INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available.

UMI

An Integrated Development Environment for Moon Processor Simulator

Andrei Elson

A Major Report

In

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2002

© Andrei Elson, 2002

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68463-6

Canada

ABSTRACT

An Integrated Development Environment for
Moon Processor Simulator

Andrei Elson

The purpose of this Master's Report is to create a software application called an Integrated Development Environment (IDE) for Moon Processor Simulator. The product is supposed to help students taking the Compiler Design course at Concordia University to debug and test assembly code that was handwritten or generated by a compiler generating moon code.

This document starts with describing the reasons and advantages of having a processor emulator to test any assembly language. It then lists specifications and requirements of the project and reasons for choosing the Java programming language for the implementation.

The paper then lists Object Oriented principles and demonstrates how they are applied during program design. The main goal of the project is to create a program that would be useful, easy to use and fast to learn. The paper describes how these objectives are achieved by following usability rules.

TABLE OF CONTENTS

TABLE OF FIGURES .... **VI**

1.0 INTRODUCTION **1**

1.1 PROJECT PURPOSE 1
1.2 MOON IDE DESCRIPTION 1

2.0 BACKGROUND **3**

2.1 THE NEED FOR AN EMULATOR 3
2.2 MOON REFERENCE 3
2.2.1 Processor *4*
2.2.2 Instructions *4*
2.2.2 Registers *6*
2.2.3 Memory *6*
2.2.4 Language Grammar *7*
2.3 PROJECT SPECIFICATION 7
2.4 WHY JAVA 8
2.5 WHY SWING 8
2.5.1 AWT Features *8*
2.5.2 Swing Features *9*
2.5.3 Comparison between AWT and Swing *9*

3.0 IDE DESIGN **10**

3.1 OBJECT ORIENTED DESIGN 10
3.1.1 Encapsulation *10*
3.1.2 Inheritance *10*
3.1.3 Polymorphism *12*
3.2 USER INTERFACE DESIGN CHALLENGES 12
3.3 USER INTERFACE DESIGN AND USABILITY 12
3.3.1 Program Installation *13*
3.3.2 Program Correctness *13*
3.3.3 Accessible and Well Written Help *13*
3.3.4 Control of the System *14*
3.3.5 System Feedback *14*
3.3.6 System Requirements *15*
3.3.7 Error Prevention *15*
3.3.8 Meaningful Error Messages *15*
3.3.9 Minimization of User Memory Load *15*
3.3.10 Clearly Marked Exits *16*
3.3.11 Shortcuts for Expert Users *16*
3.3.12 Portability *17*
3.4 MODEL-VIEW-CONTROLLER ARCHITECTURE 17

4.0 IMPLEMENTATION **18**

4.1 PROBLEMS/SOLUTIONS 18
4.1.1 Debugger Area *18*
4.1.2 Scrolling *18*
4.2 MODULES DESCRIPTION 19
4.2.1 Editor *19*
4.2.2 Help *20*
4.2.3 Instruction *21*
4.2.4 Loader *21*
4.2.5 Memory *21*
4.2.6 Parser *21*
4.2.7 Registers *22*
4.2.8 Runtime *22*
4.2.9 Symbols *22*
4.2.10 User Interface *23*
4.2.11 Utils *28*
4.3 UNIQUE FEATURES 28
4.3.1 Reflection *28*
4.3.2 Text Buffer Modification *29*
4.3.3 Method Testing *30*

5.0 SOFTWARE QUALITY **31**

5.1 CORRECTNESS 31
5.2 ROBUSTNESS 31
5.3 MAINTAINABILITY 31
5.4 REUSABILITY 32

6.0 FUTURE ENHANCEMENTS **33**

6.1 MODIFYING MEMORY CONTENT 33
6.2 BETTER SIMULATION REPORTING 33
6.3 PLUGGABLE ASSEMBLER ENGINE 33

7.0 CONCLUSION **34**

REFERENCES **35**

APPENDIX A – TOP LEVEL CLASS DIAGRAM **36**

APPENDIX B - FULL MOON IDE ON LINUX **37**

APPENDIX C – MOON IDE API **38**

TABLE OF FIGURES

FIGURE 1: DATA ACCESS INSTRUCTIONS 4
FIGURE 2: ARITHMETIC AND LOGICAL INSTRUCTIONS OF TYPE A 5
FIGURE 3: ARITHMETIC AND LOGICAL INSTRUCTIONS OF TYPE B 5
FIGURE 4: INPUT AND OUTPUT INSTRUCTIONS 6
FIGURE 5: CONTROL INSTRUCTIONS 6
FIGURE 6: LANGUAGE GRAMMAR 7
FIGURE 7: DIRECTIVES 7
FIGURE 8: INHERITANCE EXAMPLE 11
FIGURE 9: HELP DIALOG 14
FIGURE 10: ERROR MESSAGE 15
FIGURE 11: SHORTCUTS 16
FIGURE 12: MEMORY VIEWER 19
FIGURE 13: EDITORBEAN 20
FIGURE 14: ABOUTDIALOG 23
FIGURE 15: INPUT OUTPUT VIEWER 23
FIGURE 16: MEMORY VIEWER (NON-ZERO MEMORY IS DISPLAYED) 24
FIGURE 17: MOONDEBUGGER 25
FIGURE 18: REGISTERSVIEWER 26
FIGURE 19: MULTIPLEFILESSELECT 26
FIGURE 20: CHARACTER INPUT DIALOG 27
FIGURE 21: ASCII INPUT DIALOG 27
FIGURE 22: SYMBOLS TABLE 28
FIGURE 23: EDITOR BEAN API 32
FIGURE 24: TOP LEVEL CLASS DIAGRAM 36
FIGURE 25: MOON IDE ON LINUX 37
FIGURE 26: EDITOR BEAN API 38
FIGURE 27: HELPVIEWER API 38
FIGURE 28: DATAWORD-INSTRUCTION API 39
FIGURE 29: MOONLOADER, EXCEPTION API 39
FIGURE 30: MEMORY CLASSEES API 40
FIGURE 31: MOONPARSER, PARSEEXCEPTION API 41
FIGURE 32: REGISTERS API 41
FIGURE 33: RUNTIMEHANDLER, MOONRUNTIMEEXCEPTION API 42
FIGURE 34: MOONSYMBOLS API 42
FIGURE 35: MOONDEBUGGER API 43

1.0 INTRODUCTION

This section describes the project purpose and description of the implemented program.

1.1 Project Purpose

The purpose of this master's report is to develop a visual tool called "An Integrated Development Environment for Moon Processor Simulator". An Integrated Development Environment (IDE) is a software application used by developers to write, run, and debug programs. Moon IDE software consists of a code editor, compiler, loader, interpreter, debugger and additional windows to show simulation context. The tool will be used to test and debug Moon language scripts.

Dr. Peter Grogono [1] developed the Moon language at Concordia University, Canada, to help the students taking the Compiler Design course to test their compilers. Dr. Grogono also implemented an inline interpreter to test the scripts using the C programming language. The interpreter serves its purpose but lacks a visual interface. The development of a visual Moon processor simulator is the next logical step, since it would allow students to edit scripts within the IDE, test scripts for syntax errors, and load scripts. Users would be able to interact with the debugger, visually see simulated memory and registers content, and so on.

1.2 Moon IDE Description

Moon IDE is build using the Java programming language. The tool has a Swing user interface, and it supports the following operations:

- editing of Moon scripts

- detecting syntax errors
- detecting loader errors
- loading external files in a specific order
- setting and removing breakpoints inside the simulated memory
- stepping through instructions or running the instructions until the next breakpoint or the end of the a script is reached

The development of the tool is strongly influenced by usability concerns. As a result it should be easy to learn and use.

2.0 BACKGROUND

This section lists Moon processor specifications, project requirements and other relevant information about the project.

2.1 The Need for an Emulator

The purpose of any compiler design course at the university level is to develop a working compiler that would transform a high-level language into machine or assembly code. Testing such machine code can be very problematic and error prone. For example, the program might have an infinite loop; it can overwrite its own instructions, access protected memory or do some other illegal operation. Testing such an erroneous program on a regular CPU can lead to serious system crashes. Even if assembler debugger were used, it would take too much time to fix compiler errors that produced such erroneous machine code. The solution to this problem is creating an emulator of a CPU, which would execute compiler-produced code as a regular processor. There are many advantages to this approach: such a simulator can run on any CPU type and it can crash or hang up without causing the whole system to fail. An emulator will generally execute machine code slower that a regular CPU, but for educational purposes it is an ideal solution. The Moon processor and its instruction set was developed especially for those reasons.

2.2 Moon Reference

This section describes the most important details about the Moon Processor. The complete specifications for the Moon processor can be found in [12].

2.2.1 Processor

The Moon is an imaginary processor based on simplified DLX architecture [2]. The processor has four instructions for data access, twenty-nine arithmetic and logic instructions, two instructions for input and output (IO), and eight control instructions to branch and call subroutines.

2.2.2 Instructions

There are two instruction formats: Instruction A and Instruction B. Instruction A has an opcode and 3 registers, while Instruction B has an opcode, two registers and a K operand, which is a constant or a label. Before each instruction is executed, it must be loaded from the memory. This action requires 10 clock cycles. Data read or write generally also requires 10 clock cycles, unless data requested is already in Memory Buffer Register (MBR). Each instruction occupies 32 bits of memory and non data access instructions take 1 cycle to execute. Data access instructions require more cycles since memory access is a more expensive operation. Figures 1 through 7 were taken from the original Moon reference document [12]. Figure 1 shows data access instructions.

Function	**Operation**	**Effect**
Load word	lw *Ri, K(Rj)*	$R(i) \xleftarrow{32} M_{32}[R(j)+K]$
Load byte	lb *Ri, K(Rj)*	$R_{24..31}(i) \xleftarrow{8} M_{8}[R(j)+K]$
Store word	sw *K(Rj), Ri*	$M_{32}[R(j)+K] \xleftarrow{32} R(i)$
Store byte	sb *K(Rj), Ri*	$M_{8}[R(j)+K] \xleftarrow{8} R_{24..31}(i)$

Figure 1: Data Access Instructions

Figure 2 shows arithmetic and logical instructions with register operand. Figure 3 shows arithmetic and logical instructions with immediate operand.

Function	Operation	Effect
Add	add *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) + R(k)$
Subtract	sub *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) - R(k)$
Multiply	mul *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \times R(k)$
Divide	div *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \div R(k)$
Modulus	mod *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \bmod R(k)$
And	and *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \wedge R(k)$
Or	or *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \vee R(k)$
Not	not *Ri, Rj*	$R(i) \xleftarrow{32} \neg R(j)$
Equal	ceq *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) = R(k)$
Not equal	cne *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \neq R(k)$
Less	clt *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) < R(k)$
Less or equal	cle *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \leq R(k)$
Greater	cgt *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) > R(k)$
Greater or equal	cge *Ri, Rj, Rk*	$R(i) \xleftarrow{32} R(j) \geq R(k)$

Figure 2: Arithmetic and Logical Instructions of type A

Function	Operation	Effect
Add immediate	addi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) + K$
Subtract immediate	subi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) - K$
Multiply immediate	muli *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \times K$
Divide immediate	divi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \div K$
Modulus immediate	modi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \bmod K$
And immediate	andi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \wedge K$
Or immediate	ori *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \vee K$
Equal immediate	ceqi *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) = K$
Not equal immediate	cnei *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \neq K$
Less immediate	clti *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) < K$
Less or equal immediate	clei *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \leq K$
Greater immediate	cgti *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) > K$
Greater or equal immediate	cgei *Ri, Rj, K*	$R(i) \xleftarrow{32} R(j) \geq K$
Shift left	sl *Ri, K*	$R(i) \xleftarrow{32} R(j) << K$
Shift right	sr *Ri, K*	$R(i) \xleftarrow{32} R(j) >> K$

Figure 3: Arithmetic and Logical Instructions of type B

Figure 4 shows input and output instructions.

Function	Operation	Effect
Get character	getc *Ri*	$R_{24..31}(i) \xleftarrow{8} Stdin$
Put character	putc *Ri*	$Stdout \xleftarrow{8} R_{24..31}(i)$

Figure 4: Input and Output Instructions

Figure 5 shows control instructions.

Function	Operation	Effect
Branch if zero	bz *Ri, K*	$if\ R(i) = 0\ then\ PC \xleftarrow{16} PC + K$
Branch if non-zero	bnz *Ri, K*	$if\ R(i) \neq 0\ then\ PC \xleftarrow{16} PC + K$
Jump	j *K*	$PC \xleftarrow{16} PC + K$
Jump (register)	jr *Ri*	$PC \xleftarrow{32} R(i)$
Jump and link	jl *Ri, K*	$R(i) \xleftarrow{32} PC + 4; \mathrm{PC} \xleftarrow{16} PC + K$
Jump and link (register)	jlr *Ri, Rj*	$R(i) \xleftarrow{32} PC + 4; \mathrm{PC} \xleftarrow{16} R(j)$
No-op	nop	Do nothing
Halt	hlt	Halt the processor

Figure 5: Control Instructions

2.2.2 Registers

There are sixteen general-purpose registers and two system registers: Memory Buffer Register (MBR) and Memory Address Register (MAR). Register R0 always contains 0 and cannot be modified.

2.2.3 Memory

Simulated machine memory contains 5000 bytes. The memory can contain 1250 instructions or data words. The memory size can be increased, but the memory module has to be recompiled. Larger memory size results in decreased responsiveness of MemoryViewer. This problem is explained in great detail in section 4.1.2. Current memory size should be enough for any Moon program that was generated by a compiler in question.

2.2.4 Language Grammar

Figure 6 shows language grammar and Figure 7 shows loader directives.

Program	→	{Line} eof
Line	→	[Symbol][Instr | Directive] [Comment] eol
Directive	→	DirCode [Operand {"," Operand}]
Instr	→	Opcode [Operand {"," Operand}]
Operand	→	Register | Constant ["(" Register ")"] | String
Register	→	("r" | "R") Digit [Digit]
Constant	→	Number | Symbol
Number	→	["+" | "-"] Digit {Digit}
String	→	""" {Char} """
Symbol	→	Letter {Letter | Digit}
Comment	→	"%" {Char}

Figure 6: Language Grammar

Directive	Effect
entry	The following instruction will be the first to execute
align	The next address will be aligned
org K	The next address will be K
dw $K_1, K_2, ...$	Store words in memory
db $K_1, K_2, ...$	Store bytes in memory
res K	Reserve K bytes of memory

Figure 7: Directives

2.3 Project Specification

Project specifications states that the Moon IDE has to be designed using modern object oriented language such as Java or Visual C++. The IDE has to simulate the execution of a machine code by creating virtual memory, registers and symbols table. The program is also supposed to provide visual clues about the state of the simulated machine, show the content of simulated memory, registers and symbol table in a visually pleasing way. Furthermore, the program has to allow changing memory and registers content on the fly, by users typing in new values.

2.4 Why Java

The specification of the project stipulates that the program has to be build using Java or C++ programming language. The Java programming language is used for the implementation of the project for a number of reasons. Some of the motives are as follows: cross platform portability and overall simplicity of Java compared to C++, automatic garbage collection, rich swing widget set and easy to implement threads. Java also has no pointers and therefore is considered much safer than C++. Although Java programs run generally slower even with Just-In-Time compiling than native C++ programs [3], the project did not really require fast execution time. The main purpose of the program is to display the content of the simulated memory and registers.

2.5 Why Swing

There are two basic sets of components that are available for GUI designers that program in Java: Abstract Window Toolkit (AWT) and Swing. This section describes the differences between the libraries and lists reasons for choosing Swing over AWT.

2.5.1 AWT Features

AWT components depend on native operating system visual components called peers. This makes AWT widgets to have platform limitations [4]. In addition to that, AWT libraries have a limited amount of available components. AWT do not contain components such as table, tree, tabbed pane, split pane and many other useful widgets. AWT components also do not support features like icons and tool-tips.

2.5.2 Swing Features

Swing has a rich set of higher-level components such as tree, table, tabbed pane, image buttons, etc. It also includes pluggable look and feel, which allows Java programs running on one Operating System (OS) to visually look like they were running on some other OS. Swing components do not depend on peers to handle their functionality. Thus, these components are called "lightweight". Swing has a pure Java design which results in fewer platform specific limitations. Pure Java design allows for a greater range of behavior for Swing components since they are not limited by the native peers [4].

2.5.3 Comparison between AWT and Swing

Swing widget set is much richer in configuration and better looking than AWT. The AWT is also missing useful UI widgets. For example, the IDE uses the JTable component to display memory and registers content. There would be no such component available if AWT were chosen for the implementation instead of Swing. Although Swing components are slower than their AWT counterparts [4], speed is not an issue for this project. The look of the program User Interface (UI) is much more important, so Swing is chosen over AWT.

3.0 IDE DESIGN

This section describes the main features of the program design. It also covers user interface design and program usability issues.

3.1 Object Oriented Design

The architecture of the system uses object-oriented design (OOD). There are many benefits of OOD. OOD techniques enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits attributes and behaviour from existing objects. This makes object-oriented programs easier to modify. The most important principles of OOD are encapsulation, inheritance and polymorphism, whish allow for better code maintainability, improved code quality and productivity, system flexibility, scalability and information hiding [5].

3.1.1 Encapsulation

Each class in the software has a special Application Programming Interface (API), a collection of public methods that are used for object access and modification. This allows hiding of the actual implementation to the outside world and improves security of the system. The complete API for all of the classes can be found in Appendix C.

3.1.2 Inheritance

Inheritance is a programming instrument by which more-specific elements incorporate the structure and behaviour of more general elements. This allows inheriting classes to reuse operations of the parent classes. Figure 8 shows how inheritance is used in the Moon IDE. The notation used is Unified Modeling Language [6]. The diagram shows

InstructionA, InstructionB and Directive classes, which inherit operations from Instruction class. The Instruction in turn inherits from the DataWord, because DataWord objects get stored in the simulated memory.

Figure 8: Inheritance Example

InstructionA, InstructionB and Directive inherit attributes Ri, Rj, lineNumber and all operations from Instruction class. This makes inherited classes much smaller because most of the required operations are already implemented in the parent class.

3.1.3 Polymorphism

Polymorphism is a characteristic of objects that enables run-time type binding. This feature allows programmers to avoid writing explicit conditions such as if-else or switch when condition depends on object type. Polymorphism can only work with inheritance, because it is possible to overwrite and redefine methods in derived classes only. Figure 8 shows an example of polymorphism, where the `toString()` method in each class will be called according to the target object type.

3.2 User Interface Design Challenges

User interface design brings challenges to the system design. According to [7] there are the following inherent problems with any program what uses user interface: designers have difficulty learning potential user's tasks, good user interface design requires iterative approach, program that uses UI requires multiprocessing and real-time response on input events. The software must also be robust and support aborting actions. Next section explains how these challenges are solved.

3.3 User Interface Design and Usability

The Moon Debugger IDE design is strongly influenced by usability concerns. The usability is an element of software quality, and it is defined in ISO-9126 standard. This International Organization for Standardization standard is concerned with the explanation of quality characteristics to be used in the evaluation of software products [5]. Among the factors of quality user interface discussed in the standard are the following characteristics: efficiency, learnability, intuitiveness, satisfaction, effectiveness, precision and productivity. Other researchers [8, 9] defined some additional features that are important to make the user satisfied with the program. Sections 3.3.1 through 3.3.12

discuss all of the usability factors and the ways they are addressed in system and user interface design.

3.3.1 Program Installation

For the user to be satisfied with the program he must be able to easily install and uninstall the program [9]. To solve this problem, the program uses Nullsoft's product called NSIS [14]. It is freely available software that allows creating install programs. After the IDE gets installed on the target system, the user can also easily perform complete uninstall of the product.

3.3.2 Program Correctness

Program is assumed to be correct when it does the task specified in the software specifications document and gracefully handles inputs outside its domain. The program is indeed meets specifications and works in an expected manner. The program does not crash even when non-standard input is entered. One specific case of this behavior is described in section 3.3.8

3.3.3 Accessible and Well Written Help

Help is accessible from the main menu window and it is designed to look similar to the standard Windows help dialog (Figure 9). The dialog has 3 parts: content viewer, index viewer and article viewer. The content viewer allows the user to see available help titles and select and interesting item for reading it in the article area. The index viewer allows the user to type in any keyword, perform a search, and select any found help title.

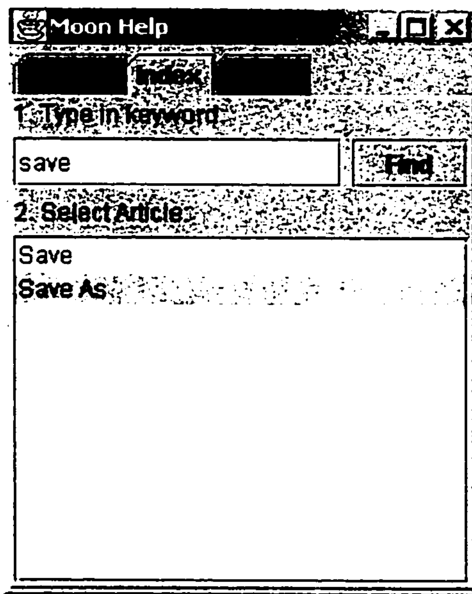

Figure 9: Help Dialog

3.3.4 Control of the System

The user must always have total control of the system [9]. The program is using multithreading to give user total charge of the system. If the Moon processor goes into an infinite loop because of the errors in the code, the user interface will still respond to the external inputs, and the user could stop the execution of the Moon Processor.

3.3.5 System Feedback

The program is using colors to provide constant feedback to the user. For instance, the location of the Program Counter is always highlighted in blue. Memory regions that contain instructions are also highlighted differently than regions that contain data. Program also highlights the register that was accessed last. These features should help users to immediately see what data values were changed and facilitate program usage. It should also make users spend less time figuring out what just happened.

3.3.6 System Requirements

During installation the user must be alerted about software and hardware requirements of the program [9]. The installation `readme.txt` file clearly states minimum software and hardware requirements.

3.3.7 Error Prevention

The simulator disallows to read or write data from memory area, which is occupied by the instruction. Although the "real" program can potentially overwrite its instructions, the Moon Debugger is educational software that is supposed to prevent this from happening. That is why it notifies the user about illegal memory access.

3.3.8 Meaningful Error Messages

The error messages in the program not only notify user of abnormal conditions but also give possible solutions to the problem. Figure 10 shows error messages generated when user inputs invalid ASCII code.

Figure 10: Error Message

3.3.9 Minimization of User Memory Load

The program hides options that are rarely used into menus, and shows options that are often used as image buttons. This technique should minimize user memory load.

The memory viewer has a mode where only non-zero (non-empty) memory is displayed. This should prevent user from extra scrolling through memory, and again decrease memory load and unnecessary hand movement because the user will not have to remember address range of previously visible memory cells.

3.3.10 Clearly Marked Exits

The program can be stopped by closing a main window or selecting the Exit submenu from the File menu (Figure 11).

3.3.11 Shortcuts for Expert Users

Figure 11 shows File menu, which has shortcuts for advanced users. Every menu/submenu in the IDE has a shortcut where an action key always corresponds to the action name. For example, shortcut for "New" command is CTRL-N, 'N' being the first letter of action command.

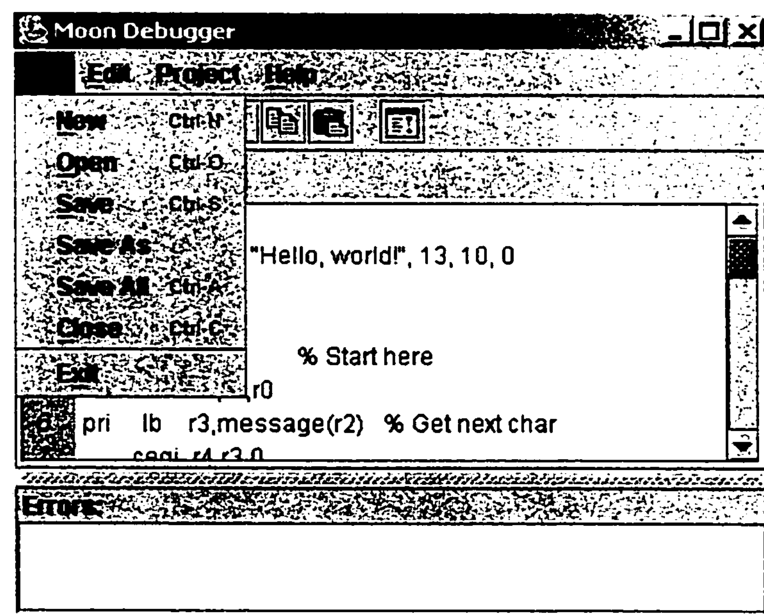

Figure 11: Shortcuts

3.3.12 Portability

The software is indeed portable since the Java programming language is used for the implementation. It makes software to be easily run on any platform that has a Java Virtual Machine (JVM) installed. JVM comes with Java Development Kit (JDK), which is freely available for download. The software was tested on Windows 98 (JDK 1.3.1_02), Windows 2002 (JDK 1.3.1_02) and Red-Hat Linux (JDK 1.3.1_01). On all tested platforms the program worked correctly. Appendix B shows a screen capture of how Moon IDE looks when running on Red Hat Linux.

3.4 Model-View-Controller Architecture

The IDE is using a Model-View-Controller (MVC) architecture [11]. MVC architecture separates visual representation (View) from data (Model). This schema allows greater decoupling between data and view. The architecture can also be used for differently displaying the same data. In the program the data models are MoonRegisters, MoonMemory and MoonSymbols classes. There are appropriate views for each model: MemoryViewer, RegistersViewer and SymbolViewer. The Controller catches events generated by the user input and modifies the data in the Model. They are three controllers that handle this: MemoryTableModel, RegistersTableModel and SymbolsTableModel.

4.0 IMPLEMENTATION

The design of classes and algorithms reflects a lot of ideas taken from [10]. This made a source code shorter and the whole system more manageable. The advices taken from the same source also improved internal structure of the code and made it easier for different developers to understand and modify source files.

4.1 Problems/Solutions

This section discusses problems that were encountered during development of the software and solutions that were found and applied for each problem.

4.1.1 Debugger Area

At first, debugging capabilities were supposed to be implemented right inside the editor (Figure 13). After careful examination, this option of the IDE was moved to the memory viewer (Figure 12). The editor's text buffer typically has directives and comments, which are not important when debugging the application. The Moon code might not be aligned, which would strain user's eyes and decrease perception during debugging phase. In addition to that, the memory viewer would store and display instructions anyway. So, it made more sense to have debugging capabilities in the memory viewer instead of in the main window editor.

4.1.2 Scrolling

After the first prototype was developed, it was noticed that the memory viewer was very slow to redisplay during scrolling (Figure 12). After analyzing the problem using JProbe Java profiler, it became apparent that the simulated Moon memory was taking too much actual physical computer memory because the Java objects it held were too big. The

objects had to be redesigned, and some extra functionality such as conversion of integers to hexadecimal and ASCII form was removed from DataWord class and placed into the utility class (`Util.java`). This made DataWord object and all its children to be truly data objects and improved the speed of the memory viewer and the responsiveness of the whole user interface. When Moon memory was set to 5000 bytes, the simulator used on average 250KB less memory with the redesigned DataWord class.

Memory Viewer

Show All Memory

ADDR	HEX	CHARS	ASCII	INT
224	lb	R3 R2 me...		
228	ceqi	R4 R3 0		
232	bnz	R4 pr2		
236	putc	R3		
240	addi	R2 R2 1		
244	j	pri		
248	addi	R2 R0 na...		
252	jl	R15 getna...		
256	hlt			
260	00000000		0 0 0 0	0
264	00000000		0 0 0 0	0

Figure 12: Memory Viewer

4.2 Modules Description

The Moon IDE implementation has 11 modules and 34 classes. This section describes each module and classes that compose that model.

4.2.1 Editor

This package contains two classes: EditorBean and EditorBeanBeanInfo. The EditorBean class is a custom editor, which is build on top of JTexArea. The component allows user to

edit text, and set and remove breakpoints (Figure 13). The EditorBeanBeanInfo class is only needed in case the EditorBean must be laoded into the Java IDE such as Borland JBuilder, IBM Visual Age or similar IDE. The file allows Java IDE to display properties of the bean so that they can be modified by the user during design time.

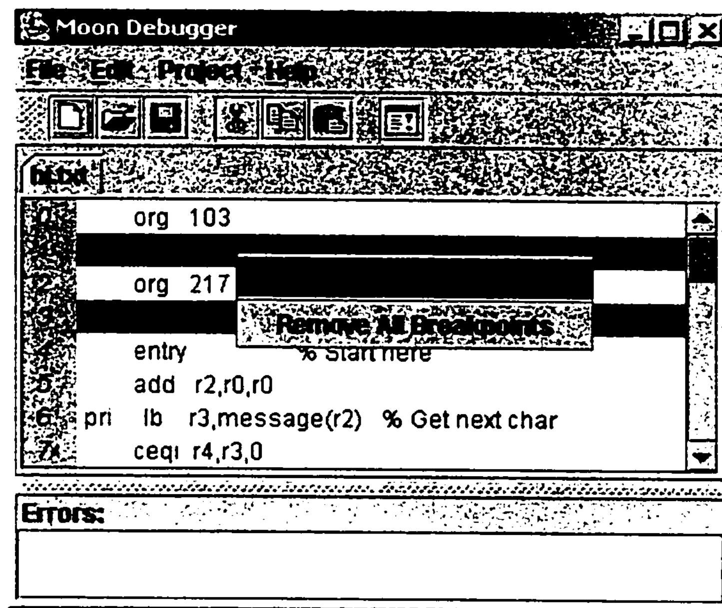

Figure 13: EditorBean

4.2.2 Help

This package contains the HelpViewer class, which displays help. It is a singleton class, which ensures that only on instance of the class can be instantiated. This feature makes sure that only one dialog is created for the whole application. This is a useful technique to reduce system resources use in case the user tries to open the help dialog multiple times or from multiple windows. The dialog is displayed on Figure 9.

4.2.3 Instruction

This package contains 5 classes: DataWord, Directive, Instruction, InstructionA and InstructionB. All of the classes in this package are data classes that represent Moon words, Moon directive and two types of Moon instructions. The classes can be seen on Figure 8.

4.2.4 Loader

This package contains two classes: MoonLoader and LoaderException. MoonLoader is used when the parser successfully parsed files, and loading to the simulated memory can be performed. The purpose of the class is to go through parsed instructions and directives, load them into memory and fill in the symbol table. If any errors are encountered such as invalid memory address or undefined symbol, the Module throws a LoaderException.

4.2.5 Memory

This package has three classes and one interface. The main class of this package is MoonMemory. It represents the simulated memory, and its purpose is to store data or instructions to be executed. MemoryTableModel is the controller that is used to convert memory content into human readable form, and then display it in the MemoryViewer (UI package). The MemoryTableModel implements MemoryChangeListener, which allows MoonMemory to notify MemoryTableModel when memory was changed at a particular address. The controller then redisplays that cell.

4.2.6 Parser

This package is used for parsing. The main class in the package, MoonParser, parses the text file that contains Moon code and checks for syntax errors. If no errors were

encountered, it returns a list off all parsed instructions and directives. If errors are encountered, the class throws a ParseException. Parse errors are displayed in the "Errors" area below text buffer (Figure 17).

4.2.7 Registers

The package is used for storing and controlling registers content. It is similar to the MoonMemory package (Section 4.2.5), and it has three classes and an interface: MoonRegisters, RegistersAccessException, RegistersChangeListener and RegistersTableModel.

4.2.8 Runtime

This package contains RuntimeHandler and MoonRuntimeException. RuntimeHandler has methods to execute one instruction at a time for "stepping" or execute all for "running". It checks if current instruction is a breakpoint, and suspends executions if it is. If a RuntimeError is encountered, such as invalid memory access, MoonRuntimeException is thrown. Runtime exceptions are displayed using a pop-up dialog.

4.2.9 Symbols

This package has classes to store and display the content of symbol table. The design of its classes is similar to that of MoonMemory (Section 4.2.5) and MoonRegisters (Section 4.2.7). The only difference is that the package does not have ChangeListener interface, because during simulation, the state of SymbolTable does not change and therefore SymbolsViewer does not need to refresh.

4.2.10 User Interface

The UI package contains all of the User Interfaces used in Moon IDE.

4.2.10.1 AboutDialog

AboutDialog gets displayed when user chooses the 'About' submenu form the 'Help' menu. Figure 14 shows the dialog.

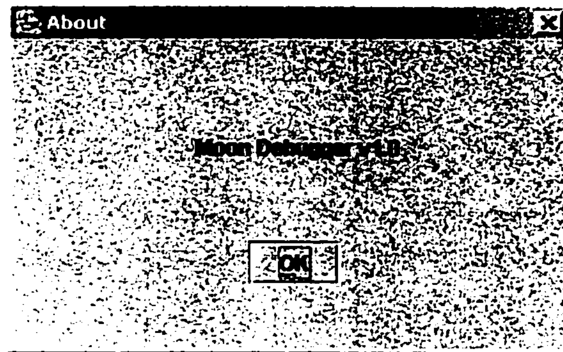

Figure 14: AboutDialog

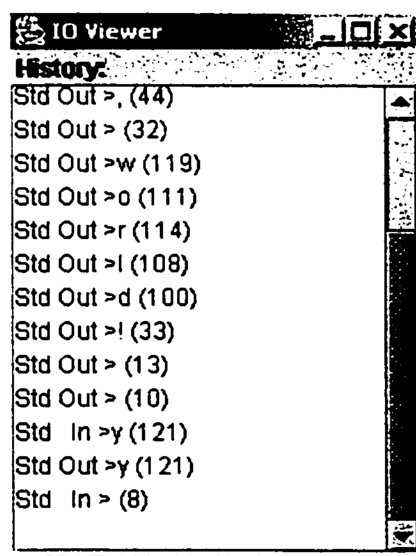

Figure 15: Input Output Viewer

4.2.10.2 IOViewer

IOViewer (Figure 15) allows users to see a history of what has been outputted and inputted. It is designed in such a way that users can scroll through history and see old values. Each line displays character originator (Standard Input or Output), character and corresponding ASCII code.

4.2.10.3 MemoryViewer

This class displays memory content, current instruction (Program Counter) and allows putting breakpoints by double clicking on the interesting instruction (Figure 16).

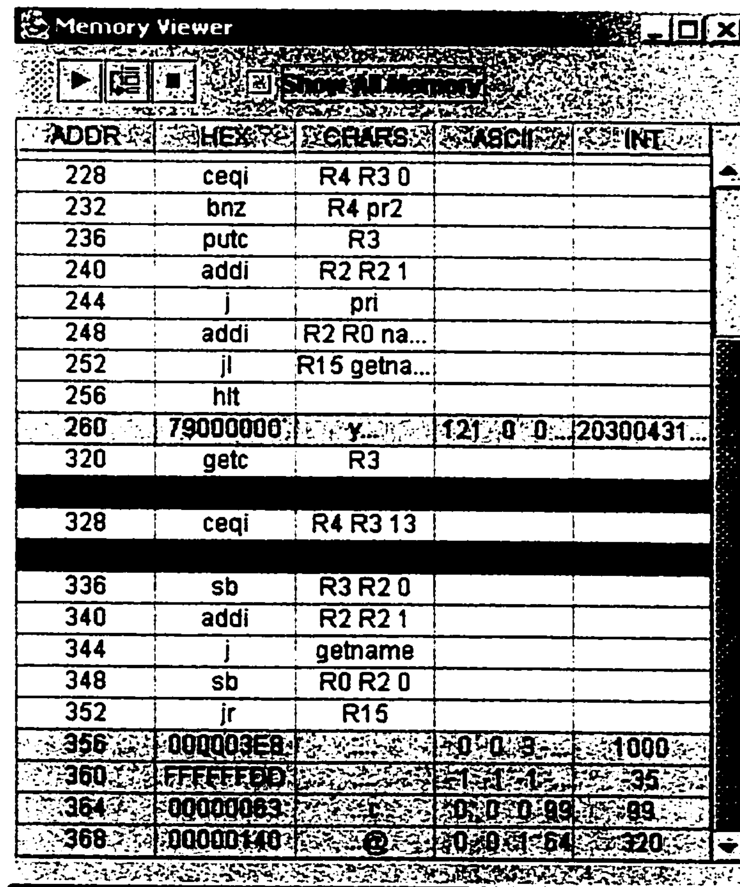

Figure 16: Memory Viewer (Non-zero Memory is Displayed)

Memory address 332 has a breakpoint, and instruction at address 324 is a current instruction. The memory viewer displays only non-zero memory to avoid extra scrolling and improve usability. The component can also display all memory (Figure 12), when user checks "Show All Memory" check box.

4.2.10.4 MoonDebuggerFrame

This is the main window of the IDE. The user can edit and compile Moon code using this component.

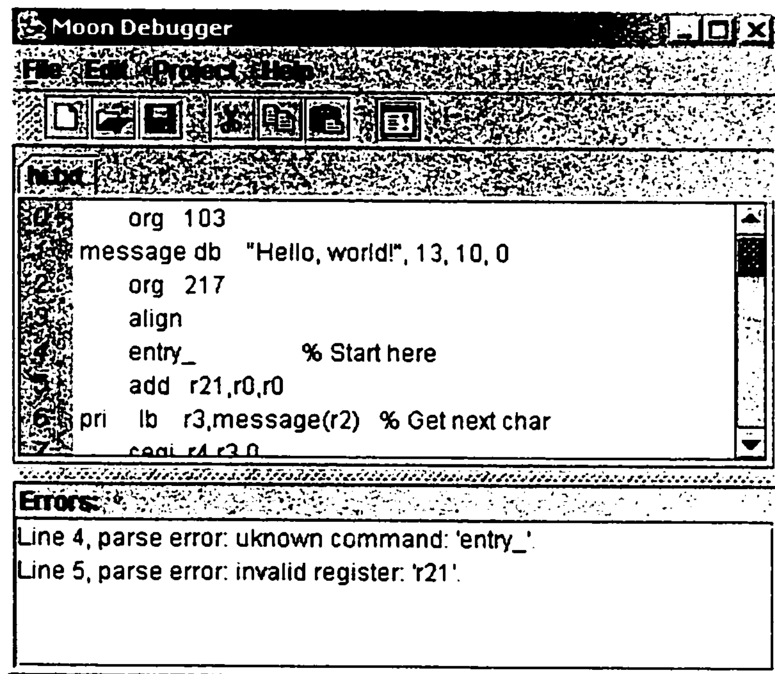

Figure 17: MoonDebugger

Figure 17 shows the MoonDebugger window with one file opened and two errors in the Moon code on lines 4 and 5.

4.2.10.5 RegistersViewer

The registers viewer shows the content of registers and highlights the last accessed register. Figure 18 shows that the last register to be accessed is R3.

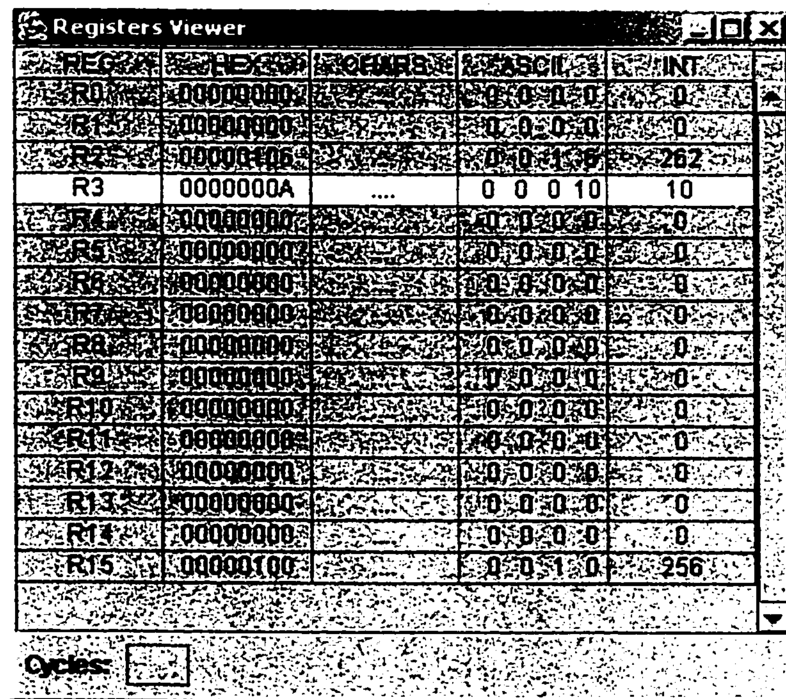

Figure 18: RegistersViewer

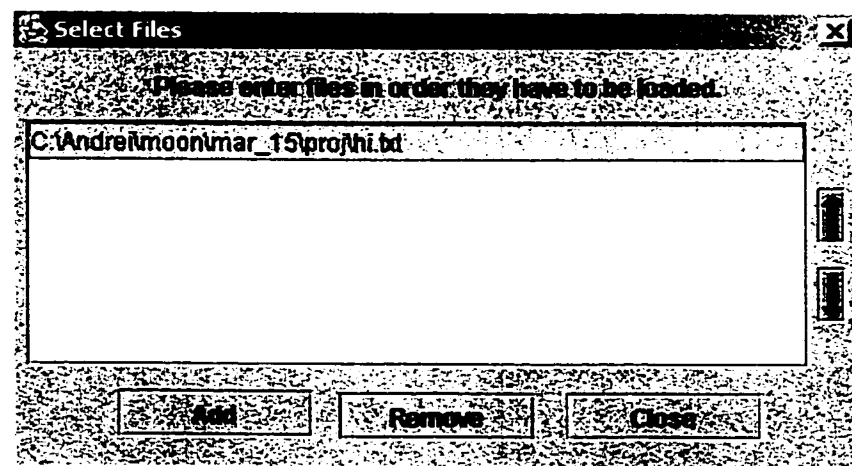

Figure 19: MultipleFilesSelect

4.2.10.6 SelectFilesDialog

When two or more files must be loaded, the user can specify what files to load and the ordering of the each load. The dialog window on Figure 19 is used for that purpose. It allows marking additional files for loading and permits the user to change ordering of the loading using buttons on the left.

4.2.10.7 StdInDialog

When an Input Instruction is encountered, the simulator opens StdInDialog. That allows the user to type in a character (Figure 20) or an ASCII code (Figure 21).

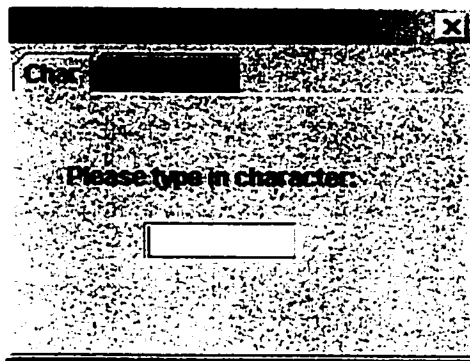

Figure 20: Character Input Dialog

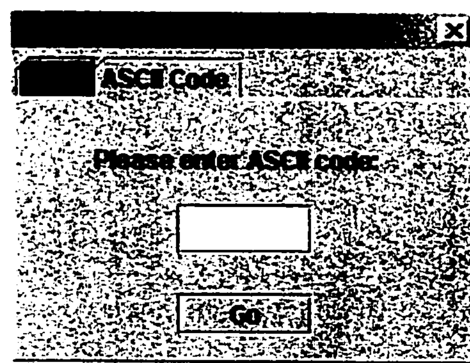

Figure 21: ASCII Input Dialog

4.2.10.8 SymbolsViewer

Symbols viewer dialog (Figure 22) displays the content of the symbols table.

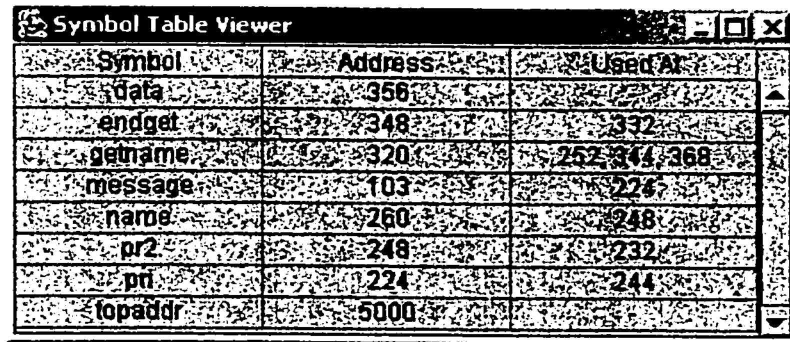

Figure 22: Symbols Table

4.2.11 Utils

This package contains utility classes with static methods that are used by some modules. The complete API for the utility classes FileUtil and Utils can be found in Appendix C.

4.3 Unique Features

This section describes interesting solutions that were used when implementing the Moon IDE.

4.3.1 Reflection

The parser uses reflection API to avoid if-else blocks by making one-to-one mapping between operation code of Moon instructions and appropriate parser methods. Reflection API allows invoking a method upon an object, even if the method is not known until run time. For example, let's assume that a function `parse_addi()` exists in a parser class. When line parser encounters instruction with opcode `addi`, it can concatenate `parse_`

and `addi` and call resulting function `parse_addi()` dynamically. Similar way of calling the right parse methods is used in MoonParser.

4.3.2 Text Buffer Modification

Every editor must check if the text was modified in order to save or not to save modifications to the file system for periodical backup or before compiling. Saving unmodified files every time wastes system resources and irritates the user. Nicely designed program has to address this issue. The simulator program uses Cyclic Redundancy Code (CRC) checksum to check if the text in the editor buffer was modified. One other design solutions would be to keep a copy of the text buffer and then compare the copy with the real content. This approach would take too much memory, since a copy of the text buffer must be kept in the memory. The second approach would be assigning a "dirty" flag to the editor and set it to true if something was typed. This approach has also a drawback because user might make a change and then undo it. The flag would tell the program what file was changed, even though in reality it was not. CRC method is the best solution because the only overhead it required was assigning a CRC value of unmodified text to each text buffer. The new CRC value can then be compared with the original CRC value, and if they are different, the text is saved in the file system. The feature is implemented using `java.util.CRC32` class. The probability that two different strings would have the same CRC value is 1 – 1/ (2^32) when strings would differ by more than 32 characters and 1 – 1/ (2^31) otherwise [13].

4.3.3 Method Testing

Some of the non-GUI classes have a main method that performs a white-box testing of the methods. If the source code were changed, the class can be tested by simply running the `main()` method.

5.0 SOFTWARE QUALITY

According to [5] there are thirteen quality issues that can be used to evaluate software systems. Some of them are correctness, robustness, maintainability, reusability, efficiency, portability, security, friendliness and understandably. Sections 5.1 through 5.4 explain how the Moon IDE software meets factors described above that were not covered in previous sections.

5.1 Correctness

The software is indeed correct, because it meets original specifications. To check for software correctness, different test scripts were created to test each instruction. The Moon IDE simulator was then tested with those test cases. The IDE was also tested against the original interpreter using different sample scripts by means of back-to-back testing. Tests results of the IDE matched results of the original interpreter.

5.2 Robustness

The software is robust, because usability ideas such as responsiveness and recoverability were used when designing and implementing software. Such practice insures that the program is reliable even in abnormal conditions. If abnormal conditions were to occur, the software protects the users by saving files before every compile.

5.3 Maintainability

The software is easily maintainable because OOD was used during its design. The program is divided into classes and packages, which are logical collections of classes.

This structure allows for easy code maintains, because code can be easily modified in one class without altering the other parts of the program.

5.4 Reusability

Some parts of the software are indeed reusable, because they were designed to be very generic. For instance, EditorBean can be used as an editor widget that supports setting and removing breakpoints and displaying line numbers in any Java program. Figure 23 shows the API of the EditorBean. The FileUtil class (Appendix C) can also successfully be used in other interactive programs. The class has functions to save and load files, to browse through the file system and to check if a given file indeed exists.

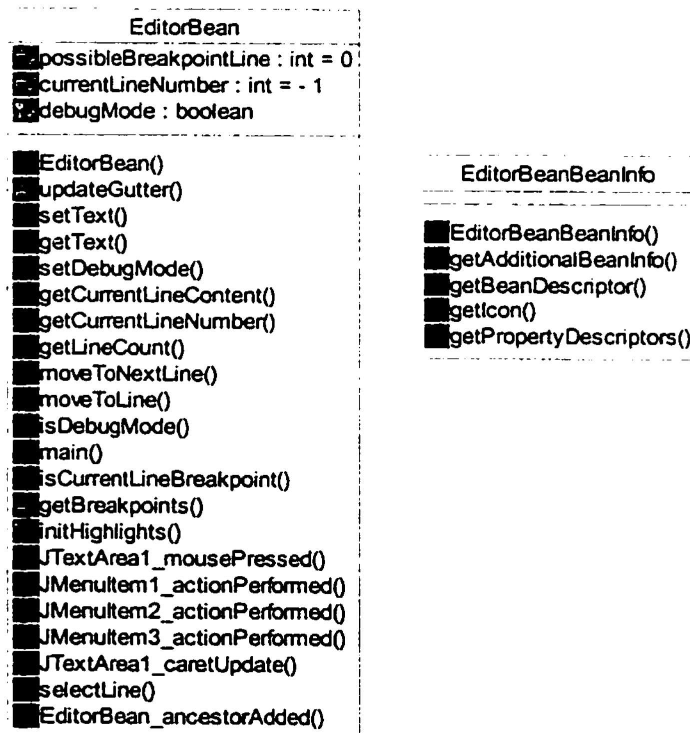

Figure 23: Editor Bean API

6.0 FUTURE ENHANCEMENTS

The Moon IDE program has a lot of room for improvement. This section discusses potential additions that can make the application even more useful.

6.1 Modifying Memory Content

There are some features of the software that are not implemented in this release. For instance, the software does not allow users to edit the content of the registers and memory. A future release might have this important feature implemented.

6.2 Better Simulation Reporting

The program might have a better statistical reporting for each run. For instance, the program might display what instructions were executed the most and what instructions caused the most cycles. To achieve that goal the program must have internal counter that records instruction types used during program execution. At the end of each run special form window can appear with statistics for each run. This information can be very useful for optimizing the compiler generating the Moon code in question.

6.3 Pluggable Assembler Engine

The IDE could be improved so that it could handle different assembly languages. The best design approach to achieve that objective would be to create a special description file for each assembly language. The file would have information such as a description of the instruction set and language grammar. The file could be parsed by the IDE, and after that, the new assembly language could be understood by the IDE. This feature would make it unnecessary to recompile the IDE if the Moon language were ever to be changed.

7.0 CONCLUSION

This report described the Moon IDE project, outlined its purpose, specifications and features. The document also highlighted the importance of OOD and usability goals and then illustrated how it affected design of the Moon IDE.

This project covered many tasks that included system analysis, architectural design. user interface design, program implementation and testing.

When developing the software, I really tried to create something very useful. During my years at Concordia I took the Compiler Design course. I remember it took me quite a bit of time to test the Moon code that was generated by a compiler I was developing. I hope that the Moon IDE program will help the students to save time when debugging Moon code and better understand the workings of a regular computer system.

REFERENCES

[1] Peter Grogono. *Home Page*. http://www.cs.concordia.ca/~faculty/grogono/ (Current March 12, 2002)

[2] J. Hennessy, D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[3] Ulrich Stern. *Java vs. C+*. http://verify.stanford.edu/uli/java_cpp.html (Current March 12, 2002)

[4] Borland Developer Support Staff. *Why Swing?* http://community.borland.com/article/0,1410,26970,00.html (Access March 12, 2002)

[5] Ian Graham, *Object-Oriented Methods, Principles & Practice*, 3rd edition, Addison-Wesley (2001).

[6] Grady Booch, James Rumbaugh , and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley (1999).

[7] Brad A. Myers. *Why are Human-Computer Interfaces Difficult to Design and Implement?* Carnegie Mellon University School of Computer Science, Technical Report CMU-CS-93-183, July 1993.

[8] John Karat. *Taking Software Design Seriously: Practical Techniques for Human-Computer Interaction Design*. Academic Press, 1991.

[9] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 1994.

[10] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] Aaron E. Walsh, John Fronckowiak *Java Bible*. IDG Books, 1998.

[12] Peter Grogono. *Moon Reference*. http://www.cs.concordia.ca/~faculty/paquet/teaching/442/moon.pdf (Current March 12, 2002)

[13] Greg Gagne. *Lecture Notes*. http://people.westminstercollege.edu/faculty/ggagne/spring2002/352/chapters/chapter3 (Current March 12, 2002)

[14] Nullsoft. *Nullsoft Install System*. http://www.nullsoft.com/free/nsis/ (Current March 12, 2002)

APPENDIX A – TOP LEVEL CLASS DIAGRAM

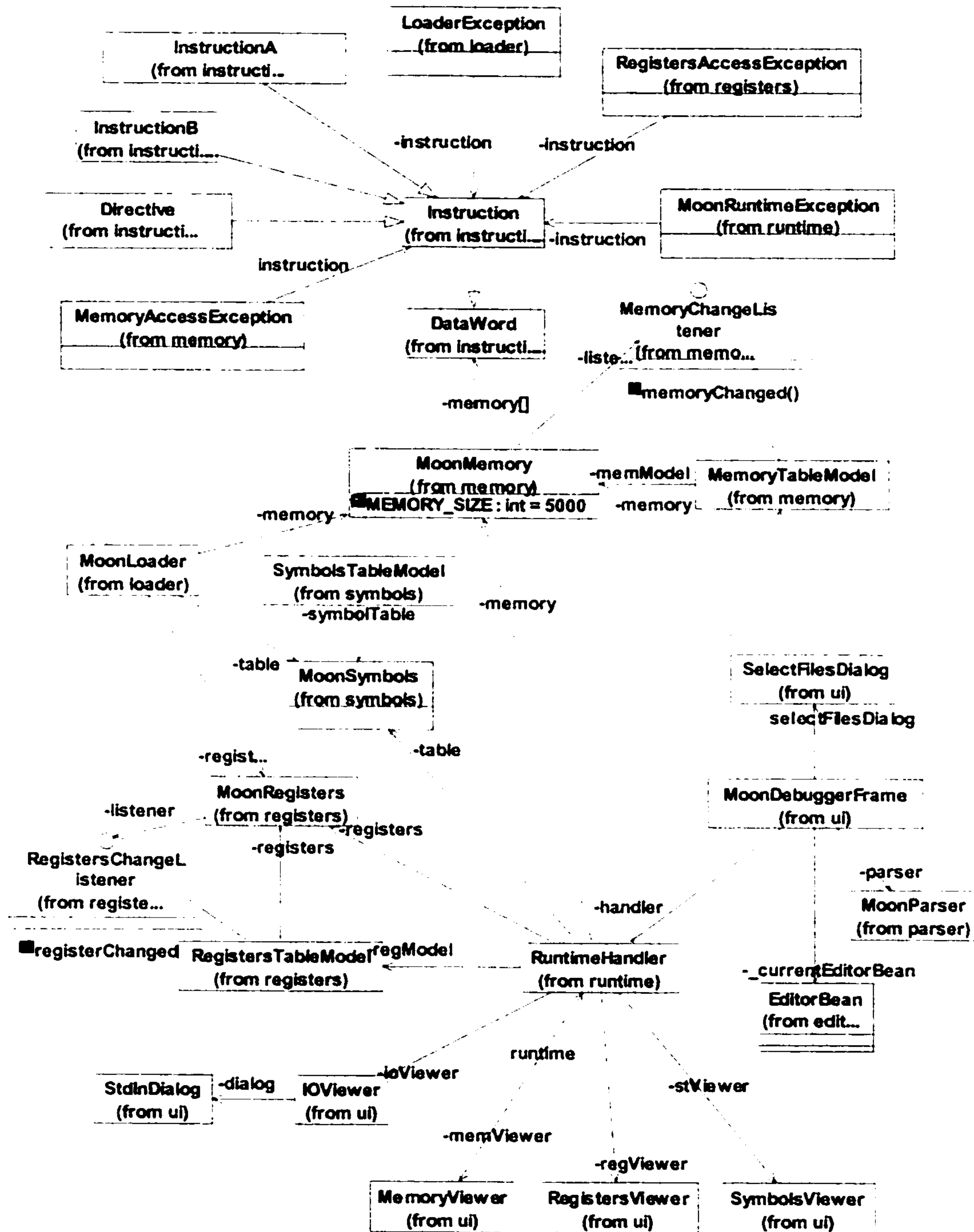

Figure 24: Top Level Class Diagram

APPENDIX B - FULL MOON IDE ON LINUX

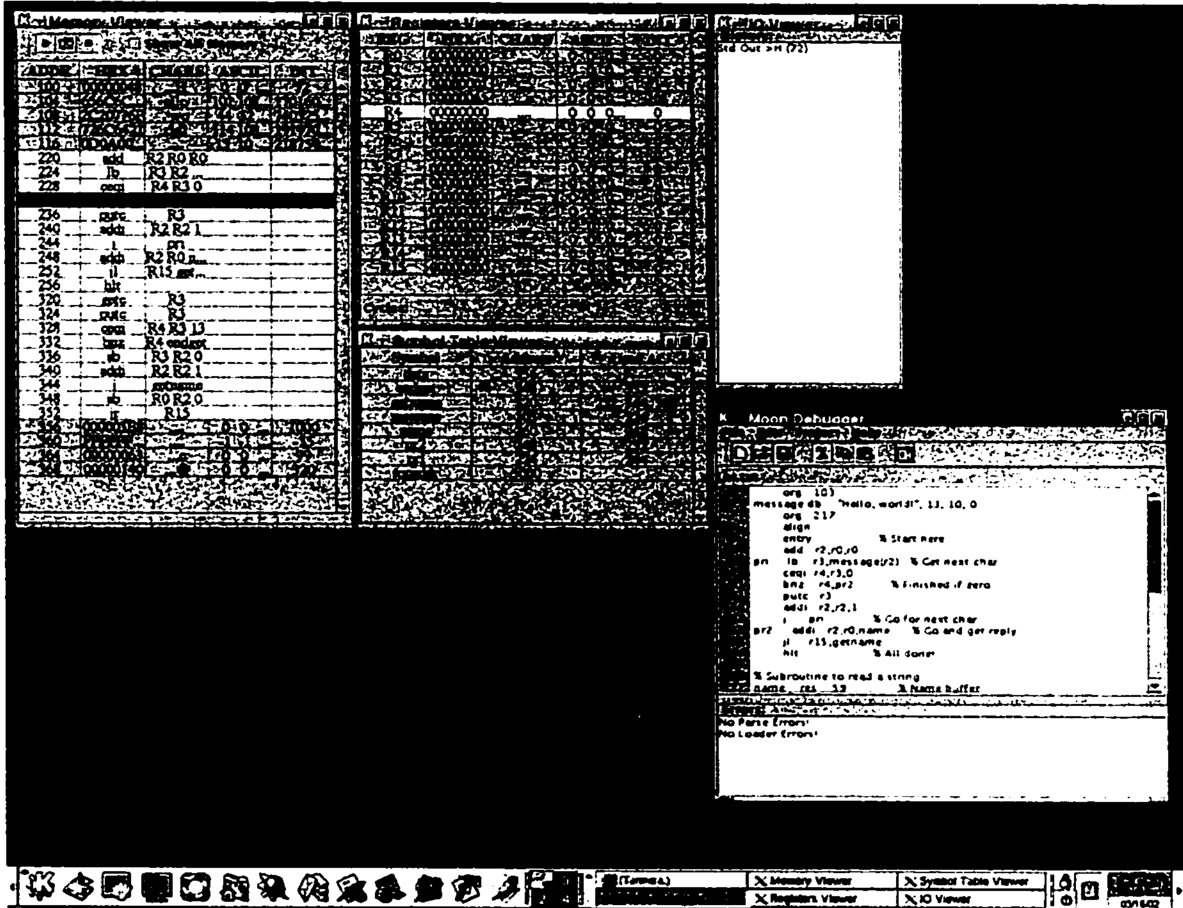

Figure 25: Moon IDE on Linux

APPENDIX C – MOON IDE API

The diagrams in this appendix displays the API for classes used in the project.

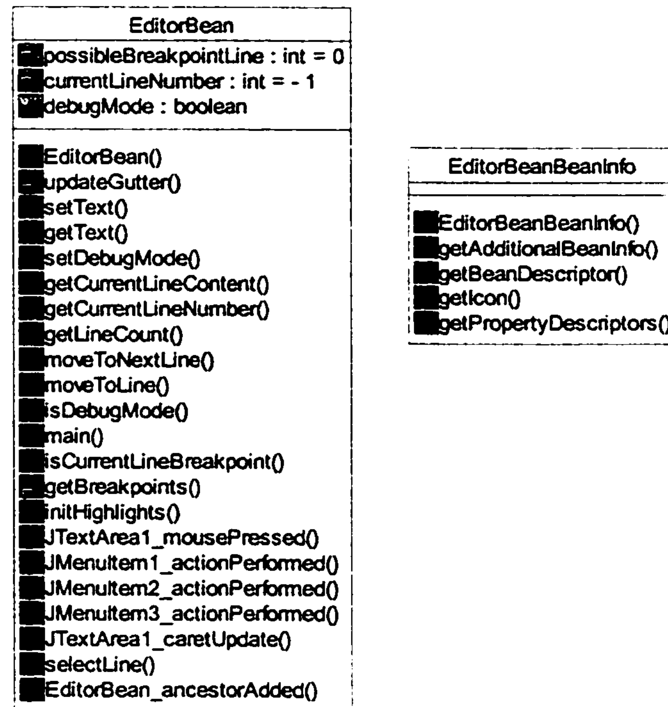

Figure 26: Editor Bean API

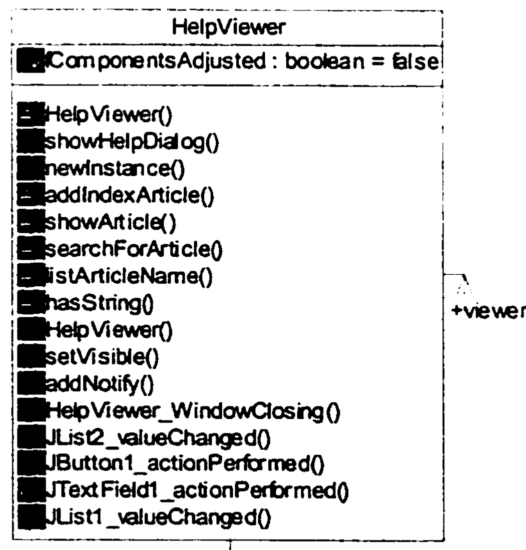

Figure 27: HelpViewer API

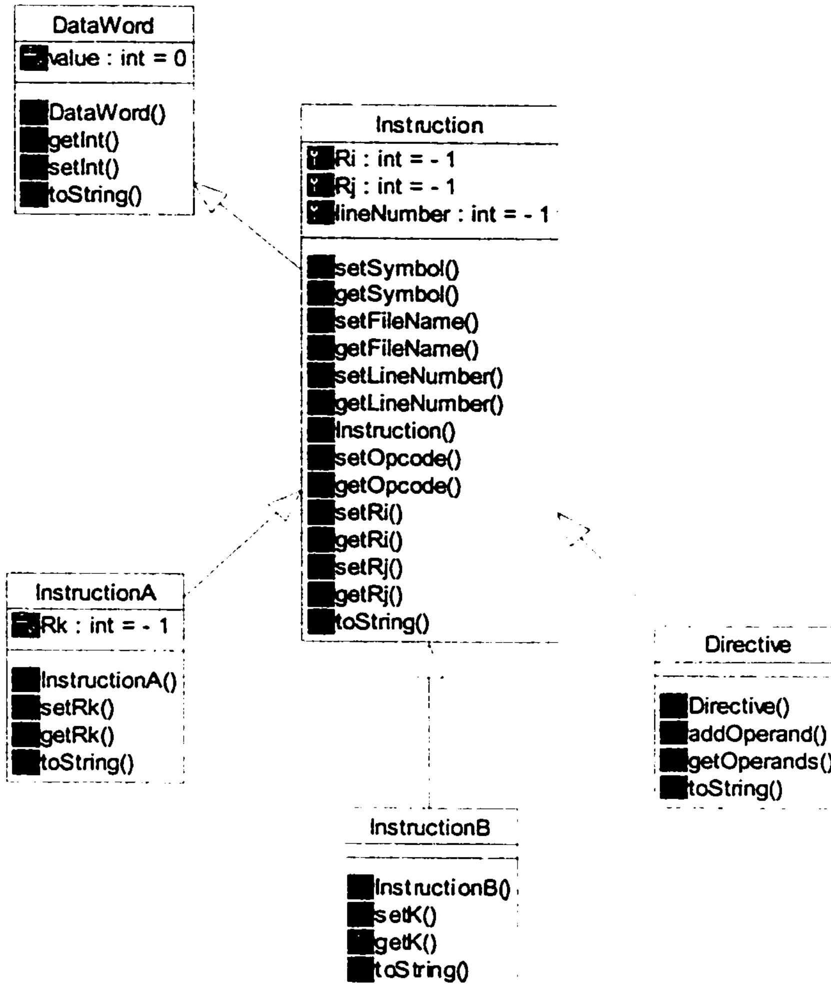

Figure 28: DataWord-Instruction API

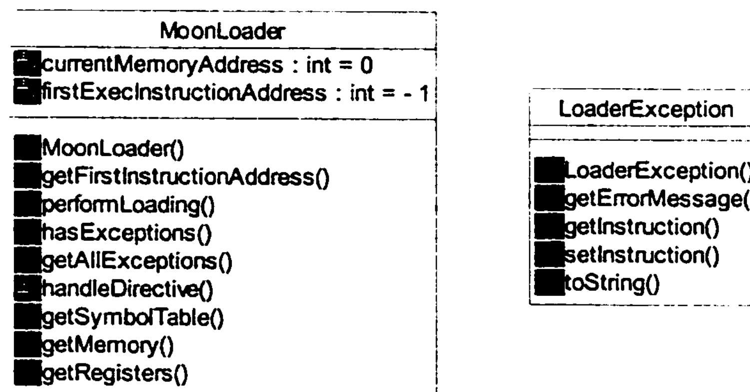

Figure 29: MoonLoader, Exception API

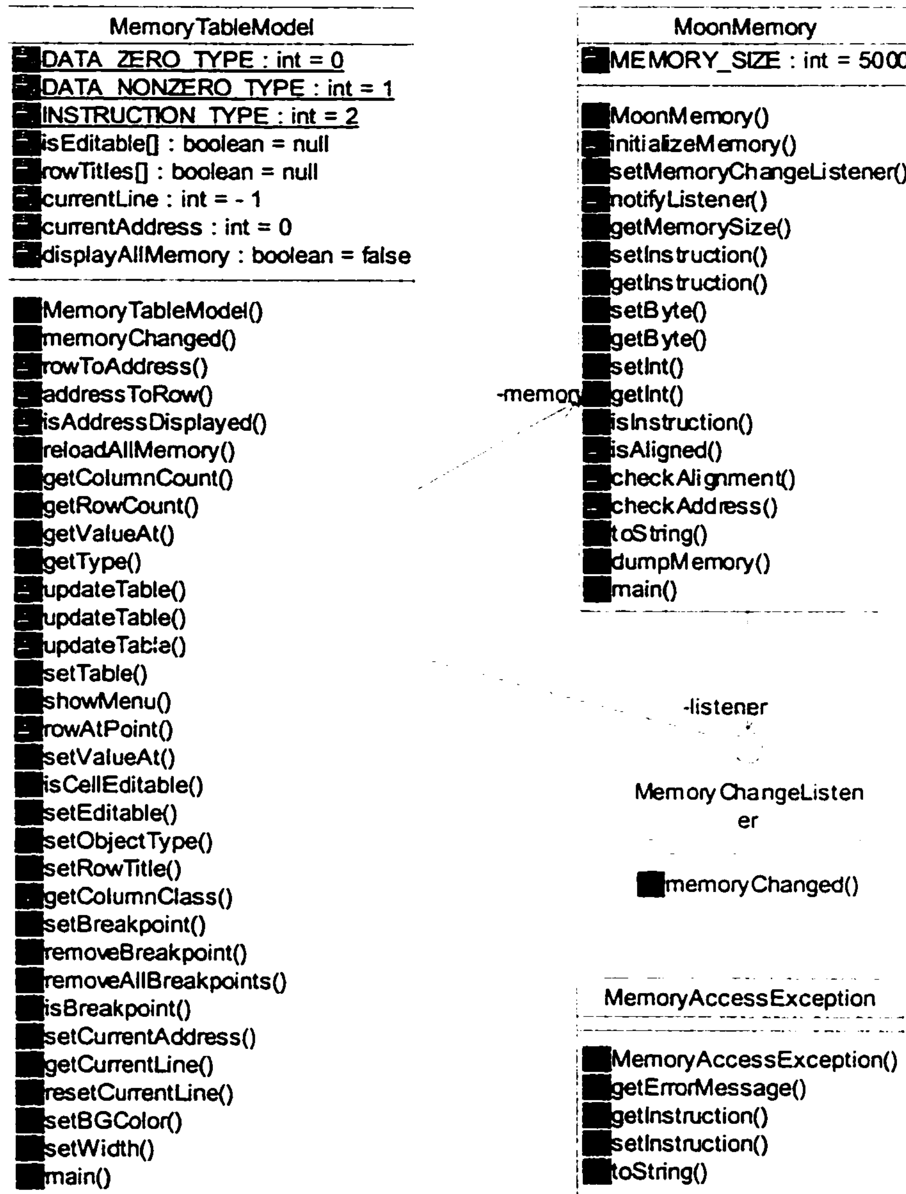

Figure 30: Memory Classes API

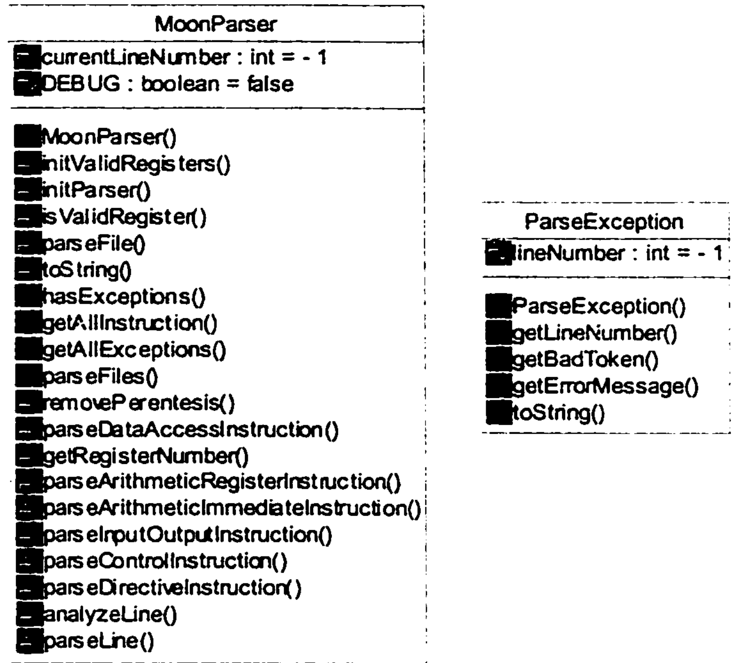

Figure 31: MoonParser, ParseException API

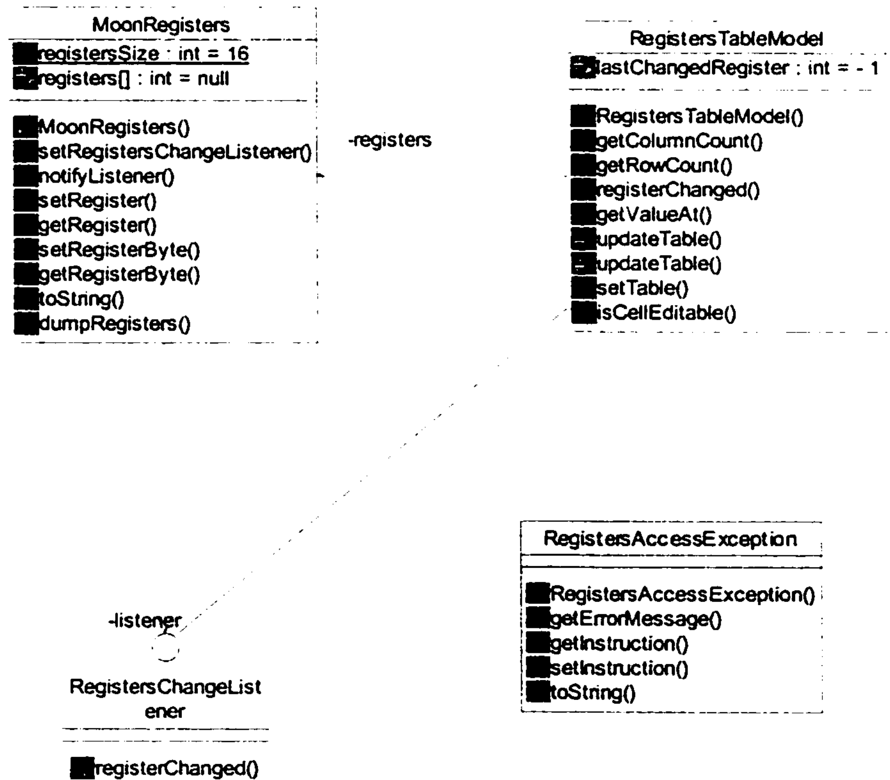

Figure 32: Registers API

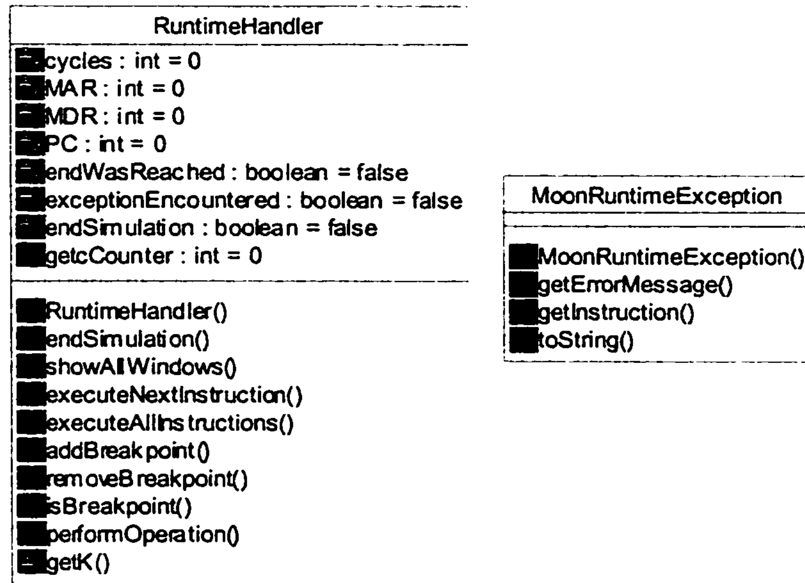

Figure 33: RuntimeHandler, MoonRuntimeException API

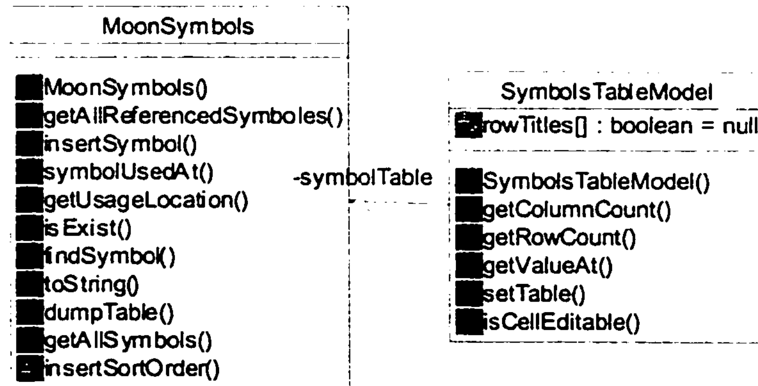

Figure 34: MoonSymbols API

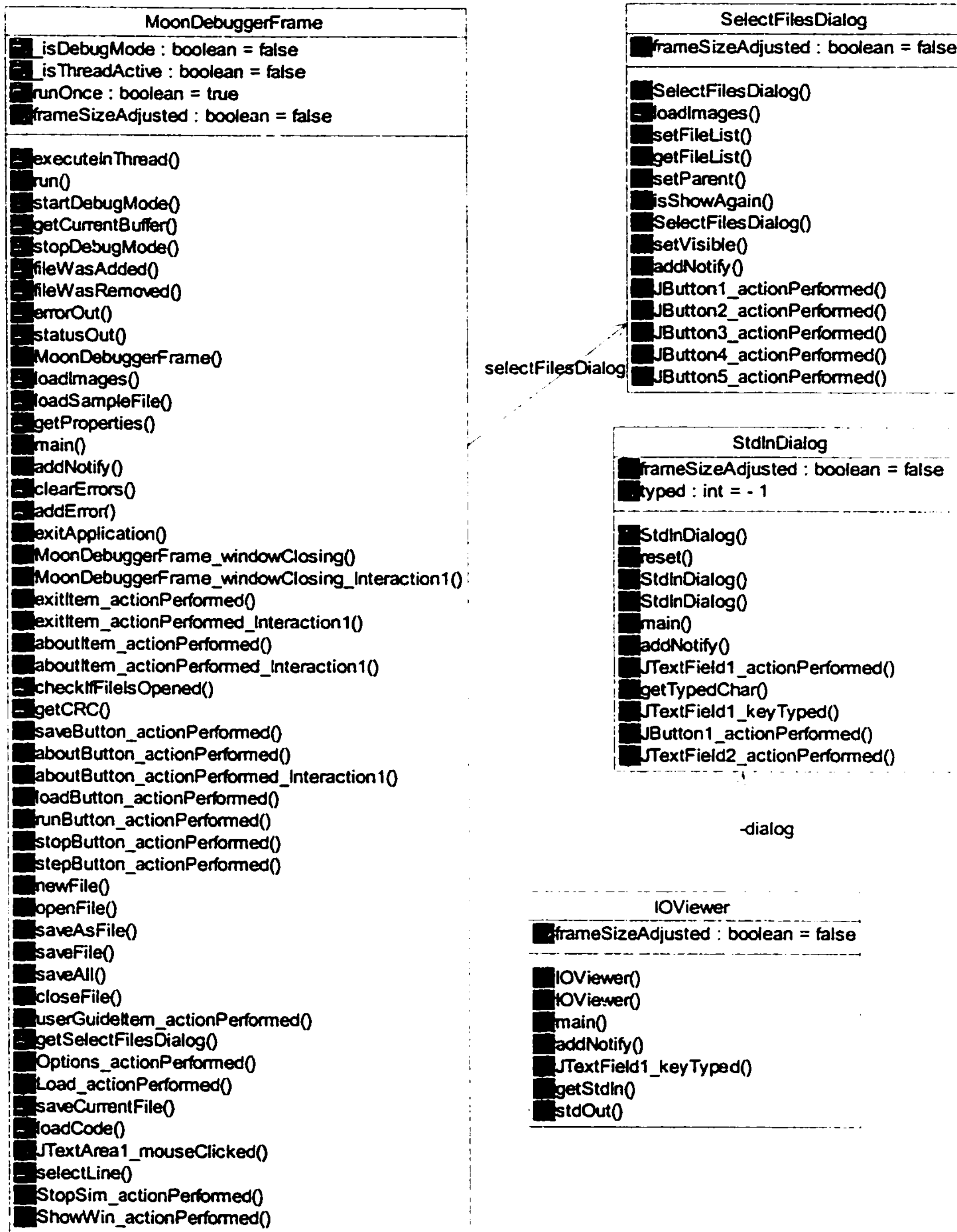

Figure 35: MoonDebugger API