An Innovative Integrated Development Environment for the Pocket PC PDA

by

Jack C. Kwok

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2001

B.S. Mathematics
Massachusetts Institute of Technology, 2001

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2002

© 2002 Jack C. Kwok. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author____
Department of Electrical Engineering and Computer Science
May 21, 2002

Certified by____
Li-Te Cheng
VI-A Company Thesis Supervisor

Certified by____
Tomas Lozano-Perez
M.I.T. Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Innovative Integrated Development Environment for the Pocket PC PDA
by
Jack C. Kwok

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Current Integrated Development Environments running on Personal Digital Assistants lack features that address the challenges in human-computer interaction with the PDA. The slow input techniques and small screen display of PDAs contribute to the overall difficulty of software development on the PDA. These problems discourage developers from using the PDA as a programming platform. An innovative IDE for the Pocket PC PDA is designed to facilitate fast prototyping and ease of development. The graphical user interface of the IDE overcomes the slow input techniques and small screen display of the PDA.

To take advantage of the capabilities of PDA, a novel design which integrates PDA personal information management capabilities with the IDE is explored. The programming language for the IDE is Ruby, a powerful general-purpose object-oriented scripting language. The PocketRuby IDE is a powerful and efficient programming development environment on the PDA.

Thesis Supervisor: Tomas Lozano-Perez
Title: Professor of Computer Science and Engineering, MIT Artificial Intelligence Laboratory

Thesis Supervisor: Li-Te Cheng
Title: Research Scientist, IBM Corporation

Acknowledgments

I would like to express my deepest appreciation to the people who helped make this work possible. First, I would like to give my sincere thanks my thesis supervisor Dr. Li-Te Cheng at IBM T. J. Watson Research Center in Cambridge for his guidance and support throughout the project. His encouragement and patience have helped made this thesis possible. I would also like to extend my gratitude to my thesis advisor Professor Tomas Lozano-Perez at MIT Artificial Intelligence Laboratory for his time and effort on supervising my thesis and for his advice in writing this thesis. Finally, I would also like to thank my family for their love and support throughout the course of my studies at MIT.

Contents

1. Introduction . . . 11
1.1 Problem Domain . . . 11
1.2 Objectives . . . 12
1.3 Thesis Organization . . . 14

2. Personal Digital Assistants . . . 15
2.1 Applications . . . 16
2.2 Challenges of Current Personal Digital Assistants . . . 17
2.3 Types of PDAs . . . 21

3. Human-Computer Interaction with PDA Devices . . . 23
3.1 The Role of the User Interface in a Software Application . . . 24
3.2 The Principles of Interface Design . . . 24
3.3 Graphical User Interface versus Command Language Interface . . . 26
3.4 Factors Affecting Legibility of Output . . . 28
3.5 Evaluation . . . 29

4. Integrated Development Environment . . . 31
4.1 The Need For PDA IDEs . . . 31
4.1.1 Rapid Prototyping of PDA Applications with PDA IDEs . . . 33
4.2 Survey of Current IDEs . . . 35

5. Ruby . . . 38
5.1 Scripting Languages . . . 38
5.2 Overview of the Ruby Language . . . 39
5.2.1 Features of Ruby . . . 39
5.2.2 Comparison Between Ruby and Perl . . . 40
5.3 Basics of Ruby Operations and Executions . . . 42
5.3.1 Ruby and The Microsoft Windows Operating Systems . . . 42
5.4 Interactive Ruby Shell . . . 43
5.5 Reasons for a Pocket PC Version of Ruby . . . 45

6. Architecture . . . 47
6.1 Overview . . . 47
6.2 Model-View-Controller Paradigm . . . 49
6.3 System Properties . . . 51

7. PocketRuby . . . 53
7.1 Ruby on PDAs . . . 53
7.2 Design Criteria for the PocketRuby Interpreter . . . 54
7.3 Porting Techniques and Technical Challenge . . . 55
7.4 Evaluation of Capability and Limitation . . . 58

8. The User Interface 60
8.1 Designing the GUI to Address Challenges on the PDA 61
8.2 Software Tools Used in the GUI Implementation 61
8.3 Overview of the Graphical User Interface of the PocketRuby IDE System 61
8.3.1 The Menu Bar 63
8.3.2 Editor Mode versus Interactive Mode 69
8.4 Addressing the PDA Input Limitation Problem 72
8.4.1 Frequently-Used Word List 72
8.5 Addressing the PDA Output Limitation Problem 75
8.6 Integration with Personal Information Management Tools 77
8.7 An Informal Evaluation 82

9. Integration 85
9.1 Communication Between PocketRuby Interpreter and the User Interface 86
9.2 The Mechanism of AppPipeDLL 86
9.2.1 The Messaging Protocol 88
9.3 Alternatives to AppPipeDLL 91

10. Conclusion 93
10.1 Summary of Work 93
10.2 Future Research Directions 96
10.3 Future Directions of PocketRuby IDE 96

Reference 98

List of Figures

Figure 1.2.1: A screen capture of the PocketRuby IDE running under an emulator 13
Figure 5.2.1: Two equivalent programs written in Perl and Ruby 41
Figure 5.3.1: Running Ruby from the command prompt 42
Figure 5.4.1: Interactive Ruby Shell running under Windows 44
Figure 6.1.1: Overview of the system architecture: the GUI and the language interpreter/compiler interconnected by a data exchange component 48
Figure 6.2.1: Model-View-Controller paradigm 49
Figure 6.2.2: Model-View-Controller paradigm with view and controller as a pair 49
Figure 8.3.1: Execution of a Hello World program in Editor mode and corresponding result displayed on the Result tab. 62
Figure 8.3.2: Execution of a loop in Editor mode and corresponding result displayed on the Result tab. 63
Figure 8.3.3: The IDE menu bar with the File menu activated. 64
Figure 8.3.4: The IDE menu bar with the Edit menu activated. 65
Figure 8.3.5: The IDE menu bar with the Tools menu activated. 66
Figure 8.3.6: The IDE menu bar with the Help menu activated. 68
Figure 8.3.7: Comparison between Editor Mode and Interactive Mode 69
Figure 8.3.8: A pop-up alert dialogue box appears when the user exits the system if the data on the code editor has not be saved. 71
Figure 8.4.1: An example of using the frequently-used word list context menu 73
Figure 8.4.2: The WordList tab 74
Figure 8.5.1: The tab control located on top of the menu bar 76
Figure 8.6.1: Ruby embedded documentation and corresponding data fields displayed under the Info tab 80
Figure 8.6.2: Adding a new contact to Pocket Outlook as an integrated feature of the IDE . . . 81
Figure 8.6.3: New entry added to Pocket Outlook address book 82
Figure 9.2.1: A high-level view of two applications exchanging data using AppPipeDLL's function calls 87
Figure 9.2.2: The GUI sending command messages to the PocketRuby interpreter 88
Figure 9.2.3: Data Exchange within PocketRuby IDE under Interactive Mode 90
Figure 9.2.4: Data Exchange within PocketRuby IDE under Editor Mode 91

List of Tables

Table 2-1: Comparison of input methods in terms of typical maximum speed 19
Table 2-2: PDA series with their pre-installed operating systems. 21
Table 2-3: Hardware specification of a Compaq iPAQ Pocket PC PDA 22
Table 3-1: The various disciplines and their contribution to HCI 24
Table 3-2: Advantages and disadvantages of command line dialogues 27
Table 3-3: Advantages and disadvantages of menu-based dialogues 28
Table 4-1: Typical PDA software development cycle using desktop IDE and PDA IDE 34
Table 4-2: PDA-hosted programming languages with currently available Pocket PC development tools and desktop development tools 36
Table 4-3: A survey of current PDA IDEs 37
Table 8-1: Association between tag names with Pocket Outlook database fields. 79
Table 10-1: Installation size of the PocketRuby IDE 94
Table 10-2: Memory Allocated for the IDE running on an iPAQ Pocket PC 95

Chapter 1

Introduction

1.1 Problem Domain

The design of personal digital assistants (PDAs) software development environments is not a new field. A variety of development tools are available for programming in different languages on the PDA.

There are issues inherent in the capabilities of PDAs affecting the ease of use of such programming environment. For example, the relatively small screen display and slow input techniques of typical PDAs greatly contribute to the challenge in making the PDA a popular programming tool. The current development tools on PDAs generally lack features that address the challenges specific to PDAs and the needs of their programmers, making software development on the PDA very difficult.

Certain limitations and challenges of PDAs can be overcome by the user interface design. A well-designed user interface should address the issues of human-PDA interactions. This thesis addresses the limitations of PDAs and explores innovative techniques and user interface design elements that improve the PDA as a software development tool.

1.2 Objectives

The main technical contributions of this thesis are the following:

- The PocketRuby Integrated Development Environment on the Pocket PC PDA
- The PocketRuby language interpreter: the first port of the Ruby language interpreter to the Pocket PC operating system. The PocketRuby language interpreter is used as a component of the PocketRuby Integrated Development Environment system.

The primary technical contribution is the development of the first integrated development environment (IDE) for the Ruby [1,5] scripting language on the Pocket PC operating system. PocketRuby IDE is the first Ruby development system that uses a graphical user interface.

PocketRuby IDE allows programmers to develop software in the Ruby scripting language directly on the Pocket PC. Thus, the entire software development cycle can be carried out within the PDA as opposed to relying on an emulator on a desktop development environment.

Figure 1.2.1 shows the PocketRuby IDE, with a user-defined class, *Foo*, and a one-line program instantiating *Foo* and executing the method, *foo*.

Figure 1.2.1: A screen capture of the PocketRuby IDE running under an emulator

Designing the graphical user interface of the handheld development system is a very challenging problem. The primary design goal of the graphical user interface is to overcome the various limitations posed by the typical handheld PDA device. The secondary goal is to take advantage of the unique capabilities of PDAs. The unique features of the PocketRuby IDE include the following:

- Support for two development modes: Editor mode and Interactive mode. Editor mode for regular evaluation and Interactive mode for fast interactive evaluation of Ruby statements.
- Features that address and solve the challenges of input and output on PDAs such as a context menu for easy entering of frequently-used words.

- An innovative approach in integrating built-in personal information management tools with the IDE to manage different aspects of software development.

The architecture of the IDE system requires modular design of system components. As a result, some properties of the IDE are code maintainability and reusability of modules. The user interface component and the interpreter component are reusable. In addition, the GUI can be readily replaced by another GUI within the system.

1.3 Thesis Organization

This thesis is divided into ten chapters. Chapter 2 introduces various types of personal digital assistants and their applications. Chapter 3 presents computer-human interaction research studies and results relevant to the PDA. Chapter 4 introduces IDEs running on PDAs. Chapter 5 gives an essential overview of the Ruby scripting language and the main reasons Ruby was chosen as the language for the IDE. Chapter 6 presents the high-level architecture of the PocketRuby IDE system. Chapter 7 describes the process of creating the PocketRuby interpreter. Chapter 8 describes the graphical user interface of PocketRuby IDE and explains how its features address the limitations and challenges on the PDA. Chapter 9 details the integration of the PocketRuby interpreter with the graphical user interface. Chapter 10 concludes this thesis by presenting an overall summary of contribution and a discussion of possible future work.

Chapter 2

Personal Digital Assistants

Personal digital assistants (PDAs) have enjoyed phenomenal popularity in recent years. PDAs were pioneered by Apple Computer, which introduced the Newton MessagePad in 1993. Shortly after, several manufacturers created similar products. Since the introduction of the Palm Pilot series of PDAs by Palm Inc., the demand for PDAs has resulted in enormous growth. Currently, major PDA manufacturers include Hewlett-Packard, Compaq, Toshiba, Sharp, Handspring, Palm, and Sony. There is an emergence of wireless PDA-like devices such as the Blackberry e-mail device and the Handspring *Treo* PDA-phone device. The market for PDAs is enormous and manufacturers introduce new PDA models to the market every few months.

PDAs offer several advantages over desktop and laptop computers. PDAs offer higher mobility than desktop and laptop computers. Because of their small size and light weight, PDAs are more convenient to carry around than other portable computers. Therefore, PDAs can be the ideal computing platform for highly mobile users. Another advantage concerns boot speed. Compared to PDAs, laptop computers require much longer time to boot up. Additionally, a laptop with a fully-recharged battery typically lasts for only a few hours while some PDAs last for days after each recharge. Finally, there are situations, such as in meetings, where typing is not

generally very welcomed because of it could be distracting to the speaker and the audience. Writing with a stylus on a PDA is much less conspicuous that typing on laptops.

PDAs are not likely to replace laptop computers in the near future because of their small screen size and the lack of a speedy and efficient input technology. Most PDAs rely on pen-based text input: the user writes with a stylus instead of typing on a keyboard. Handwriting is inherently slow and handwriting recognition is often inaccurate. In Section 2.2, the challenges and limitations of PDAs will be discussed.

2.1 Applications

The earliest PDAs originally functioned as personal digital organizers. PDAs were generally designed to provide personal information management (PIM) tools such as the calendar, to-do list, and address book. The main appeal of the PDA organizer was the vast amount of PIM data stored in a highly mobile device.

With advances in integrated circuits and storage technologies, computational power and storage capabilities of PDAs have drastically increased. Moore's law states that processing power for integrated circuits doubles for a given price every 18-month period. One can observe PDA's processor speed roughly following Moore's law in the past decade. There is an exponential increase in processing speed of PDA processors. The processor clock speed of the first PalmPilot PDA was a few megahertz whereas some current PDA processors feature clock rates of hundreds of megahertz. Memory has also enjoyed increased storage capacity for a given price. The first PalmPilot had 512 kilobytes of memory whereas a new Compaq iPAQ Pocket PC has 64

megabytes of built-in memory, not to mention an optional memory expansion slot which can carry up to 1 gigabytes of solid-state flash memory.

Increases in processing and storage power along with the advent of full-color high-resolution display have extended the PDA's capability as a computing device. Multimedia capabilities, such as the ability to play videos, are possible in newer PDAs. Some PDAs combine computing, telephone/fax, and networking capabilities.

There are numerous real world software applications on the PDAs, many of which address the need in the areas of technical support, system administration, and consulting services. For example, IBM has developed server-monitoring software (SNAPP) [12] running on the Palm and a security application called Wireless Security Advisor [11] running on the Compaq iPAQ Pocket PC. Compaq's Project Mercury [10] created an add-on unit to the iPAQ with a digital camera for technicians to use for on-site troubleshooting. The Pocket PC 2002 operating system includes features that appeal to IT administrators, such as virtual private networking and remote server control [13, 14].

2.2 Challenges of Current Personal Digital Assistants

The inherent hardware limitations of PDAs pose difficulties as a software development platform. The following limitations apply to PDAs in general.

Input/Output Limitations:

- Small Screen Size

A high-end Compaq iPAQ PDA has a resolution of 240 pixels by 320 pixels. Because of the limited resolution, concise presentation of relevant information is crucial to an interface design's success.

- Relatively Slow Input

Most PDAs have configurable physical built-in buttons for quick access to common applications such as address book and calendar, but these buttons cannot be used for text input. A typical PDA mainly relies on an over the screen digitizer and a stylus for text inputs. In fact, stylus-based input serves the purpose of both text input and pointing/positioning (for moving the cursor, menu selection and so on). The stylus-based input technique is very well suited for pointing/positioning such as menu selection and pushing buttons etc., but it is not ideal for fast input of long messages.

The transparent digitizer is layered directed over the liquid crystal display (LCD). The function of the digitizer is to translate the movement of the stylus into a series of (x,y) coordinates. Pattern recognition and processing algorithm then translates the series of coordinates into characters, numbers, and symbols. It is widely accepted that handwriting recognition is a very complex problem. As discussed in the next section, the accuracy of stylus-based input is affected by hardware factors, software factors, and human factors.

There are PDAs with built-in microphone and speech recognition capability. Speech recognition technology on the PDA, such as Microsoft MiPad [16] and Conversay Embedded

Speech Recognition [17], is in very early stage. Speech recognition is a young discipline with many difficult unsolved problems. One of the main problems with speech recognition is that speech is continuous. Normal speech seldom contains silent breaks between words. Therefore isolating the boundaries of individual words in continuous speech is very difficult. Other problems include background noise, redundant utterances, and variation between speakers [13]. While speech recognition is a very promising input technology for the PDA, there are still many technical obstacles to overcome.

Another disadvantage is that using speech recognition in public could potentially create a social disturbance.

Table 2-1 shows the comparison of different text input methods to a computer [8]. Normal handwriting is roughly 4 to 5 times slower than normal spoken English. When voice recognition matures into a stable technology, it is likely that it will replace handwriting as the primary method of input for PDAs. However, at present, handwriting is the standard means of text input.

Text Input Method to Computer	Typical maximum speeds (words per minute)
Normal handwriting	20-60
Regular QWERTY keyboard	80-100
Normal spoken English	180-200

Table 2-1: Comparison of input methods in terms of typical maximum speed

There are PDAs with built-in miniature QWERTY keyboards (such as the Sharp Zaurus SL-5500 PDA). However, these keyboards are difficult to use, not to mention theirs ergonomic deficiency. They have tiny keys and a cramped layout compared to a typical computer keyboard.

Typing on a built-in PDA keyboard makes use of only two thumbs whereas typing on a regular keyboard uses all ten fingers. Therefore, the typical maximum typing speed on a PDA keyboard is expectedly much lower than that of a full-size computer keyboard.

- Limited Input Accuracy

Both hardware factors and human factors contribute to the inaccuracy of stylus-based input. As an example of how hardware factors plays a part in the problem, parallax caused by the stylus tip and the display point on the screen being physically separated by the thickness of the display and the digitizer surfaces cause a slightly confused perception between the observed handwriting in comparison with writing on paper [13]. Additionally, digitizer introduces imperfections into handwriting [22, 23]. Some are caused by limitations or calibration errors within the digitizer technology. The deficiency can be static (present when the pen is stationary) or dynamic (dependent upon the pen's speed and direction of movement) [21]. In terms of human factors, handwriting styles used to record the English language vary widely. The writing can be composed of a mixture of individual separated characters and joined cursive characters; letters can be written in upper or mixed case; letters can be formed vertically or at a slant. No two persons have the exact same handwriting. In fact, the same person's handwriting is often different from time to time. Therefore, even with sophisticated pattern recognition and processing algorithms, errors occur.

Processing Limitations:

- Limited Storage Capacity

A high-end Compaq iPAQ Pocket PC has 64 MB of SDRAM. The limited storage space will be a constraint on the allowable memory footprints of the software applications.

- Limited Processing Power

A high-end Compaq iPAQ Pocket PC runs with a 200 MHz CPU, whereas a typical desktop runs with a 1 to 2 GHz CPU. Though PDA hardware technologies are advancing rapidly, system resources (such as processing speed and memory) are limited. The software applications must expose important functionality without consuming excessive resources.

2.3 Types of PDAs

PDAs are commonly categorized by their operating systems (OS). Table 2-2 shows PDAs with their pre-installed operating systems. It should be noted that a PDA pre-installed with a particular OS does not necessarily imply incompatibility with a different OS. As an example, the Compaq iPAQ has been shown to run an embedded version of Linux.

PDAs	Operating Systems
Palm series, Handspring Visors series, Sony CLIE series.	Palm OS
Compaq iPAQ series, HP Jornada series, Casio Cassiopeia series.	Microsoft Pocket PC OS
Agenda VR3 series	Linux-VR OS
Sharp Zaurus series	Lineo Embedix Linux OS

Table 2-2: PDA series with their pre-installed operating systems.

Palm OS and Microsoft Pocket PC OS are closed-source proprietary software. Linux-VR OS is the first open-source OS for PDAs. Linux-VR is an embedded version of Linux designed to

run on NEC VRSeries devices including the Agenda VR3 PDA. The Sharp Zaurus is another Linux PDA but runs the Lineo Embedix Linux OS. Linux PDAs are relatively new and have not enjoyed much popularity.

The decision on which OS to use for the IDE is basely on the hardware capability of the PDAs that run a particular OS. Strong emphasis is placed on processing speed and memory capacity. Programming environments can be quite memory hungry because of garbage collection. Linux and Palm PDAs generally have less memory and slower processors than Pocket PC PDAs. Therefore, Pocket PC is chosen as the operating system platform for the PocketRuby IDE.

The Compaq iPAQ Pocket PC was chosen as the testing device for the PocketRuby IDE. The iPAQ has probably enjoyed the greatest popularity among Pocket PC PDAs. Additionally, the iPAQ is arguably the most powerful PDA in terms of its hardware capability. Table 2-3 shows the hardware specification of an advanced iPAQ PDA model released this year (2002).

Model	iPAQ H3870
Processor	206-MHz Intel StrongARM SA-1110 32-bit RISC Processor
Memory (RAM)	64MB
Memory (ROM)	32MB
Wireless	Integrated BlueTooth, Infrared.
Display	TFT LCD 64K colors
Dimensions	5.3” x 3.3” x 0.62”
Weight	6.7oz

Table 2-3: Hardware specification of a Compaq iPAQ Pocket PC PDA

Chapter 3

Human-Computer Interaction with PDA Devices

Human-Computer Interaction (HCI) is the study of the relationships which exist between human users and the computer systems they use in the performance of their various tasks.

HCI is a multi-disciplinary field because HCI seeks to understand the computer system, the human user, and the task the user is performing. The ability to develop a computer system requires knowledge in computer engineering and programming languages. An understanding of the user requires knowledge in human behaviors, of social interaction, of environment, attitudes, motivation, among other things [15]. An understanding of task requires a means of identifying what is being done and why, and in what type of environment [15]. Table 3-1 shows the various disciplines and their contributions to HCI.

In this chapter, selected HCI research studies and results which are relevant to PDA devices are presented. An understanding of past research studies in HCI will guide the design of the GUI for the PocketRuby Integrated Development Environment. In particular, the principles of user interface design will be discussed in Section 3.2. Section 3.3 discusses the merits of the two main types of interface, the graphical user interface and the command language interface. Section 3.4 discusses output legibility. Section 3.5 discusses the evaluation process of a user interface.

Discipline	Contributions	Discipline	Contributions
A.I.	Help facilities, Modeling the user	Psychology	Understanding the user, modeling the user
Computer science and engineering	Faster machines, faster systems, means of building better interfaces	Sociology	Groupware
Physiology	Physical capability	Art	Aesthetic appeal
Philosophy	Creating consistency	Design	User interface layout
Ergonomics	Equipment design	Linguistics	Language for commands

Table 3-1: The various disciplines and their contribution to HCI

3.1 The Role of the User Interface in a Software Application

The user interface mediates between the users and the computer system. It reflects the system model to the users and translates their intentions into system activities [15].

The user interface is arguably one of the most important parts of a software application. The user interface is certainly the most visible part of the application. No matter how much time and effort is put into writing and optimizing internal code, the usability of the application depends on the user interface.

3.2 The Principles of Interface Design

The principles of interface design are presented here because they guide the design of the PocketRuby IDE's user interface. The main principles of interface design are naturalness, consistency, relevance, supportiveness, and flexibility [15].

- *Naturalness*

A good interface appears to be natural. The interface should reflect the user's task syntax and semantics: it should be in a natural language for the task involved and should appear to be structured according to that task. In addition, the interface should be self-explanatory.

- *Consistency*

The interface should reinforce the user's expectations from previous interactions with that system or with similar systems. It should be consistent in its requirements for input and should have consistent mechanisms for the user to make any demands on the system.

The user should not be expected to learn one method for one area of the system and then another method for somewhere else.

- *Relevance*

The interface should not request redundant information. It should require minimum of user input and should provide the minimum of system output necessary for the completion of the user's task. The on-screen information should be short and relevant and at the same time it must make sense to the user.

- *Supportiveness*

The interface should provide adequate information to allow the user to operate and to perform the task. There should be status feedback that provides information to help the user continue with the task. For example, the user should be able to tell what he could do with the system at a given time, and he should aware of what the system is doing.

New users need much more help that do experts. The interface designer should know in advance the level of understanding the user has of both the system and the task to provide adequate support.

- *Flexibility*

The interface should accommodate differences in user requirements and in user preferences. It should provide a variety of support levels and should also allow personalized output formats.

3.3 Graphical User Interface versus Command Language Interface

Two major means allow a user to communicate with a computerized system. First, the process could be carried out by linguistic manipulation, that is by typing in commands. Second, it can be done by the direct manipulation of objects, that is by using some sort of pointing device [15]. The first type of interface is a command language interface (also known as command line interface) and the second type is the graphical user interface.

A command language interface system offers a prompt which expects the user to input a command. The input could range from a single character to a word to a sentence. The commands and parameter keywords have to be chosen with care as it is imperative to follow the rules of the command syntax.

In a command line interface, command language dialogues are used. The user must type syntactically correct strings of words without help from the system. The distinctive feature of any command language dialogue is that no explicit support is provided to the user to show him the allowable set of commands. Instead, the user is expected to know these commands. One implication is that the user must memorize command names that he wishes to use. The user must avoid typing errors because of the lack of error handling. The reliance on the user's knowledge of possible commands imposes a high memory load.

Command line dialogues offer certain advantages. Command line dialogues offer high speed for expert users. Additionally, command line dialogues are generally concise and precise expressions which make very suitable interfaces in natural language speech recognition system.

Advantages	Disadvantages
Fast	Needs regular use
Efficient	High memory load on the user's part
Precise	Poor error handling
Concise	Involves typing or other means of text input

Table 3-2: Advantages and disadvantages of command line dialogues

One advantage of a GUI is that there is no need for the user to memorize the names of the commands and the syntax/format of the command language. The system is said to have a low memory load.

Within most GUIs, the user issues commands through menu selections. The GUI's menu system eliminates the burden of having to memorize command names and their syntax. No text input is required since menu selection can be done by simply tapping the stylus at the selection. The menu system offers a significant advantage over command line dialogues for a PDA device as it avoids the slow input speed and input accuracy problems described in Chapter 2.

A disadvantage of menus-based dialogues is that they can take up substantial display space on the relatively small PDA screen. In addition, data entry is not possible with menus but there are other GUI components designed for the task of data entry. Finally, menus do not provide flexibility. For example, only one selection can be made at a time.

Advantages	Disadvantages
No typing	Not flexible
Low memory load	Consume screen space
Well-defined structure	Not suited for data entry
	Consume processing resources

Table 3-3: Advantages and disadvantages of menu-based dialogues

A disadvantage of a GUI over a command language interface is that the GUI could consume a significant portion of total computation and memory resources. These resources are shared among all running application threads. The interpreter could take longer to return results with a GUI running at the same time. The effect is possibly more pronounced on PDA systems which have significantly less computation and memory resources than desktop computers.

The advantages and disadvantages of both menu-based and command dialogues systems were presented above. Menu-based dialogues within a GUI is more suitable than command language dialogues for the PocketRuby IDE because only the menu-based system solves certain limitation issues of PDAs.

3.4 Factors Affecting Legibility of Output

In user interface design, one consideration is legibility of the output. It is apparent that there is a difference between reading text from a display screen and reading text from paper. A number of studies have been carried out to quantify the difference. The conclusion from these studies was that reading from a CRT display screen is up to 30 percent slower than from paper and was also less accurate and caused more fatigue [27, 28].

There is a lack of research studies specifically in the area of PDAs output legibility, but it is safe to assume that PDAs generally suffer more problems in legibility than regular computer display screens. This is due to the relatively small screen size and low resolution of PDA screen displays compared to desktop or laptop computer displays. Designing a PDA user interface requires even more care and attention in ensuring output legibility.

Generally, mixed case letters are more legible than upper case. The size of the script is a factor that affects legibility. For the average person the size of the text should be such that it creates a visual angle of greater than 20 minutes of an arc [8]. In addition, the line, letter, and word spacing all combine to affect legibility. A page filled with text with little word spacing is hard to read. Line spacing should give not less than 50 percent of character height between top and bottom of adjacent lines.

Positive contrast (i.e., black text on white background) has been shown to improve legibility [8]. The number of words on each line also affects legibility. Experiments have shown that between 8 to 15 words per line produces the optimum legibility [8]. The dot matrix representation of a character affects legibility as each character is represented by a series of dots on the screen. Therefore, the font size must be chosen with care.

3.5 Evaluation

Evaluation is relied on quite heavily on HCI. It is hoped that evaluation will eliminate any problems that might be present in the systems. Evaluation of an interface is parallel to system testing in software engineering. It is the process by which the interface is tested against the needs and practices of the user. Thus, the evaluation should be done on a representative group. Often evaluation is inaccurate because it is done on people who already know too much about the

systems or people who are system designers themselves. Selection of a representative user-testing group must be done with care. There are multiple quantitative methods to carry out evaluations such as experiments, questionnaires, interviews and so on.

Evaluation should be done early. Design decisions can be difficult to reverse if evaluation appears late. The evaluation process should be on-going: design, evaluate, redesign and so on.

Chapter 4

Integrated Development Environment

An integrated development environment (IDE) is a programming environment that has been packaged as an application program, typically consisting of a code editor, a compiler, and a debugger. Some also have graphical user interface builders. The IDE may be a standalone application or may be included as part of one or more existing and compatible applications. IDEs provide a user-friendly framework for many programming languages, such as Visual Basic, C++, and Java. IDEs for developing HTML applications are among the most commonly used. For example, Macromedia DreamWeaver and Microsoft FrontPage are IDEs that automate tasks involved in web site development.

In this chapter, there will be examples to demonstrate uses a PDA-hosted IDE. The advantages and disadvantages of a PDA IDE over a desktop IDE will be examined. Finally, a survey of the existing PDA IDEs will be presented.

4.1 The Need For PDA IDEs

A PDA IDE is a PDA-hosted integrated development environment system for a PDA-hosted language. A PDA-hosted language is simply a language that runs on a PDA platform. Perl CE, Python CE, NS Basic, and Squeak [9] are examples of PDA-hosted languages

for which PDA IDEs have been developed. A PDA IDE system provides a framework for programming in a particular language.

A PDA IDE for a powerful modern programming language addresses needs in many areas including:

- Technical support and system administration
- IT consulting services
- Education settings

For example, an IBM Global Services consultant arrives a customer site with a PDA to troubleshoot a client's server scripts. He uses wireless Ethernet to receive code, then edits code on his PDA, and finally submits the fixed code back.

The programmer benefits from a PDA IDE with which he can use the PDA IDE to perform code analysis of desktop-based applications. The programmer can perform software testing on small changes of selected parts of the program code and later synchronize the code with a larger desktop-based software project.

A PDA can offer advantages in development of desktop applications, server-hosted applications, and PDA applications. A PDA can be used as a mobile platform to edit pieces of code or create desktop applications. A PDA could be used as a terminal client to edit a server-hosted application such as editing servlets and JavaScripts on a web server.

Research studies on classroom applications on the PDA designed to help student-teacher interaction in collaborative educational settings has been conducted [7]. One advantage of PDA over other computing platforms is PDAs offer a more cost effective solution than laptop computers. A good PDA IDE should contribute to the area of computer science education in

collaborative settings. For example, students and instructors can transmit program source code using wireless Ethernet, BlueTooth, or infrared. Such IDE facilitates sharing of ideas among students and instructors while working on a software engineering project. As another way to make use of highly mobile computing platforms in education settings, students and instructors can transmits instructions to experimental devices in a laboratory. For example, with their PDAs, students write and send instruction codes to a mobile robot to perform a physics or chemistry experiment in a laboratory.

4.1.1 Rapid Prototyping of PDA Applications with PDA IDEs

A PDA IDE promotes rapid prototyping of PDA applications. As shown in Table 4-1, software development cycle on the PDA IDE consists of fewer steps than the desktop development cycle for PDA applications. The shorter development cycle has a positive impact on the speed of development.

Desktop IDE Development Cycle	PDA IDE Development Cycle
Step 1. Edit source code files on desktop IDE Step 2. Compile source code into binary code Step 3. Install binary code on an desktop emulator Step 4. Test binary code on emulator (observe program behavior on test cases and verify correctness of program) Step 5. If behavior does not fit specification/criteria, debug code (and go back to step 1) Step 6. Install binary code on PDA Step 7. Test binary code on PDA (observe program behavior on test cases and verify correctness of program) Step 8. If behavior does not fit specification/criteria, debug code (go back to step 1). Otherwise, product is ready for release.	Step 1. Edit source code files on PDA IDE Step 2. Compile source code into binary code Step 3. Run binary code on PDA (observe program behavior on test cases and verify correctness of program) Step 4. If behavior does not fit specification/criteria, debug code (and go back to step 2) Otherwise, product is ready for release.

Table 4-1: Typical PDA software development cycle using desktop IDE and PDA IDE

A disadvantage of the desktop IDE development cycle is that the initial testing stage of PDA applications development relies on a software emulator. The emulator's function is to emulate the behavior of the actual PDA device. In practice, program behavior on the emulator is often not identical to behavior on actual PDAs. The differences in hardware (e.g. CPU

architecture, memory, I/O devices etc.) between the desktop computer and the PDA prevent perfect emulation of the PDA. Additionally, software bugs in the emulator itself further make perfect emulation impossible. A function that works on the device might not work on the emulator at all. In the case when the programmer implements functionality not supported in the emulator, the software application might crash under the emulator [6]. To debug the application, the binary code must be downloaded and tested on the PDA. Using the emulator for testing is often not very accurate or convenient.

The alternative is development on a PDA IDE. The entire development cycle, including coding, testing, and debugging, is completely undertaken on the PDA device. A shorter development cycle (shown in Table 4-1) promotes rapid prototyping. Therefore, a PDA IDE with properly designed functionality and usability could potentially increase the programmer's efficiency. One disadvantage of a PDA IDE is there are fewer processing resources to run and debug a program. Another disadvantage of a PDA IDE is the challenges of text input and small screen size discussed in Section 2.2.

4.2 Survey of Current IDEs

Although there is a wide variety of desktop IDEs for developing PDA applications, there is a relatively limited variety of PDA IDEs (see Table 4-2). Perl CE, Python CE, NS Basic, and Squeak are among the few languages for which IDEs on Pocket PC have been developed. Even fewer IDEs exist on Palm PDAs, which generally have less memory than a Pocket PC. A PDA IDE typically consists of a simple code editor linked with a language compiler or interpreter. A lack of features geared toward mobile programming on PDAs possibly has made such PDA IDEs

less appealing. Table 4-3 describes some of the deficiencies in existing IDEs which might discourage programmers from using a PDA IDE.

Language	Pocket PC IDE	Desktop IDE
Smalltalk-80	Squeak (Pocket PC version)	(None)
BASIC	NS BASIC	MS Embedded Visual Basic
C	PocketC text editors	MS Embedded Visual C, PocketC
C++	(None)	MS Embedded Visual C++
Python	Python CE	(None)
Perl	Perl CE	(None)
Scheme	Pocket Scheme	(None)
Java	(None)	1. The Jeode EVM supports Sun's PersonalJava™ and EmbeddedJava specifications. 2. Microchai is an HP implementation of Sun’s VM for PocketPC and embedded devices [18].

Table 4-2: PDA-hosted programming languages with currently available Pocket PC development tools and desktop development tools

A typical PDA IDEs surveyed in Table 4-3 lacks a graphical user interface design addressing the issues and challenges related to programming on handheld devices. Issues include the speed and accuracy of text input. A description of general issues and challenges for PDAs was presented in Section 2.2. An adequate and well-designed graphical user interface of a PDA IDE should address these challenges.

IDE/ Language	Description
Perl CE/ Perl	Installation difficult: Perl libraries need to be copied over from standard distribution. IDE does not offer much features besides simple code editing, file loading/saving.
Pippy/ Python	Pippy runs on PalmOS only. It has a very simple UI with no scrolling ability. There is a Pocket PC port called Python CE.
LispMe/ Scheme	LispMe runs on PalmOS only. The UI of LispMe is not intuitive. It has no scrolling ability. Only one result is displayed at a time and the old result is erased when the new result is displayed. Input is difficult.
PocketC Development/ PocketC	PocketC runs on Palm and Pocket PC. PocketC Development contains a very simple editor for code editing. Entering code is difficult as there is no feature to alleviate the difficulty of input.

Table 4-3: A survey of current PDA IDEs

Chapter 5

Ruby

Ruby is a pure object-oriented scripting language designed from the ground up to support the Object-Oriented Programming (OOP) model. Ruby is open source, distributed under GNU General Public License (GPL), and under active development by the Ruby developers community. The Ruby interpreter is written in the standard C programming language. With the Ruby C API, it is relatively easy to add extension libraries to Ruby. Background knowledge of Ruby is a prerequisite to understanding the PocketRuby interpreter and the PocketRuby Integrated Development Environment (IDE). The purpose of this chapter is to present relevant background information. The chapter consists of an overview of the Ruby language, the background technical knowledge about Ruby, and the reasons Ruby was chosen for the IDE system.

5.1 Scripting Languages

There is no standard definition of a scripting language. According to the definition from the *Free Online Dictionary of Computing*, a scripting language is a loose term for any language that is weakly-typed or untyped and has little or no provision for complex data structures. A program in a scripting language is often interpreted.

Some common scripting languages are Perl, Python, AppleScript, C Shell, and Tcl. In general, scripting languages offer some advantages over application languages such as Java and C++.

5.2 Overview of the Ruby Language

Ruby is a scripting language created by Yukihiro Matsumoto [5]. While Ruby is a weakly-typed interpreted language, Ruby is capable of handling complex data structures. Matsumoto's goal was to make the language faster and easier than existing scripting languages such as Perl and Python [5]. According to Matsumoto, Ruby follows the *principle of least surprise*, meaning that features in Ruby work just like what programmers would expect them to work [5].

Ruby is a general-purpose language. Ruby has been used for applications from text processing, XML applications, and general system administration to artificial intelligence, machine-learning research, and as an engine for exploratory mathematics [2].

Ruby is portable. Ruby has been ported to many platforms including UNIX, DOS, Windows, OS/2, etc. Some important features of Ruby will be discussed next.

5.2.1 Features of Ruby

Ruby is a pure object-oriented scripting language. The language has a simple syntax which should be familiar to programmers of modern programming languages. For example, the following factorial function written in Ruby illustrates Ruby's simple syntax:

```
def factorial (n)
  if n == 0
    return 1
  else
    return n * factorial (n-1)
  end
end
```

Ruby comes with a set of bundled class libraries that cover a variety of domains, from basic data types to thread programming. Besides bundled libraries, there are unbundled libraries. Exception handling is supported.

Ruby has full object-oriented functionality. OOP features such as inheritance, polymorphism, singleton method, and mix-in are implemented in Ruby.

5.2.2 Comparison Between Ruby and Perl

Ruby syntax and design philosophy are heavily influenced by Perl. This section contains a comparison of Ruby and Perl.

Ruby was purely object-oriented from the beginning whereas Perl's OOP features were added to the non-OOP Perl as an afterthought. As a result, the object-oriented programming (OOP) features of Perl are less elegant and are harder to use than those of Ruby. Ruby's OOP provides a number features that Python lacks or is still working toward: a unified type/class hierarchy, metaclasses, and the ability to subclass everything. Ruby only supports single inheritance, but it has a powerful mixin concept: a class definition may include a module, which inserts that module's methods and constants into the class.

Ruby uses less punctuation ($,@,%, etc.) than Perl (See Figure 5.2.1). Ruby uses prefixes ($,@) to denote variable scope, not data type. Ruby interprets newline characters or

semicolons as the ending of a statement. Thus, it is not necessary to append a semi-colon at the end of each statement as long as a new line is used. Ruby programs are generally less cryptic than Perl. The high readability of Ruby programs makes them easier to maintain.

Perl is staticly typed (except for references) while Ruby is dynamicly typed. Types of variables and expressions are determined at runtime as well as class and method definitions. The dynamic programming aspect of Ruby allows a program to generate programs within itself and execute them.

Perl

```
@array = (1, 2, 3);
puts $array[1];
%hash = ('foo1' => 'bar1',
            'foo2' => 'bar2',
            'foo3' => 'bar3');
puts $hash{'foo1'};
```

Ruby

```
array = [1,2,3]
puts array[1]
hash = {'foo1' => 'bar1',
          'foo2' => 'bar2',
          'foo3' => 'bar3'}
puts hash['foo1']
```

Figure 5.2.1: Two equivalent programs written in Perl and Ruby

One similarity is both Ruby and Perl are extensible through class libraries. The Ruby C API allows developers to extend on Ruby. Since Ruby is relatively new, the number of available libraries is currently less than that for Perl.

Ruby has extensive support for regular expressions, a feature that Ruby has borrowed from Perl..

5.3 Basics of Ruby Operations and Executions

All versions of Ruby, except PocketRuby, do not make use of any graphical user interface. Instead, Ruby is generally run from a command line interface such as the MS-DOS command console and UNIX console. For example, to execute a ruby program file named *hello.rb*, one would enter *ruby hello.rb* on the command line prompt (As in Figure 5.3.1). Ruby supports command line options that give the user control over the behavior of the interpreter. For example, the user can turn the debugger on, among other options. The list of command line options can be found in *Ruby in a Nutshell* [5].

Figure 5.3.1: Running Ruby from the command prompt

5.3.1 Ruby and The Microsoft Windows Operating Systems

Ruby was originally written for Portable Operating System Interface (POSIX) environments, which enables it to take advantage of all of the UNIX system calls and libraries.

Unfortunately, Microsoft Windows operating systems do not provide a POSIX environment by itself, so an emulation library is required to provide the necessary functions.

There are two Ruby binaries that run on 32-bit versions of Windows: *ruby-cygwin32* and *ruby-mswin32*. The most common one, *ruby-cygwin32*, uses *cygwin* which is a UNIX emulation layer that provides substantial UNIX API functionality. The emulation is contained in one dynamic linked library cygwin1.dll. When compiling Ruby for Windows, cygwin1.dll is linked to Ruby to provide the UNIX API functionality. *ruby-cygwin32* is compiled by GNU C Compiler (GCC). The advantage of *ruby-cygwin32* is that it runs very much like Ruby on UNIX. The *ruby-cygwin32* port also works well with extension libraries. However, *ruby-cygwin32* emulation library does not work under Windows CE [20]. Therefore, *ruby-cygwin32* is not readily portable to the Pocket PC operating system.

An alternative to *ruby-cygwin32* is *ruby-mswin32* [19]. *ruby-mswin32* is compiled by Microsoft Visual C++. A known shortcoming of *ruby-mswin32* is that certain functions which Ruby on UNIX has cannot be used. After proper modifications, *ruby-mswin32* does compile and run on the Pocket PC. Therefore, *ruby-mswin32* was used as the foundation for porting Ruby to the Pocket PC.

5.4 Interactive Ruby Shell

Interactive Ruby Shell (IRB) is a Ruby module where the user can interactively evaluate Ruby expressions and see the results immediately. Figure 5.4.1 shows a simple example of using IRB within a DOS command prompt in Windows.

Interactive interpreter allows very fast testing of ideas without the overhead of creating test files as is typical in most programming languages. Interactive evaluation of Ruby expressions is possible because of the dynamic property of Ruby and the ability to evaluate a program within a program. IRB is a very useful feature of Ruby and comes with all standard distributions of Ruby. A similar feature is implemented in the PocketRuby IDE.

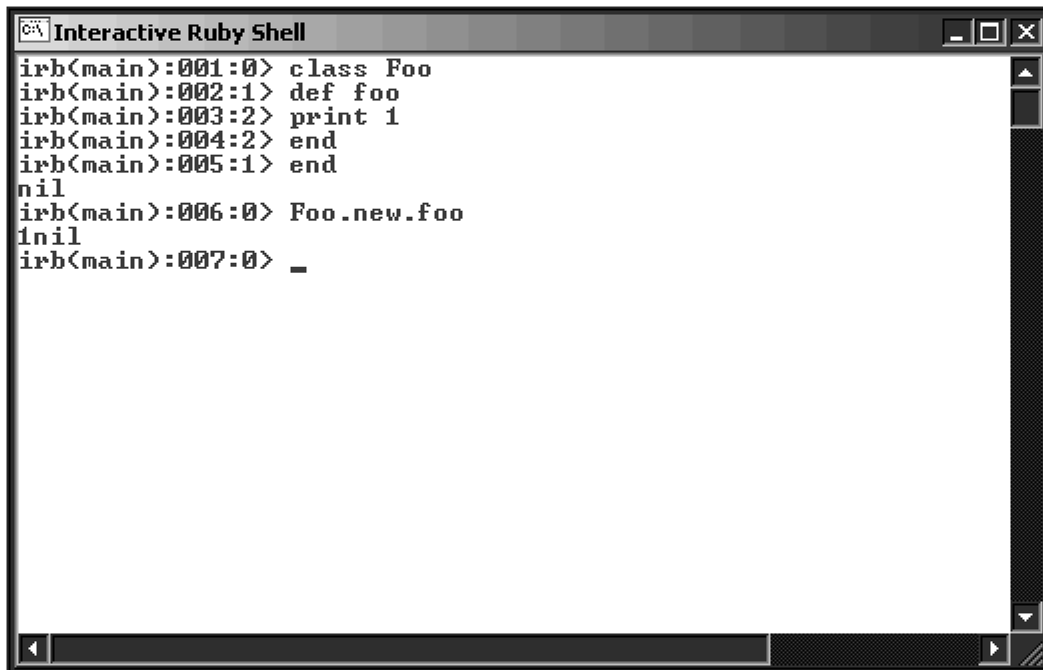

Figure 5.4.1: Interactive Ruby Shell running under Windows

5.5 Reasons for a Pocket PC Version of Ruby

Ruby was chosen as the language for our IDE for the following reasons.

Language

- Ruby programs are typically more concise than their Perl, Python, or C++ counterparts [2]. This is ideal for the PDA for two reasons. Ruby scripts occupy small space on the memory-limited PDA. Most importantly, the compact code size decreases the amount of work for code input which affects the efficiency of development on a PDA.

- Ruby is a powerful but easy-to-use language [4]. Ruby is used around the world for applications from text processing, XML applications, and general system administration to artificial intelligence, machine-learning research, and as an engine for exploratory mathematics [2]. It is currently the most popular scripting language in its native Japan.

- Ruby is an interpreted language. When prototyping and testing code, an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle. Ruby code can be modified on the fly [1] making Ruby perfect for rapid prototype development.

- Ruby is a relatively new language and no implementation on the Pocket PC platform exists. Thus, PocketRuby will be unique.

Portability

- Ruby is under open source development. Most of the Ruby source code can be modified and distributed freely under the GNU General Public License. A Ruby IDE for Pocket PC (released under the same license) will have a positive impact to the Ruby community.

- Existing Microsoft Windows ports of Ruby make the process of porting to the Pocket PC platform relatively easier.

Audience

- The Ruby community is growing. Ruby is currently the most popular scripting language in Japan. There has been repeated interest within the Ruby community on a PDA version of Ruby.

- The project of porting Ruby to Pocket PC was announced in June 2001. Since then, there have been significant interests expressed from the Ruby community.

Chapter 6

Architecture

In this chapter, the high-level system architecture of the PocketRuby Integrated Development Environment (PocketRuby IDE) is presented. The architecture is very general and can be used in other IDE systems. The high-level system design is independent of the programming language that the system is designed for. A great deal of effort has been put into designing and implementing the architecture. The architecture determines how the Ruby interpreter is connected to its graphical user interface (GUI). The details of the system architecture of the IDE will be discussed. In addition, the model-view-controller paradigm on which the design of the system is based will be explained.

6.1 Overview

The architectural view of the PocketRuby IDE is illustrated in Figure 6.1.1. The three components of this architecture are the language interpreter or compiler, the GUI, and the data exchange component.

A compiler is a program that converts another program from some source language (or programming language) to low level machine language. An interpreter is a program which executes other programs. The user interface (UI) mediates between the user and the system. It

handles user inputs (via keyboard, mouse, and other input devices) and translates those inputs into commands to the system. The UI also outputs information such as results from the system.

Each component in the system has a modular design. The GUI in Figure 6.1.1 is responsible for mediating between the human user and the language interpreter/compiler. It is connected to the language interpreter/compiler by means of an intermediate data exchange component. The data exchange component's purpose is to get data across from the interpreter to the GUI for display and for the GUI to supply information to the interpreter.

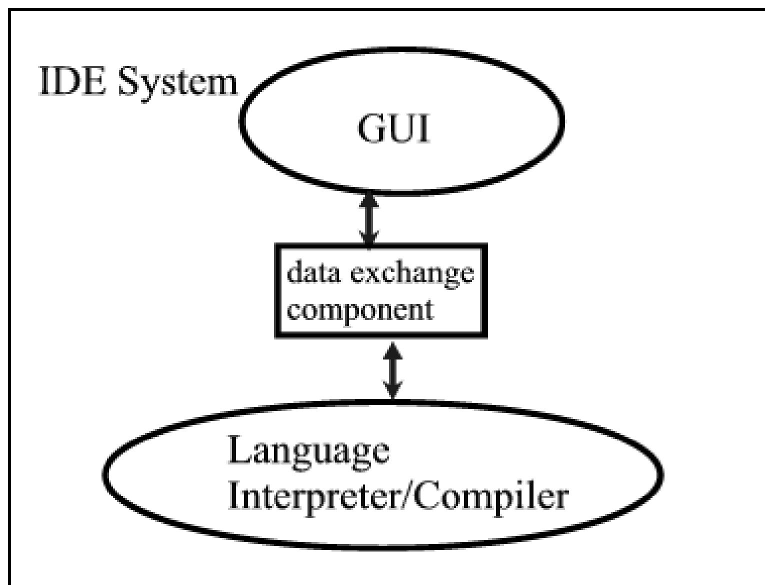

Figure 6.1.1: Overview of the system architecture: the GUI and the language interpreter/compiler interconnected by a data exchange component

6.2 Model-View-Controller Paradigm

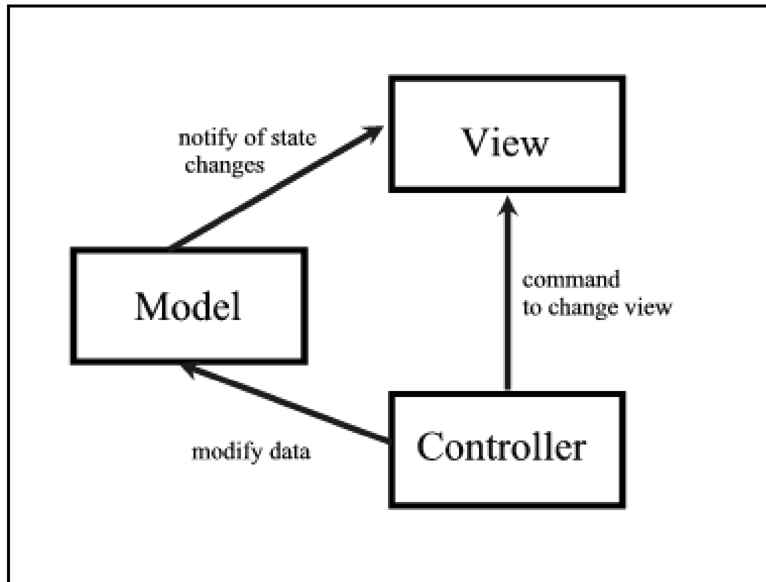

Figure 6.2.1: Model-View-Controller paradigm

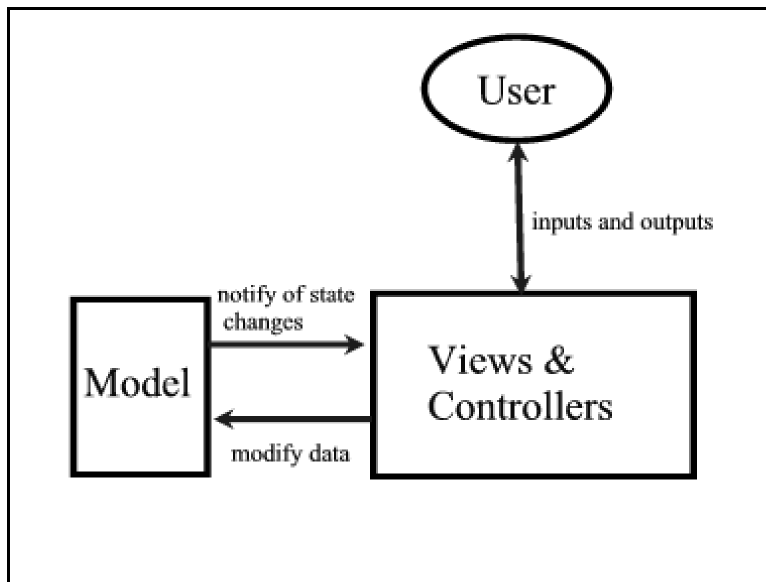

Figure 6.2.2: Model-View-Controller paradigm with view and controller as a pair

The model-view-controller paradigm (MVC), originated in Smalltalk-80, allows the construction of modular user interface that separates the interface (view) from the control elements (controller) and from the application code (model).

- The *model* manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
- A *controller* is an input component. It interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate.
- A *view* is an output component. It manages graphical and textual displays based on information from the model.

In practice, the view and controller often occur in a pair as most GUI components include both the view and the controller. Figure 6.2.2 illustrates this case.

One key idea of MVC is that the model assumes absolutely no responsibility on how results are presented in the view. This is important because MVC affects the reusability of application code and it increases maintainability by allowing easy additions or modifications of view. This paradigm allows the construction of modular user interface separate from the application code.

The MVC paradigm dictates some important design decisions of the PocketRuby IDE. Applying the MVC paradigm to the system in Figure 6.1.1, the model is the PocketRuby interpreter. The view and the controller make up the GUI. The data exchange component allows the controller to modify the model and the model to send results to the view.

Most GUI components, such as buttons, text boxes, menus, pop-up menus, combo boxes, and list boxes, include both view and controller elements. Therefore, the controller and view

occur in pairs. The controller-view relationship is supported internally by the GUI components. In Chapter 8, the GUI and its components will be discussed.

The PocketRuby interpreter notifies the GUI application of state changes within the model. The GUI application is responsible for deciding how to display that information visually on the PDA's screen. This supports the model-view relationship within the MVC paradigm.

Finally, the GUI application issues certain commands and data to the interpreter based on user inputs on the GUI. This supports the controller-model relationship.

6.3 System Properties

The architecture of the PocketRuby IDE system is based on the MVC paradigm described in Section 6.2. The following are some desirable properties of the IDE:

Property 1: Modular User Interface

The MVC paradigm allows modular construction of user interface. An essential feature of the architecture is that the user interface is interchangeable meaning that a GUI designer could create a new GUI to readily replace an existing GUI without any changes to other components of the system. The requirement for connecting the language interpreter to a GUI is that both must adhere to a communication protocol.

Property 2: Interchangeable language interpreter/compiler component

The IDE system is highly flexible. Not only is the GUI module interchangeable, the language module is interchangeable as long as certain requirement is met. In fact, the GUI module is capable of handling multiple interpreters. For example, a user can command the IDE to

switch to Perl mode from Python mode. In that case, the user will have the choose of language and will also be able to select the tools that are specific to the particular language chosen.

Property 3: Reusability of Modules

The modular design of the components allows components to be readily reused. The GUI module, the interpreter/compiler module, and the data exchange module are all reusable.

Property 4: Code Maintainability

As a result of the modularity of the system design, code is easier to maintain. This is crucial for open-source systems in which developers are encouraged to contribute to bug fixes and improvement to future versions of the system.

Chapter 7

PocketRuby

One technical contribution of this thesis is PocketRuby, the first Pocket PC port of the Ruby language interpreter. In this chapter, the design criteria of the PocketRuby interpreter will be explained. The techniques and challenges of porting the Ruby interpreter to the Pocket PC will be detailed.

The Pocket PC port of the Ruby interpreter as well as the IDE are named PocketRuby. To avoid confusion, within the scope of this chapter, PocketRuby refers to the PocketRuby interpreter, unless otherwise noted.

7.1 Ruby on PDAs

PocketRuby is the first Pocket PC port of the Ruby interpreter. While PocketRuby is a unique Pocket PC port, there is a Ruby port for Linux PDAs named *AgRuby*. *AgRuby* is included as pre-loaded software on the *Agenda VR3* Linux PDA. As with Ruby on the desktop, *AgRuby* uses a command line interface. PocketRuby is different from all other versions of Ruby in that it interacts with the user through a graphical user interface.

7.2 Design Criteria for the PocketRuby Interpreter

The PocketRuby interpreter must satisfy these important design criteria:

1. The PocketRuby interpreter should preserve essential core functionality of the Ruby language.
2. The interpreter should contain minimal modifications to the original Ruby source code. Changes and additions made should be fully documented.
3. The interpreter must comply with a set of communication specifications to establish and maintain interconnection with the user interface.
4. The interpreter must successfully run under the target Pocket PC processor architectures (for instance: ARM for Compaq iPAQ, SH3 for HP Jornada, and MIPS for Casio Cassiopeia).

PocketRuby need not behave with perfect resemblance to Ruby, but it should preserve the essential core functionality of the Ruby language. For the purpose of this thesis, it is acceptable that certain functionality of the language, particularly features that are operating system specific, are not fully implemented. However, the basic core functionality such as arrays, regular expression, arithmetic, class/method definitions, iterators, exceptions, to name a few, should be functional.

The second criterion is necessary because the more unnecessary changes to the source code, the less readable the source code becomes. Additionally, undocumented changes tend to make the source code less comprehensible. Since the source code of the PocketRuby project is

available as open source software to the Ruby community, changes should be clearly documented for developers to contribute to future versions of PocketRuby.

New stable releases of Ruby become available every few months. It will be desirable to incorporate the new features and bug fixes of new releases into future versions of PocketRuby. To prevent the potential danger of introducing new bugs into PocketRuby in the process, it is important to have a clear history of changes made in all previous versions of PocketRuby.

Finally, the last criterion states that the interpreter is useful only if it runs on the targeted architecture(s).

7.3 Porting Techniques and Technical Challenge

As explained in Section 5.3.1, PocketRuby is based on *ruby-mswin32*. Porting the mswin32 version of Ruby to the Pocket PC involves the following steps:

Step 1. Modifications to Makefile.

Step 2. Find a procedure, variable, or include directive preventing successful compilation and provide partial or temporary empty implementation. Recompile. Repeat Step 2 until the no compilation error occurs.

Step 3. Limited preliminary testing.

Step 4. Connect PocketRuby to graphical user interface.

Step 5. Extensive testing of the various functionality of PocketRuby.

Step 6. Implementation of missing libraries and functions.

The purpose of the first two steps is getting the interpreter code to compile under a specific Pocket PC architecture.

A Makefile is used in the compilation process because Ruby is traditionally compiled using the UNIX *make* utility. The UNIX *make* utility is a tool for organizing and facilitating the update of executables or other files which are built from one or more constituent files. On Windows platforms, Microsoft *nmake* is the analogous tool to the UNIX *make* utility.

The programmer specifies relationships between source, object, library and executable files in the Makefile for use by *make* or *nmake*. The Makefile defines the relationships among the constituent and target files of a project by listing the required files for each target file, and stating the shell commands that must operate on the required files to create or update the target(s). For example, the Makefile of PocketRuby can accommodate various Pocket PC architectures target (ARM, MIPS, SH3, among others) by specifying the shell command that invoke the appropriate C compiler for a target Pocket PC architecture.

The second step involves modifications to ms-win32 source code. The goal is to successfully compile the source into an executable for a target Pocket PC architecture. Since the mswin-32 version was originally compiled with Microsoft Visual C++, the compiler chosen for compilation of PocketRuby was Microsoft Embedded Visual C++, a programming development tool from the freely available Embedded Visual Tools 3.0 package. Embedded Visual C ++ is a software development tool for the Pocket PC operating system while Visual C++ is for Windows development. However, Embedded Visual C++ recognizes only a subset of the C libraries and functions Visual C++ recognizes. Therefore, the mswin-32 source code contains function calls not recognizable by the Embedded Visual C++ compiler.

The solution is to identify each unrecognizable procedure and subsequently provide implementation. For example, the functions for directory and file access, _write, _open, _close, _chmod are unrecognizable, causing errors during the compilation process. Providing complete and accurate implementation to missing procedures and libraries is time consuming. Due to lack of time, only empty body implementation are provided. As it was learned later, most of the basic core functionality behaved correctly on the Pocket PC with this approach.

After completion of the first two steps, PocketRuby should be installable on the Pocket PC. Preliminary testing was performed before further implementation was undertaken. The purpose was to ensure that the interpreter correctly execute at least simple programs. Testing is very limited in this stage as the lack of a user interface makes direct testing impossible. Ruby originally depends on a command language interface to interact with the user. Without a command line interface on the Pocket PC, PocketRuby cannot accept inputs or display outputs. Testing must be done by manually placing test code inside Ruby evaluation procedures.

Testing on the actual Pocket PC device is not preferable due to efficiency issues. Individual downloads to the device accumulate to significant amount of time. An alternative was testing on the Pocket PC emulator provided by the Embedded Visual Studio package.

After the initial testing, the fourth step involves connecting PocketRuby to the GUI. The design and implementation of the GUI will be discussed in the next chapter. The integration between the interpreter and the GUI will be explored in details in Chapter 9.

The final steps of the porting process involve extensive testing of the functionality of the PocketRuby interpreter and implementing the missing libraries and functions. Almost all testing was performed on the emulator. Complete implementation of libraries was not completed due to lack of time.

7.4 Evaluation of Capability and Limitation

An evaluation was conducted to show the degree to which the core functionality of the Ruby language was preserved in the final PocketRuby implementation.

Testing of PocketRuby was initially performed on the desktop Pocket PC emulator. Later on, PocketRuby was compiled for the ARM architecture and tested on a Compaq iPAQ. There was no noticeable difference in behavior between the emulator and the iPAQ other than speed. Due to lack of time, PocketRuby has not been tested on other architectures. However, the current version of PocketRuby is expected to run under all Pocket PC architectures (for example: MIPS, ARM, and SH3) supported by Embedded Visual C++.

PocketRuby was consistently shown to preserve most of the core functionality of the original Ruby interpreter after extensive testing was performed. However, there are many classes and methods in Ruby. A complete and exhaustive testing of each method is impossible in the duration of this project.

Most of PocketRuby's built-in classes have been shown to function properly. Table 7-1 lists many important built-in classes in PocketRuby and their functional status. It should be noted that certain classes do not currently function. For example, PocketRuby's file access methods in class *IO* and class *File* do not currently work. Attempts to open or write to files within a Ruby program will likely result in error messages or unexpected program behavior due to lack of actual implementation of some libraries and functions. This deficiency is fixable. However, the task of fixing all problems will take more time than allowed in the duration of this project.

Built-in Classes	**Functional Status**
Array	Functional
Dir	Not Functional
Exception	Functional
File	Not Functional
Fixnum	Functional
Float	Functional
Integer	Functional
IO	Not Functional
Object	Functional
Regexp	Functional
String	Functional
Thread	Functional
Time	Functional

Table 7-1: PocketRuby's built-in classes and their functional status

Chapter 8

The User Interface

Designing the graphical user interface (GUI) of the PocketRuby Integrated Development Environment is a very challenging problem by itself. PDAs are not designed for input intensive tasks such as software coding. Available PDA input technologies carry inherit limitations in speed and ease of input. A well-designed user interface for input intensive software application must address this serious input challenge. Besides input limitations, limited visual output area poses another significant challenge. The typical PDA screen offers a significantly lower screen resolution than typical desktop or laptop displays. This major limitation directly affects the quantity and quality of visual information simultaneously presented to the user. For example, the length and width of Ruby source code that can be displayed on the screen at a time are limited by the resolution of the screen. To overcome the problem, the GUI's integrated code editor employs horizontal and vertical scroll bars. The quantity of information displayed on the editor will be dependent on various factors such as the font size, the font type, the size of the text box, and so on. These factors in turn affect the legibility of output. Design decisions were made following user interface design principles and results from HCI research.

8.1 Designing the GUI to Address Challenges on the PDA

In the survey of existing IDEs in Section 4.2, it was concluded that GUIs of existing IDEs are generally inadequate. They lack features that address the input/output challenges with PDAs discussed in Section 2.2.

PocketRuby IDE GUI's main design goal is to overcome these challenges. As we will see, features of the IDE help with input speed and display of information on the small LCD screen.

8.2 Software Tools Used in the GUI Implementation

Microsoft Embedded Visual Basic (eVB) was the primary development tool used in the GUI implementation. EVB is part of the freely available Microsoft Embedded Visual Tools 3.0 software package. The language used by eVB is a subset of that used by the desktop version of Visual Basic. The eVB language is an interpreted language. EVB was chosen because it offers rapidly built GUI prototypes. Additionally, eVB allows the GUI designer to build GUIs with consistent look-and-feel across the Pocket PC platform. An alternative to using eVB is to build the GUI from the ground up using C/C++, which would require much more work than building from existing components available within eVB. Therefore, the decision was to use eVB to implement the GUI.

8.3 Overview of the Graphical User Interface of the PocketRuby IDE System

The PocketRuby IDE is the first Ruby development environment with a GUI. The GUI of the PocketRuby IDE is pictured in Figure 8.3.1 and 8.3.2. Figure 8.3.1 illustrates the execution

of a simple objected-oriented *Hello World* program. The left side shows the integrated code editor and the right side shows the result of the execution. As another example, Figure 8.3.2 depicts an execution of an iterative loop.

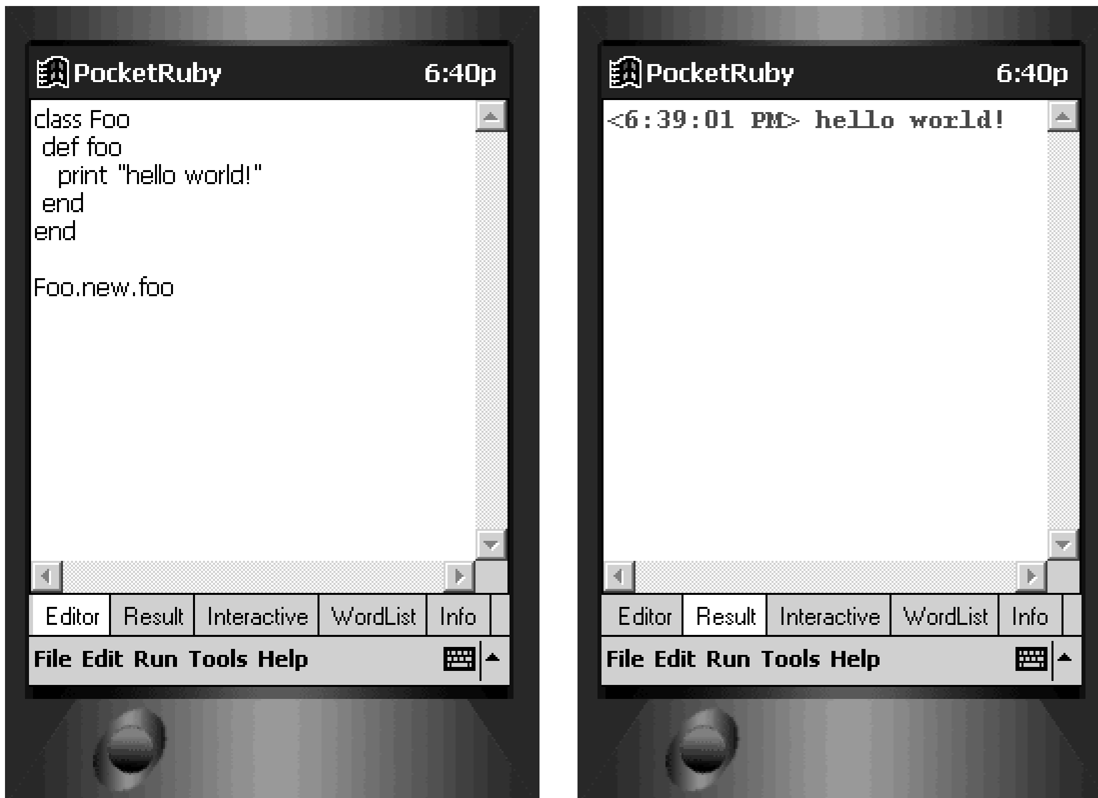

Figure 8.3.1: Execution of a Hello World program in Editor mode and corresponding result displayed on the Result tab.

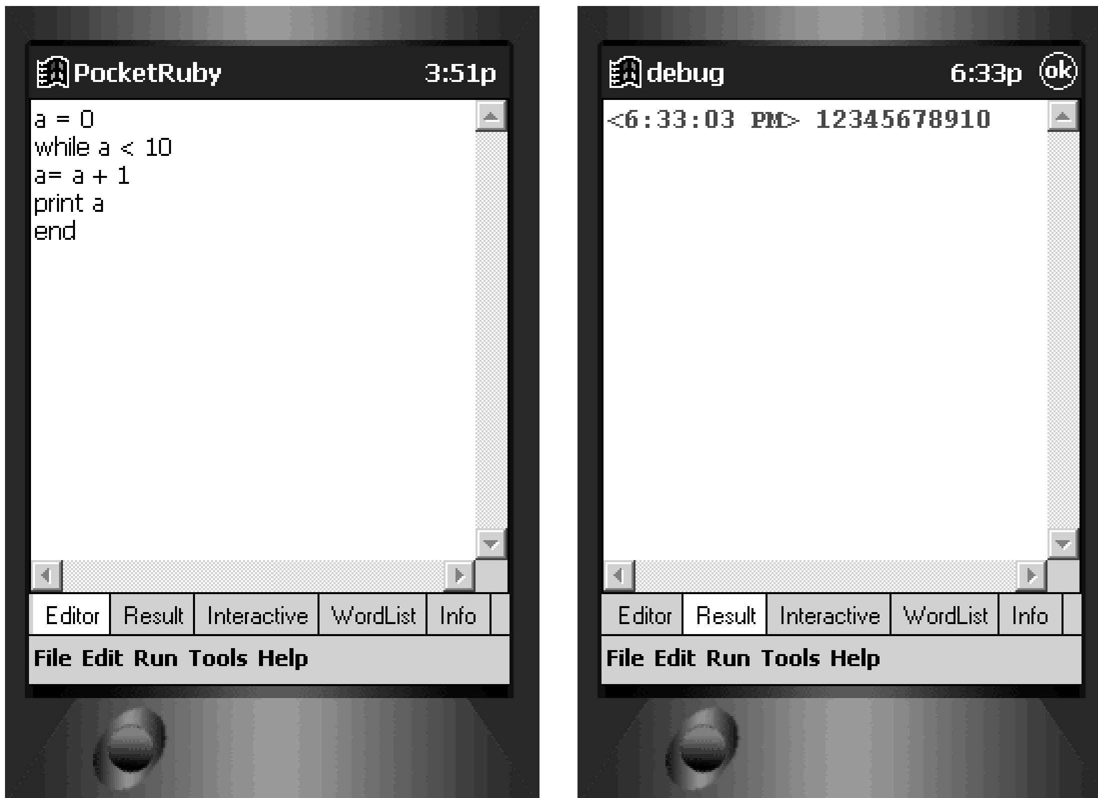

Figure 8.3.2: Execution of a loop in Editor mode and corresponding result displayed on the Result tab.

8.3.1 The Menu Bar

The GUI features a standard Pocket PC menu bar that allows the user quick and intuitive access to various user commands.

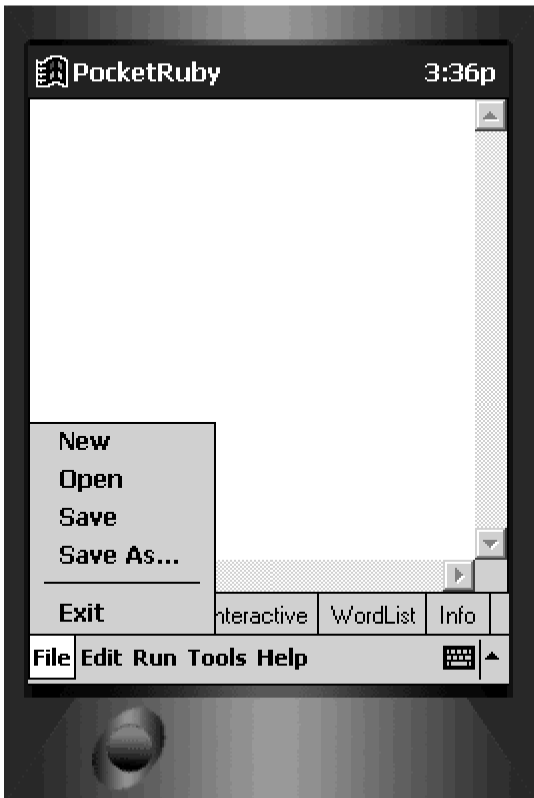

Figure 8.3.3: The IDE menu bar with the File menu activated.

The function of the menu bar is both to accept user commands and to present the user an organized view of the available commands. On the screen, the menu bar is the horizontal stripe located at the bottom of the GUI as depicted in Figure 8.3.3. For menu bars on a PDA, a bottom location is better than a top location (typical in desktop applications). The reason is the user's writing hand would naturally block the line of sight to the screen while accessing a top menu bar. Other than difference in appearance and location, the eVB menu bar behaves almost identically to a desktop menu bar.

The IDE menu contains menu categories such that user can intuitively find the appropriate command button. The IDE menu bar contains 5 menus (*File, Edit, Run, Tools, Help*)

representing different categories of commands. Each contains its respective menu command buttons. Sub-menus were not present since they might clutter the small screen display.

The *File* menu supports the following file-management commands for the integrated code editor: *New* for creating a new empty file, *Load* for opening an existing file, *Save* and *Save As...* for saving modified or newly created files. When loading a file, the user is presented with a directory browser to choose the file to be opened as an alternative to typing in the file name and location. The *Exit* command terminates the application.

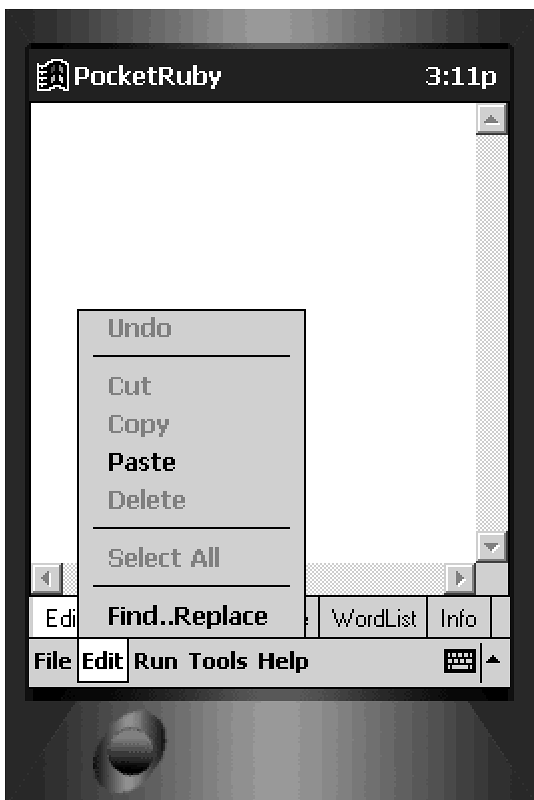

Figure 8.3.4: The IDE menu bar with the Edit menu activated.

The *Edit* menu (Figure 8.3.4) supports standard Clipboard operations (*Cut*, *Copy*, *Select All*, *Paste*, and *Delete*). Additionally, the *Undo* command allows the user to undo a previous command. Finally, the *Search and Replace* feature allows the user to:

- Search for a particular string throughout the current file
- Fast recursive replacement of all occurrences of a string throughout the file
- Slow word-by-word replacement: prompt the user the option to replace at each occurrence

The *Run* menu contains commands for execution of Ruby programs.

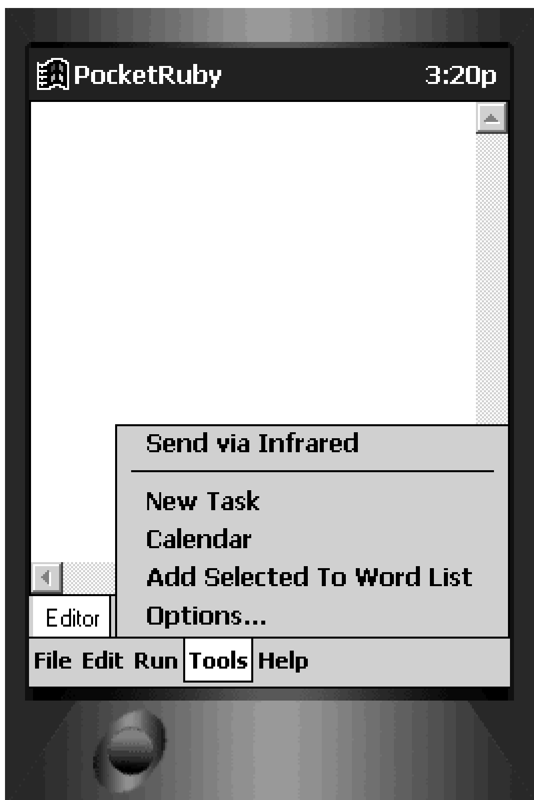

Figure 8.3.5: The IDE menu bar with the Tools menu activated.

The *Tools* menu (shown in Figure 8.3.5) contains the infrared communications command which allows two Pocket PC PDAs to send source code to each other. Most PDAs have built-in infrared transceivers. The IDE sends and receives program source files using the irDA (industry standard Infrared Data Association) protocol via Winsock ActiveX control. Sharing via IR is useful in collaborative work settings. This menu also contains tools designed to ease the user's tasks. For example, it allows direct access to the to-do list and calendar of the PDA to keep track of scheduling aspects of software projects. Integration with the personal information management (PIM) tools will be discussed fully in Section 8.6. The *Options...* button brings up an options screen for customization of the IDE, such as font size and font type in the code editor, among other options.

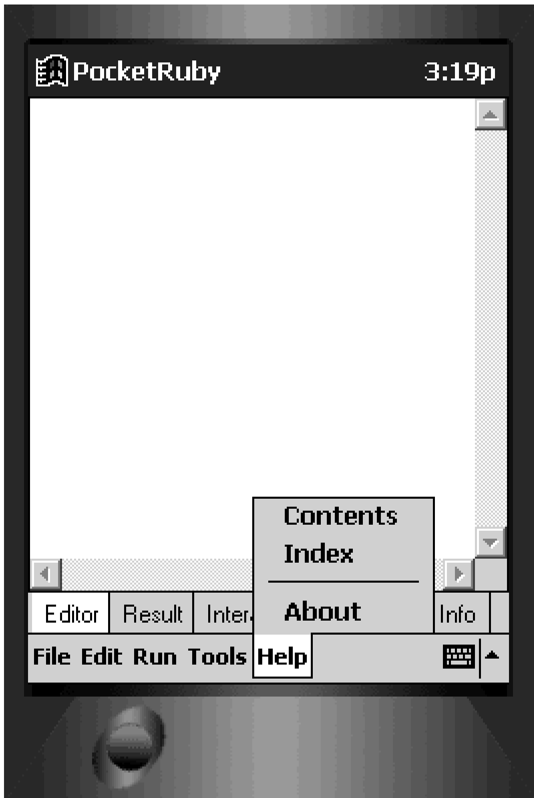

Figure 8.3.6: The IDE menu bar with the Help menu activated.

The *Help* menu (shown in Figure 8.3.6) provides PocketRuby IDE's help documentation and Ruby language reference/documentation. The help information is represented as HTML. HTML representation was chosen because hyperlinks allow easy navigation. Additionally, the HTML files can be readily processed and displayed by the built-in Pocket PC version of Microsoft Internet Explorer.

8.3.2 Editor Mode versus Interactive Mode

A unique feature of the PocketRuby IDE is the support of two different development modes: the Editor mode and the Interactive mode. Each mode addresses different programming needs. The Editor mode interprets a Ruby script in its entirety while interactive mode interprets Ruby code a line by line as the user enters each line. Figure 8.3.7 shows screen shots of the Editor mode and the Interactive mode.

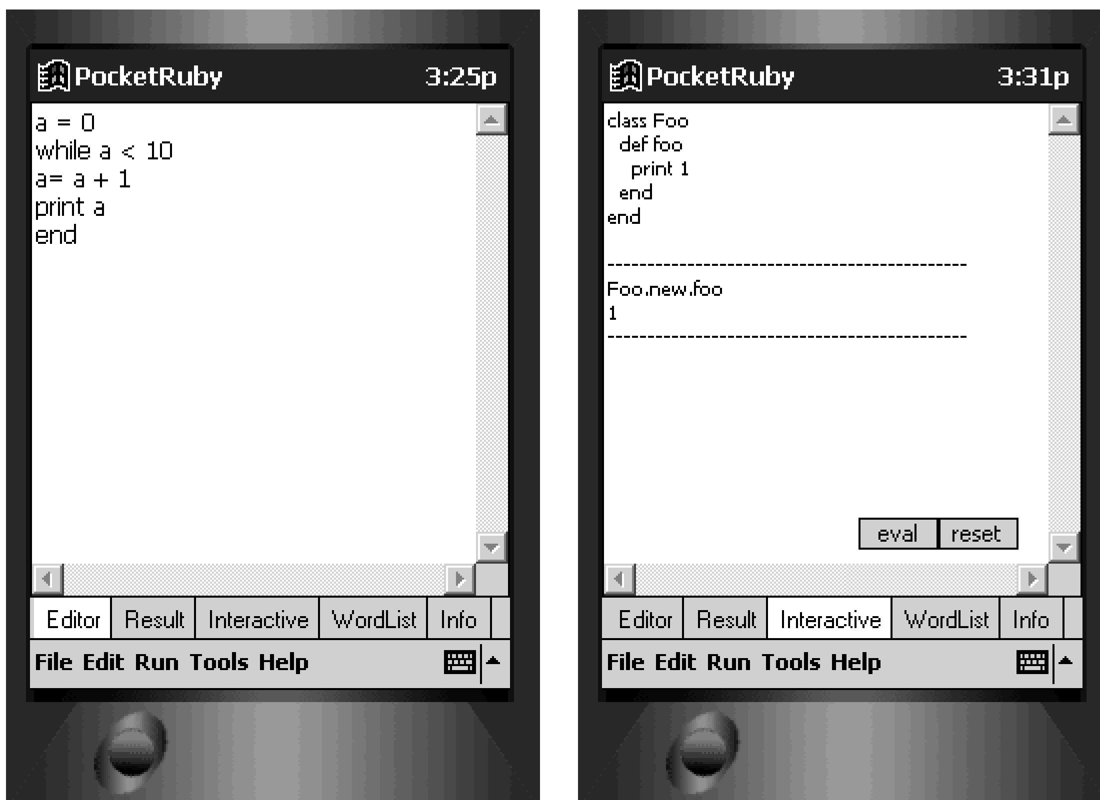

Figure 8.3.7: Comparison between Editor Mode and Interactive Mode

(1) Editor Mode

Editor mode is the default mode when the IDE starts up. The integrated code editor is active in its ready state. If the user has switched to other tabs such as the *Interactive* tab, tapping

on the *Editor* tab will reactivate Editor mode. When the Editor mode is activated, the integrated code editor occupies the bulk of the screen. The integrated code editor is the area in which code is displayed and where the programmer edits the source code files.

Integrated Code Editor

The IDE includes an integrated code editor for editing source files (The code editor also functions as a basic word processor. For example, it can be used to edit code documentation). The integrated code editor supports common file operations such as saving a file, loading a file, creating a new file, and closing a file. It supports common editing tools such as copy, cut, and paste. In addition to the basic functionality, the editor incorporates features that partially overcome the screen size limitation problem and input difficulty inherent with current handheld devices. These features will be discussed in Section 8.4 and Section 8.5.

The integrated code editor has built-in safeguards against accidental lost of unsaved or modified files due to user negligence. The IDE issues on-screen pop-up alert messages when there is unsaved data and that data might be discarded. For example, when the user terminates the IDE or opens another file before saving the current modified file, a pop-up alert message warns the user by prompting the user to choose among a set of specific actions which lets the user to save or discard changes. See Figure 8.3.8.

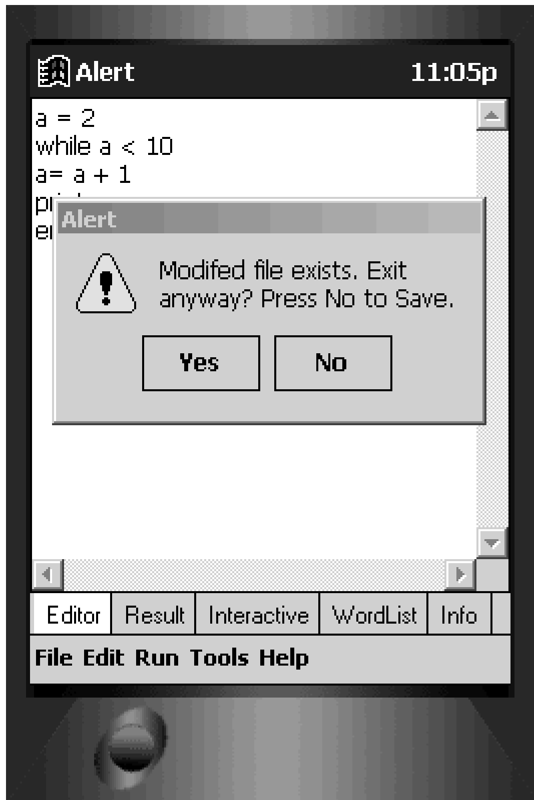

Figure 8.3.8: A pop-up alert dialogue box appears when the user exits the system if the data on the code editor has not be saved.

(2) Interactive Mode

Tapping on the *Interactive* tab activates Interactive mode. See Figure 8.3.7. The design of Interactive mode in the PocketRuby IDE was inspired by Interactive Ruby Shell (IRB) discussed in Section 5.4. Unfortunately, there is no simple way to port IRB to the Pocket PC as IRB relies on registering user keystrokes in real-time and there is no keyboard on the Pocket PC. Interactive mode was made to simulate the behaviors of IRB to provide similar benefits as IRB.

Using interactive mode presents certain advantages over editor mode. Interactive mode allows faster prototyping. The development process could be accelerated by writing and testing code within interactive mode concurrently.

8.4 Addressing the PDA Input Limitation Problem

As discussed in Section 2.2, slow input speed and limitations in accuracy are significant challenges for typical PDAs with current input technologies. Handwriting recognition is the primary means of text input for the Pocket PC PDAs. Certain features of the IDE have been designed to alleviate the problem by reducing the amount of user text inputs required in writing code.

8.4.1 Frequently-Used Word List

Usable in both programming modes, a frequently-used word list context menu serves as a shortcut to frequently-used words. Context menus (also known as pop-up menus) are typically invoked when the user right-clicks on the mouse (or other pointing devices) within a desktop Windows application. On the Pocket PC, standard context menus are usually invoked by a technique known as tap 'n hold: holding the stylus on the screen for an extended period of time (typically two to five seconds). PocketRuby IDE's GUI displays frequently-used words as items on the context menu. After the user has made a selection on the context menu, the selection is automatically inserted at the current cursor position as illustrated in Figure 8.4.1. The list is user customizable. Activating the *WordList* tab allows the user to add or remove words or set of related keywords sequence (such as *if…else… end*) from the list (as shown in Figure 8.4.2). Each time a user makes an addition or deletion from the list, the new list is permanently saved for future use.

Alternatively, the user can add a word to the context menu with a shortcut on the menu bar. The user first selects any word on the code editor and then taps on the *Add Selected to Word List* menu command under the *Tools* menu (Refer back to Figure 8.3.5). This is very fast method to add a word to the list.

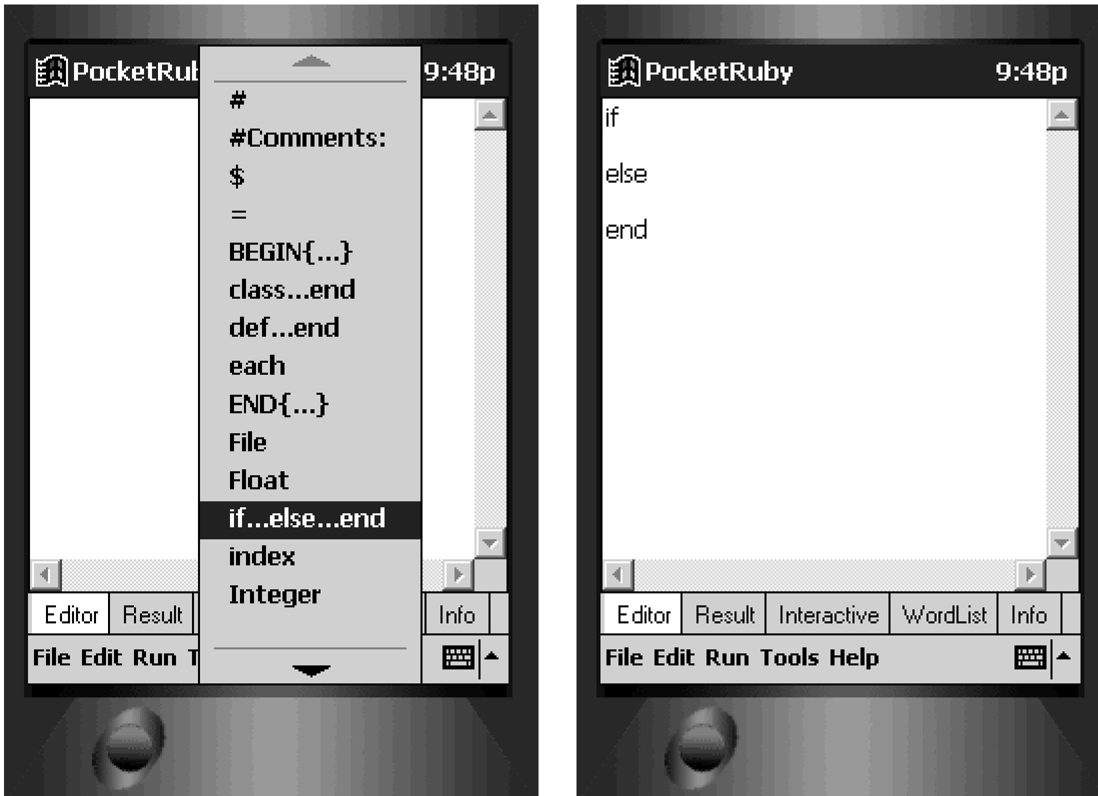

Figure 8.4.1: An example of using the frequently-used word list context menu

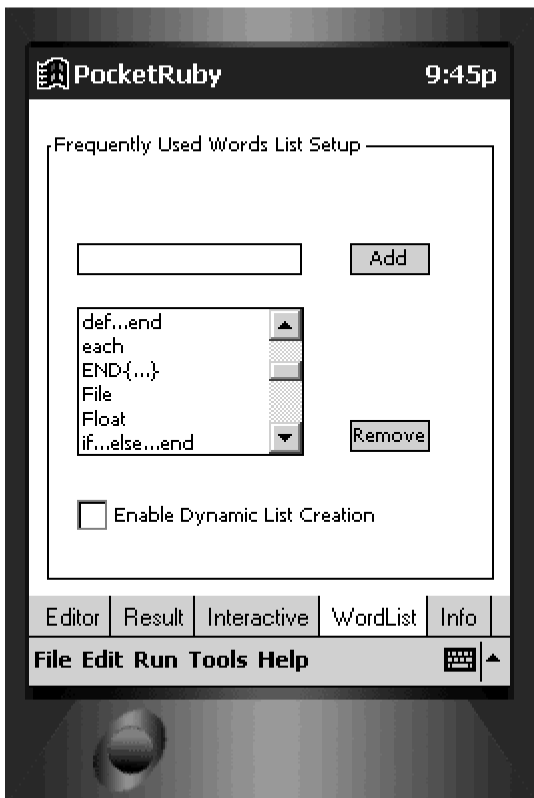

Figure 8.4.2: The WordList tab

While a statically-defined word list saves the user a lot of time in typing frequently used words, the IDE supports the dynamic creation of word list to further save the time of the user. The IDE checks the current file for newly declared variables and most frequent words within the file and dynamically inserts those words in the context menu. Dynamic creation of list includes words that the user is likely to use again based of input history.

In essence, the frequently-used word list is a GUI feature that allow the user enter words without relying on handwriting recognition. Using the context menu is a fast and accurate means of entering words. Although no quantitative studies have been conducted, this feature should

considerably speed up the code input process. Time saved from text input results in increased speed of development.

8.5 Addressing the PDA Output Limitation Problem

The output limitation of PDAs was discussed in Section 2.2. As PocketRuby IDE contains many features, the GUI determines the layout of those features and how the user accesses those features. Guided by the design principles of naturalness and relevance discussed in Section 3.2, the GUI should satisfy the following criteria:

- The GUI should provide an intuitive layout so that the user can easily and quickly get to the appropriate place to perform a task.
- The presentation of information should maximize the use of the small display area. The information should be kept concise and relevant.

The GUI layout characterized by the 5-section tab control of PocketRuby IDE (shown in Figure 8.5.1) satisfies the above criteria. A tab control is analogous to the dividers in a notebook or labels in a file cabinet. To access and view the content area under a particular tab, the user simply taps on the corresponding tab label. The five tab labels are *Editor*, *Result*, *Interactive*, *Wordlist*, and *Info*, in that order.

The tab control allows switching between different views (such as Editor mode and Interactive mode) with one tap of the stylus. When the result of a computation is ready for display, the GUI switches to the *Result* tab automatically. The *WordList* tab is shown on Figure 8.4.1 and its function was explained in a previous section. The use of the *Info* tab will be

explained in Section 8.6. This notebook dividers style layout is intuitive and the user should not experience many problems finding the correct tab to perform a task.

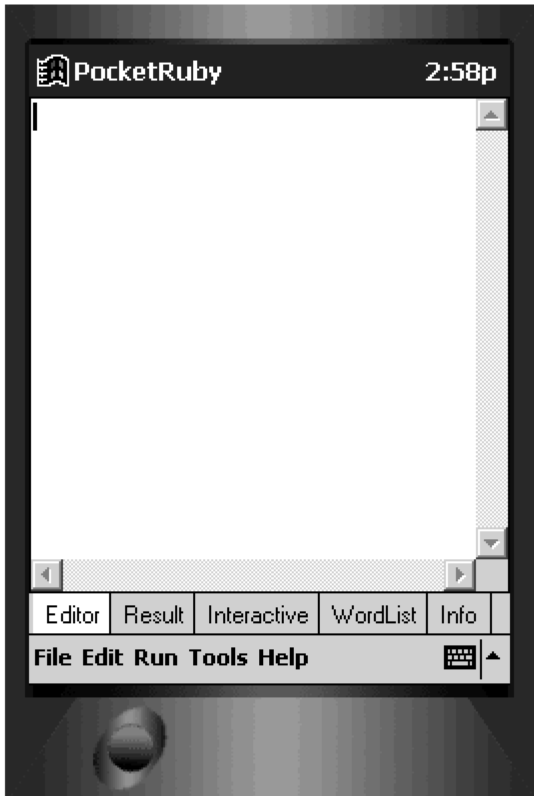

Figure 8.5.1: The tab control located on top of the menu bar

An alternative design approach is adding an additional menu on the menu bar. In that case, the user is required to make the appropriate selection on the menu. One disadvantage of this approach is that it takes more work from the user to switch between, say, Interactive mode and Editor mode. The process would require at least two taps from the user: one for bringing up the menu, the other for choosing the menu item. Compared to the previous design, switching

between different views takes twice as much work. Not to mention that the menu bar already has five menus each containing a number of menu items. Adding an additional menu might clutter up the menu bar possibly making it more difficult for the user to find a particular command. One advantage of this approach is that the screen space that would have been occupied by the tab control is freed up.

The first design is more effective in overcoming the output limitation problem. Therefore, it was chosen as the final design.

8.6 Integration with Personal Information Management Tools

PDAs were originally invented for contact, time, and task management, as well as for note taking. The PocketRuby IDE is designed to take advantage of the capabilities of PDA personal management tools. The time and task management aspects of a programming project can be managed through personal information management tools such as the calendar, to-do list, and address book. For example, a programmer issues a command in the IDE that scans through the headers of all source files in a large software project. It extracts the name and the e-mail address of the author, if present, in each source file and adds new entries to the contact list if necessary. Besides the name and e-mail, information such as projects the person has been involved in and what his expertise are can also be gathered into the address book. The user for the IDE can easily access the contact information of the entire team working on the project. Or the user can issue commands in the IDE to call the contact list's search function to look for programmers with certain expertise.

The implementation makes use of the built-in embedded Ruby documentation format. The original motivation of embedded documentation was code documentation. Using Ruby's

embedded documentation format, a utility suite named *rdtool* converts this documentation into a variety of output formats. Our implementation does not use *rdtool* but a simple parser was implemented which will be discussed later.

Embedded documentation is any block of text that starts with the *=begin* tag and ends with the *=end* tag. The block contains various other tags and associated text. Embedded documentation is a special form of programming comments. Any embedded documentation is striped off before the code is passed to the PocketRuby interpreter.

The tags used by the IDE have self-explanatory names: *=Lastname* -- the last name of the author, *=Firstname* -- the first name of the author, *=Email* -- the e-mail address of the author, *=Copy* -- the copyright or licensing information, *=Version* -- the version number, *=Note* -- any additional information, and *=Lastsaved* -- system-generated date/time information.

Within the GUI, a parser scans through any embedded documentation and extracts information under each tag listed above. The extracted content is organized and displayed when the *Info* tab is activated as shown in Figure 8.6.1.

An advantage of using embedded documentation is extensibility. New tags can be easily created. It is also straightforward to update the parser so that the new tags are accounted for.

PocketRuby has access to the address book of Pocket Outlook, a Pocket PC personal information management application. Entries in the address book can be directly manipulated through the Pocket Outlook API. As shown in Figure 8.6.2, the IDE prompted the user before creating a new entry in the address book after a lookup search for the author in the Pocket Outlook address book returned no match. Figure 8.6.3 shows the system-generated entry in the Pocket Outlook address book.

Table 8-1 shows embedded documentation tags in PocketRuby IDE and associated entries in the Pocket Outlook address book.

Embedded Documentation Tag Names	**Pocket Outlook Database Fields**
=Lastname	Last name field in the address book entry
=Firstname	First name field in the address book entry
=Email	E-mail field in the address book entry
=Note	Note section in the address book entry

Table 8-1: Association between tag names with Pocket Outlook database fields.

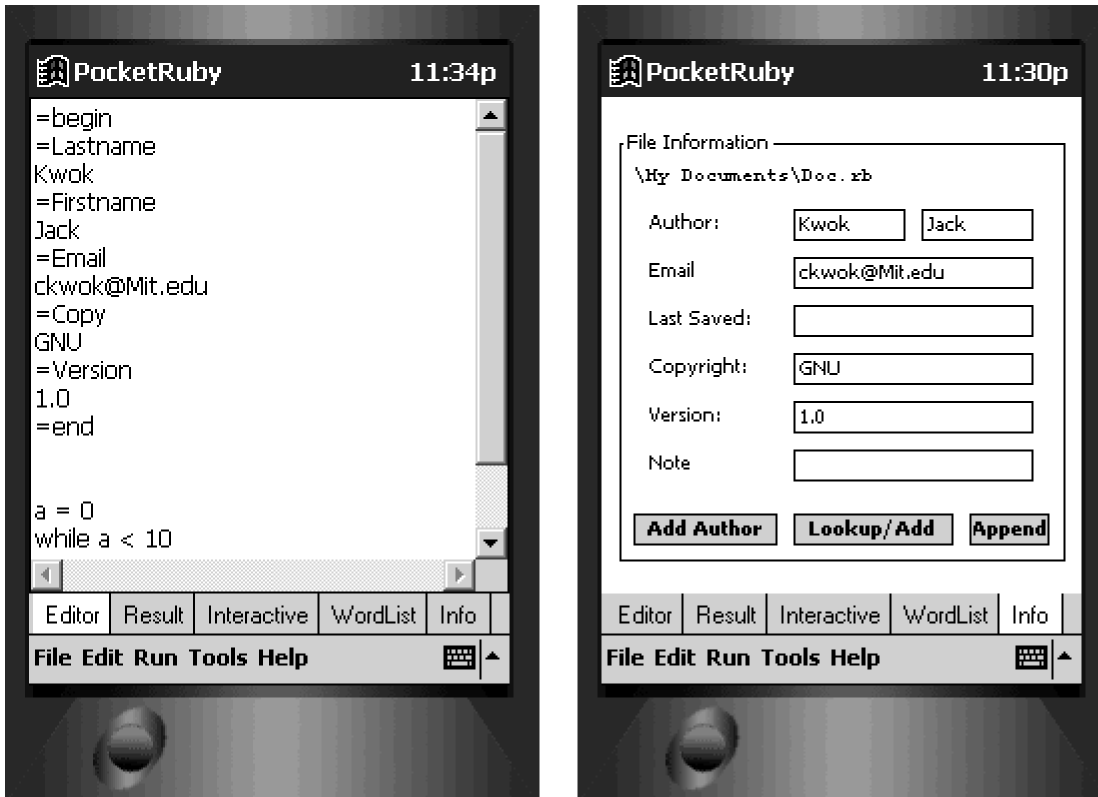

Figure 8.6.1: Ruby embedded documentation and corresponding data fields displayed under the Info tab

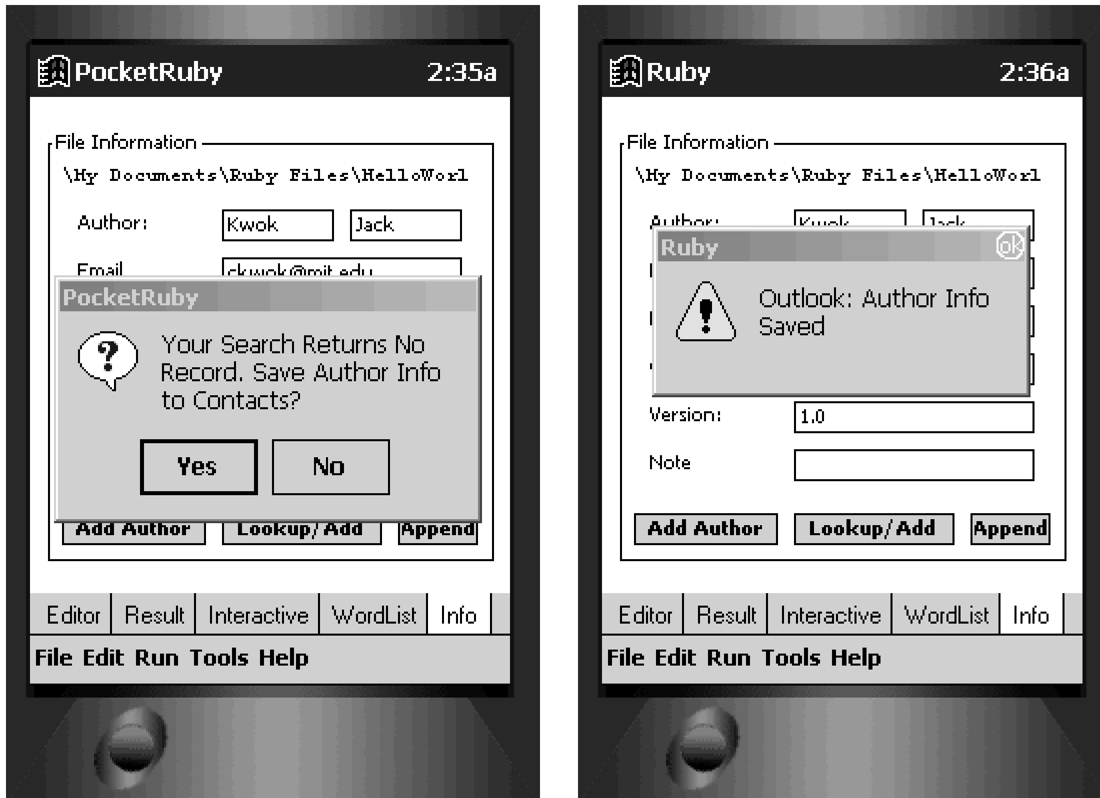

Figure 8.6.2: Adding a new contact to Pocket Outlook as an integrated feature of the IDE

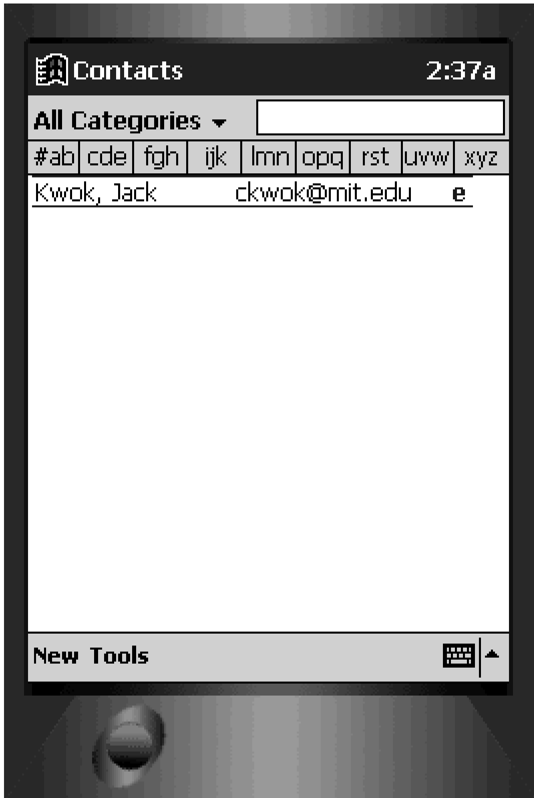

Figure 8.6.3: New entry added to Pocket Outlook address book

8.7 An Informal Evaluation

Using the principles of user interface designed outlined in Section 3.2, PocketRuby IDE GUI is evaluated.

- *Naturalness*

The PocketRuby IDE GUI is mostly self-explanatory. Advanced features consult the help documentation. It should be apparent to the user how the basic features of the IDE work simply by using them.

- *Consistency*

The GUI elements (scroll bars, menu bar, combo boxes, and so on) are all from Embedded Visual Basic, which provide a consistent "look and feel" across many common Pocket PC applications. The menu bar selection system is set up so that it is similar to typical menu bars in other applications.

- *Relevance*

Inactive menu buttons are grayed out to signify that the corresponding command is not currently available at the time. For instance, "Paste" is grayed out when there is nothing on the operating system's Clipboard. The user does not have time to read unnecessary materials so the GUI's on-screen information is short and relevant.

- *Supportiveness*

The IDE help system provide instructions on using the various features of the IDE. Additionally, it provides documentation of the Ruby language.

Another support feature is the IDE's safeguards against accidental lose of files. For example, pop-up dialog boxes warn the user of unsaved data when exiting the IDE application.

- *Flexibility*

Currently, the IDE supports a certain degree of customizations to accommodate the preferences of different users. For example, the IDE could allow adjustments to font size and

font type within the integrated text editor. In addition, the frequently-used word list is user customizable.

The IDE was designed following the principles of user interface design. Unfortunately, no formal evaluation with a user testing group was undertaken on the GUI because of limited time and resources.

Chapter 9

Integration

The main technical contribution of this thesis is the Integrated Development Environment for Ruby on the Pocket PC. In Chapter 7, issues related to the PocketRuby interpreter were discussed. An important issue was the PocketRuby interpreter by itself was not readily usable on the Pocket PC because of the absence of a suitable command line interface (such as a DOS prompt or UNIX prompt) on the Pocket PC. A graphical user interface that exposes the functionality of the PocketRuby interpreter was designed and implemented. In Chapter 8, we showed that the graphical user interface of the PocketRuby IDE eliminated the need for a command line interface and at the same time provided a variety of features that greatly eased the software development process on a PDA device. In this chapter, the implementation techniques that integrate the PocketRuby interpreter with the GUI will be explained. A great deal of effort has been put into designing and implementing the interconnection that connects the PocketRuby interpreter to a user interface. AppPipeDLL, a program designed to enable communication between two Pocket PC applications, will be explained in detail. In order for two applications to understand each other's messages, a protocol is necessary. The design of the protocol will be discussed. Some interesting programming techniques in the implementation will also be explained.

9.1 Communication Between PocketRuby Interpreter and the User Interface

The MVC paradigm discussed in Section 6.2 affects the design of the interconnection between the GUI and the interpreter. To support the model-view relationship in the MVC paradigm, the PocketRuby interpreter must be able to somehow notify the user interface of its internal state changes. To support the controller-model relationship, the GUI must be able to modify the PocketRuby interpreter.

9.2 The Mechanism of AppPipeDLL

AppPipeDLL is a small program written in C++ that establishes 2-way or 1-way communication between two Pocket PC applications. AppPipeDLL supplies a "data pipe" between the applications. The data are text strings composed of ASCII characters.

AppPipeDLL was originally created for the PocketRuby IDE system. However, AppPipeDLL is generally usable with any two Pocket PC applications.

The system depicted in Figure 6.1.1 contains a data exchange component connecting the UI and the language interpreter. AppPipeDLL is the core of that data exchange component.

AppPipeDLL is implemented as a dynamic linked library (DLL). A DLL is a collection of small programs, any of which can be called when needed by a larger program. The advantage of using DLLs is that, because they don't get loaded into random access memory together with the main program, space is saved in RAM. When a DLL is needed, it is loaded and run. DLL files are dynamically linked with the program that uses them during program execution rather than being compiled with the main program.

AppPipeDLL relies on the Windows/PocketPC Clipboard to allow one application to pass a text message to another application.

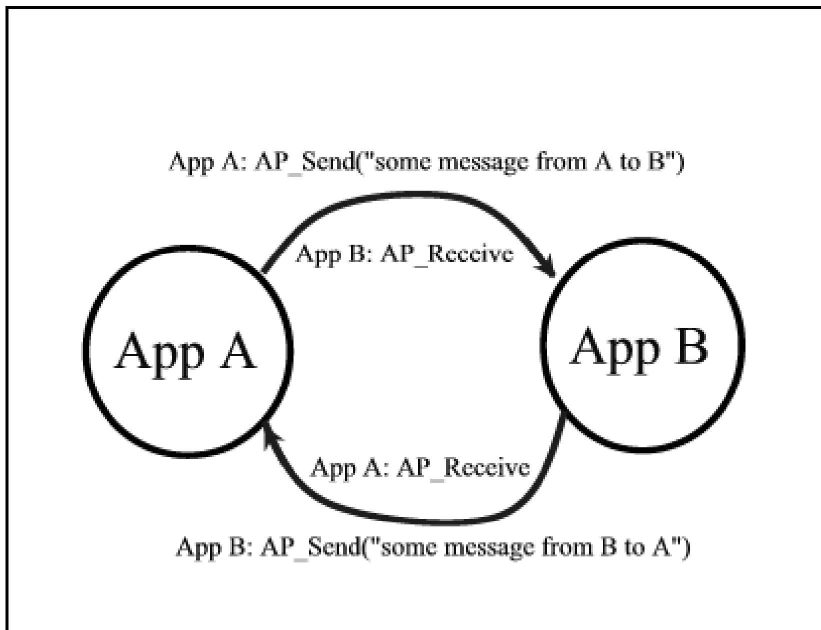

Figure 9.2.1: A high-level view of two applications exchanging data using AppPipeDLL's function calls

Figure 9.2.1 illustrates the process data are exchanged between two applications using AppPipeDLL. Application A and B exchange data by sending/receiving text messages. First, App A, the sender, calls AP_Send("<some message from A to B>"). This results in the message being copied by AppPipeDLL to the Clipboard. When a string of text is copied from within an application, a low-level window event called WM_COPY is fired by the application to an edit control to copy the current selection to the Clipboard. The WM_COPY event is fired and App B, the receiver, registers the event. When App B registers a WM_COPY event, it calls AP_Receive. AppPipeDLL retrieves the message from the Clipboard and AP_Receive returns the message string. To send message in the opposite direction, App A and App B switch their roles as sender and receiver.

The main requirement for using AppPipeDLL is that the receiver can listen to the WM_COPY event. While implementing the PocketRuby interpreter, event listeners were added to the interpreter for listening to the WM_COPY event.

9.2.1 The Messaging Protocol

A discussion of the actual messages between the GUI and the interpreter is presented.

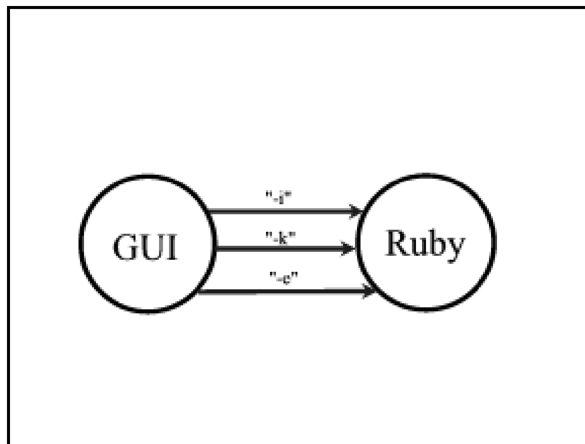

Figure 9.2.2: The GUI sending command messages to the PocketRuby interpreter

The following are messages the GUI sends to the PocketRuby interpreter via AppPipeDLL:

Command Messages

- Interpreter termination message ('-k') commands the interpreter to terminate itself. (Figure 9.2.2)
- Interactive mode activation message ('-i') commands PocketRuby to switch to Interactive mode if it is in Editor mode.

- Interactive evaluation command ('-e') signals that there is a Ruby statement waiting to be executed within Interactive mode.

Other Messages

- Locations of Ruby source files for evaluation:

  The absolute path of the Ruby program file to be processed (within Editor Mode)

  The absolute path of the file with Ruby code to be executed interactively (within Interactive Mode)

The GUI sends Ruby the full path location of the Ruby program file for the first case. For the second case, the GUI sends the full path location of the file containing the code to be evaluated interactively.

On the other hand, the GUI component cannot directly receive AppPipeDLL messages from the PocketRuby interpreter component. The reason is Embedded Visual Basic programs do not have built-in events listeners for the WM_COPY event, which is a requirement for AppPipeDLL. A solution to this problem is reached by having the GUI periodically checks a set of plain-text files where PocketRuby can record notification messages to. These messages notify the GUI of state changes within PocketRuby. When the system checks for messages approximately every half a second, the overhead cost of this periodical check is negligible. The latency from the time the data is written to the time the data is read is at most half a second.

Notification messages the interpreter sends to the GUI:

- Notification of successful initialization of PocketRuby interpreter.
- Notification that program execution was successful and results and/or error messages have been written (or updated) to a set of special files.

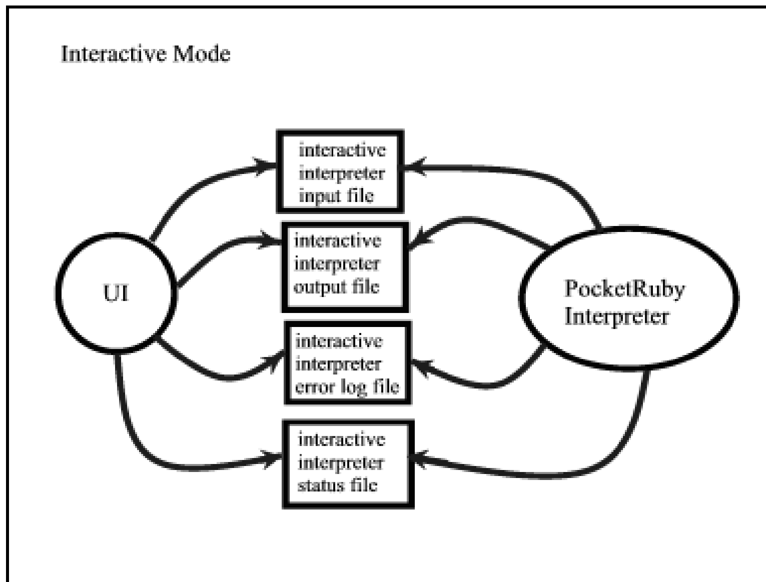

Figure 9.2.3: Data Exchange within PocketRuby IDE under Interactive Mode

Figure 9.2.3 depicts the flow of data in the system while the IDE is in Interactive mode. When the IDE is in Interactive mode, it is ready to evaluate Ruby statements interactively. The process is as follows. First, the UI copies the Ruby code from the UI to the *interactive interpreter input file*. Then the PocketRuby interpreter reads the Ruby code from the *interactive interpreter input file*, evaluates the Ruby code, and writes the results to the *interactive interpreter output file*. If there is any error, error statements are written to the *interactive interpreter error log file*. PocketRuby signals the completion of each evaluation by writing a special notification message to the *interactive interpreter status file*. The UI checks the *interactive interpreter status file* periodically. Upon reading the notification message, the UI reads from the *interactive interpreter output file* and *interactive interpreter error log file*. Finally, the results and error messages are displayed on the UI.

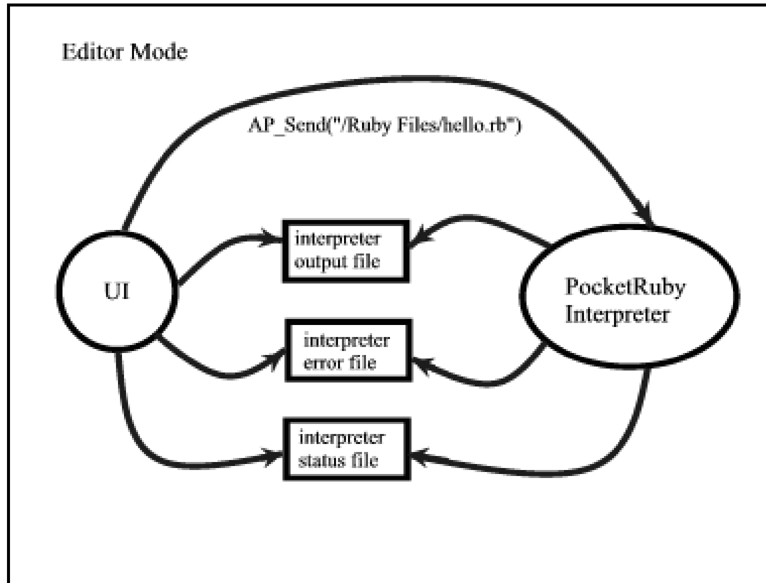

Figure 9.2.4: Data Exchange within PocketRuby IDE under Editor Mode

Figure 9.2.4 shows the interconnection mechanism of the PocketRuby IDE system under editor mode. Since the GUI implemented in eVB cannot directly receive AppPipeDLL messages, PocketRuby notifies the GUI of state changes by writing data to a special set of plain-text files. When the PocketRuby interpreter has completed an evaluation, the interpreter writes the results to the *interpreter output file* and any error message to the *interpreter error file*. To notify the GUI that new data have been written to *interpreter output file* and/or *interpreter error file*, PocketRuby writes a special notification message to the *interpreter status file*. The UI, upon receiving notification that PocketRuby interpreter has written data to those files, reads the data from the *interpreter output file* and *interpreter error file*.

9.3 Alternatives to AppPipeDLL

AppPipeDLL is certainly not the only method to enable communication between two Pocket PC applications. The following are alternatives that can accomplish the task of communications. Their shortcoming(s) are explained:

- Using TCP/IP -- Communication based on the TCP/IP protocol could be used to connect the IDE and Ruby by simulating them as a client-server pair. The overhead cost of integration effort and time is very high. Thus, it is not suitable for our purpose.
- Using commercial ActiveX controls -- There are closed source commercial applications that enable programs to communicate using Microsoft's ActiveX Control Technologies. Incorporating closed source commercial products into our open source project in unacceptable.
- Integrated Ruby editor written in C/C++ -- Since Ruby is completely written in C, it is possible and straightforward to add an editor to it. While this approach is feasible, the problem lies in that C is not an optimal language for implementing graphical user interfaces.

Chapter 10

Conclusion

A powerful PDA-hosted programming environment is an invaluable mobile software development tool that serves the needs in various areas including technical support, IT consulting, and education settings, to name a few. In this thesis, PocketRuby Integrated Development Environment has demonstrated its capability as a software development tool. In this chapter, a summary of work and future directions will be presented.

10.1 Summary of Work

The technical contribution is the PocketRuby Integrated Development Environment. PocketRuby IDE brings a powerful language to the Pocket PC. The PDA IDE allows the programmer to develop Ruby programs directly on the PDA. The characteristics of this innovative PDA IDE include:

- Applying research results in the area of human-computer interaction with the PDA to guide the design of the GUI for the IDE. The GUI follows principles of user interface design.
- GUI addressing the challenges and limitations of PDAs. GUI overcomes the slow input of PDA and small display screen with innovative GUI features and layout.
- Small memory footprints. The PocketRuby IDE installation consumes only 0.79 megabytes, which is a small share on a 32 or 64 megabyte Pocket PC. Table 10-1 shows the installation

size of each component of the IDE system. An experiment was conducted to measure the amount of additional memory required to run the IDE. First, all running programs are terminated and a snapshot of the total amount of program memory usage is taken with the Pocket PC's (under the *Settings/Systems* menu) built-in memory utility. Then, the IDE is started and a snapshot of the total amount of memory usage is taken again. The difference of the two memory usage should be roughly the amount of memory allocated to the IDE. The second column of Table 10-2 shows memory allocation to the IDE with no interpreter active. The third column shows memory allocation with the interactive interpreter running. The experiment was repeated a few times and the average memory allocation was around 2 or 2.5 megabytes depending on whether the interpreter is running. Ruby scripts are not being executed when these measurements are taken because the total memory usage will be affected by the additional amount of memory a Ruby script needs. The low memory requirement of the IDE is an advantage on a PDA with limited memory.

Module of the IDE	**Storage memory allocation (in megabytes)**
PocketRuby interpreter	0.63
GUI (eVB program)	0.15
AppPipeDLL.dll	0.01
Total	**0.79**

Table 10-1: Installation size of the PocketRuby IDE

Trial	Memory Allocated (in megabytes) for IDE when interpreter has not been invoked.	Memory Allocated (in megabytes) for IDE with interactive interpreter running.
I	1.91	2.35
II	1.81	2.48
III	2	2.51
Average	**1.91**	**2.45**

Table 10-2: Memory Allocated for the IDE running on an iPAQ Pocket PC

- A system architecture based on the MVC paradigm. Some properties resulting from the paradigm include: modular user interface, interchangeable interpreter, and code maintainability.
- Novel approach to integrate the personal information management tools of PDAs with the IDE. The IDE takes advantage of the personal information management capabilities of PDAs. For example, the IDE allows direct access to the to-do list and calendar of the PDA to keep track of scheduling aspects of software projects. In addition, the IDE uses the existing embedded Ruby documentation format to store personal information and other relevant information within the Ruby file. It then parses the documentation and extract the contents. Information such as name and e-mail of the author can be added as entries in Pocket Outlook's address book. In additions, he user can search the Pocket Outlook database within the IDE.

10.2 Future Research Directions

Human-computer interaction with PDA devices is a new discipline. Future research studies in the area will help the development of better graphical user interfaces not only for PDA IDEs but also for PDA applications in general.

Future research could be conducted to study speech recognition's potential role in the interaction between a PDA integrated development environment and its user. For example, an integrated speech activation menu system could supplement the normal menu bar by translating the user's verbal commands into system actions. Speech recognition might help entering code and even replace the stylus. Speech recognition algorithms could take advantage of the simpler syntax of a computer programming language as opposed to the English language.

HCI is a constantly changing field. The designer might need to take into considerations the new results from HCI research before designing the user interface.

10.3 Future Directions of PocketRuby IDE

PocketRuby has much potential for growth, as it is open source and extensible IDE. The PocketRuby interpreter and IDE are released under the same GNU General Public License (GPL). The complete source code of PocketRuby, the PocketRuby IDE, and their respective documentation is freely available for copying and modification under the terms and conditions of GPL. The source code and binaries of PocketRuby interpreter and the PocketRuby IDE currently reside in the official Ruby concurrent version system (CVS) repositories.

Some recommendations for future development of the IDE are as follows:

- Support for multiple language interpreters/compilers within the same IDE to create a universal PDA IDE. The PocketRuby IDE can be readily made to support other language interpreters.

The architecture of the IDE allows replacing the PocketRuby interpreter with any language interpreter as long as certain requirements are met (for using AppPipeDLL). A future version of the IDE could be extended to support multiple language interpreters and compilers.

- Explore new GUI features and tools. New features and tools can be readily integrated into the system as the IDE is open source and extensible. Explore new features and invent new tools that would improve the usability of the IDE and improve the ease of development on the PDA. In additions, the IDE would benefit from formal usability testing and evaluation.
- Explore completely new GUIs. The GUI is replaceable module in the IDE architecture so it is possible to substitute the existing GUI with a new GUI. In addition, usability testing can be conducted to evaluate the strengths and weaknesses of the different GUI designs.

Reference

[1] THOMAS, D. and HUNT, A. (2001) Programming Ruby The Pragmatic Programmer's Guide, Addison Wesley.

[2] THOMAS, D. and HUNT. A. (2001, Jan) Programming in Ruby, Dr. Dobb's Journal.

[3] MYERS, B. *et al.* Collaboration Using Multiple PDAs Connected to a PC Proceedings CSCW'98: ACM Conference on Computer-Supported Cooperative Work, November 14-18, 1998, Seattle, WA. pp. 285-294.

[4] Drake, J. Programming in the Ruby Language , IBM DeveloperWorks. URL: http://www-106.ibm.com/developerworks/linux/library/l-ruby1.html

[5] Matsumoto, Y. (2001) Ruby In a Nutshell, O'Reilly & Associates Inc.

[6] Havewala, A. (1999, Mar) The Windows CE Emulator, Dr. Dobb's Journal.

[7] Jipping, M. *et al.* (2001, Feb) Using handheld computers in the classroom: laboratories and collaboration on handheld machines Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education. pp. 169-173.

[8] G. Leedham (1991) Input/output Hardware, Engineering the Human-Computer Interface, McGraw-Hill Book Company, pp. 187-219.

[9] Squeak programming system:URL http://www.squeak.org

[10] Compaq Corporation's Project Mercury:URL http://www.crl.research.digital.com/projects/mercury/

[11] IBM Wireless Security Advisor: URL http://www.research.ibm.com/gsal/wsa/

[12] IBM System Network, Analysis, and Performance Pilot (SNAPP): URL http://www.alphaworks.ibm.com/aw.nsf/frame?ReadForm&/aw.nsf/techmain/67B6BC441F40E0DA88256863006FAA65

[13] G. Leedham (1991) Speech and Handwriting, Engineering the Human-Computer Interface, McGraw-Hill Book Company, pp. 220-245.

[14] J. Kang, T. Muter, (1989) Reading Dynamically Displayed Text, Behavior and Information Technology, pp. 32-42.

[15] C. Faulkner (1998) The Essence of Human-Computer Interaction, Prentice Hall, New York.

[16] X. Huang, A. Acero, C. Chelba, L. Deng, J. Droppo, H. Hon, et al. (2001, April) MIPAD: A next generation PDA prototype, Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing.

[17] Conversay Advanced Symbolic Speech Interface (CASSI) Embedded Speech Recognition White Paper. Conversay Corp. URL http://www.conversay.com/

[18] HP MicrochaiVM. HP Computer News April 2001. URL http://www.hp.com/communications/network/broadband/broadbandinfolib/chai.pdf

[19] Mswin32 Ruby. URL http://www.dm4lab.to/~usa/ruby/index_en.html#mswin32

[20] CYGWIN project URL http://sources.redhat.com/cygwin/

[21] A. Downton. (1991) Engineering the Human-Computer Interface, McGraw-Hill Book Company, pp. 235-245.

[22] J. Ward, M. Phillips. (1987) Digitizer technology: performance characteristics and the effects on the user interface, IEEE Computer Graphics and Applications, Apr., pp. 31-34.

[23] M. Meeks, T. Kuklinski (1990) Measurement of dynamic digitizer performance, in Computer Processing of Handwriting, R. Plamondon and C.G. Leedham (eds), World Scientific Press, Singapore, pp. 88-110.

[24] G. Bristow (1986) Electronic Speech Recognition: Techniques, Technology and Applications, Collins, London.