	3	4	5
g	gr	gra	grap	grape	grapef
					↑

0	1	2	3	4	5
gr	gra	grap	grape	grapef	grapefr
					↑

Fig 8.4 (i) and (ii) - examples for the UndoManager

0	1	2	3	4	5
g	gr	gra	grap	grape	grapef
		↑			

0	1	2	3	4	5
g	gr	gra	grar		
			↑		

Fig 8.4 (iii) and (iv)- examples for the UndoManager

Hence the implementation of the add method as shown in Fig 8.5:

com.jgrindall.logo.views.UndoManager.java:

```
public void add(AMemento m){
    clearForwards();
    if(myList.size()==MAX){
        myList.remove(0);
    }
    else{
        ptr++;
    }
    myList.add(ptr, m);
}

private void clearForwards(){
    for(int i=myList.size()-1; i>ptr; i--){
        myList.remove(i);
    }
}
```

Fig 8.5 - adding to the UndoManager

The final class diagram is shown in Fig 8.6.

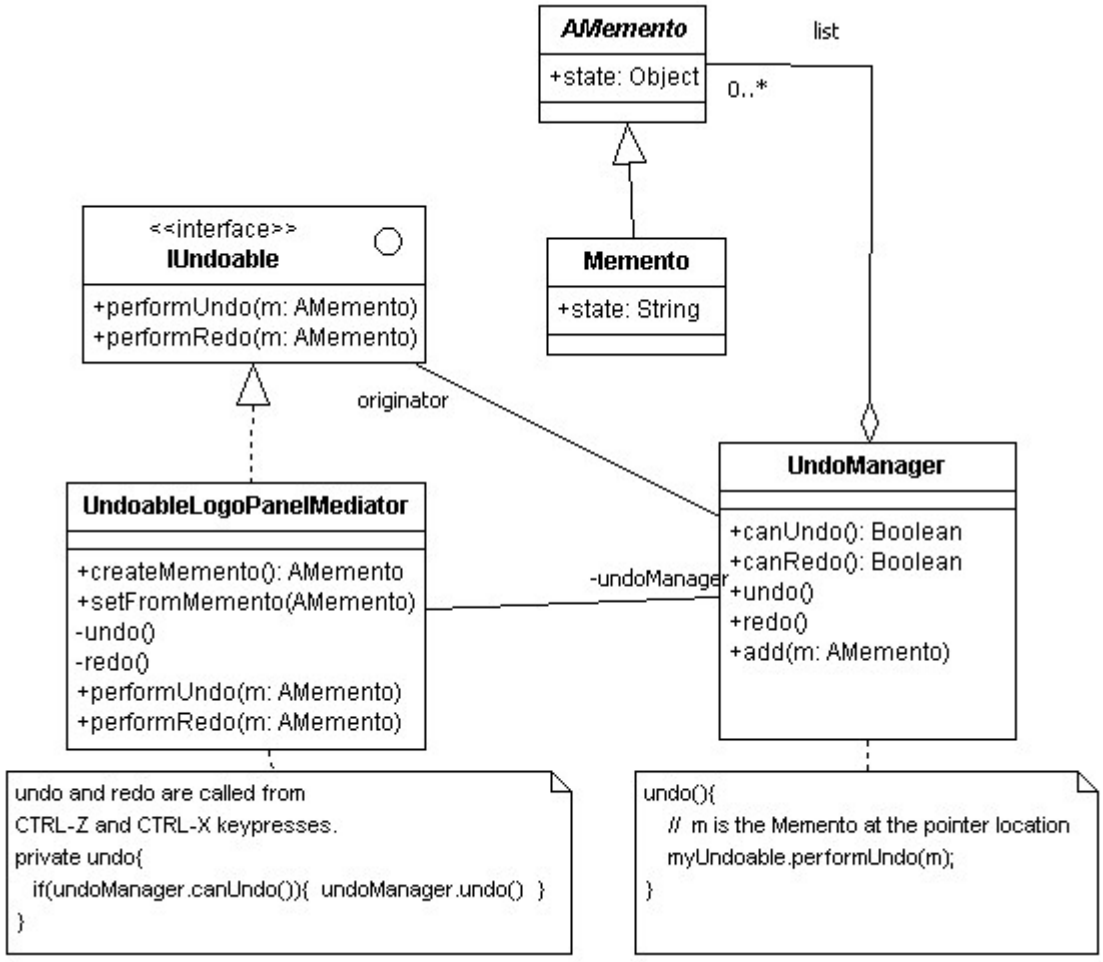


Fig 8.6 - final class diagram for the UndoManager

8.3 Implementing the TurtleCanvas

The TurtleCanvas is implemented as a 2000×1000 pixel canvas. One serious problem I encountered was dealing with the user resizing the available window *as the Logo is being drawn*. It is not possible to take a copy of the Graphics and repaint it into the new enlarged space, since Java will not even have drawn pixels if they lie outside of the clipped region. Only the pixels that were visible before the user resized were drawn! The solution I found is to use a 2000×1000 pixel BufferedImage to store all the graphics (lines and the green triangle which represents the turtle) rather than drawing them directly into the Graphics of the swing component. When the canvas is resized I listen for this event and repaint the graphics from the BufferedImage into the Graphics of the canvas. It works nicely - all pixels are redrawn and the origin is always nicely centred. The pseudocode (relating to the file com.jgrindall.logo.views.components.TurtleCanvas) is shown in Fig 8.7:

```
// fired when the component is resized
private void onComponentResized(){
    w = max( width now, previous width)
    h = max( height now, previous height)
    create new buffered image
    copy old pixels from the rectangle (0,0,w,h)
    replace the old buffered image with the new one
    repaint()
}

// overridden swing method
public void paintComponent(Graphics g){
    super.paintComponent(g);
    g.drawImage(myBufferedImage,0,0,this);
}

private void drawForward(double d){
    Graphics2D gc = myBufferedImage.createGraphics();
    gc.drawLine(...)
    repaint()
}
```

Fig 8.7 - drawing and resizing the TurtleCanvas

8.4 Implementing the LogoTextEditor and the 'store' Command

When the Logo is changed it needs to be stored so that it is available to the rest of the application - for example when we want to send it to the robot. The place to do this in PureMVC is in a Proxy. To achieve it the following technique is used:

- A `store()` method in ALogoPanelMediator, the base class for all my Mediators, implemented as:

```
sendNotification(AppFacade.STORE_LOGO, logoString, null);
```

- A Notification called STORE_LOGO which holds the String in its body.
- The STORE_LOGO Notification is mapped to a Command to perform the logic, namely StoreLogoCommand which stores the String inside a Proxy for that purpose, called LogoProxy.
- The LogoProxy is registered with the PureMVC framework in the StartupCommand
- The store method is overridden in the UndoableLogoPanelMediator as:

```
protected void store(){
     super.store();
     undoManager.add(  saveToMemento()  );
}
```

- It is overridden in the decorator classes as:

```
protected void store(){
     wrappedMediator.store();
}
```

The fact that Logo is stored centrally means that the interface ILogoSource in Fig 7.12 is not needed. Any interested parties can retrieve the Logo from the Proxy in a coherent fashion.

8.5 Drawing Logo (UC_testLogoLocally)

When the draw button is pressed the first Notifications that are sent are to clear any currently displayed error messages and graphics, namely the following:

- CLEAR_DRAWING (listened for by the TurtlePanelMediator, to clear all graphics)
- CLEAR_ERROR (listened for by the LogoPanelMediator to clear any highlights)
- CLEAR_ERROR_TEXT (listened for by the ErrorPanelMediator to remove error messages)

Only then is a Notification called PROCESS_LOGO sent. This notification is mapped to a Command called ProcessLogoCommand. Actual processing happens in LogoProcessingProxy whose responsibility is to syntax check the Logo and then use the JavaCC parser to begin walking the parse tree.

An additional interface called ILogoProcess is used, so that the rest of the application has a reference

typed to the interface rather than to LogoProcessingProxy itself - see Fig 8.8:

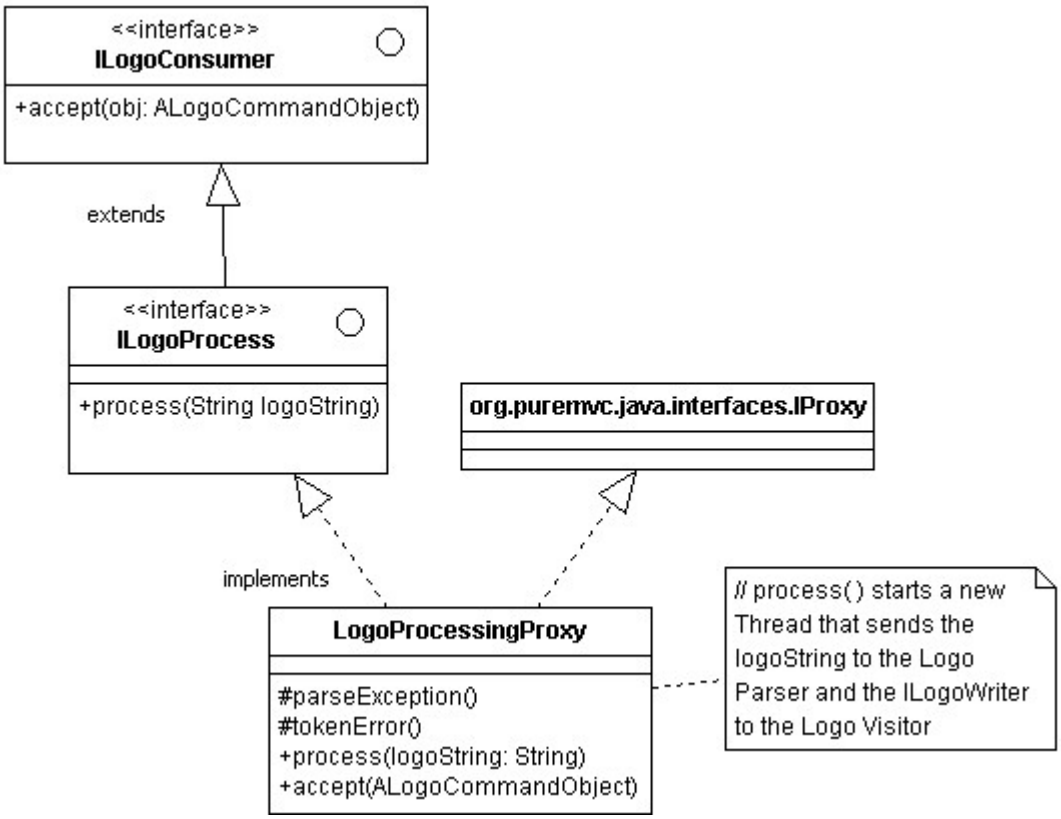


Fig 8.8 - class diagram for processing Logo

The final implementation of the 'draw logo' use case consists of a sequence of steps as shown in Fig 8.9:

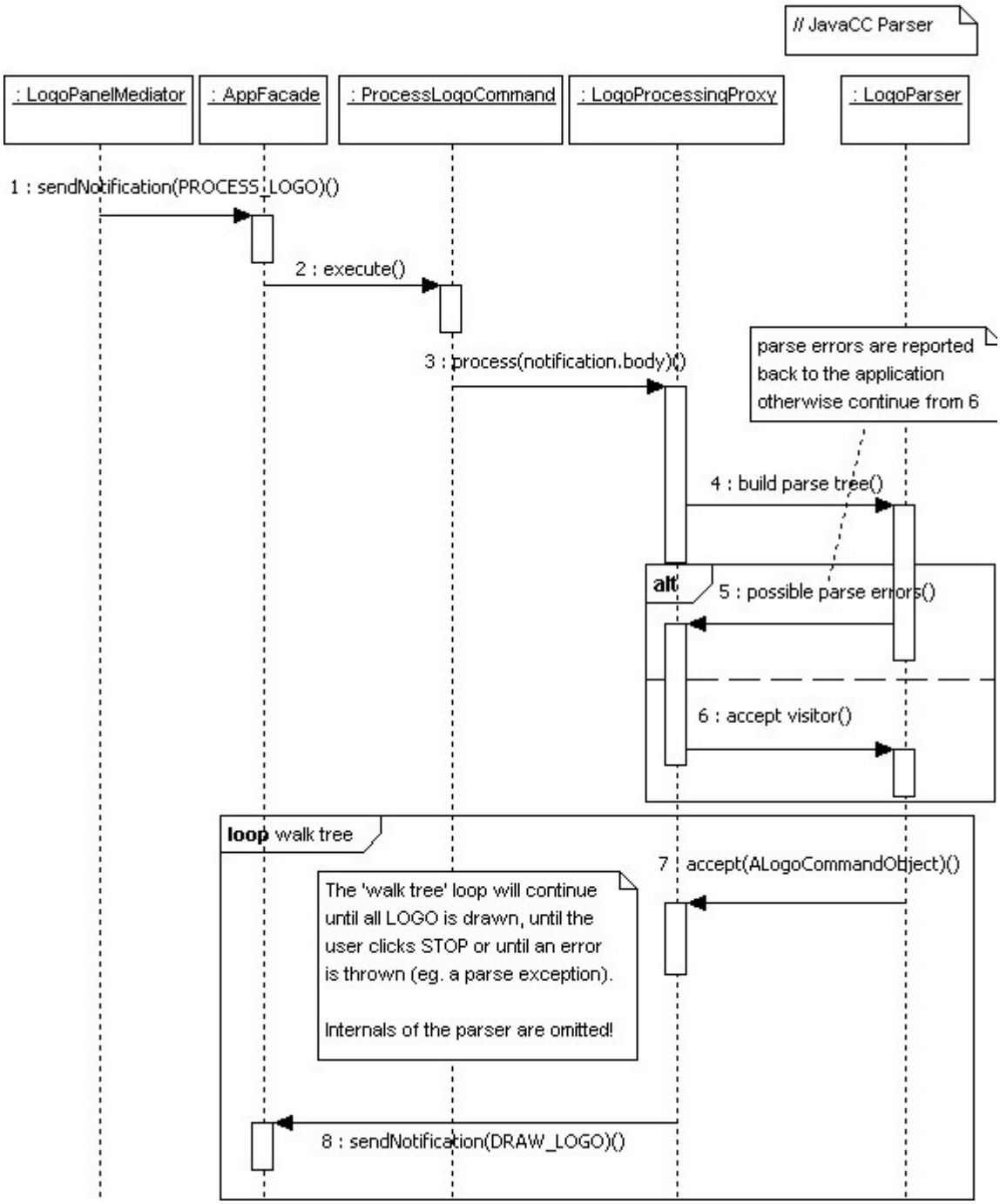


Fig 8.9 - sequence diagram for drawing Logo

1. User clicks 'draw', LogoPanelMediator sends PROCESS_LOGO Notification, with a body which contains the Logo as a String.
2. AppFacade executes ProcessLogoCommand.
3. ProcessLogoCommand sends process() message to LogoProcessingProxy, passing the Logo as a String from the body of the Notification.
4. JavaCC parse tree built
5. If syntax errors, HIGHLIGHT_ERROR, PARSE_ERROR and STOP_DRAWING_UPDATE_BUTTONS notifications are sent to show the error in the Logo

text area, to display the error message in the ErrorPanel and to re-enable the draw button.

6. Otherwise the parser accepts the visitor.
7. When the parser visits a Fd or Rt node, it is accepted by the LogoProcessingProxy (which implements ILogoConsumer)
8. The LogoProcessingProxy sends a Notification which is listened by the TurtleCanvasMediator to actually draw the Logo.

8.6 Stopping Logo drawing (UC_stopLocally)

Even without "while" statements (which are not included in the grammar yet) and having restricted the depth of recursion, Logo programs can still run for an unbounded length of time (eg. long, nested repeat loops). It is necessary to be able to stop them. This is achieved in a similar way to the previous section - namely when the stop button is clicked a STOP_DRAWING Notification is sent, mapped to a StopDrawingCommand which retrieves the ILogoProcess object and calls its stop() method (see Fig 8.11)

Since the LogoProcessingProxy kicks off the JavaCC Parser in a new Thread, we need a way to stop the Thread. Since Thread::stop() is deprecated I use a flag called "active" (see also [30] - "Most uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running.").

Recall from Section 4.2.7 that when the Parser visits a FdStatementNode or a RtStatementNode, it passes an ALogoCommandObject to an ILogoConsumer using its accept() method. The technique for stopping drawing Logo is as follows:

- Whenever the parser visitor visits *any* node (not just a Fd or Rt node) it calls a check() method on the ILogoConsumer (Fig 8.10)
- The ILogoConsumer can throw an exception if necessary (for example we will throw an exception if the "active" flag is set to false).

Fig 8.10 - the check() method can be used to stop the Parser at any time

The exception is thrown and the thread in which we are processing the logo terminates safely. ParseException is subclassed to com.jgrindall.logo.javacc.StopParseException.java to indicate to the application that the error message should *not* be reported to the user. This subclassing is not very tidy but it means that we do not need to mess around with the auto-generated JavaCC code that throws ParseExceptions.

The fact that the check() method is called on *any* Node means that programs that are in 'infinite' loops but not actually drawing anything (no Fd or Rt statements) can also be stopped. The final class diagram is shown in Fig 8.11.

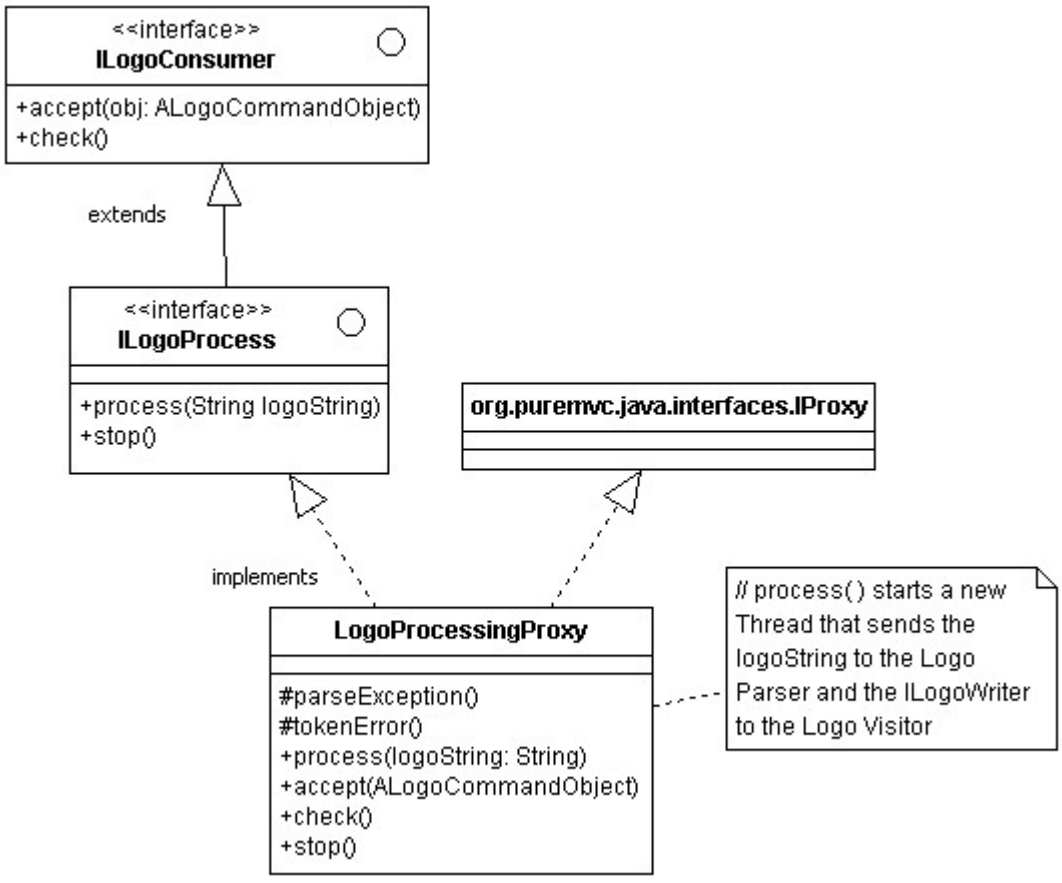


Fig 8.11 - final class diagram for stopping Logo processing

8.7 Sending Logo to the backend (UC_executeOnRobot)

When the user clicks the 'Send to robot' button a Notification called SEND_TO_ROBOT is sent, which is mapped to a Command called SendToRobotCommand. This command performs syntax checking on the Logo and if no exceptions are caught, it sends the Logo over the Socket. To utilize ObjectOutputStreams, an abstract serializable base class called ASocketObject is used. LogoSocketObject extends ASocketObject and holds the Logo to be drawn by the robot (Fig 8.12)

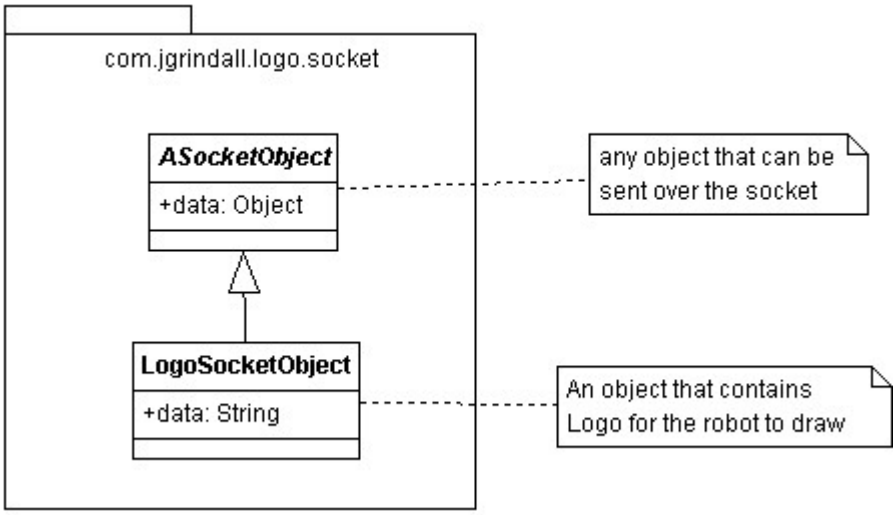


Fig 8.12 - objects that can be sent over the socket

A Proxy called SocketProxy (Fig 8.13) deals with sending all information over the Socket (see com.jgrindall.logo.proxy.SocketProxy.java and com.jgrindall.logo.proxy.ISocket.java). It implements java.awt.Runnable so that, in a separate thread, the Socket can listen for input from the backend.

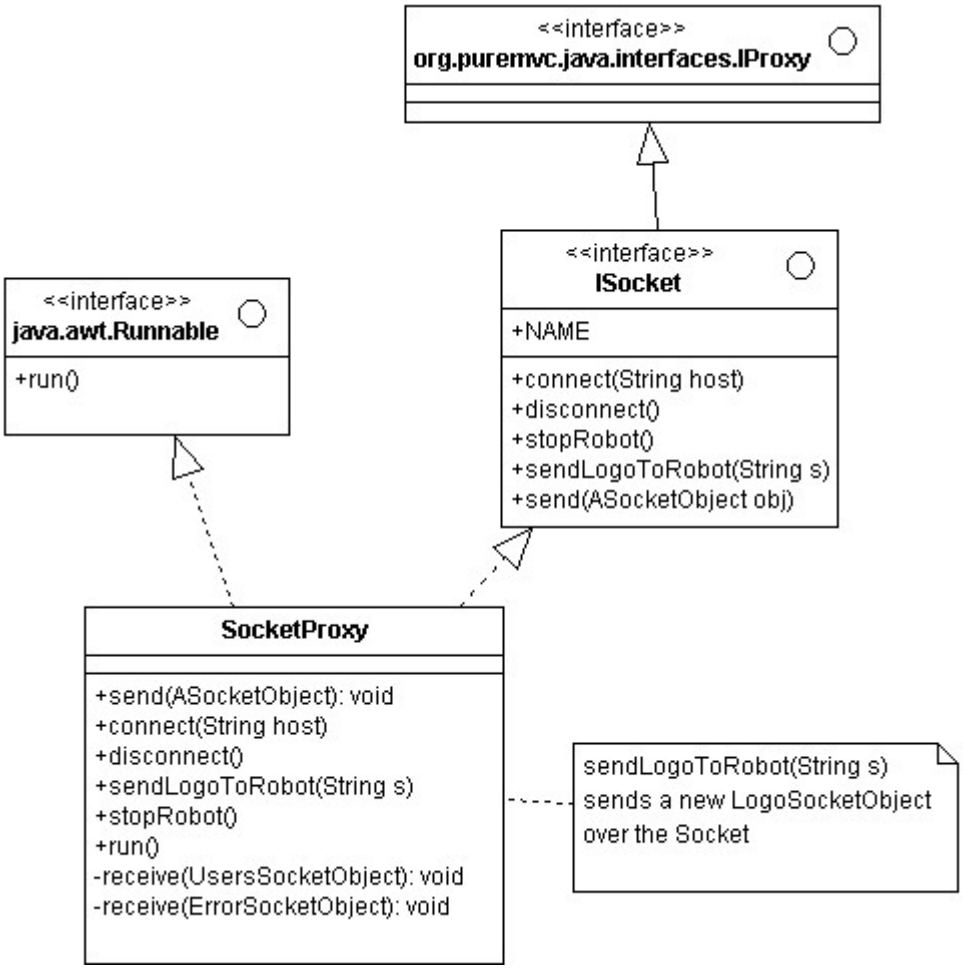


Fig 8.13 - the SocketProxy manages all communication over the socket

The interface ISocket is used for loose coupling - in case a different method of sending and receiving data is required at a later date. All other references to the SocketProxy are typed to the interface.

A sequence diagram for the entire process of sending Logo to the backend is shown in Fig 8.14:

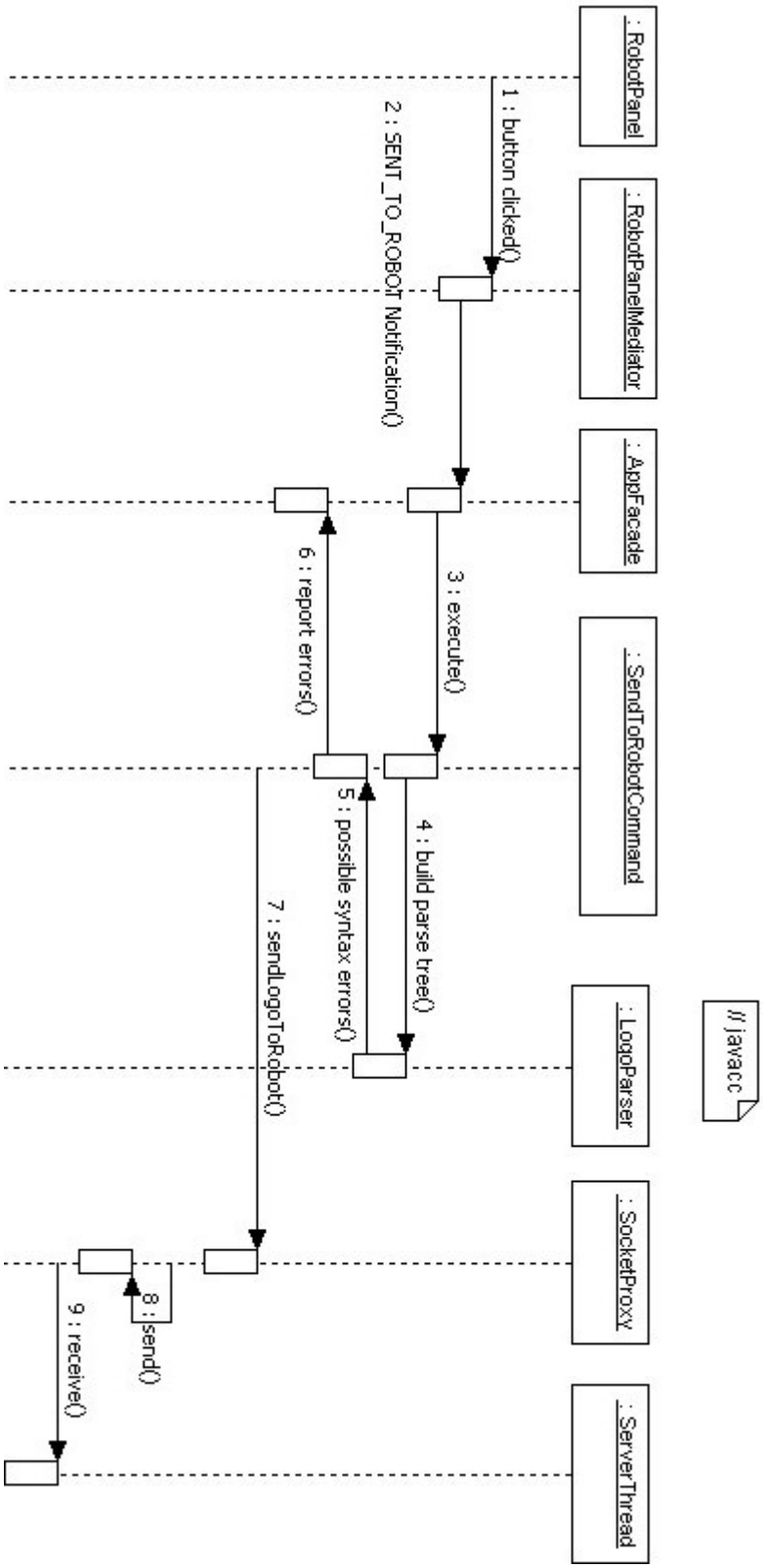


Fig 8.14 - sending logo to the backend

1. The swing button is clicked
2. SEND_TO_ROBOT Notification is send, the body contains the Logo string.
3. AppFacade maps this to SendToRobotCommand and executes it
4. The JavaCC parser takes the string and builds the Parse tree.

5. Any syntax errors are caught
6. Syntax errors are reported to the user using other Notifications (not shown)
7. The SocketProxy is retrieved, and is sent the Logo.
8. A LogoSocketObject containing the Logo is sent over the Socket
9. The back end receives the LogoSocketObject.

8.8 Sending Logo to the robot - the Producer/Consumer model

It took a lot of trial and error to get Logo sent from the Server class to the robot, and then executed on the robot correctly. At first I tried a complicated technique of terminating the execution of the LogoParserVisitor and storing the node I had reached, and restarting the Visitor at that node when required. This failed and was in retrospect a silly idea, because the Java stack (ie. the history of the depth first traversal from all the `node.childrenAccept(this, data))` calls is completely lost! I then started researching ways in which the actual thread itself (in which the Parser and Visitor are executing) can be paused somehow while the robot executes the Logo, essentially as in Fig 8.15.

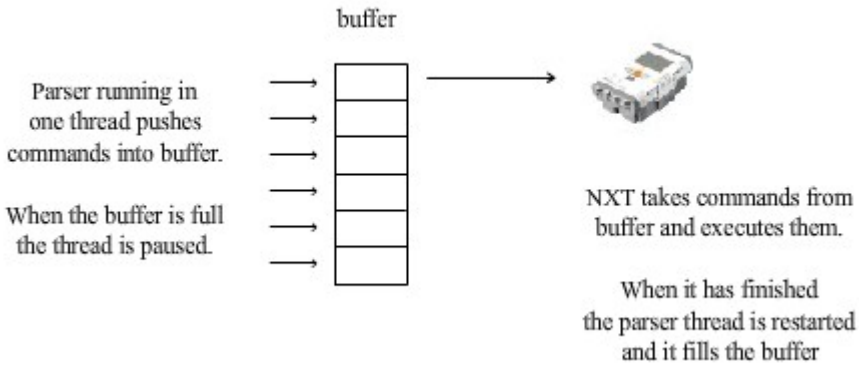


Fig 8.15 - sending commands from the backend to the NXT

I found the solution in [19] and [26], using the so-called Producer/Consumer model. The final design as a class diagram is shown in Fig 8.16, but it is better explained in the final 'thread diagram' Fig 8.17 (I use the same kind of arrow diagram found in [19], in which dark arrows represent threads, rectangles represent locks and ovals actions to be performed)

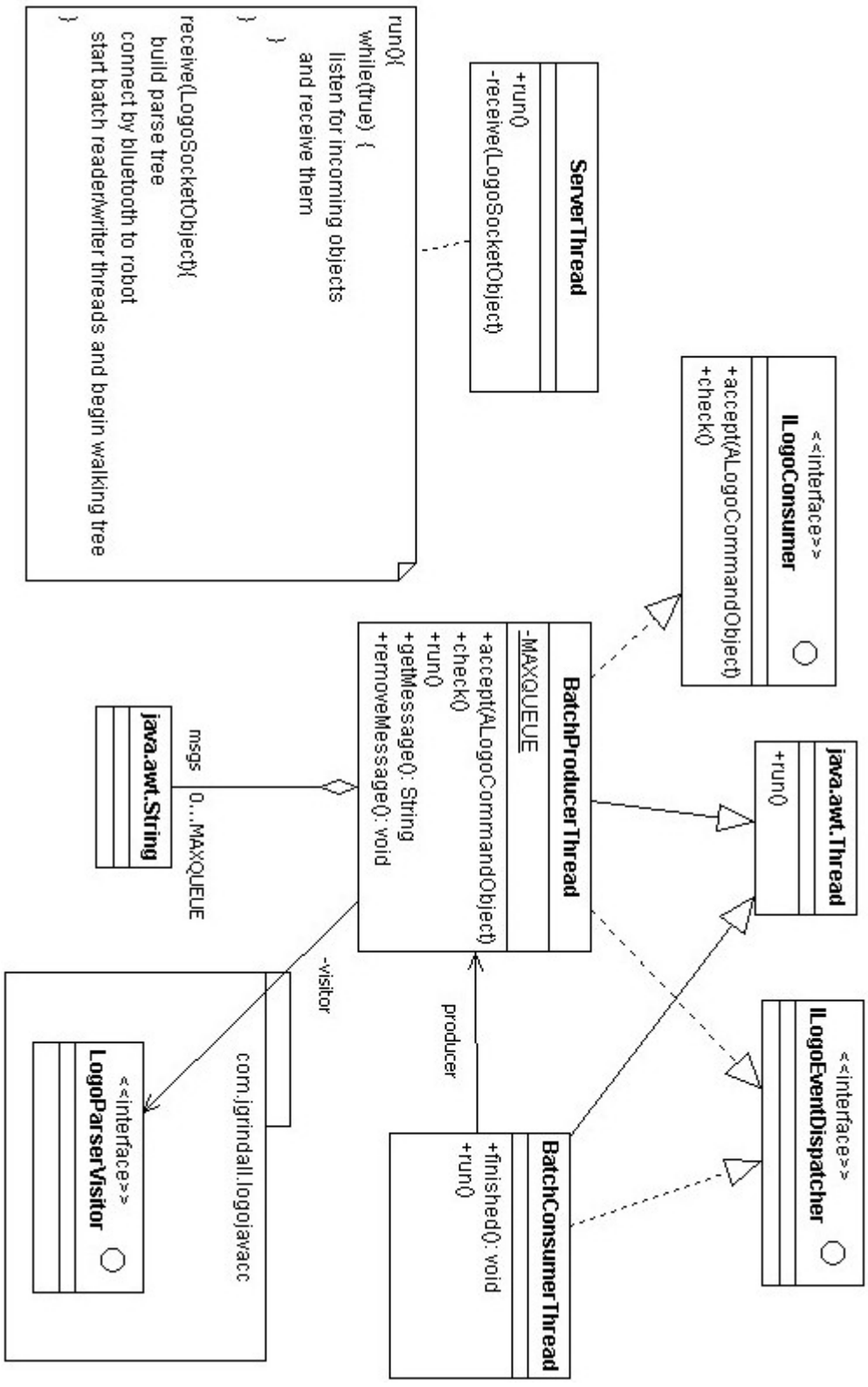


Fig 8.16 - class diagram for sending Logo commands from the backend to the NXT

The BatchProducerThread class is the Producer. It implements ILogoConsumer because accepts ALogoCommandObjects from the parser. Rather than doing anything with these objects it simply pushes them into a FIFO list of messages. When the list becomes full the Consumer (BatchConsumerThread) is woken up and takes messages out of the queue using getMessage(). This is

sent to the robot and when the robot replies, the Consumer removes the message from the queue and wakes up the Producer.

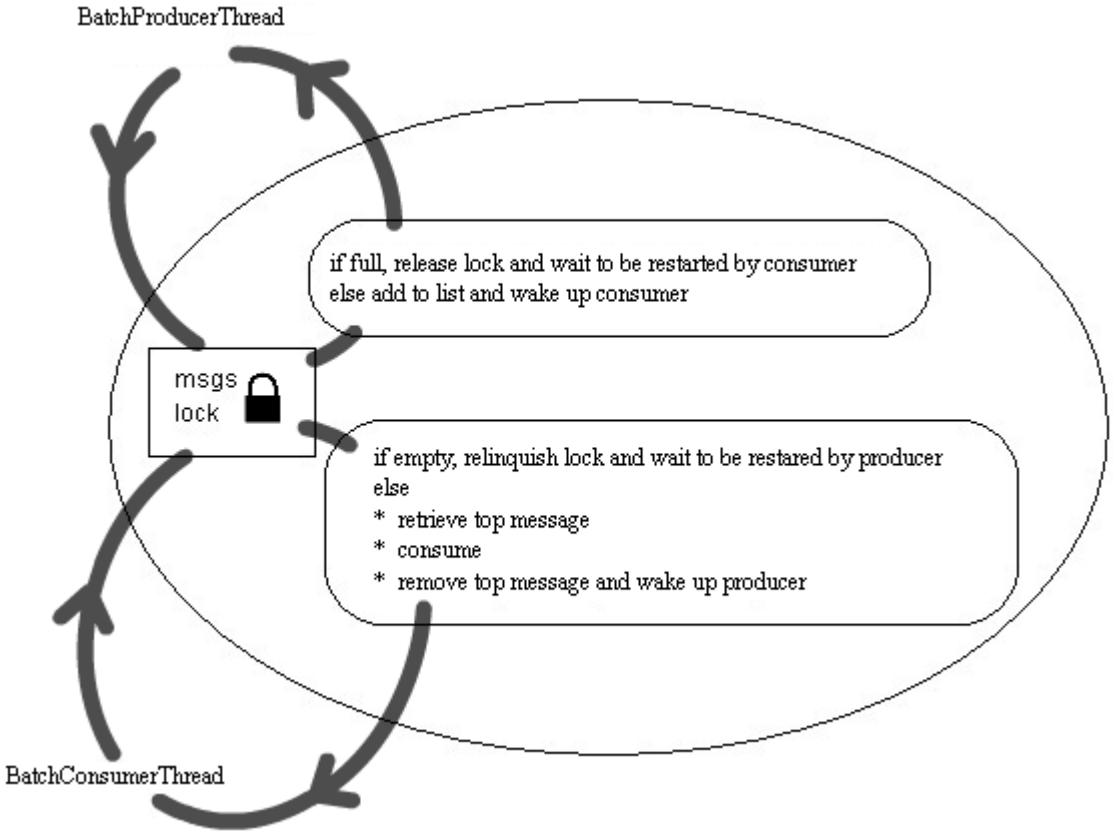


Fig 8.17 - threads and the Producer/Consumer model for sending Logo commands from the backend to the NXT

The message queue (called 'msgs') is the lock object on which all the methods (add, retrieve, remove) are synchronised. According to [26] (with my italics) we have the following definitions:

A wait() invocation results in the following actions:

- If the current thread has been interrupted, then the method exits immediately, throwing an InterruptedException. Otherwise the current thread is blocked.
- The JVM places the thread in the internal and otherwise inaccessible wait set associated with the target object.
- The synchronization lock for the target object is released *but all other locks held by the object are retained...*

A notify() invocation results in the following actions:

- If one exists, an arbitrarily chosen thread, say T, is removed by the JVM from the internal wait set associated with the target object. *There is no guarantee about which waiting thread will be selected when the wait set contains more than one thread.*
- T must reobtain the synchronization lock for the target object which will always cause it to block at least until the thread calling notify releases the lock. It will continue to block if some other thread obtains the lock first.
- T is then resumed from the point of its wait.

A notifyAll() invocation works in the same way as notify() *except that the steps occur (in effect simultaneously) for all threads in the wait set for the object. However, because they must acquire the lock, threads continue one at a time.* [26]

The italics can be ignored for our application because there are precisely two threads competing for the lock on 'msgs'. Since 'wait' and 'notify' are called from within synchronized blocks, we know that the thread that calls these methods owns the lock, so there can be one or zero threads in the wait set. Notify and notifyAll are the same for us - either will do the following:

- If called by the producer (after adding a message) and the consumer is in the wait set, pass execution to the consumer, otherwise stay with the producer and insert more messages.

- If called by the consumer (after removing a message) and the producer is in the wait set, pass execution to the producer, otherwise stay with the consumer.

In reality we know that the producer will be far faster than the consumer but we can be sure that when one thread has been added to the wait set, eventually it will call for the lock and will be restarted. After some investigation I use MAXQUEUE = 1, ie. a batch size of 1 or in other words commands are sent to the back end one by one rather than in batches. This is so that the robot can be stopped immediately (see section 8.11).

8.9 Sending Logo to the robot - abstracting the robot

Once the Consumer has retrieved a message it needs to send it to the robot and wait for a response before retrieving the next message. To ensure that the code isn't limited to a Lego NXT robot and also doesn't depend especially on the *current version* of Lejos, all code dealing specially with the NXT is abstracted out into a separate class (Fig 8.18)

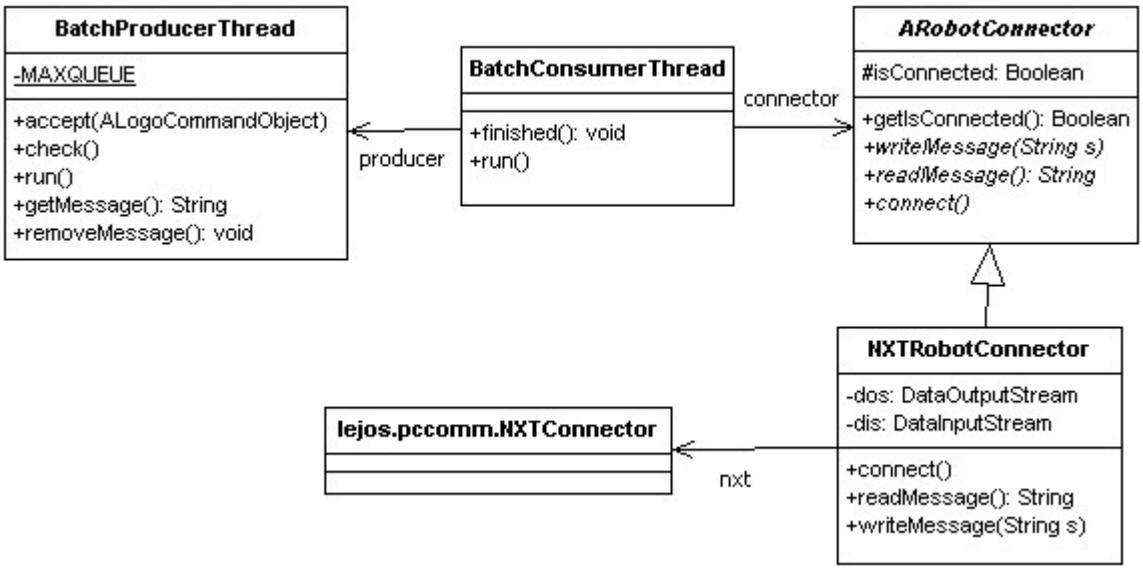


Fig 8.18 - abstracting the robot by using ARobotConnector and NXTRobotConnector

The ARobotConnector class is abstract, and its methods connect, writeMessage and readMessage are also abstract. These three methods are the only ones needed for communication to any robot, not just an NXT. The concrete NXTConnector class implements these methods in the special case of using an NXT robot (using the lejos.pccomm.NXTConnector class from Lejos).

8.10 Java running on the Robot

Unfortunately, Lejos does not include ObjectOutputStream/ObjectInputStream from java.io, so it is necessary for the robot to read *character* streams. My work around is to encode the LogoCommandObjects as Strings. The algorithm for encoding is contained in a new class called com.jgrindall.logo.comm.LogoCommandUtils.java (Fig 8.19). This class contains a number of static methods and constants for encoding a command into a string and decoding a string back into a command. This class is used by the backend and the robot and is defined in a separate shared library project.

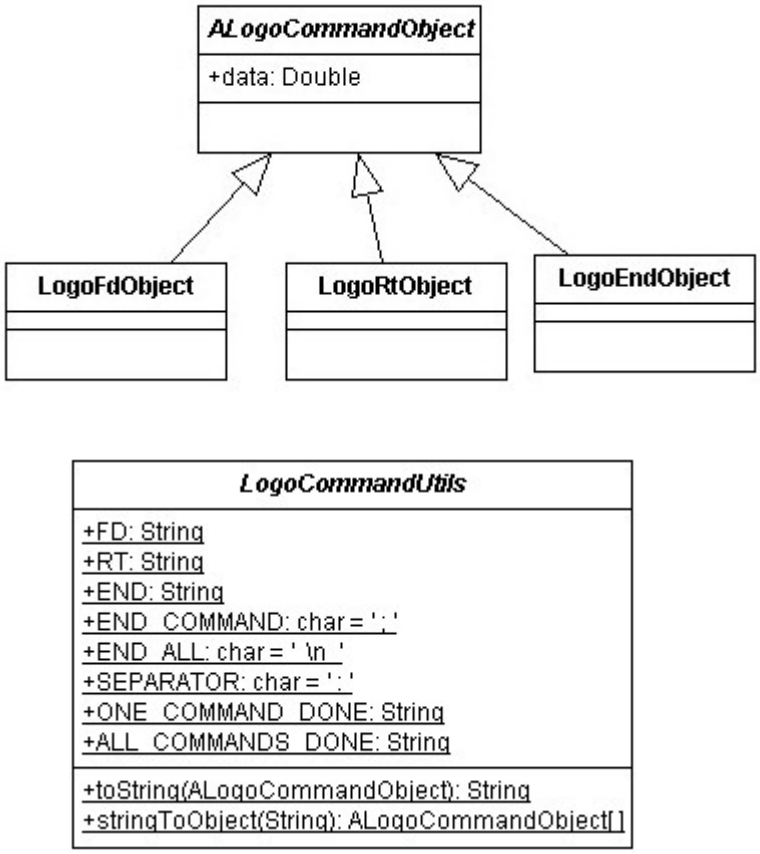


Fig 8.19 - ALogoCommandObjects are encoded as Strings to be sent to the robot

A LogoFdObject for "forward 100" is encoded as "fd:100;" and similarly for LogoRtObjects (for example "rt:90;"). I use the LogoEndObject to indicate to listening threads that the end has been reached and their run methods should terminate, and also to tell the robot to stop listening for commands. It appears in LogoParserVisitorImpl.java when the parent ProgramNode is visited:

com.jgrindall.logojavacc. LogoParserVisitorImpl.java:

```
public Object visit(ProgramNode node,Object data) throws ParseException{

    node.childrenAccept(this, data);

    /* the end object is used to indicate to anyone listening that the
    program has ended.For example we might have to re-enable the draw
    button in the GUI.
    */

    ((ILogoConsumer)data).accept(new LogoEndObject());
}
```

Fig 8.20 - adding a LogoEndObject after all Logo has finished executing

When the string of instructions reaches the Robot I need to interpret them into meaningful instructions. I do this in a new class called an "Interpreter" which encapsulates the logic behind translating a string of

characters into instructions. I also encapsulate the motion (forward/right) into what I call Action classes. The final design allows for more generality and separation of interests than Fig 7.25 and is shown in Fig 8.21, with explanation following:

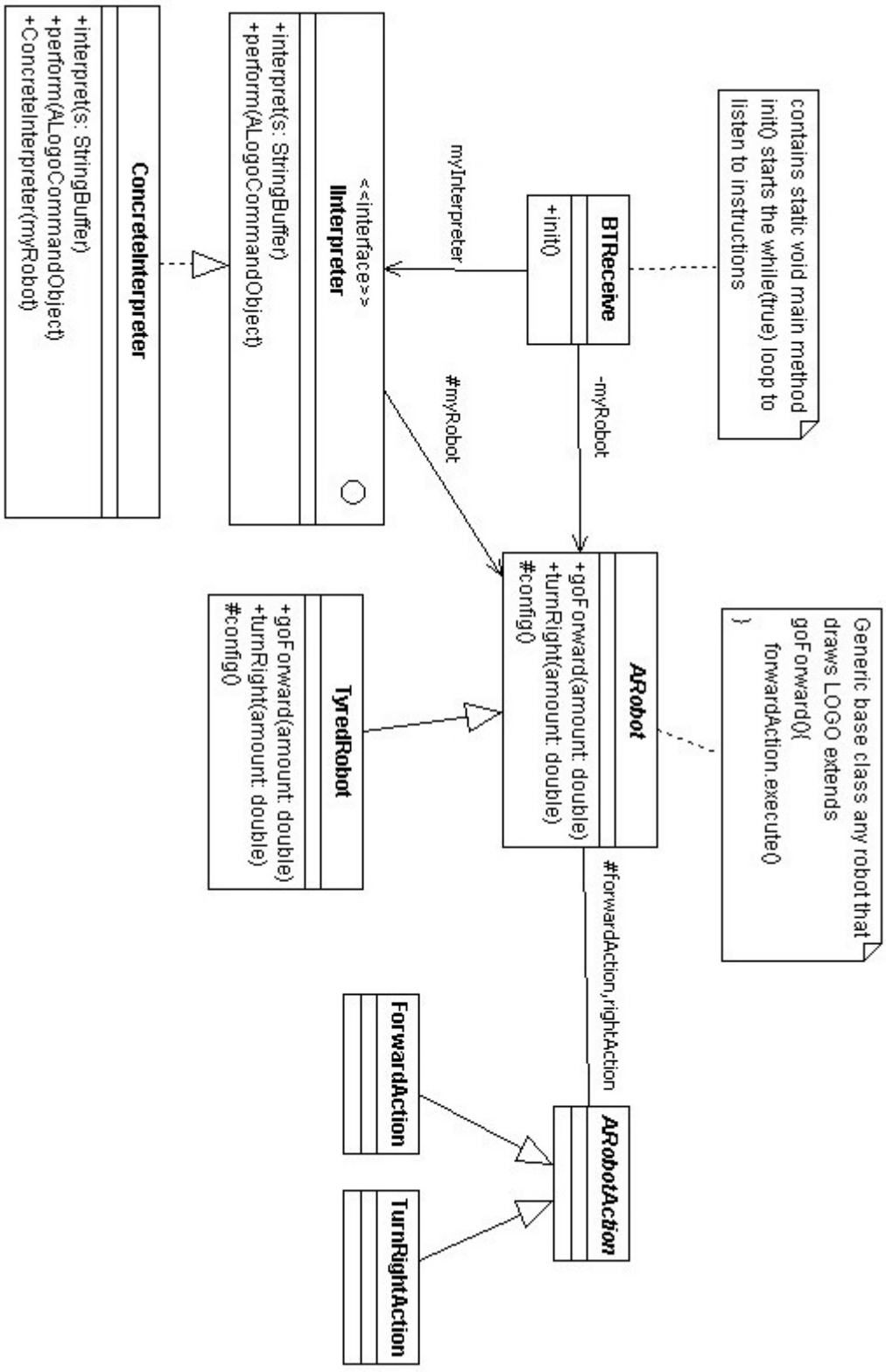


Fig 8.21 - Implementation class diagram for the robot; an interpreter to encapsulate strings and convert them into actions. Actions to encapsulate motion.

Explanation:

- The received messages are made sense of by an object of type IInterpreter.
- The implementation of this is in a separate class, ConreteInterpreter.
- The interpreter has a reference to an object of type ARobot (abstract)
- Any ARobot object has a forwardAction object and a rightAction object that deal with moving the wheels.

The main class BTReceive.java is as shown in Fig 8.22:

com.jgrindall.logorobot.BTReceive.java (pseudocode)

```
while(true){

    wait for bluetooth connection and instantiate input and output streams

    while(true){

        read from input stream up to \n, and put into StringBuffer s

        myInterpreter.interpret(s)

        a RobotException is thrown when the LogoEndCommand is read.
        It is caught, 'doneall' is sent back in the output stream.
        The streams are closed and the inner loop terminates

        Else write 'doneone' to the output stream
    }

}
```

Fig 8.22 - main loops running on the Robot

The ConcreteInterpreter::interpret() method is implemented by using the LogoCommandUtils class (see Section 8.10), namely LogoCommandUtils.stringToObject(). This outputs an ArrayList of ALogoCommandObjects. Finally, ARobot implements goForward(d) by delegating to forwardAction.execute(d). When the robot has executed a command it sends a LogoCommandUtils.DONE_ONE_COMMAND message back along the stream and the 'consume' method illustrated in Fig 8.17 is called - this actually removes the command from the 'msgs' queue for the producer/consumer pair and allows the next command to be added.

The LogoEndCommand (see Fig 8.20) is not really interpreted into a motion - but rather an exception of type RobotException is thrown which plays a sound (a double beep), sends a DONE_ALL_COMMANDS message back along the stream (which stops the producer and consumer threads), breaks the robot out of the inner while loop and back into the outer while loop (waiting for the

next connection to be initiated).

I have also thrown a RobotException when the robot is told to move for anything longer than 10 seconds. This is a simple way of stopping commands such as “fd 1000000000” from executing.

It is not ideal to have a reference to the robot in BTReceive, a reference to the robot in the Interpreter and a reference to the interpreter in BTReceive. Perhaps having an event dispatched from the Interpreter would reduce the coupling here.

8.11 Stopping the robot (UC_stopRobot)

When the 'stop robot' button is clicked the robot should stop executing Logo and should go back to an idle state, waiting for the next set of communication. Perhaps the user has noticed a problem or has started a long loop that they want to stop. For example, Logo such as :
“`rpt 10000 [fd 0.5 rt 0.5]`” work great on-screen (you get a circle) but does not work well on the robot since the pauses in between executing a command take far longer than the movement themselves, and the motors on the robot do not really move 10cm if you tell them to move 0.001cm 10000 times! As usual this is accomplished by mapping a Notification to a Command which is executed - namely STOP_ROBOT and com.jgrindall.logo.commands.StopRobotCommand.java. This command executes SocketProxy::stopRobot() as shown in Fig 8.13, which sends a StopRobotSocketObject (which extends ASocketObject) across the Socket - see Fig 8.23.

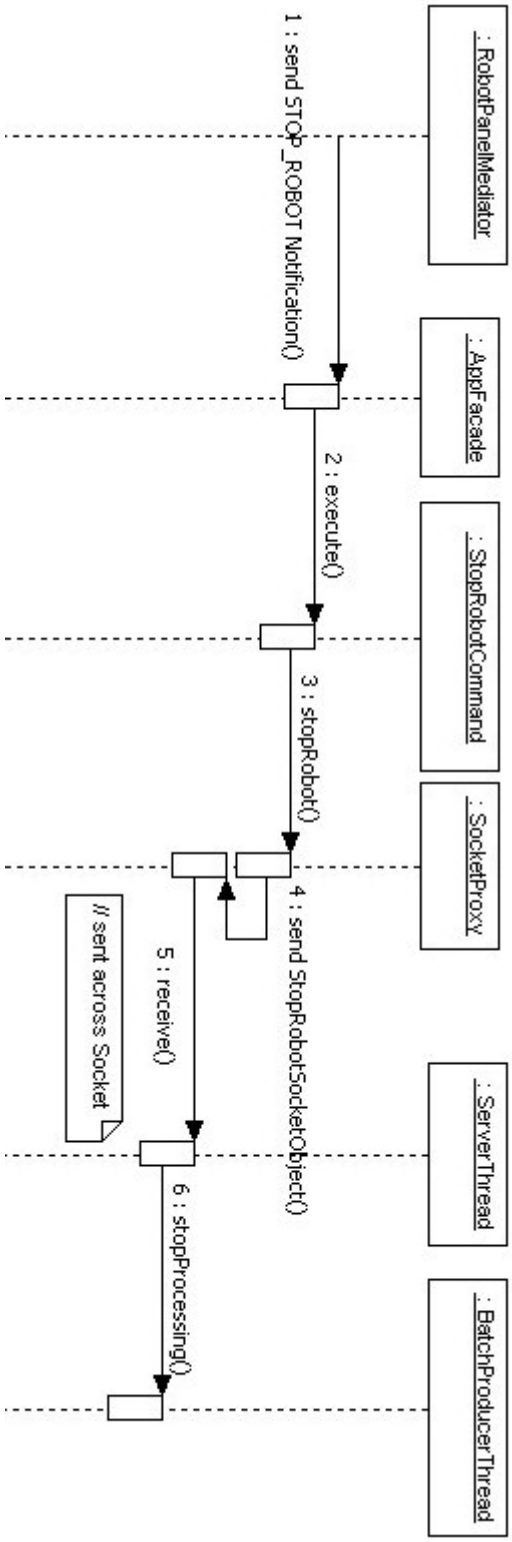


Fig 8.23 - sending the stop robot command

When the *stopProcessing()*message is received by the Producer all further “accept()” or “check()” calls received from the Logo Parser are blocked and rather a LogoEndObject is added to the msgs array. When the consumer takes this object the robot will stop and send the “ALL_COMMANDS_DONE” message back, just as would happen if the Logo had terminated naturally (see

com.jgrindall.logo.server.BatchProducerThread.java for the implementation details)

If the robot encounters a run-time syntax error then this is reported to the user and once again, the LogoEndObject is sent to the robot. Both the Producer and Consumer threads implement ILogoEventDispatcher, so they can dispatch events back to the socket server classes.

9 Iteration 2 - Design

Iteration 2 allows multiple users to communicate with the Robot. The users are still unauthenticated and they cannot save or load their files. Users can request and relinquish the robot (see Fig 3.3) and the backend maintains a list of users, as well as who is controlling the robot.

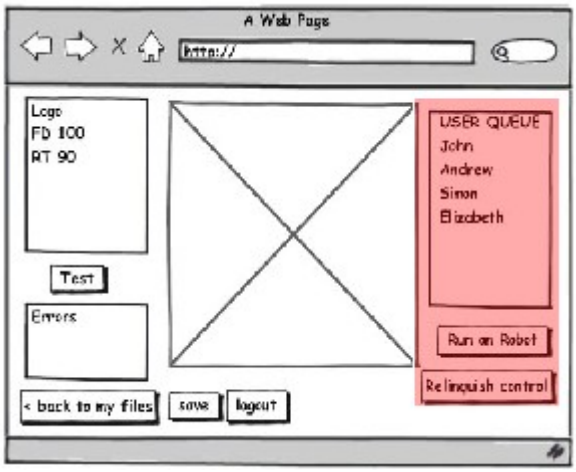


Fig 9.1 - Rough design for the GUI

9.1 Objects sent over the Socket

The new information that needs to be sent to the backend is:

- Register a new user, store them in the system in the 'not waiting to use the robot' queue.
- A user requests control of the robot. Push the user into the 'waiting to use the robot' queue , or if no-one is using the robot, give them access straight away.
- A user relinquishes control of the robot. Remove the user from being in control of the robot and move the user to the 'not waiting to use the robot' queue.
- A user logs out, either by clicking on logout, or closing the application. Remove the user from the system.

The class diagram from iteration 1 is updated as shown in Fig 9.2.

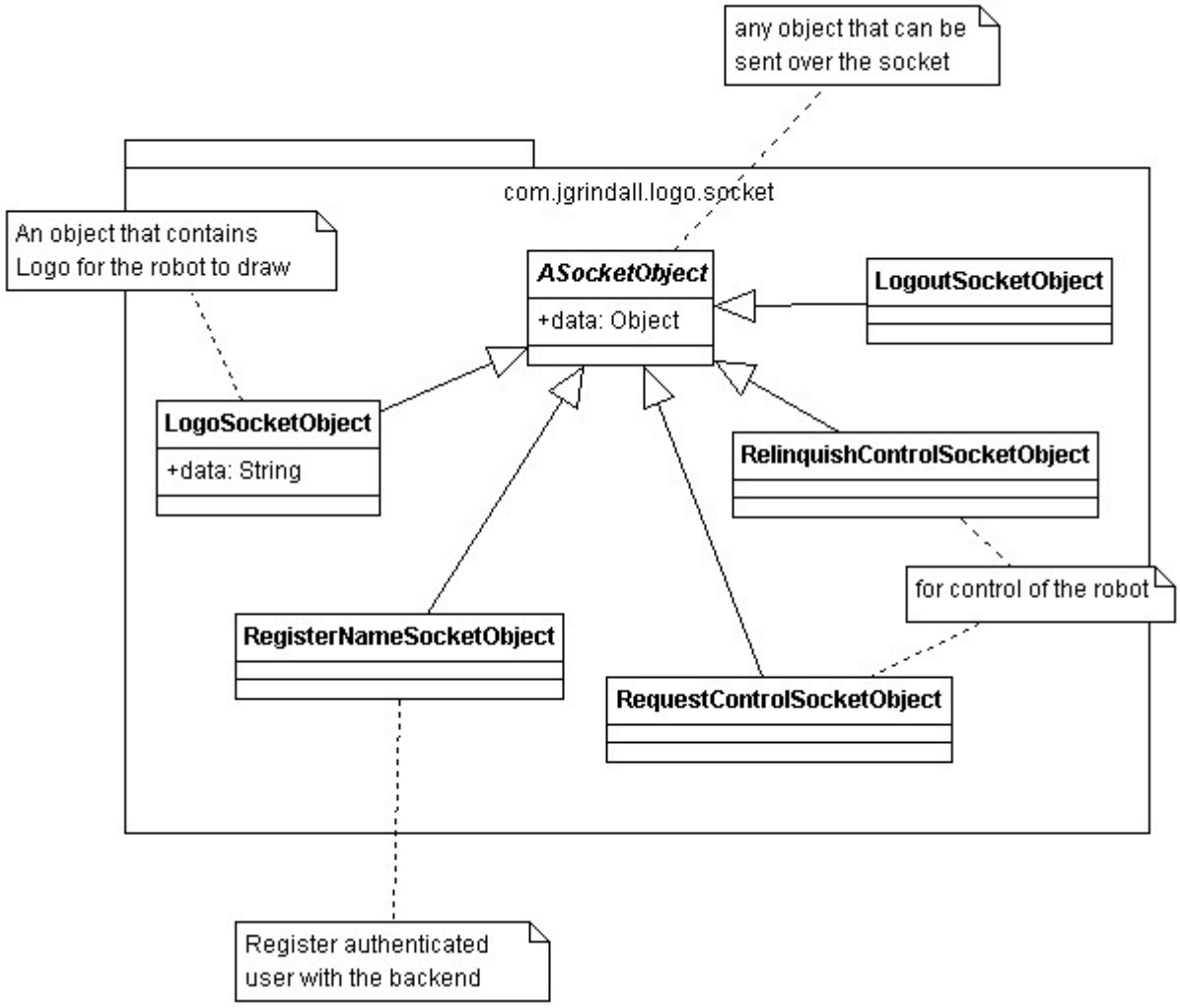


Fig 9.2 - hierarchy of SocketObjects for Iteration 2

9.2 The user list (UC_updateUserList)

Users need to be represented in the system by a class called User, and a user list needs to be maintained on the backend as described in Fig 3.3. Each logged-in user needs to know who (including themselves) is in the waiting queue, and the not waiting queue. If the user is in the waiting queue the user needs to know if they are in charge of the robot (at the top of the queue) and also if they *are* in charge of the robot, whether the robot is executing logo or not. The consequences of this information are shown in Fig 9.3.

	Request button enabled?	Relinquish button enabled?	Send to robot button enabled?	Stop robot button enabled?
Not in the waiting queue	**TRUE**	false	false	false
In the waiting queue but not at the top	false	false	false	false
In the waiting queue and at the top and the robot is executing	false	false	false	**TRUE**
In the waiting queue and at the top and the robot is not executing	false	**TRUE**	**TRUE**	false

Fig 9.3 - enabled states of buttons

The design is shown in Fig 9.4

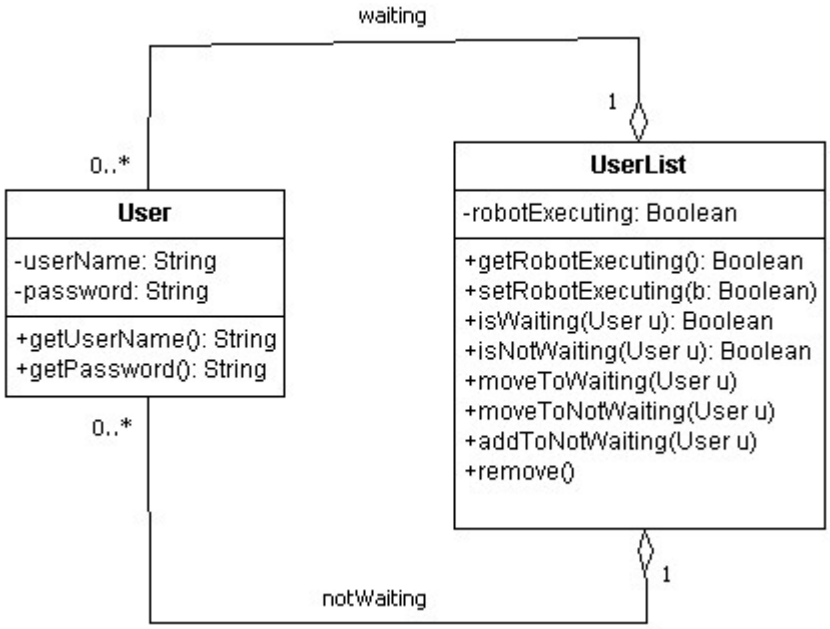


Fig 9.4 - the user list containing waiting and not waiting queues of users.

10 Iteration 2 - Development

10.1 Requesting and relinquishing control of the robot (UC_requestControl and UC_relinquishControl)

Notifications are broadcast when the request and relinquish buttons are clicked. They are linked to Commands to contain the business logic. Specifically, notification names REQUEST_ROBOT and RELINQUISH_ROBOT are linked to RequestRobotCommand and RelinquishRobotCommand classes in the com.jgrindall.logo.commands package, which use the Socket to send RequestControlSocketObject's and RelinquishControlSocketObjects.

I decided that the simplest way to send the information back to the clients regarding who is logged in, who is waiting/not waiting and who is using the robot right now is to send the *entire list* across the socket rather than updates. If a huge number of people were logged in it would make sense to come up with a cleverer design to ensure that all user lists remain in synch, sending objects that encapsulate changes rather than the entire list. However, the product is intended for small groups using one robot in a classroom. The Server stores the user list in a class that extends ASocketObject, and the entirety of this object is sent over the socket.

The SocketProxy (the class that deals with all data in and out of the application over the Socket) is updated to include receiving messages from the server, namely errors (for example run time parse exceptions or IO errors - held in ErrorSocketObjects) and the UserSocketObject. See Figs 9.5 and 9.6.

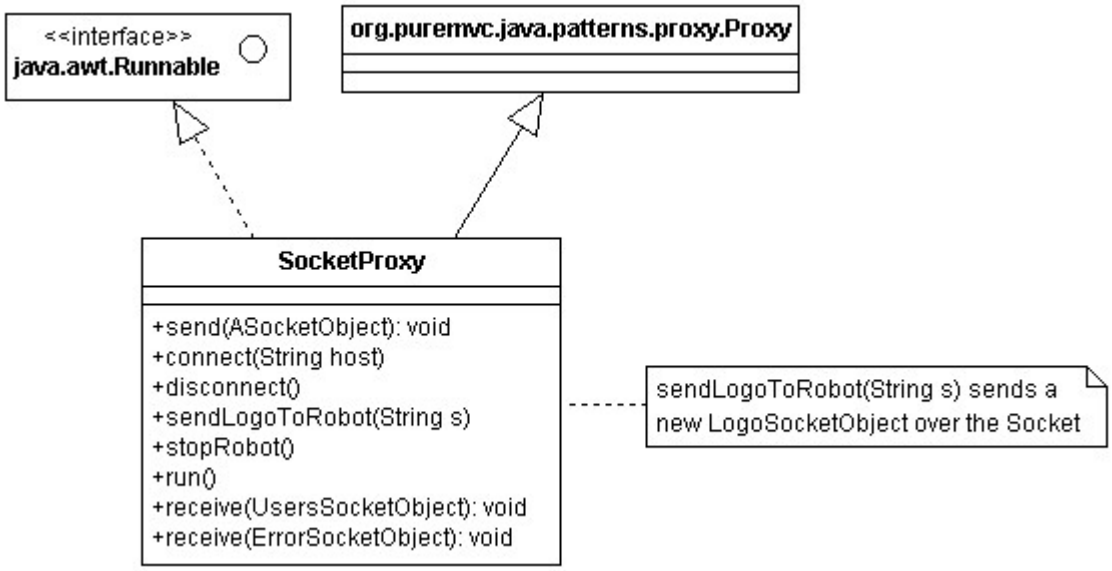


Fig 9.5 - final class digram for the SocketProxy

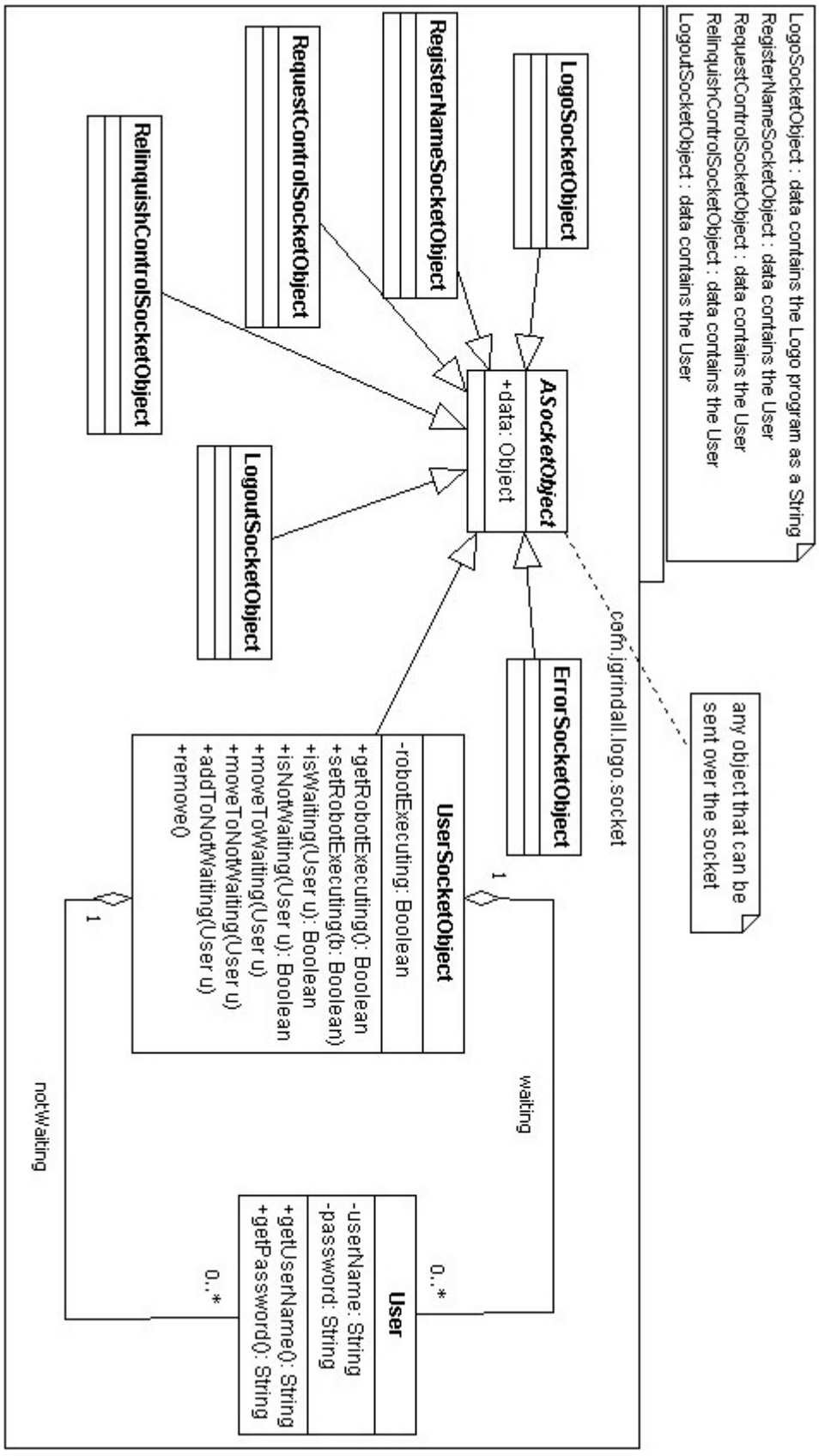


Fig 9.6 - objects that can be sent over the Socket.

A sequence diagram showing the entire sequence of operations in the UC_requestRobot use case is shown in Fig 9.7 and explained below:

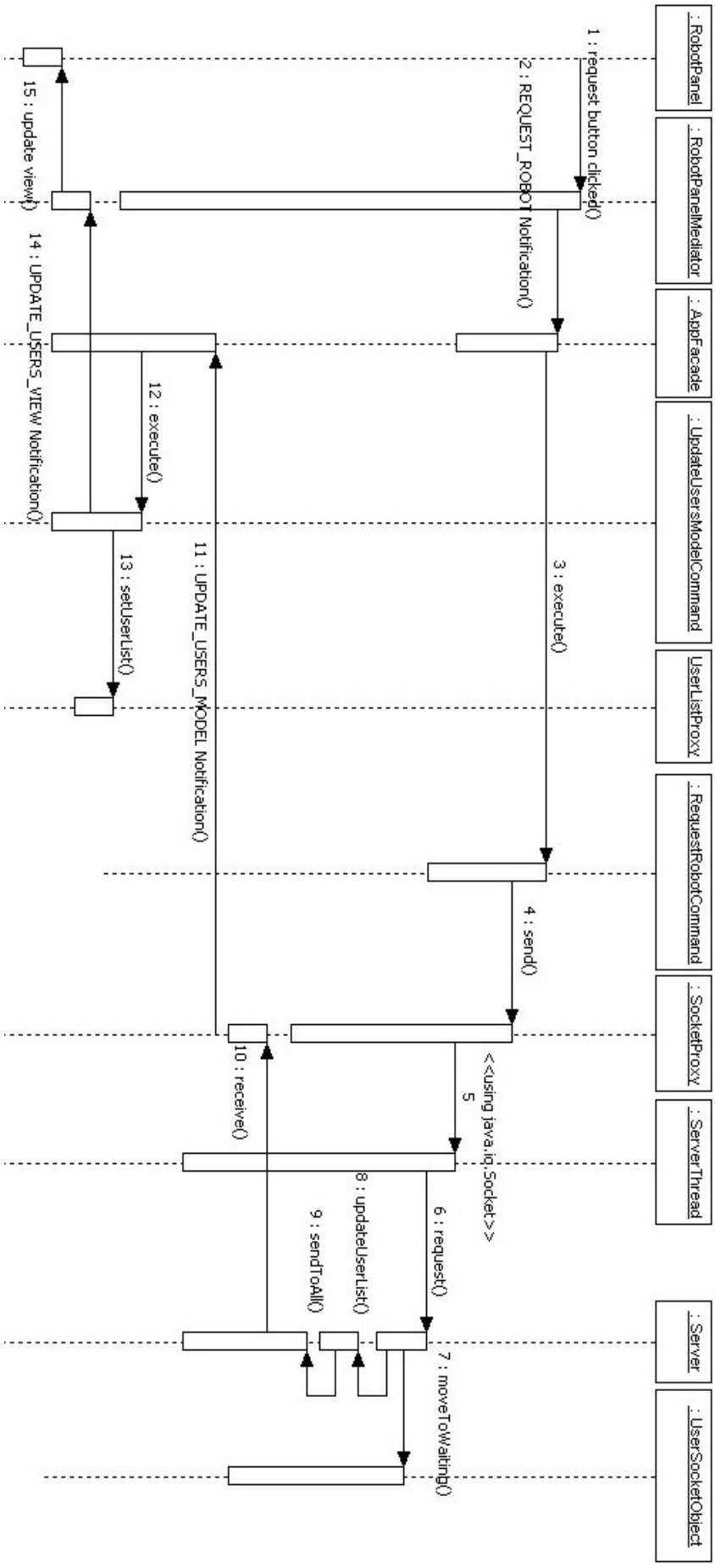


Fig 9.7 - requesting the robot

1. The 'request robot' swing button is clicked in the RobotPanel (the right hand side swing component containing the user list and the request/relinquish/send buttons)
2. A REQUEST_ROBOT Notification is sent
3. The RequestRobotCommand is executed.
4. A message is sent to the SocketProxy to send a RequestRobotSocketObject
5. A RequestRobotSocketObject is sent over the Socket
6. The Server (which maintains the user list) is told that a user has requested the robot
7. The user is moved from the not waiting queue to the waiting queue
8. updateUserList() is called...
9. ... which sends the UserListObject to all members logged in
10. The SocketProxy (listening in its run() method) receives the UsersocketObject.
11. It sends a Notification to store the users in a Proxy so that it can be obtained later by any part of the application
12. UpdateUserModelCommand is executed
13. The proxy stores the new users list
14. A Notification is sent telling any listening Mediators to update their view.
15. The RobotPanelMediator tells its Swing component to update its view.

10.2 Navigation, logging in and out

Navigation between the three main screens (Login, MyFiles, CreateLogo) is not as simple as it first seems. The Swing components are all created immediately when the application is launched. The background syntax checking should not be started if the user has not logged in, and the Socket should also not be connected. On log-out the syntax checking should be stopped and the Socket closed. A simple way to achieve this would be to use Notifications - start/stop background syntax checking and open/close the Socket. It is not ideal to have all the classes instantiated even before the user logs in (and might never succeed to log in), so I developed a cleaner solution, explained below.

The SocketProxy is registered in the StartUpCommand along with all other proxies, but it is not connected until you log in successfully. When the users clicks on the log in button an ATTEMPT_LOGIN Notification is sent, and the AttemptLoginCommand is executed. The username and password are contained in the body of the Notification in a User object which is forwarded to the UserProxy. In iteration 2 the authentication is 100% hard-coded - username and password combinations {userAAA, userAAA} are accepted (AAA any characters). In iteration 3 this would be replaced with a call to a database.

After a successful login, a LOGIN_SUCCESS Notification is sent and the LoginSuccessCommand executed which in turn does three things:

- Sends a Notification called START_BUILD_APP which is heard by the MainPanelMediator. Mediators for the MyFiles screen and the CreateLogo screen are created and registered. They do not exist up to this point.
- Send a Notification called GOTO_MY_FILES which is heard by the MainPanelMediator and the user moves to the MyFiles screen.
- The Socket in the SocketProxy is connected.

On log-out a command called LogoutCommand is executed, which does the following:

- Stops any Logo being drawn
- Clears the Logo text area, any errors on the text area, the error panel and the turtle graphics
- Sends a Notification called STOP_APP which is heard by the MainPanelMediator. The Mediators for the MyFiles screen and the CreateLogo screen are unregistered - they no longer exist in the PureMVC framework and can be garbage collected.
- Send a Notification called GOTO_LOGIN which is heard by the MainPanelMediator and the user moves to the Login screen.
- The Socket in the SocketProxy is disconnected (the listening thread returns from its `run()` method and a LogoutSocketObject is sent over the Socket.

To allow certain cleaning up tasks before Mediators are unregistered, the Mediator class from the org.puremvc.java.patterns package is extended by a completely new class called AMediator (com.jgrindall.logo.views.AMediator).

All of the Mediators in the Logo application extend this class and it contains a `destroy()` method which subclasses implement by:

- deleting any event listeners they may have created

- iterating through their children Mediators and destroying them
- destroying any objects specific to themselves. The Mediator for the Background syntax checking stops the syntax checking thread for example.
- The Decorator classes (See section 10.3.2) implement this by destroying themselves as well as calling `wrappedMediator.destroy()`.

10.3 The AdminApp

One missing aspect of the use cases in Section 3 is the application that is running on the machine that is connected to the robot. Running the com.jgrindall.logo.server.Server.java class (see Section 7.5) is simple to do using NetBeans, or at the command prompt, but I decided to create a small Swing application that is run by the Administrator (the person using the machine to which the robot is connected). The functionality is:

- Start and stop buttons to initiate and stop the Server class
- Two text panels:
  - One to contain messages to report to the Admin such as people logging in and out
  - Another showing the list of users logged on to the system.

The application also writes date, time and error messages to a log file (robologo.log) to help in diagnosing any problems. A screenshot of this application is found in Appendix A.

10.4 Final PureMVC classes - a reference

For reference, a summary of all the elements in the application follows:

Proxies and their purpose

LogoProxy	Store the Logo string to make it accessible to the rest of the application. Updated by the StoreLogoCommand.
LogoProcessingProxy	Uses the Parser to process Logo. Receives LogoCommandObjects (Fd, Rt) from the Parser and sends DRAW_LOGO notifications to the TurtleCanvas
SocketProxy	Send and receive all messages across the socket
UserProxy	Store a reference to the user logged in and check authentication (not yet implemented)
UserListProxy	Store a reference to all users logged in

Notifications and their purpose

STARTUP	Required by PureMVC, described in section 7.1.6). Results in StartUpCommand being executed.
GOTO_CREATE GOTO_MYFILES GOTO_LOGIN	Listened for by MainPanelMediator - for navigating between the three main screens.
ATTEMPT_LOGIN	Sent when a user clicks on the log in button. Results in the AttemptLoginCommand being executed. The body contains the username and password.
LOGIN_SUCCESS	Success results in LoginSuccessCommand being executed.
STORE_LOGO	Sent when the Logo is edited. Results in the StoreLogoCommand being executed, the body contains the Logo.
PROCESS_LOGO	Sent when the user clicks 'draw'. Results in the ProcessLogoCommand being executed.
SEND_TO_ROBOT	Sent when the user clicks 'send to robot'. Results in the SendToRobotCommand being executed.
STOP_ROBOT	Sent when the user clicks 'stop robot'. Results in the StopRobotCommand being executed.
REQUEST_ROBOT	Sent when the robot clicks 'Request'. Results in the RequestRobotCommand being executed.
RELINQUISH_ROBOT	Sent when the robot clicks 'Relinquish'. Results in the RequestRobotCommand being executed.
LOGOUT	Sent when the user clicks Logout. Results in the LogoutCommand being executed.
LOGIN_SUCCESS	Sent when the user has been authenticated.
CLEAR_ERROR	Listened for by the LogoPanelMediator - clears any error highlighting. For example when the user clicks draw.
CLEAR_ERROR_TEXT	Listened for by the ErrorPanelMediator - clears any error messages. For example when the user clicks draw.
CLEAR_LOGO	Listened for by the LogoPanelMediator - clears any Logo, for example when the user logs out.
CLEAR_DRAWING	Listened for by the TurtlePanelMediator - clears any turtle graphics. For example when the user clicks draw.
START_BUILDAPP	Register Mediators for the my files screen and create screen, and start the syntax checking
STOP_APP	Remove the Mediators for the myFiles screen and the create screen, stop any syntax checking.
SHOW_POPUP	Listened for by the MainPanelMediator - displays a popup message.
PARSE_ERROR	Listened for by the ErrorPanelMediator and LogoPanelMediator - results in an error message being displayed and error line highlighted.
UPDATE_USERS_MODEL	Results in UpdateUsersModelCommand being executed. Body contains the list of users.
UPDATE_USERS_VIEW	Listened for by the RobotPanelMediator - updates the view of the user list.

BAD_LOGO_SENT_TO_ROBOT	Sent when the SendToRobotProxy fails to parse the Logo.
DRAW_LOGO	Sent by the ProcessLogoProxy when it receives an error in drawing Logo on screen. Body contains a LogoCommand (forward or right). Listened for by the TurtleCanvasMediator.
START_DRAWING_UPDATE_BUTTONS, STOP_DRAWING_UPDATE_BUTTONS	Listened for by the LogoPanelMediator - enables/disables the draw/stop buttons when the user clicks start, stop and when Logo has finished drawing on screen.
HIGHLIGHT_ERROR	Listened for by the LogoPanelMediator - draws a highlight on a specific line.
DISABLE_SOCKET_BUTTONS	Listened for by the RobotPanelMediator - disables all the request/relinquish/sent to robot buttons.
GO_BACK_CLICKED	Results in GoBackCommand being executed.

Mediators and their Notification interests

AMediator	none
LogoPanelMediator	CLEAR_LOGO, START_DRAWING_UPDATE_BUTTONS, STOP_DRAWING_UPDATE_BUTTONS, HIGHLIGHT_ERROR
LoginPanelMediator	none
CreatePanelMediator	none
EditorPanelMediator	CLEAR_ERROR_TEXT, PARSE_ERROR
ErrorPanelMediator	none
FilesPanelMediator	none
MainPanelMediator	GOTO_CREATE_SCREEN, GOTO_FILES_SCREEN, STOP_APP, GOTO_LOGIN_SCREEN, LOGIN_FAIL, LOG_OUT, START_BUILDAPP
RightPanelMediator	none
RobotPanelMediator	UPDATE_USERS_VIEW, BAD_LOGO_SENT_TO_ROBOT, DISABLE_SOCKET_BUTTONS
TurtlePanelMediator	DRAW_LOGO, DEBUG, CLEAR_DRAWING

Commands and their purpose

StartUpCommand	Register proxies and Mediators
AttemptLoginCommand	Send username and password to UserProxy for authentication.
LoginSuccessCommand	Send START_BUILDAPP Notification, GOTO_MY_FILES Notification, and connect socket.
StoreLogoCommand	Take Logo from body of notification and store in LogoProxy so anyone can access the Logo.
ProcessLogoCommand	Take Logo from LogoProxy and send it to the LogoProcessingProxy.
SendToRobotCommand	Take Logo from LogoProxy, and pass too SocketProxy.

StopDrawingCommand	Stops the LogoProcessingProxy from processing Logo (terminates the `run()` method)
StopRobotCommand	Tell SocketProxy to send the StopRobot object over the Socket.
RequestRobotCommand	Tell SocketProxy to send the RequestRobot object over the Socket.
RelinquishRobotCommand	Tell SocketProxy to send the RelinquishRobot object over the Socket.
LogoutCommand	Disconnect the socket, log out the user, clear the Logo and any drawing and any errors.
UpdateUsersModelCommand	Take user list from body of notification and store in the UserListProxy.
GoBackCommand	Executed when the users clicks 'go back' from the CreateLogo screen. Stops any drawing happening, clears any Logo and errors and returns user to the MyFiles screen.

11 Deployment

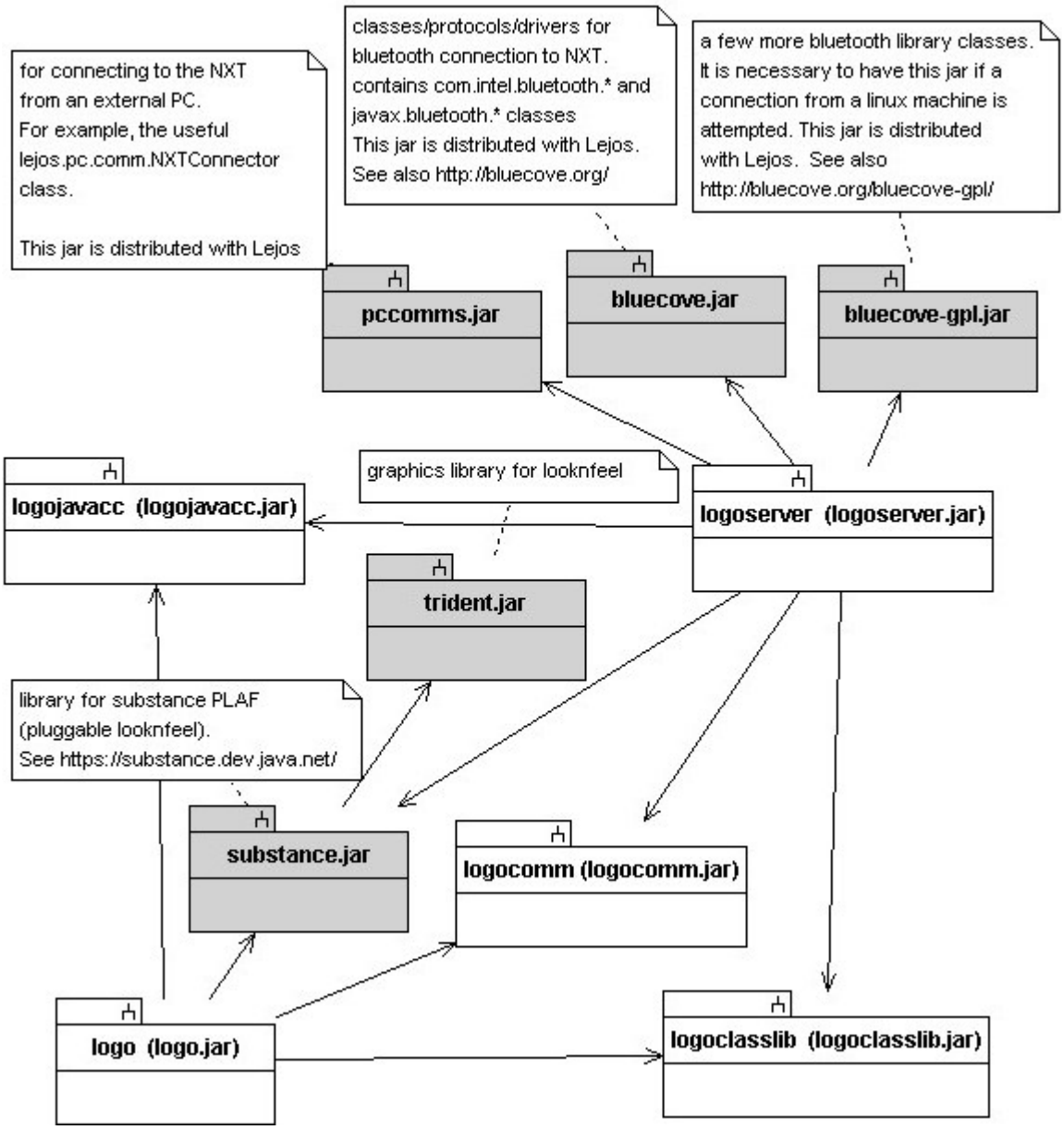
11.1 Projects, libraries and dependencies

I developed the application in separate projects in NetBeans 6.8. The code splits into the following projects and packages:

Project name and description	Packages	Description of package
logo The Swing client	com.jgrindall.logo	Contains main executables for deployment as exe and applet. Also the main swing containers.
	com.jgrindall.logo.commands	Contains all PureMVC Command classes
	com.jgrindall.logo.images	Contains a singleton which loads png files for icons
	com.jgrindall.logo.math	Contains one class for performing some geometry to draw the turtle.
	com.jgrindall.logo.proxy	Contains all PureMVC Proxy classes
	com.jgrindall.logo.threads	Contains thread class for background syntax checking
	com.jgrindall.logo.utils	Contains various helper classes - Memento, TextLocationObject, class for loading look'n'feel for example
	com.jgrindall.logo.views	Contains PureMVC Mediators
	com.jgrindall.logo.components	Contains swing components
	org.puremvc.java.*	PureMVC library - included as code rather than jar
logoclasslib A class library used by the client and the backend	com.jgrindall.logo.socket	A shared class library containing classes used by the client and backend - namely objects sent over the socket.
	com.jgrindall.logo.event	Contains classes used to dispatch custom events throughout the system. For example, ILogoEventDispatcher, ILogoEventListener and ILogoEvent.
logojavacc Logo Interpreter plus helper classes	com.jgrindall.logojavacc	JavaCC interpreter
logocomm Shared class library. Logo command objects (fd, rt) and utilities to work with them.	com.jgrindall.logojavacc.comms	Classes shared by client, and backend and robot relating to use of the interpreter - for example LogoFdObject, LogoRtObject

logorobot Lejos-compiled Java running on the robot	com.jgrindall.logo.robot	Code running on the robot.
	com.jgrindall.logo.robot.actions	Actions that the robot can perform - namely Fd and Rt actions.
logoserver The backend	com.jgrindall.logo.server	The backend, multithreaded socket server which connects to clients, maintains user list and connects to robot by bluetooth.

Each project is built to give a jar file, and the dependencies amongst the projects (and other 3rd party libraries) are illustrated in Fig 11.1:



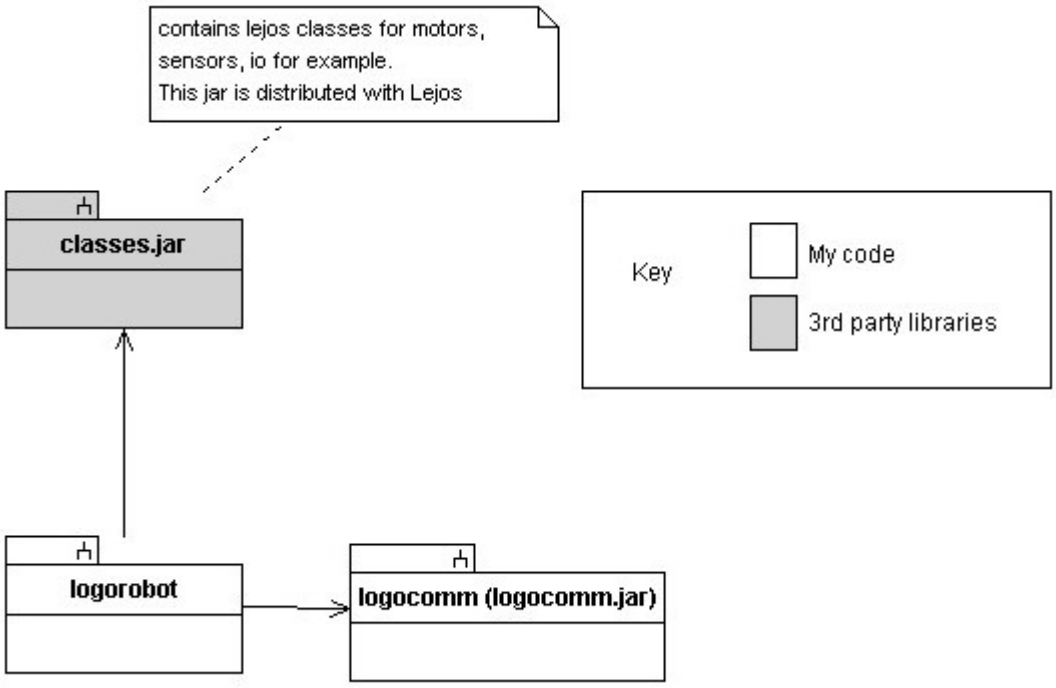


Fig 11.1 - final dependencies between projects and compiled jar files

11.2 Deploying the application

As illustrated in Fig 5.1, the application and database (the database is part of iteration 3) are to be hosted on one central server. If iteration 3 were finished, the application would be hosted on a web application server, and the database connectivity would have to be implemented in a server side language. But since iteration 3 is not finished, it is currently just html files, hosted on www.johngrindall.com/robologo.

The application could be deployed as a standalone, an applet or a JAWS (Java web start - see [28], [33]) application. The Java sources include main classes that extend both JFrame and JApplet, so either is possible.

The HTML pages on www.johngrindall.com/robologo contain some background information and a link to launch the JAWS application. I have not included the applet because I found the user experience terrible; the page did not refresh properly and the page often became unresponsive. JAWS has the other advantage that applications are downloaded and cached so that they can be launched quickly, and also outside of the browser.

The file to be run on the machine that is connected to the robot (ie. starting up the Socket server) is the AdminApp class, which starts the Socket and displays information to the user. Originally I packaged all the files as a .war file (web application archive) to be run on Tomcat, and I used an extra line in the web.xml file called a context listener to kick off the socket server (See [5]). I realised this is very over the top - and now simple .bat (for Windows) and .sh (Linux) scripts are used to launch the application (java -jar launches the file)

To perform the final build and deployment I created one more folder structure on my local machine containing the following:

i) logoKeystore - a keystore that can be used to sign the jar files with a certificate for JAWS.
ii) A script to build the final folder structure for distribution

The script is found in Appendix E. It copies the jar files from the locations inside the projects' build folders, signs them (for JAWS security), and copies them to the folder 'dist' (see iii)

iii) A folder called 'dist' that contains the actual final files

This consists of:

1. The .nxj file that runs on the NXT robot
2. A folder called 'appserver' that contains the main application files to be run on johngrindall.com (containing html files, the jnlp file (JAWS file, see Appendix E) and relevant jars.
3. A folder called 'robotserver' that contains the files to be run on the machine which is connected to the robot. This contains the logoserver.jar file and other relevant jar files. It also contains scripts (.bat and .sh) to start the application. Double clicking a jar works on Windows but not on Linux. Details for how these files should be run are found in the user manual (Appendix F) and in the readme file in the dist folder.

11.3 Bluetooth connections

Connecting the NXT robot to any computer is an awkward task. The first bluetooth dongle I bought refused to connect outright from either my Windows Vista machine, or my Linux Mint machine. A forum posting directed me to another brand of dongle which works fine on Windows provided you say “cancel” when Windows tries to install the driver (!) It is awkward to set up on my Linux laptop because the NXT has a four digit pin and Linux kept displaying a popup (for a fraction of a second)

telling me to enter a 6 digit pin. Connecting the other way round (searching for laptop from NXT) has never worked for me. Eventually the method I got working in Linux Mint is to set the PIN on the NXT as 1111, deselect the “set PIN as 1111” checkbox on Linux, and enter 1111 as a 'custom' pin. Installing the necessary bluetooth drivers is also necessary but my particular distribution came with them pre-installed so I have done little research on the subject.

12 Testing

12.1 Unit tests

I carried unit tests for the JavaCC interpreter and for the decoding/encoding of commands in the class LogoCommandUtils (which translates logo command objects into strings and back - see 8.10). These are the main 'algorithm' parts of the system.

Testing of the interpreter is found in section 4.3. I tested the LogoCommandUtils object using a test class com.jgrindall.logo.comms.test.Main.java, found in the Appendices.

As pseudocode, it is shown in Fig 12.1:

com.jgrindall.logo.comms.test.Main.java:

```
public Main(){
   Vector<String> tests = new Vector<String>();
   // fill vector with correct and incorrect strings.
   // loop through each test case
   for(int i=0;i<=tests.size()-1;i++){
      String test = tests.elementAt(i);
      try{
         // convert string to arraylist
         // loop through arraylist
         // convert each to a string and concatenate
         // check the answer matches the input
       catch(Exception e){
         // output exception
      }
   }
}
```

Fig 12.1 - testing the LogoCommandUtils class

Strings (correct and incorrect) are converted into array lists of objects and then converted back. It is not possible to check if the original and final strings are equal because a number “2” is converted to “2.0” for example - so checked by eye.

My text cases are as follows:

```
// correct strings
tests.add("fd:100;");
tests.add("fd:78.9875;");
tests.add("fd:78.9875;rt:65;");
tests.add("fd:78.9875;rt:65.0;fd:-19.8;");
tests.add("rt:78.9875;rt:65.0;fd:-19.8;end:0;");
tests.add("end:0;");
tests.add("rt:78.9875;rt:65.0;fd:-19.8;rt:-100.0;end:0;");

// incorrect strings
tests.add("fd:100");                // ; missing
tests.add("fd 78.9875;");           // : missing
```

```
tests.add("fd:;rt:65;");            // extra ;
tests.add("");                      // empty is not allowed
tests.add("end:0;;");               // empty command
tests.add("end");             // missing data
tests.add("rt:78.9875;rt:65.0;fd:-19.8;rt:-100.0end:0;"); //; missing
```

12.2 System tests

The System tests include testing the functional and non-functional requirements.

12.2.1 Functional requirements

The functional requirements are found in Section 3. The use cases were tested as follows:

12.2.1.1 Entering and editing Logo (UC_editLogo, UC_undoEdit, UC_redoEdit)

Test entering Logo

Test #	Test case	Expected	Actual	Pass/fail/notes
1	Test large file (not Logo)	Text area should remain responsive. Errors should be detected and displayed to the user correctly. The LineNumberedBorder should function correctly (no overlapping)	As expected.	Pass I tested with 10000 lines of Java. Going over 50000 lines does make the text area very sluggish.
2	Test large file (Logo that also has to be parsed)	As above	As expected	Pass. I tested with 10000 lines of correct Logo. Going over 50000 lines does make the test area very sluggish.
3	Test character set. I tested a mixture of Arabic, French and UTF-16 Chinese.	Token errors	As expected	Pass. I am checking that the app doesn't crash when it tries to process the characters.

Fig 12.2 - testing entering/editing Logo

Test undo/redo

Test #	Test case	Expected	Actual	Pass/fail/notes
4	Test simple undo and redo, using typed characters.	Should undo up to 8 times, and redo up to 8 times if you haven't typed.	As expected.	Pass.
5	Test undo and redo using copy/paste commands	As above, the fact that the text is copied or pasted should have no effect.	As expected	Pass.
6	Test undo and redo using very large copy/paste commands	As above.	As expected	Pass. I tested with copy/paste sections of Logo of size 4000 characters.

Fig 12.3 - testing undo/redo

12.2.1.2 Drawing Logo on screen (UC_testLogoLocally, UC_stopLogoLocally)

All 42 tests that used to test the interpreter (see section 4.3) were also run on the on-screen turtle. All executed successfully (apart from the fact that commands such as fd 0.01 don't do much!) I also made a number of extra test cases to check the multithreading and the graphics capabilities.

These are:

Test #	Test case	Purpose	Expected	Actual	Pass/fail/notes
7	rpt 100000[rpt 1000000[fd 50 rt 90]]	Infinite loop to check app remains responsive.	App remains responsive. Logo can be stopped.	App is responsive but becomes sluggish. Logo can be stopped.	?Fail
8	rpt 1000000[rpt 100000[fd 10000]]	Check what happens to graphics with overflow.	App remains responsive. Logo can be stopped. Straight line is drawn (north)	App is responsive but becomes sluggish. Logo is correctly drawn and can be stopped	?Fail
9	proc drawPoly(:n,:s) rpt :n [fd :s rt (360/:n)] end drawPoly(5,1000000000000000)	As above.	Large Pentagon?	Pentagon is never finished.	? This is presumably due to overflow inside the Java language.

Fig 12.4 - testing drawing Logo on screen.

Notes:

- The application becomes sluggish because even though the processing is done off the Swing

EDT (Event dispatching thread), the repaint() method *is* called on the EDT. I'm not sure if repainting a Swing component can happen off the EDT, it needs more research.

- It is certainly possible to use Java rounding errors (underflow, overflow) to make various "incorrect" patterns from the Logo. I can't do anything about this, neither do I consider them bugs.

12.2.1.3 Logging in and out (UC_logIn, UC_logOut)

Test #	Test case	Expected	Actual	Pass/fail/notes
10	correct username/password	Successful login	As expected	Pass
11	incorrect username/password	Error message - wrong username/password	As expected	Pass
12	empty username/password	Error message - wrong username/password	As expected	Pass
13	multiple logins	Error message - allowed to login, but not allowed to connect to Socket	As expected	Pass
14	Try to connect to robot on wrong IP address	Allow login but error message for Socket	As expected	Pass
15	logout normally (while not drawing nor in charge of robot)	Redirected to log in screen. User deleted from other users' user list.	As expected, but unnecessary error message if connected to Socket.	? Pass I tested with 6 users logged on
16	logout while drawing on-screen	Redirected to log in screen. User deleted from other users' user list.	As expected, but unnecessary error message if connected to Socket.	? Pass I tested with 6 users logged on
17	logout while in charge of robot while not executing on robot.	Redirected to log in screen. User deleted from other users' user list and control of robot passes to next person waiting.	As expected, but unnecessary error message	? Pass I tested with 6 users logged on
18	logout while in charge of robot and executing on robot.	User deleted from other users' user list. Robot stops. Control of robot passes to next person waiting.	As expected	Pass I tested with 6 users logged on

19	close application without logging out while neither drawing nor in charge of robot	User deleted from other users' user list.	As expected	Pass I tested with 6 users logged on
20	close application without logging out while drawing but not in charge of robot	User deleted from other users' user list.	As expected	Pass I tested with 6 users logged on
21	close application while in charge of robot but not executing	User deleted from other users' user list.	As expected	Pass I tested with 6 users logged on
22	close application while in charge of robot and it is executing	User deleted from other users' user list. Control of robot passes to next person waiting.	As expected	Pass I tested with 6 users logged on
23	Test number of users logged on	Normal behaviour	As expected	Pass I tested 20 users logged on simultaneously

Fig 12.5 - testing logging in and out

Notes

- I have not tested pressing the back button while executing on the robot. This test is missing.
- Sometimes the robot stops after executing the command it is doing, not immediately. Since I have limited all commands to 10 seconds I think this is acceptable. In fact the robot beeps when it is ready for the next set of commands (from any user) and I have not tested what happens if commands are sent in between the time when the robot beeps and when it is ready for commands.
- Tests 17,18,21,22 need a lot more testing, with more users on different machines and in combination with other test cases.
- More load testing is probably needed - a large group of people all interacting with the robot for a long period of time.
- It is possible for the NXT to switch itself off if the batteries run out.

12.2.1.4 Requesting and relinquishing control (UC_requestControl, UC_relinquishControl, UC_updateUserList)

Test #	Test case	Expected	Actual	Pass/fail/notes
24	request, with some people in waiting queue	Name added to waiting list. Request button disabled.	As expected	Pass I tested with 4 users logged on
25	request, with waiting queue empty	Robot control granted. Request button disabled, robot buttons enabled.	As expected	Pass I tested with 4 users logged on
26	relinquish with no-one in waiting queue	Robot control removed, all request buttons enabled.	As expected	Pass I tested with 4 users logged on
27	relinquish with some people in waiting queue	Robot control removed, next person is waiting queue is granted control and their buttons are update	As expected	Pass I tested with 4 users logged on

Fig 12.6 - testing request/relinquish

12.2.1.5 Robot functionality (UC_UC_executeOnRobot, UC_stopExecutionOnRobot)

I re-ran all 42 tests used to test the interpreter (see section 4.3) on the robot. The results are shown in the "Test on Robot" column of Appendix D. As described in section 8.10, when the robot is told to turn the motors on and wait 'x' seconds, it rejects this command if x is greater than 10. This accounts for test case number 38. Commands such as "fd -100" turn the motors on backwards, which makes my robot fall over sometimes, but this is a fault of the Lego not the Logo!

I also ran some extra test cases:

Test #	Test case	Expected	Actual	Pass/fail/notes
28	fd 100000	Error message "number too large". Robot returns to idle state.	As expected.	Pass
29	rt 10000	Error message "number too large". Robot returns to idle state.	As expected.	Pass
30	rt -90 fd -100	Robot should turn left 90 degrees and move backwards 100.	As expected.	Pass.

Fig 12.7 - testing execution on robot

12.2.1.6 Test communications/IO

Test #	Test case	Expected	Actual	Pass/fail/ notes
31	Log in and try to connect to server without it having been started	Error message but can proceed and edit Logo and test it on screen.	As expected. Can log out log in again if Server is started in between	Pass
32	Socket shut down while logged in but not in charge of the robot	? Error message	All users receive an error message. They would be able to save their work. The robot buttons are not disabled, they probably should be. If the Socket is restarted it will have the wrong user list!	Fail
33	Socket shut down while logged in and in charge of the robot	? Error message	As above	Fail
34	Socket shut down while executing on robot	? Error message	Error message is displayed. Robot cannot be stopped executing. Stop button is enabled but does not work. Null pointer exception thrown.	Fail
35	Running multiple instances of the socket server	Error message	As expected	Pass
37	Turn off robot/stop .nxj program while executing	? Error message	Stop robot button is still enabled but does not work. Null pointer exception thrown.	Fail
38	Send Logo to robot when not running .nxj program	? Error message	'stop robot' button is enabled but not functional. If clicked it becomes disabled but no-one can use the robot from now on without complete restart.	Fail
39	Send Logo to robot when not switched on	? Error message	Error message. 'run on robot button' enabled. If the .nxj program is run now, it can be executed	Pass
43	Move robot out of bluetooth distance while executing	? Error message	Robot pauses executing while out of range and resumes when back in range. (Needs more testing)	Pass

Fig 12.8 - testing communications/IO

Notes:

- All the above need a lot more testing; for example it takes a second or so for the user to get feedback from test case 39, what if something else happens in the interim?

12.2.2 Non-functional requirements

Performance

I performed the following speed tests for drawing Logo, sending Logo to the robot and updating the user list:

<table>
<tr><th>Test #</th><th>Test case</th><th>Time taken</th></tr>
<tr><td>44</td><td>Drawing line segments
rpt N [fd 50 rt 90]</td><td>
<table>
<tr><th>N (× 10^6)</th><th>Time (approx, in seconds)</th></tr>
<tr><td>0.1</td><td>≈ 1</td></tr>
<tr><td>0.5</td><td>≈ 5</td></tr>
<tr><td>1</td><td>≈ 12</td></tr>
<tr><td>5</td><td>≈ 70</td></tr>
<tr><td>10</td><td>≈ 140</td></tr>
</table>
* Windows Vista, 2Ghz, 3GB Ram
</td></tr>
<tr><td>48</td><td>Time taken between clicking send and robot moving</td><td>≈ 1 second</td></tr>
<tr><td>49</td><td>Time taken between relinquishing robot and user list updating other users' view.</td><td>A fraction of a second.
I tested with 20 users logged in on two different machines on the same WLAN.</td></tr>
</table>

Fig 12.9 - testing performance

- Drawing appears to be linear in N (approx 1.4N in units of millions and seconds on my laptop). I think this is fine for a school student. It could with some optimisation - the entire 2000×1000px is being redrawn because this is the only way I could get the resize functionality working.

Security

Iteration 3 is not started so security issues with user-names and passwords are not relevant yet The Socket is hopelessly insecure - anyone could log on to the same server and port. I don't see this as a problem because the application is meant to be used in a classroom situation.

Portability & OS

I tested combinations of the Socket server running on a Windows Vista and a Linux Mint machine, and downloading the jnlp file/running the main application on the Windows Vista machine, a Windows XP machine and Linux Mint machine. I tested in Firefox and IE8. No errors were found - it is the right version of JRE that is presumably vital and all my machines have an up to date version. I have not tested

what happens in Java is not installed but of course it would not work.

Reliability

The application is reliable for drawing Logo on screen and it is *very unreliable* when it comes to unexpected events such as things being switched off (see failed test cases above!). The reliability is far too low to consider the software ready for use in a classroom with real users! See the conclusions chapter below for more information.

Look and feel / GUI

I am very happy with the look and feel of the application. The substance look'n'feel (See [31]) is not particularly bright or colourful, but it is ok. I think the icons and the help file are reasonably user friendly. The available area for drawing Logo on screen is too small when the application is viewed at its default size 800x600, but can be increased easily. It is annoying that you cannot copy/paste from the help file, zoom in/out on the Logo window and clear the graphics, but these would all be easy to implement.

13 Conclusions

13.1 Summary of progress / comparison to project aims

Iterations 1 and 2 of the project have been completed. Iteration 3 (the database containing user authentication and storing files) has not been started. User authentication is currently hard-coded into the application. Files cannot be saved and loaded; the “My Files” screen is a dummy screen at the moment and starting a new Logo file is the only way to proceed to the next screen.

Iterations 1 and 2 (the core functionality of users being able to work locally and on a robot) have been completed successfully. I am not disappointed that Iteration 3 is not started - I think it is the simplest and least interesting. I found the project a enjoyable and a huge learning experience, and with hindsight, dealing with all the situations and bugs shown in Section 12.2.1.6 would probably have been too ambitious.

13.2 Requirements / Lifecycle activities

The requirements were useful at the start of the project but now they do not look detailed enough. The admin app was not described at all - I thought that the application would be deployed on a server and that was that, whereas in reality the application needs to be started/stopped.

I did not realise at the outset that by far the most serious problems with the code would be in stopping Logo being drawn after it had started, and what happens when users log out or just close their windows, rather than the relatively simple problems of getting Logo to the robot to start with! A lot more detail is needed in the use cases regarding error handling and handling unexpected situations. This is why many of my tests list the expected result as unknown. It would have been better to spend more time on the detail of the requirements and on writing test cases soon after listing the requirements rather than at the end. It may also have been better to include more detail regarding background syntax checking and error reporting in the use cases.

In retrospect, Iteration 1 was far too large - it would have been better to make it much more fine grained and to include 'executing Logo locally' as one iteration, and 'executing Logo on the robot' as a second.

Subversion was very useful - reverting files when I went up a cul-de-sac, or if an old version had something working and I needed to get the details. It was especially useful for storing this document and the UML diagrams, and for peace of mind of course.

Although writing Lejos programs requires a little bit of tinkering with ANT to add libraries, if time allowed I would have learnt a little more about ANT and used it to build my jars and the final files for distribution. Since there are only 6 jar files it was easy to build them in the right order by hand.

My design and implementation chapters are very similar - a lot of my project was sketching designs and then experimenting and adapting them. I cannot include all the improvements I made because the write up would be far too long. I learnt a huge amount about PureMVC, interpreters, streams and threads - none of which I knew very much of before starting the project.

I did not write test cases before starting development, this would have made the testing chapters easier and would have exposed several problems such as being able to stop the robot mid-processing, and thinking about what happens when a command such as “fd 1000000000” is successfully sent to the robot and it begins to execute it.

13.3 Design & implementation

13.3.1 The Logo interpreter

The interpreter was more of a conceptual challenge than something that involved a great deal of coding. The grammar file and the visitor file are both quite small, but at the outset of the project I envisaged writing the interpreter 'by hand'. Learning about interpreters and JavaCC was a significant chunk of the time spent on the project. I think the final interpreter is of very high quality - the testing was very thorough and the errors it outputs are quite useful. “Extra” errors such as division by zero and Java stack overflow errors are well handled.

13.3.2 PureMVC

I was not familiar with PureMVC before starting the project. I like the framework for many reasons. It certainly does force the programmer to separate logic, model and view. This helps keep the applications

classes loosely coupled - parts of the system essentially subscribe to notifications they are interested to without knowing or caring who sent them. It also means that another developer coming to the code will be able to understand the code quite easily; once one has understood the vocabulary and three basic ideas of the framework (data is managed in Proxies, views are managed by Mediators, Commands contain logic and use Proxies, Notifications trigger Commands) and has a list of the classes used such as those I have put in Section 10.4, another developer can quite quickly understand exactly how the code works.

I also like the way that entire subsections of the application can be switched on and off by registering or un-registering Mediators and/or Proxies. For example, switching off the background syntax checking and the Socket when a user logs out was something that occurred to me quite late on in the project and it was very easy to implement because the GUI, mediators, model and commands are all very loosely coupled.

I have some reservations about how I have used PureMVC. I have placed a lot of code into Proxies. My Proxies tend to do a lot of work rather than just store data. Perhaps it is better to use a Proxy as a gateway for data in/out and actually do the work in other classes.

The way that PureMVC registers Mediators and Proxies by “NAME” is also a cause for concern. Swapping one implementation of a class/part of a class for another is hampered by this. I have tried to use interfaces where possible (for example ISocket, ILogoProcess and ALogoPanelMediator) but have resisted putting an interface for every single such class since PureMVC already results in a lot of classes!

The NAME variable also means that adding multiple instances of the same class would be difficult. Imagine that I decided to have separate panels for each procedure rather than having just one text field. Presumably this would require a number of Mediators with names such as TextAreaMediator1, TextAreaMediator2 etc. Presumably they would be interested in the same Notifications and so to distinguish which object is meant to act on each would be awkward. The documentation [13] suggests using the extra 'type' parameter of a Notification to do this, but it seems messy and against the spirit of

OOP.

Another way in which the framework goes against the idea of OOP is that objects do not really encapsulate their data and their methods to act on it. Rather, an object is spread around between at least three classes. I think this is not a problem with PureMVC in particular but is part of the MVC pattern itself, but one possible problem with PureMVC is that any class could get hold of any Proxy or any Mediator and change something. It also means that sections of code cannot be easily stripped out and re-used in other projects for example - rather than having a public API the subsystems rely on the PureMVC framework being used again, and the same '*interaction → mediator → notification → command*' communication being used again.

Since starting this project I have used the Actionscript 3 port of PureMVC and I think it is well suited to developing GUIs *rapidly*. It is simple to add and remove functionality using PureMVC, but I am not sure that it is suited to large re factorings.

13.3.3 Drawing Logo locally

I am very happy with the on-screen drawing. It is fast enough and I am particularly pleased with the way the graphics are re-drawn (see section 8.3). The multithreading works acceptably well. At some point the Swing EDT redraws the screen and this is why the application becomes sluggish. I would need to research ways around this, if any.

One major problem I faced throughout was how to stop processes in different threads executing and I think the check( ) method in the ILogoConsumer class works well (See 8.6).

Regarding code quality, I do not especially like the way that the parser is stopped; it basically throws Exceptions rather than being stopped nicely, but I decided to stay with the JavaCC generated files (which only throw ParseExceptions) and work around this. In fact a lot of the 'stopping things' behaviour is handled by trying to tidy up after throwing and Exception. This is not a good way of managing the complicated behaviour and results in the failed test cases in Section 12.2.1.6

13.3.4 Executing Logo on the robot

I feel the code running on the robot is of quite high quality - I tried to keep in mind that the robot should

be abstracted as much as possible, even though the NXT is the only one I own. The software is very buggy when it comes to being able to recover from unexpected errors (robot switched off, out of range, someone closes a window). These all cause serious errors to the entire application. I am a little disappointed that I have no way of solving these, and I think in retrospect trying to stream messages back and forth was a bad idea and resulting in bug explosion. See 13.4 for more details.

13.3.5 User Management

This aspect of the backend is working nicely apart from the serious errors described above (for example user using the robot closes his window). I could replace all the synchronization with thread safe collection classes, but it was useful for me to learn about thread safe programming!

13.4 Technologies

I am reasonably happy with my choice of Java, but towards the end of the project I realised I had not spent enough time researching options. For example, I discovered a Ruby library that can send commands to the NXT directly (rather than having the problems of streaming). Avoiding streaming would have made the application simpler, more reliable, more modular and easier to test.

I spent a little time experimenting with using nxjc.bat and nxjupload.bat from Java (Process myProcess = Runtime.getRuntime().exec("batch file here");). Combined with some way of translating from Logo to Java, it would have been great to write Java to a file and then compile and send the Logo as one executable to the NXT. That way the server can just perform one communication to the NXT and the user presses the run button - much simpler than my streaming. Perhaps having the visitor pattern and the parse tree done it would have been possible. Fd statements would be converted into “Motor.A.start() thread.sleep(ms)” and a procedure would be translated into “private void functionName (int arg1, int arg2 ...) { // statements here }. Variables would be translated into class variables.

I would focus on the choice of technology (all Java and a lot of streaming) as the main problem in the project. It caused numerous problems that would vanish if I researched a better way of getting Logo commands onto the robot in one go.

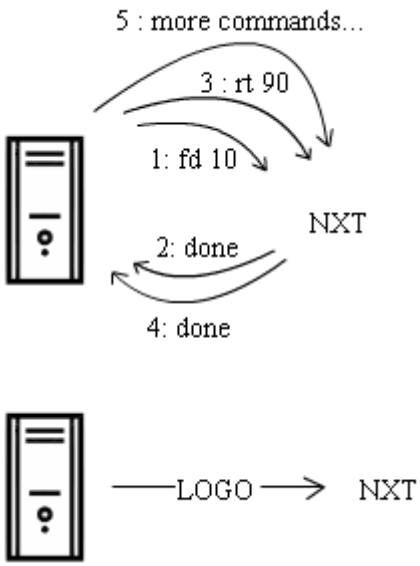


Fig 13.1 - streaming vs sending in one go. The latter would be more reliable.

I did *try* to get the entire parser onto the robot, but Lejos does not support many of the classes used by JavaCC (RunTimeExceptions need to be removed and Hashmaps and System.getProperty() cannot be used and must be replaced). I tried a tiny grammar, consisting of just one token [a-z]. I could get the code to compile but I could not get the code to run on the NXT - I kept getting confusing errors about Lexical errors , “” encountered after : “”. It could be something to do with the end of line characters but I tried \n, and \r\n and was unable to progress.

If I were to improve the project or do it again I would also look at Remote Procedure calls rather than Socketing. I didn't research these much because I had enough research to do (!) but it may have paid off in the long run. I am pretty sure that RPC are not easily configured on the machine that connects to the robot however; the Socket is certainly easy to start and stop.

13.5 Future work / suggested improvements

Apart from the above, other improvements I would have made to the code if time allowed are:

- The system should try to reconnect to the back end if it fails first time (perhaps an error message and a “trying to reconnect...” message.

- The application is currently quite strongly coupled to the JavaCC Parser. If another parser was

to be used (for example the alternative ANTLR I mentioned in Section 4.2) it would require a lot of rewriting.

- There are no real configuration files; it would be better to have the port (5000) loaded from an external file, as well as some way of configuring the robot (for example the speed will be different on different floor surfaces and might need changing). This could be done by Logo statements (perhaps 'setspeed') or by addng to the AdminApp.

- Regarding the JavaCC parser, I think it would be very straightforward to add extra Logo commands such as if/else and returning values from procedures. Boolean data variables would be stored as 0/1 for true/false and a BooleanExpressionNode would be a special Expression node whose numerical value is interpreted as a Boolean. If/else would be implemented as an IfElseNode, that would be visited in a similar way to all other nodes, would evaluate the BooleanExpressionNode in the if statement and visit the other nodes as appropriate.

- Adding pen up/pen down would also be straightforward - an extra physical motor is needed on the robot and LogoPenUpObject and LogoPenDownObject would need to extend ALogoObject in the same way as the LogoFdObject and LogoRtObject do at the moment. The TurtleCanvas would need to store a variable indicating whether the pen is up or down and adjust the x/y position of the turtle without drawing a line if the pen is up. Similarly, changing pen colours could be achieved quite easily (although not so easily on the robot).

- It may be better to allow procedures to be defined anywhere in the code - currently a procedure has to be defined before use. I think this would be easy to do, simply pre-process and move all procedures to the top before parsing.

- I have used the error messages that come for free with JavaCC (of the form "u*nexpected [on line 6, was expecting blah blah blah*..." but these are probably too vague and complicated for younger users. It would be better to try and provide more natural language errors and provide some "Did you mean..." hints.

- Background syntax checking is working nicely, but it would be better if it were not activated every 5 seconds, but rather 5 seconds after the last user interaction. I think this would be easy to do; the thread can contain a 5 second timer that is reset and started after any user interaction, and triggers one syntax-check. User interactions within the 5 seconds will reset the timer.

- The UndoManager is simplistic at the moment, especially as the history is lost when the text is changed during a sequence of undos. Some kind of undo tree that makes a new branch for such changes would be more powerful, but probably too complex for children. XXX

- The extensions listed in section 3.4.4 (chat room, a web cam connected to the robot, printing/exporting as image) would not be difficult to add. If the user owns a robot (and does not need to use a Socket connection at all) more work is needed - the producer/consumer pair would need to be abstracted out into a new class that could be used on the client side too. The current design does use event dispatching to listen for events from the producer/consumer, so this is loosely coupled and is ready for reuse.

13.5.1.1 Error reporting & error codes

I am happy with the reporting of errors to the user - the red highlighting (see Appendix A) and the exclamation marks make the application user friendly. I have also tried to make sure that errors such as "failed to find robot" are reported to the user as pop-ups, and for more serious errors (such as those in section 12.2.1.6) I have some added error codes to most errors to make them more easily understood. I think the way that the AdminApp writes to a log file is also useful.

13.6 Known bugs

The known bugs are:

- The "substance" look and feel library [31] throws Exceptions occasionally, about "changing the state of a non-idle timeline". These are caught by the substance library.
- The failed test cases in section 12.2.1

Bibliography

[1] 2SimpleSoftware (2010) *Making simple, powerful and creative software - 2XControlNXT* http://www.2simple.com/2controlNXT/ (Last accessed 11 Sept 2010)

[2] Aho A, Sethi R, Ullman J. (1986) *Compilers. Principles, techniques and tools*. Prentice Hall International.

[3] Anderrson L. (2009) *Compiler construction.* http://www.cs.lth.se/EDA180/2009/examples/CalcAnalysis/ (Last accessed 22 Sept 2010)

[4] antlr.org (2010) *ANTLR Parser Generator.* http://www.antlr.org/ (Last accessed 24 Sept 2010)

[5] Apache Software Foundation (2010) *ServletContextListener (Servlet API Documentation)* http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/ServletContextListener.html

[6] Barnes, D. (2002) Teaching introductory Java through LEGO MINDSTORMS models. Proceedings of the 33rd SIGCSE technical symposium on Computer science education. Feb 27 - Mar 03 2002.

[7] Department for Education and Employment (1999) *Unit 3.* http://nationalstrategies.standards.dcsf.gov.uk/node/18598 (Last accessed 11 Sept 2010)

[8] Durbin, P. *How do I have line numbers in my JTextArea?* http://www.esus.com/docs/GetQuestionPage.jsp?uid=1326&type=pf (Last accessed 11 Sept 2010)

[9] Freeman, E, Freeman E. (2004) *Head First Design Patterns*, O'Reilly

[10] Gamma E, Helm R, Johnson R, Vlissides J. (1985) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley

[11] Garret A, Thornton D. (2005) *A web based programming environment for Lego Mindstorms robots*. ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference

[12] Hall, C. (2008) *PureMVC - About*. http://puremvc.org/content/view/67/178/ (Last accessed 11 Sept 2010)

[13] Hall, C (2008) *PureMVC Implementation idioms and best practices.* http://puremvc.org/component/option,com_wrapper/Itemid,174/ (Last accessed 24 Sept 2010)

[14] Hamilton G. *Dr. Geoff Hamilton CA448 Compiler Construction 1*. http://www.computing.dcu.ie/~hamilton/teaching/CA448/notes.html (Last accessed Sept 11 2010)

[15] Java 2 Standard Edition. (2010) Class EventListenerList. *http://download.oracle.com/javase/1.5.0/docs/api/javax/swing/event/EventListenerList.html*

[16] Java.net (2010) *JavaCC:JJTree reference*. https://javacc.dev.java.net/doc/JJTree.html (Last accessed 22 Sept 2010)

[17] Kamvysselis M, Lueck J, Rohrs C. *RoboLogo:Teaching Children how to program Interactive Robots.* http://web.mit.edu/manoli/robologo/www/robologo.html

[18] Lawhead P, Duncan M, Bland C, Goldweber M, Schep M, Barnes D, Hollingsworth R. (2003) *A road map for teaching introductory programming using LEGO Mindstorms robots*. ACM SIGCSE Bulletin v.35 n2, June 2003

[19] Lea, D. (2008) *Concurrent Programming in Java (2nd Edition).* Addison Wesley

[20]lejos.sourceforge.net (2010) *Overview (Lejos NXJ API documentation)* http://lejos.sourceforge.net/nxt/nxj/api/index.html (Last accessed Sept 22 2010)

[21]lejos.sourceforge.net (2010) *Lejos, Java for Lego Mindstorms (Lejos NXJ API documentation)* http://lejos.sourceforge.net (Last accessed Sept 22 2010)

[22]lejos.sourceforge.net. (2010) *Communications*. http://lejos.sourceforge.net/nxt/nxj/tutorial/Communications/Communications.htm

[23]Logotron Educational Software. *Imagine Logo (Primary)*. http://www.r-e-m.co.uk/logo/?Titleno=25541 (Last accessed 22 Sept 2010)

[24]Martin, D. *Web Enabled Robot Control using Direct Input and Automated Control Patterns* http://code.google.com/p/webenabledrobotcontrol/wiki/Overview (Last accessed)

[25]Martin, D.*Web Enabled Robot Control using Direct Input and Automated Control Patterns* http://code.google.com/p/webenabledrobotcontrol/downloads/detail?name=bsc4_dmartin_project_interim_report_v1.pdf&can=2&q= (Last accessed 22 Sept 2010)

[26]Niemeyer P, Peck J. (2002) *Exploring Java (2nd Edition)* O'Reilly

[27]Norvell, T. Java CC Tutorial. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/ (Last accessed 11 Sept 2010)

[28]Oracle (2006) *Java TM Web start v1.5.0 Frequently asked questions* http://download.oracle.com/javase/1.5.0/docs/guide/javaws/developersguide/faq.html (Last accessed 11 Sept 2010)

[29]Oracle (2010) *Networking - Chat/IM problem*. http://forums.sun.com/thread.jspa?threadID=5290961 (Last accessed 22 Sept 2010)

[30]Oracle (2009) Java Thread Primitive Deprecation. http://download.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html (Last accessed 22 Sept 2010)

[31]Oracle (2010) *substance: Substance Java Look and Feel.* https://substance.dev.java.net/ (Last accessed Sept 24 2010)

[32]Oracle (2010) *UndoManager.* http://download.oracle.com/javase/1.4.2/docs/api/javax/swing/undo/UndoManager.html (Last accessed Sept 23 2010)

[33]Oracle (2010) Java Web Start 1.5.0 Developer Guide http://download.oracle.com/javase/1.5.0/docs/guide/javaws/developersguide/contents.html (Last accessed Sept 22 2010)

[34]Paine, J. How to Generate a Tree-Building Parser in Java using JJTree : http://www.j-paine.org/dobbs/jjtree.html

[35]Papert, S. (1980) *Mindstorms. Children, computers and powerful ideas.* Harvester Press.

[36]Papert, S. (1993) *The children's machine. Rethinking school in the age of the computer. Children, computers and powerful ideas.* Harvester Wheatsheef.

[37]RajaSekar A. *Writing an Interpreter using JavaCC*. http://www.anandsekar.com/2006/01/15/writing-a-interpretter/

[38]Reeni, S. (2010) *JavaCC Home.* https://javacc.dev.java.nat. (Last accessed 11 Sept 2010)

[39]Resnick, R. (1994) *Turtles, termites and traffic jams*. MIT Press.

[40]Reuben, H (2010) *1969 - The Logo Turtle - Seymour Papert, Marvin Minsky at al (American)*

http://cyberneticzoo.com/?p=1711 (Last accessed 11 Sept 2010)

[41] Star UML. (2010) *Star UML - the open source UML/MDA platform.* http://staruml.sourceforge.net/en/

[42] Terrapin Software (2010) *Terrapin Logo.* http://www.terrapinlogo.com/terrapin-logo.php (Last accessed 11 Sept 2010)

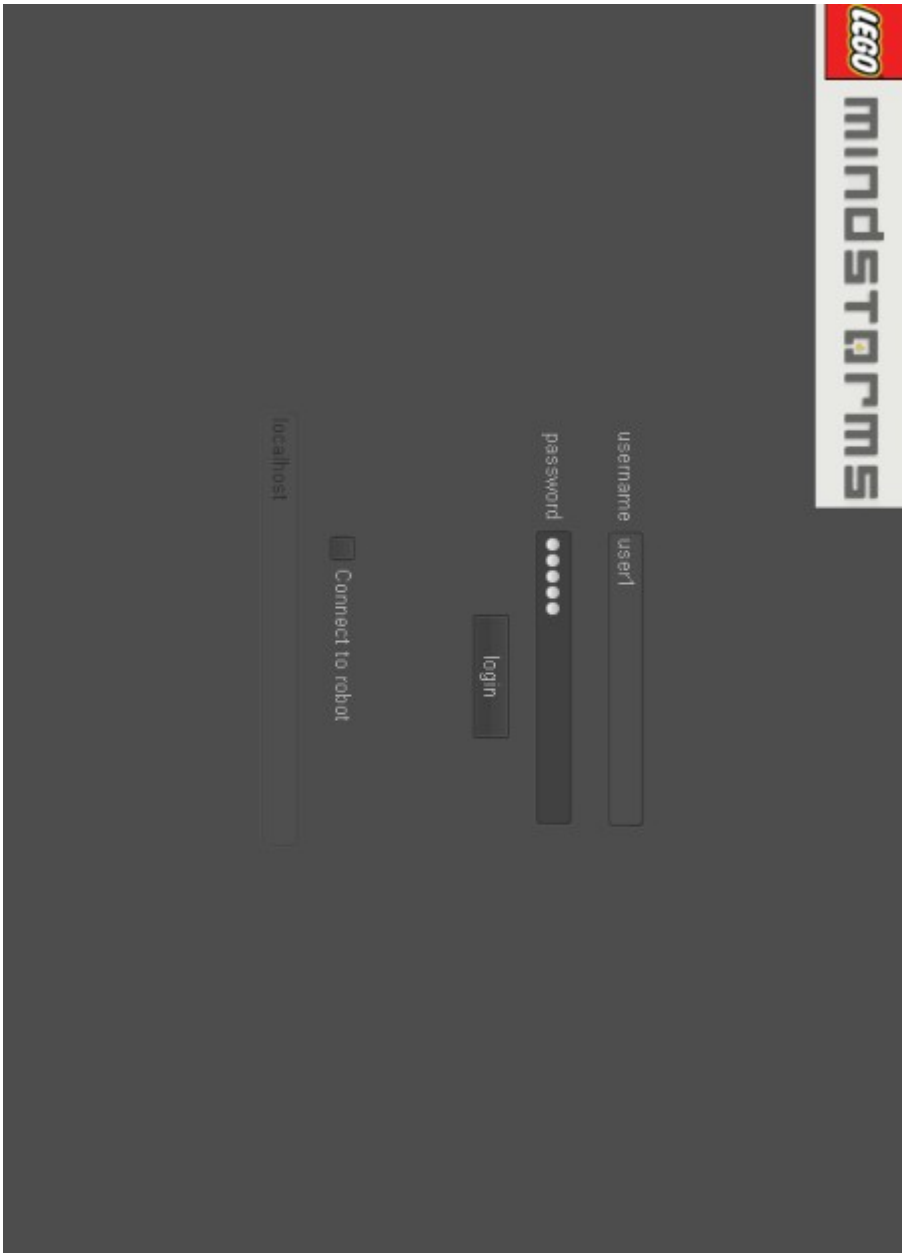
[43] Terrapin Software (2010). *Logo project ideas.* http://www.terrapinlogo.com/project-ideas.php (Last accessed 22 Sept 2010)

[44] Topologika (2010) *Screen Turtle 2 - Logo which is quick to learn. Turtle Graphics.* http://www.topologika.com/index.php?page=product.php&file=screen_turtle_2 (Last accessed 11 Sept 2010)

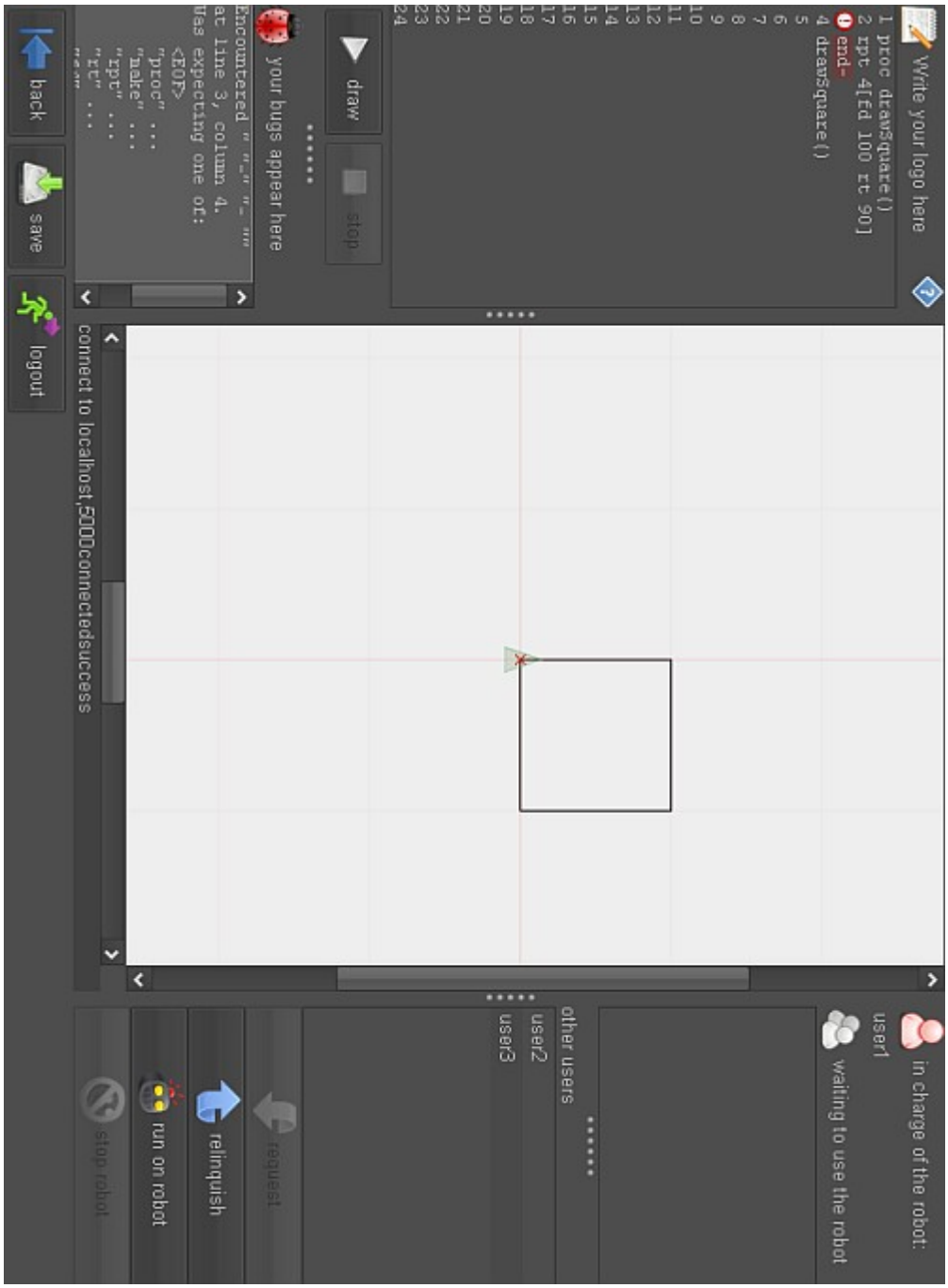
[45] Tortoise SVN (2010). *tortoisesvn.tigris.org*. http://tortoisesvn.tigris.org/ (Last accessed 22 Sept 2010)

Appendix A - Screenshots

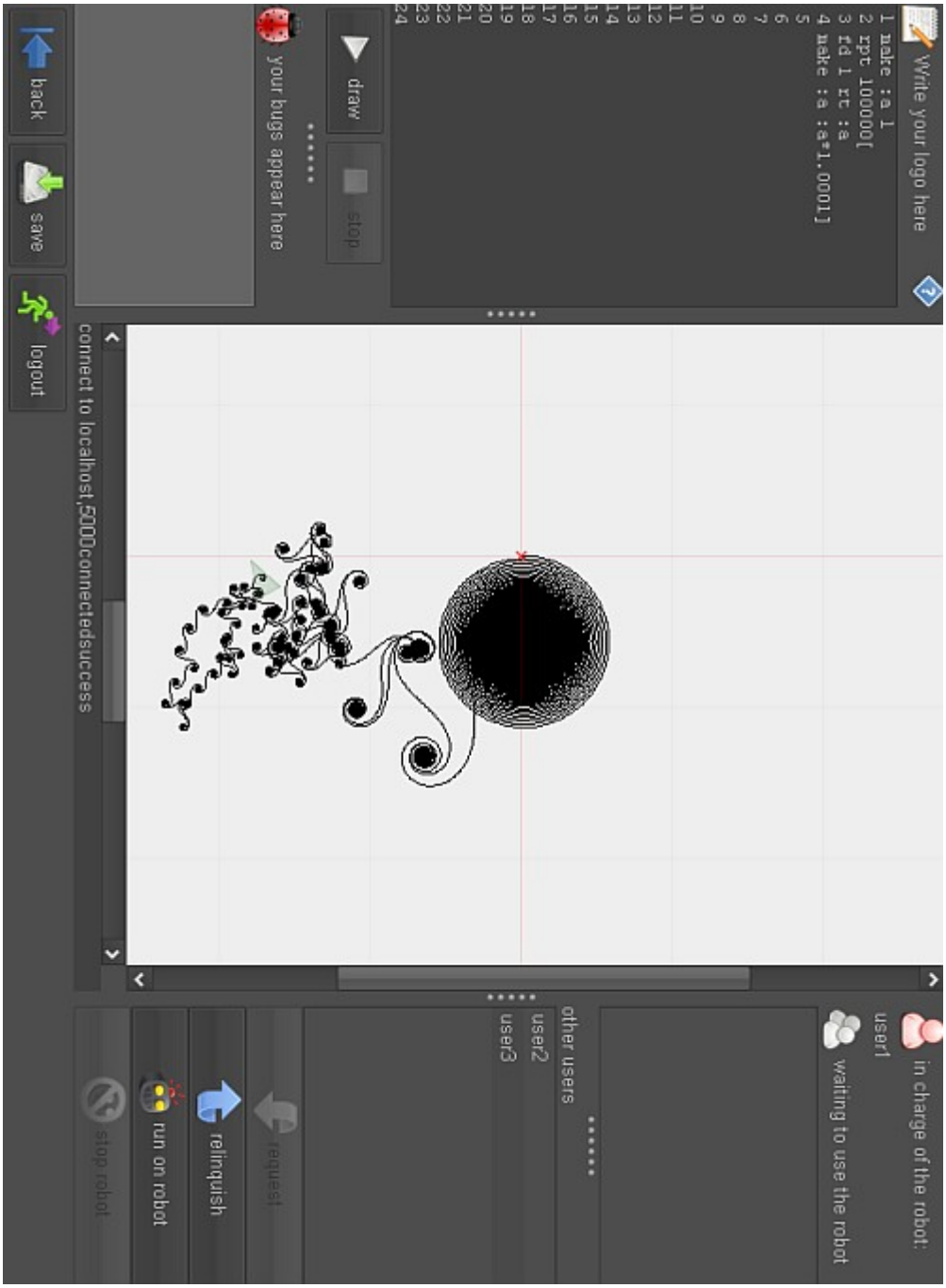
Screen shots of the final application follow. I do not have the right to use the Lego Mindstorms logo but the other icons are freeware. Many more screenshots and examples of Logo can be found on www.johngrindall.com/robologo.



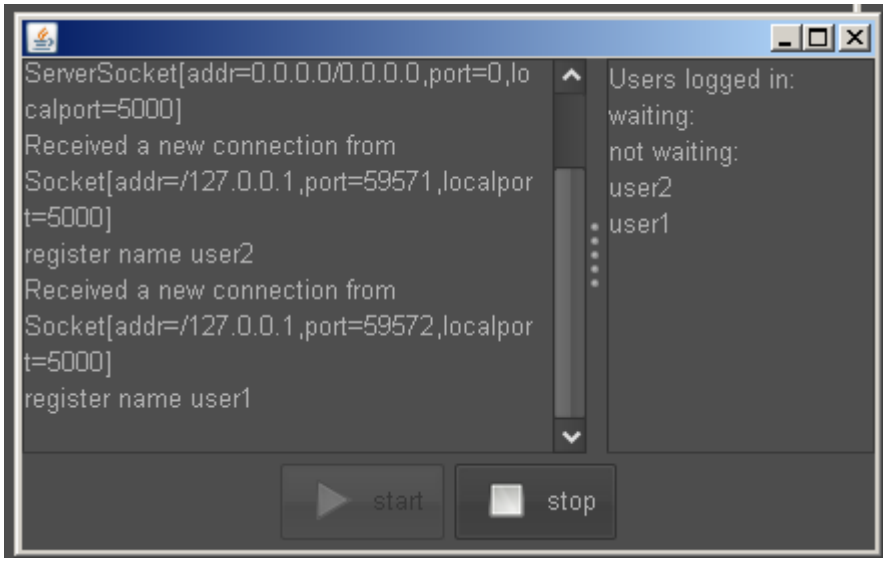
The log in page. The user chooses whether to connect to a robot or not



The create logo page. An error is displayed on line 3.
The user is in charge of the robot and users 2 and 3 are waiting.



Some interesting Logo.



The backend socket server. Run on the machine that connects to the NXT.

Appendix B - Use cases

User authentication package

Use case name	UC_logIn includes UC_updateUserList
Description	This use case begins with the user at the login page. The user is not logged on to the system. The user identifies himself/herself, by username and password, which are verified as authentic by the system and the user is forwarded to the next page (My Files)
Actors	User, Persistence, User Manager
Precondition	User is not logged in.
Postcondition	User is logged in and has been redirected to the 'my files' screen. Until the User logs out the User can see any security protected parts of the system and will see correct data relating to that User's work.
Primary path	1. User enters username and password 2. User clicks log in button 3. Persistence verifies and authenticates 4. User Manager updates the user list (include UC_updateUserList) for all users 5. User is forwarded to My Files page.
Alternate paths	
Exception paths	Identification incorrect - user receives error message and is not allowed to log in. User is already logged in - user is not allowed to log in twice - error message Database connection error - handled appropriately/fail gracefully

Use case name	UC_logOut includes UD_updateUserList
Description	This use case begins when the user is logged into the system, at some stage of working through the 'my files' and the 'create logo' screens.

Actors	User, User Manager
Precondition	User is logged in.
Postcondition	User is logged out and has been redirected to the my files/create logo screen. The user is logged out, cannot access any protected areas of the system and is removed from the system. If the user was in control of the robot, the control is relinquished. If the user is executing drawing, it is terminated. The user is removed from the 'users logged in' list that other users can see.
Primary path	1. User clicks logout 2. User Manager updates the user list (include UC_updateUserList) for all users. 3. The User is sent to the log in screen.
Alternate paths	If the User is executing Logo this is stopped. If the User is in charge of the robot this should be relinquished and the next user on the 'waiting list' should be granted control.
Exception paths	Database connection error - handled appropriately/fail gracefully
Notes	If the user closes the window without logging out the same should happen (robot relinquished)

File management package

Use case name	UC_saveFile
Description	The user is working on a file and clicks the save button. The contents of the file are saved and the old file is overwritten in the database.
Actors	User, Persistence
Precondition	User is logged in and working on a Logo file.
Postcondition	Logo file contents are updated in the database.
Primary path	1. User clicks save 2. Persistence updates file contents 3. Success message sent to User
Alternate paths	
Exception paths	Database connection error - handled appropriately/fail gracefully

Use case name	UC_openFile
Description	The user is looking at the 'my files' page and selects one of the files to open in the Logo editor.
Actors	User, Persistence
Precondition	User is logged in and located at the 'my files' page.
Postcondition	Logo file contents are loaded and the user can begin editing them.
Primary path	1. User selects file to re-open for editing 2. Persistence loads contents of file into System 3. User is directed to Create Logo screen, containing the loaded file.
Alternate paths	
Exception paths	Database connection error - handled appropriately/fail gracefully

Use case name	UC_deleteFile
Description	The user is looking at the 'my files' page and selects one of the files to delete. An 'are you sure' message is displayed and the user clicks yes.

Actors	User, Persistence
Precondition	User is logged in and located at the 'my files' page.
Postcondition	Entries relating to the particular Logo file are removed in the database. The my files page is refreshed.
Primary path	1. User selects file 2. User clicks delete 3. System displays “are you sure” message 4. User selects yes 5. Persistence removes/sets flag in database corresponding to file 6. “File deleted” message is displayed 7. My Files screen is refreshed, showing one fewer file.
Alternate paths	
Exception paths	Database/web error - handled appropriately

Use case name	UC_createNewFile
Description	The user is looking at the 'my files' page and chooses to create a new file. The user chooses the file-name (not the same as any already created). A new (empty) file is created.
Actors	User, Persistence
Precondition	User is logged in and located at the 'my files' page.
Postcondition	New entries in the database are created for the new file, containing empty Logo. The my files page is refreshed to show the new files.
Primary path	1. User clicks “create new file” 2. User enters information such as description/file name (meta data) 3. Persistence creates row in database for the new file 4. Use is forwarded to Create Logo screen.
Alternate paths	
Exception paths	Database/web error - handled appropriately

Use case name	UC_viewMyFiles
Description	The user located on the Create Logo screen, or the log in page and navigates to the My Files screen. The files belonging to the user are displayed. If the files have not been loaded yet, or if the user has deleted a file the page needs to be reloaded.
Actors	User, Persistence
Precondition	
Postcondition	None
Primary path	1. User 2.
Alternate paths	
Exception paths	Database/web error - handled appropriately

Logo Editor package

Use case name	UC_testLogoLocally includes UC_syntaxCheckLogo, UC_interpretLogo
Description	The user is located on the create logo screen. The user has written logo in the text box and clicks 'draw'. The on-screen turtle draws the Logo.

Actors	User, Interpreter, Syntax Checker
Precondition	
Postcondition	Drawing Logo commences. The stop button is enabled, the draw button is disabled.
Primary path	1. User clicks “draw” button. Draw button is disabled. Stop button is enabled. 2. Logo is syntax checked, include UC_syntaxCheckLogo 3. Logo execution begins, include UC_interpretLogo
Alternate paths	
Exception paths	The Logo has a syntax error - this is reported to the user and drawing Logo does not commence. The play button is enabled, and the stop button is disabled. The Logo has a run-time error - the Logo draws until the error is reached. It is reported to the user. The play button is enabled, and the stop button is disabled.
Notes	It is a requirement that the application is multithreaded - drawing Logo can continue while Logo is edited.

Use case name	UC_stopLogoLocally
Description	Logo is drawing on-screen. The stop button stops the interpreter walking the parse tree and drawing stops.
Actors	User
Precondition	Logo is drawing
Postcondition	The draw button is enabled, the stop button is disabled.
Primary path	1. User clicks stop button 2. Logo drawing stops. 3. Draw button is enabled, stop button is disabled.
Alternate paths	None
Exception paths	None

Use case name	UC_editLogo
Description	The user is located on the 'create' screen. The user changes the Logo in the text box, by typing or by performing a copy/paste. The system stores the edit.
Actors	User
Precondition	
Postcondition	The before and current state of the text is stored in the system so that it can be undone.
Primary path	1. User edits Logo, typing, deleting, copy/pasting. 2. The System stores the edit
Alternate paths	None
Exception paths	None

Use case name	UC_undoEdit
Description	The user is located on the 'create' screen and uses CTRL-Z to undo the previous edit.
Actors	User
Precondition	
Postcondition	The Logo text area is updated to show the previous state of the Logo.

Primary path	1. The User clicks CTRL-Z. The previous state of the Logo is retrieved and displayed to the user.
Alternate paths	If there are no previous states, no change occurs to the text field.
Exception paths	None

Use case name	UC_redoEdit
Description	The user is located on the 'create' screen and uses CTRL-Y to redo the previous undone edit.
Actors	User
Precondition	
Postcondition	The Logo text area is updated to show the 'undone' state of the Logo.
Primary path	1. The User clicks CTRL-Y. The previous undone edit is redone and displayed to the user.
Alternate paths	
Exception paths	
Notes	If there are no undone states, no change occurs to the text field.

Logo Interpreter package

Use case name	UC_interpretLogo includes UC_syntaxCheckLogo
Description	Logo is interpreted after being checked for correct syntax. The statements of the Logo are output one by one (forward 10, right 10, forward 10....)
Actors	Syntax Checker, Interpreter
Precondition	
Postcondition	
Primary path	1. Include UC_syntaxCheck 2. Interpret Logo - output messages 3. Repeat 2 until finished
Alternate paths	It is possible for the Logo to never terminate, in which case this use case must be terminated.
Exception paths	
Note	I have not included “UC_stopInterpretingLogo” - it would simply mean stopping this use case and is not something that the user will consider a 'goal'. It is a consequence of UC_stopLogoLocally and UC_stopExecutionOnRobot.

Use case name	UC_syntaxCheckLogo
Description	Syntax checking happens in different places of the application. It happens automatically when the user types into the Logo text area, it happens before Logo is drawn on screen and it happens before Logo is sent to the Robot.
Actors	Syntax Checker
Precondition	None
Postcondition	None
Primary path	1. A string is sent to the Syntax checker. 2. The syntax checker checks the syntax
Alternate paths	If the Logo is syntactically wrong this is reported to the part of the System that started the syntax checking.

Exception paths	

Robot functionality package

Use case name	UC_executeOnRobot includes UC_interpretLogo
Description	The user sends Logo to the robot to be drawn by the NXT.
Actors	User, Robot, Syntax Checker, Interpreter
Precondition	User is logged in, located at Logo editing page, in control of the robot and it is not executing
Postcondition	The robot begins executing the Logo. The 'execute on robot' button is disabled, the 'stop execution on robot' button is enabled.
Primary path	1. User clicks 'execute on robot' 2. Logo is syntax checked locally (include UC_syntaxChekcLogo) 3. Logo is sent to the backend 4. Include UC_interpretLogo 5. The Robot draws the Logo the user has sent 6. When it has finished the 'execute on robot' button is enabled, the 'stop execution on robot' button is disabled.
Alternate paths	It is possible for the robot to never finish - in which case UC_stopExecutionOnRobot is used.
Exception paths	Syntax checking fails - errors in Logo – handle and report to user. Enable 'send to robot' button, disable 'stop execution on robot' button. END. Error communicating to robot – to be handled gracefully (it could be switched off, out of range, not running the right listener program, other error or robot). Enable 'send to robot' button, disable 'stop execution on robot' button. END.

Use case name	UC_stopExecutionOnRobot
Description	The robot is drawing Logo. The user clicks 'stop'. The robot stops, returns back to the idle state (waiting for Logo) and the 'send to robot' is enabled.
Actors	User, Robot
Precondition	The robot is drawing Logo -
Postcondition	The 'execute on robot' button is enabled, the 'stop execution on robot' button is disabled.
Primary path	1. User clicks 'stop execution on robot' button 2. Robot stops executing robot. 3. Reset buttons
Alternate paths	
Exception paths	Error communicating to robot – to be handled gracefully (it could be switched off, out of range, not running the right listener program, other error or robot). Enable 'send to robot' button, disable 'stop execution on robot' button. END.

User management package

Use case name	UC_requestControl includes UC_updateUserList
Description	The user is logged in, working on a Logo file, and is not control of the robot.

	The user requests to be placed in the queue of people waiting to use the robot.
Actors	User, User Manager
Precondition	User is logged in and located at Logo editing page and not in control of the robot.
Postcondition	The User is moved to the 'waiting to use robot' list. Request button is disabled.
Primary path	1. User clicks on request button 2. User Manager moves user from the 'not waiting' queue to the 'waiting' list. 3. Include UC_updateUserList 4. The request button is disabled
Alternate paths	If the 'waiting for robot' queue was empty, the user is granted control of the robot. The request button is disabled and the 'execute on robot' and 'relinquish' buttons are enabled.
Exception paths	Communication error - handle gracefully and display error message

Use case name	UC_relinquishControl
Description	The user is logged in, working on a Logo file, and is in control of the robot. The User has finished working on a file and relinquishes control. The next person in the 'waiting' queue is given control of the robot.
Actors	User, User Manager
Precondition	User is logged in and located at Logo editing page and in control of the robot. The robot is not executing Logo.
Postcondition	The robot begins executing the Logo.
Primary path	1. User clicks on the 'relinquish' button 2. User manager moves user into the 'not waiting' queue. 3. The next person waiting is given control. 4. Include UC_updateUserList 5. The request button is enabled.
Alternate paths	
Exception paths	Communication error - handle gracefully and display error message

Use case name	UC_updateUserList
Description	The User Manager actor triggers this use case - for a variety of reasons (see above), the User Manager refreshes the display of all people logged in to the system.
Actors	User Manager
Precondition	None
Postcondition	All logged in users see the latest state of the user list.
Primary path	1. User Manager sends message to all logged in users containing information about the current user list.
Alternate paths	
Exception paths	Communication error - handle gracefully and display error message

Appendix C - .jjt grammar file

```
options {
   MULTI=true;
   VISITOR=true;
   NODE_DEFAULT_VOID=true;
   NODE_EXTENDS="BaseNode";
   NODE_PREFIX="";
   NODE_PACKAGE="com.jgrindall.logojavacc";
   OUTPUT_DIRECTORY = "C:/Users/John/Documents/NetBeansProjects/logojavacc/src/
                       com/jgrindall/logojavacc";
   STATIC=false;
   VISITOR_EXCEPTION="ParseException";
   TRACK_TOKENS=true;
}

PARSER_BEGIN(LogoParser)
package com.jgrindall.logojavacc;
public class LogoParser {

}
PARSER_END(LogoParser)

SKIP :       { " " }
SKIP :       { "\t" }
SKIP :       { "\n" | "\r" | "\r\n" }

TOKEN  :     { < PROC            :     "proc"                    > }
TOKEN  :     { < END             :     "end"                     > }
TOKEN  :     { < MAKE            :     "make"                    > }
TOKEN  :     { < VARNAME         :     <COLON><ID>               > }
TOKEN  :     { < RPT             :     "rpt"                     > }
TOKEN  :     { < RT              :     "rt"                      > }
TOKEN  :     { < FD              :     "fd"                      > }
TOKEN  :     { < ID              :     (<ALPHA>)+                > }

TOKEN   :       { < DECPOINT            :   "."                          > }
TOKEN   :       { < PLUS                :   "+"                          > }
TOKEN   :       { < MINUS               :   "-"                          > }
TOKEN   :       { < TIMES               :   "*"                          > }
TOKEN   :       { < DIVIDE              :   "/"                          > }
TOKEN   :       { < COLON               :   ":"                          > }
TOKEN   :       { < COMMA               :   ","                          > }

TOKEN   :       { < LBRACKET            :   "("                            > }
TOKEN   :       { < RBRACKET            :   ")"                            > }
TOKEN   :       { < LSQR         :   "["                                 > }
TOKEN   :       { < RSQR         :   "]"                                 > }

TOKEN   :       { < NUMBER       :
(<DECPOINT>(<DIGIT>)+)    |    ((<DIGIT>)+(<DECPOINT>(<DIGIT>)+)?)     > }

TOKEN :       { < DIGIT          :   "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"  > }
TOKEN :       { < ALPHA          :   (["a"-"z"] | ["A"-"Z"])  > }

ProgramNode start() #ProgramNode : {}
{
    statement_list()
    <EOF>
    {
        return jjtThis;
    }
}
void expression()#ExpressionNode : {}
{
    multiplicativeExpression() ( plusexpression()   |   minusexpression()  )*
}
void plusexpression()#PlusExpressionNode(2):{}
{
    <PLUS>multiplicativeExpression()
```

```
}
void minusexpression()#MinusExpressionNode(2):{}
{
    <MINUS>multiplicativeExpression()
}
void multiplicativeExpression(): {}
{
    unaryExpression() (    timesterm()    |    divterm()      )*
}
void timesterm()#TimesExpressionNode(2):{}
{
    (<TIMES> unaryExpression()  )
}
void divterm()#DivideExpressionNode(2):{}
{
    (<DIVIDE> unaryExpression() )
}
void unaryExpression(): {}
{
     <MINUS> numberExpression()#NegateExpressionNode
     |
     numberExpression()
}
void numberExpression():{}
{
    varname()
    |
    number()
    |
    <LBRACKET> expression() <RBRACKET>
}
void varname()#VariableValueNode : {Token t;}{
    t = <VARNAME>
    {
       jjtThis.addToData("name",t.image);
    }
}
void number()#NumberNode : {Token t; Double d;}
{
    t=<NUMBER>
    {
       d = Double.parseDouble(t.image);
       jjtThis.addToData("value",new Double(d));
    }
}
void fndefine_statement()#FnDefineNode : {}
{
    <PROC>
   (fn_name()   <LBRACKET> ( var_list()  )? <RBRACKET> )
   inside_fn_statement_list()
    <END>
}
void var_list()#VarListNode:{}
{
   varname() (<COMMA> varname() )*
}
void statement_list()#StatementListNode:{}
{
    ( statement() )*
}
void inside_fn_statement_list()#FnStatementListNode:{}
{
    (  inside_fn_statement()  )*
}
void inside_fn_statement():{}
{
   fncall_statement() | make_statement() | fd_statement()
   |  rt_statement() | rpt_statement()
}
void statement():{}
{
```

```
   inside_fn_statement() | fndefine_statement()
}
void rpt_statement()#RptStatementNode:{}
{
    <RPT> expression() <LSQR> inside_fn_statement_list() <RSQR>
}
void fncall_statement()#FnCallNode:{}
{
      fn_name()<LBRACKET> (  expression_list()   )?  <RBRACKET>
}
void expression_list()#ExpressionListNode:{}
{
    expression() (<COMMA>  expression()  )*
}
void fn_name()#FnNameNode:{Token t;}
{
    t=<ID>
    {jjtThis.addToData("name",t.image);}
}
void fd_statement()#FdStatementNode:{}
{
    <FD>expression()
}
void rt_statement()#RtStatementNode:{}
{
    <RT>expression()
}
void make_statement()#MakeStatementNode:{}
{
    <MAKE>
    vardefn()
    expression()
}
void vardefn()#VariableDefnNode : {Token t;}{
    t = <VARNAME>
    {
        jjtThis.addToData("name",t.image);
    }
}
```

Appendix D - testing plan for Logo parser

Appendix E - deploy.bat and the jnlp file

deploy.bat - Windows script used to build the final files for distribution.

```
SETLOCAL ENABLEDELAYEDEXPANSION

REM  take relevant jar files from projects
copy ..\logo\dist\logo.jar                          lib\
copy ..\logojavacc\dist\logojavacc.jar              lib\
copy ..\logoclasslib\dist\logoclasslib.jar  lib\
copy ..\logoserver\dist\logoserver.jar              lib\
copy ..\logocomm\dist\logocomm.jar                  lib\

REM copy the file running on the robot
copy ..\logorobot\build\*.nxj                       dist\

REM sign them all (with the same certificate)
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\logo.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\logocomm.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\logojavacc.jar
      jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\logoclasslib.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\logoserver.jar
      jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\substance.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\trident.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\bluecove.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\bluecove-gpl.jar
            jgrindall
jarsigner -storepass ***pwd*** -keystore logoKeystore lib\pccomm.jar
            jgrindall

REM copy them
copy lib\logo.jar              dist\appserver\
copy lib\substance.jar         dist\appserver\lib\
copy lib\trident.jar           dist\appserver\lib\
copy lib\logojavacc.jar        dist\appserver\lib\
copy lib\logocomm.jar          dist\appserver\lib\
copy lib\logoclasslib.jar      dist\appserver\lib\

copy lib\logoserver.jar        dist\robotserver
copy lib\bluecove.jar          dist\robotserver\lib\
copy lib\bluecove-gpl.jar      dist\robotserver\lib\
copy lib\pccomm.jar            dist\robotserver\lib\
copy lib\logocomm.jar          dist\robotserver\lib\
copy lib\logoclasslib.jar      dist\robotserver\lib\
copy lib\logojavacc.jar        dist\robotserver\lib\
copy lib\pccomm.jar            dist\robotserver\lib\
copy lib\substance.jar         dist\robotserver\lib\
copy lib\trident.jar           dist\robotserver\lib\

REM timestamp for my use
echo created %date% %time% > dist\robotserver\version.txt
echo created %date% %time% > dist\appserver\version.txt

@echo off
echo RoboLogo log file > dist/robotserver/robologo.log

copy "C:/Users/John/Documents/univ_westminster/project/WORK/userManual.odt"
```

```
dist\userManual.odt
copy "C:/Users/John/Documents/univ_westminster/project/WORK/userManual.pdf"
dist\userManual.pdf

set path="C:\Program Files\WinRAR\";%path%

cd dist

rar a robologo.zip -r *.*
```

robologo.jnlp (Java web start file)

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://www.johngrindall.com/robologo/"
href="logo.jnlp">
  <information>
    <title>RoboLogo</title>
      <icon href="robot.jpg"/>
      <icon href="legoMindstorms.png" kind="splash"/>
    <vendor>jgrindall</vendor>
    <homepage href="http://www.johngrindall.com" />
    <description>RoboLogo</description>
    <offline-allowed />
  </information>
  <resources>
    <j2se version="1.4+" />
    <jar href="logo.jar" />
      <jar href="lib/logoclasslib.jar" />
      <jar href="lib/logojavacc.jar" />
      <jar href="lib/substance.jar" />
      <jar href="lib/logocomm.jar" />
      <jar href="lib/trident.jar" />
  </resources>
  <security>
    <all-permissions />
  </security>
  <application-desc main-class="com.jgrindall.logo.Main" />
</jnlp>
```

Appendix G - Source code

The source code is divided into projects and packages, as described in section 11.1

The PureMVC library is not included, but it is included on the CD which accompanies this project

JavaCC generates one Node file for each kind of Node. I have included the file for DivideExpressionNode, the rest are almost identical (two one-line constructors that just call the superclass, and one jjtAccept method that is also a one-liner, “return visitor.visit(this, data);” All the nodes are found on the CD.