ABSTRACT

WALLIS, MICHAEL DAVID. JavaTutor – A Remotely Collaborative, Real-Time Distributed Intelligent Tutoring System for Introductory Java Computer Programming – A Qualitative Analysis. (Under the direction of Dr. James C. Lester).

One-on-one human tutoring has long been established as a highly effective method of instruction. In attempts to bring this type of instruction to a larger audience, substantial amounts of research and development have been dedicated toward the creation of Intelligent Tutoring Systems (ITSs). The field of introductory computer programming has received much attention from the ITS community. The research performed in this thesis explores the potential use of web-centric, Rich Internet Application (RIA) technologies for creating a remotely collaborative, distributed ITS, JavaTutor, which provides an integrated development environment that is directed toward human-to-human tutoring in the field of introductory Java computer programming. JavaTutor brings a student and tutor together to collaboratively engage one another while developing a solution to a programming task using the Java programming language. Code authorship is performed in a web-based environment and is further supported by remote code compilation and execution. Textual dialogue allows communication between students and tutors. A series of tutorial sessions were held to elicit input and feedback from human participants about their experiences using JavaTutor, specifically for identify potential issues and ways to further improve the system.

JavaTutor – A Remotely Collaborative, Real-Time Distributed Intelligent Tutoring System for Introductory Java Computer Programming – A Qualitative Analysis

by
Michael David Wallis

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

____
Dr. James C. Lester
Chair of Advisory Committee

____
Dr. Kristy E. Boyer

____
Dr. Eric N. Wiebe

DEDICATION

To Angela, Aidan, Payton, and Savannah.

BIOGRAPHY

Michael David Wallis was born in Fayetteville, North Carolina on September 14, 1976. He graduated from Pine Forest Senior High School in 1994. He obtained a Bachelor of Science degree in Computer Science from North Carolina State University in 2002. Since 2000 he has been employed as a software engineer with several companies including Stingray Software, a division of Rogue Wave software, Peopleclick, Applied Research Associates, Cyberspace Solutions LLC, and Northrop Grumman Corporation. He returned to North Carolina State University in 2006 and began graduate studies in Computer Science. Michael married Angela Blackmon in 2005, and their three children Aidan, Payton, and Savannah, were born in 2006, 2009, and 2011, respectively.

ACKNOWLEDGMENTS

My wife Angela and my three children Aidan, Payton, and Savannah, have sacrificed greatly over the years in my pursuit of furthering my education. Angela has always encouraged me, especially during the many stressful and demanding times and never waved in her support. I will forever be grateful for all she has sacrificed. My son, Aidan, who has asked me many times while being tucked into bed, “Are you going to work on school now,” may now go to bed without worrying about what daddy is working on. To my eldest daughter, Payton, who may now have all of her father’s attention in order to play “princess” and “dress up.” To my youngest daughter, Savannah, may you always be my surprise blessing. My mother, Wanda Jones, and stepfather, Jimmy Jones, have continually provided encouraging words for when I had my doubts. My father, Thomas M. Wallis Sr., and stepmother, Barbara Wallis, have always helped me to stay grounded and to focus my attention on what matters most. My brother, Thomas M. Wallis Jr., who I have always looked to for inspiration, for all of his sacrifices and how he has made such a positive impact on the lives of my children during the time of this work. To my extended family, Elaine Blackmon, Robert and Daynelle Headen, Lynn and Eleanor Felton provided support and reassurance many times over. My friends and “family” at Knightdale Baptist Church have helped to spiritually guide me in all aspects of my life.

Dr. James Lester, my advisor, has provided me with the upmost enthusiasm and encouragement throughout my graduate studies at NCSU. He has shown consistent, unwavering dedication in supporting me, my family, and my efforts in completing this work. Dr. Lester is an exemplary advisor who has provided me with guidance and direction. I cannot thank him enough for all he has done.

My colleagues in the IntelliMedia Research Group provided tremendous amounts of input, support, and guidance. Dr. Kristy Boyer, whom without her inspiration and leadership, this work would not have been possible. I am forever grateful for all she has done. Robert Phillips for his role in the early stages of JavaTutor and for planting the seeds for the project to emerge into what it is today. Chris Mitchell for his logistical support in helping to administrate the JavaTutor tutorial sessions and for further research into JavaTutor-relevant

technologies. Bradford Mott for his guidance and direction on applicable technologies. Additional colleagues who participated in this work include Eunyoung Ha, Alok Baikadi, Joe Grafsgaard, Jennifer Sabourin, Andy Smith, Robert Taylor, and Donnie Wright. NCSU CSC departmental staff, namely Carol Allen, for arrangement and scheduling of CSC room locations. I extend my many thanks to these individuals for their work and participation for helping to carry out this research.

I would also like to thank one of my former employers, Applied Research Associates, Inc., and my current employer, Northrop Grumman Corporation, for providing financial support for sustaining my academic and research work. Both companies encourage their employees to further expand their scholastic backgrounds and to seek out innovative and educational opportunities.

TABLE OF CONTENTS

LIST OF TABLES viii
LIST OF FIGURES ix
CHAPTER 1 Introduction 1
CHAPTER 2 Background and Related Work 2
2.1 Distributed Learning and Relevant Technologies 2
2.2 Remote Collaboration 4
2.3 Software Development Environments 4
CHAPTER 3 System Research, Design, and Development 12
3.1 JavaTutor Server 13
3.2 JavaTutor Client 17
3.3 Authoring Java Code 19
3.4 Code Compilation 20
3.5 Student Program Execution 21
3.6 Adobe Messaging Services 26
CHAPTER 4 Tutorial Sessions 31
4.1 Participants 31
4.2 Participant Interaction 31
4.3 Problem-Solving Task 32
4.4 Procedure 34
4.5 System Configuration 34
CHAPTER 5 Results and Analysis 35
5.1 User Interface and System Behavior 35
5.2 Participant Actions and Programming Task Logistics 41
5.3 Participant Knowledge 42
5.4 Programming Task 43
5.5 Recommended Improvements 44
CHAPTER 6 Conclusion 46
6.1 Summary 46

6.2 Limitations 47
6.3 Future Work 47
6.4 Concluding Remarks 48
GLOSSARY 49
REFERENCES 50
APPENDICES 55
APPENDIX A: Select Materials for Study 56

LIST OF TABLES

Table 1. Regular Expressions for User Code Extraction 22

Table 2. JavaTutor Top-Down Prioritized Channels 29

Table 3. Programming exercise (Version 1) for tutorial sessions 56

Table 4. Programming exercise (Version 2) for tutorial sessions 58

Table 5. Participant post-tutorial session questionnaire 60

LIST OF FIGURES

Figure 1. High-Level JavaTutor System Architecture 13
Figure 2. JavaTutor Login Screen 18
Figure 3. JavaTutor IDE 19
Figure 4. Java Import Declaration 22
Figure 5. Java Class Declaration 22
Figure 6. JavaTutor Server Template Excerpt 24
Figure 7. JavaTutor Server Template Excerpt - Post Main Body Replacement 24
Figure 8. Cross Domain Socket Policy 25
Figure 9. Adobe BlazeDS/LiveCycle Flex Message Flow - Producer to Consumer(s) 27
Figure 10. BlazeDS/LiveCycle Data Services Server Architecture 27
Figure 11. JavaTutor IDE Command Buttons 36
Figure 12. PseudoThread Elapsed Time Computation 37

CHAPTER 1

Introduction

Human-to-human tutoring has been shown to be a highly effective method for delivery of educational and instructional material. Studies of expert human tutoring revealed significant learning gains shown to be more effective than classroom instruction (Bloom, 1984). Replicating the process has been a goal of many educational and artificial intelligence researchers, particularly in the design and development of Intelligent Tutoring Systems (ITS). Several domains have seen the emergence of ITSs, particularly in areas of physics (*e.g.*, ITSpoke (Litman & Silliman, 2004), AutoTutor (Grasser *et al.*, 2003), Andes (VanLehn *et al.*, 2005)), mathematics (*e.g.*, Geometry Explanation Tutor (Aleven *et al.*, 2003)), medicine (*e.g.*, CIRCSIM (Evens *et al.*, 2001)), and computer programming (*e.g.*, ProPL (Lane & VanLehn, 2004), JITS (Sykes & Franek, 2003). No ITS to date has been shown to be as effective as human tutors but all have strived to meet or exceed the same level of effectiveness (VanLehn, 2008). The work reported in this thesis attempts to provide research into existing computer programming ITSs and to implement a prototype system for supporting remotely distributive, real-time collaboration between a student and tutor in the domain of Java programming.

The remainder of this document is organized as follows. Chapter 2 presents background and related work on distributed learning, the types of systems that provide environments for learning, remote collaboration, and specific types of software development environments. Chapter 3 addresses the JavaTutor system research, design, and development, which covers researched technologies and implementation details. Chapter 4 describes the tutorial sessions that provided user input and feedback for later system evaluation. Chapter 5 provides the results and analysis of the findings from the tutorial sessions, specifically those addressing system issues and recommended improvements. Chapter 6 summarizes the thesis and presents recommendations for future work.

CHAPTER 2

Background and Related Work

2.1 Distributed Learning and Relevant Technologies

The emergence of remote learning and distributed computing systems over recent decades has yielded several diverse and influential approaches to learning, many of which have provided a foundational basis for current learning systems. Through the use of distributed learning systems, geographically, temporally, and culturally displaced participants can work together in an efficient and structured manner. Instructors and students are no longer required to be physically present within the company of one another as advances in distributed computing have made it possible to provide instruction materials and learning environments that better serve the needs of all participants. Advances in distributed learning systems have progressed from one-size-fits-all approaches to being more robust and adaptive to individual learners.

2.1.1 Desktop (thick-client) applications

The most prolific and common type of applications are locally installed and physically reside and run on a hosting machine. Such applications can be made to be network-centric (or network aware) by employing one of many available networking technologies including socket-based communication, remote method invocation (RMI), or web services. Each of these technologies ultimately provide the data communication needed for distributed computing but do so at the expense of software development time, future maintenance and integration, scalability, and network communication overhead. Most software developers start out with or have most of their experience developing applications for a desktop environment. Thus, it is not surprising to see ITSs being developed as desktop applications.

2.1.2 Web applications (HTML)

With the advent of the web, web-based learning environments for content delivery and eLearning have quickly emerged and have been shown to be quite effective in reaching broad ranges of audiences. Relevant technologies have made their way to the web and have enabled web pages to behave and function in a more responsive, asynchronous manner, render a more visually appealing display, and to be more adaptive to end users (*e.g.*, AJAX, JQuery). Web-based environments range from static formats where content is geared toward a "one-size-fits-all" approach with all content being fixed to formats that are highly-dynamic, responsive, and interactive. Modern web-based learning environments have gone the way of this later format as technology has evolved, leading to more effective ways for learning and for environment customization.

2.1.3 Rich internet applications (RIA)

In efforts to create a more rich environment for the end user, usually mimicking traditional desktop applications, Rich Internet Applications (RIAs) are served over the web, usually via a plug-in that is embedded into internet browsers, and are usually coupled with supporting client-side scripting technologies (*e.g.*, JavaScript). Several platforms have been created for supporting RIAs and have varying shares of market penetrability. The most dominant RIA technologies include Adobe Flash, Microsoft Silverlight, Oracle JavaFX, and the Eclipse Rich Client Platform (RCP). Usage and/or market share of these types of technologies, minus Eclipse RCP[1], is at 95.71% for Adobe Flash, 66.23% for Microsoft Silverlight, and 76.37% for Java[2]. Creating systems using these technologies make distributed client/server model easier for software developers as they provide many programming paradigms that are commonly found in traditional desktop application development. As a result, a reduction in the amount of time taken to learn and to apply these technologies can be achieved, thus yielding the potential for a more productive software development experience. Many ITSs

---

[1] Usage data was not available at the time writing.

[2] http://www.statowl.com/custom_ria_market_penetration.php

researched and developed thus far have used Java Applets or Java in a desktop environment and have been less visible in RIA-based systems.

2.2 Remote Collaboration

Bidirectional, real-time Java source code composition, compilation, and execution thereof between a student and a tutor are essential in the JavaTutor environment. In any given JavaTutor tutorial session and regardless of a participant's role, immediate feedback of what action has been carried out by one participant (*e.g.*, keyboard typing events including insertion and removal from the Java tutor source code editor or invocation of the Java compiler or execution commands) must be immediately portrayed to the corresponding participant in order to simulate a true one-on-one, collocated tutorial session.

2.3 Software Development Environments

2.3.1 Disjoint/isolated systems

In its most raw form, developing software can be achieved through the use of a simple text editor and a compiler. Depending on the programming language being used, additional steps to compilation may be required (*e.g.*, C++ development includes linking and assembling phases while Java uses an intermediate bytecode representation in a Java Virtual Machine (JVM)). Using this type of development environment, software developers interact directly with a compiler, thus requiring knowledge of the compiler's command line interface. While this type of development environment tends to keep the software developer working in a non-automated, strictly console-oriented manner (*e.g.*, without the aid of various features and benefits of an Integrated Development Environment (IDE)), the developer must first acquire and install each piece of software before actual software development may proceed. Combine this with the lack of program interoperability, remote collaboration, and student/tutor real time communication facilities (*e.g.*, textual chat) an environment of this type is not readily sufficient to satisfy the requirements of JavaTutor. For example, take the text editor component in this type of system and examine its basic, fundamental purpose: to provide functionality for editing text-based documents. Expecting the text editor to integrate with a

compiler, a remote collaboration system, or provide bi-directional communication would be beyond the scope of this intended purpose. The same type of analysis can be made for each of the components that make up multi-component systems. A compiler is not meant to provide remote collaboration, a chat client is not meant to compile code, and a text editor is not meant to provide bi-directional communication. Integrating the various components in this type of development environment usually proves to be quite a difficult task. Some components may have restrictive licenses that do not permit any modifications or derivative works be produced while other components may be commercial software and are not released in a source code-only format. In a worst case scenario, some components may be complete black boxes and cannot be modified in any form whatsoever. Given the numerous challenges and issues present in this type of software development model, particularly for developing an application such as JavaTutor, another more robust and scalable software development model must be used.

One of the JavaTutor project's goals is to enable a student to learn to program in Java, without having to require the student to first install a JVM or a given text editor. Instead, the student should be able to open their preferred web browser, navigate to the JavaTutor web site, provide their authentication credentials, and if successful, be redirected to a web page to begin writing, compiling, and running Java code, all without requiring the student install any additional software. Students may not have the permissions necessary to install software on the local machine (*e.g.*, working on a computer in a computer lab as a non-administrative user), thus subsequently preventing the student from establishing the required components of this type of software development model. This limitation establishes a critical blocking issue that cannot be overcome by merely substituting another or modifying an existing software development tool(s). JavaTutor cannot, through design and development alone, bypass this issue and thus has been chosen to be deployed in a web-based, online manner.

2.3.2 Web-based/online systems

Many online systems exist for writing code, compiling, and in some cases, execution of such code. Nearly all of these web-based, client-server distributed systems tend to follow a standard design of "Write, Submit, Run," with each system varying slightly in their

implementations. Being that these systems emerged in the context of the web, designers and developers of these systems had to simulate traditional thick-client application behaviors and appearances using web-based technologies such as HTML, AJAX, and JavaScript. While developing for the web has matured over the past two decades, particularly for asynchronous behavior and due to the aid of supplemental technologies (*e.g.*, JQuery, JSON, Web Services, XML), depending on the scope and complexity of desired features, implementing such systems can require substantial levels of time and effort in order to reach maturity. Despite these challenges, there do exist benefits that are not typically offered by traditional thick-client, desktop applications such as, but not limited to, maintaining central control of application management and overall operation and being readily available to anyone with a web browser (possibly with cookies enabled) and an internet connection.

On the most basic end of the spectrum, systems such as JXXX Compiler Service (Tschalär, 2011) accept Java source code and library files via web-based file uploading and allows for the user to specify various options for source code compilation (*e.g.*, debugging tables, optimization, warning suppression, verbosity, and deprecation). Following successful parsing and compilation phases, JXXX Compiler Service provides the user with the results of said parsing and compilation and if no errors were found, several links are provided for downloading the resulting class files, including separate hyperlinks for compressed archives containing the class files. Systems such as JXXX Compiler Service provide the capability to compile Java source code only and do not allow for the user to execute online. Instead, if the user wants to run their code, he or she must first download the generated files and execute them locally. Given this constraint, the user will be required to obtain and install a corresponding version of the Java Runtime Environment (JRE).

Moving away from the file-based transfer model to more web-centric systems where capabilities include, at least a minimum of, HTML controls for user supplied Java code, buttons and/or menus for initiation of source code compilation, and possibly the option of online program execution. User code is entered via an HTML TextArea control, which is sometimes customized with programmable-friendly fonts and syntax highlighting for simulating more authentic source code authoring tools and development environments. The

most common and widespread methodology used to achieve such styling and behavior is through the use of the JavaScript programming language (also known as ECMAScipt) and Cascading Style Sheets (CSS). When used within a web browser, JavaScript enables client-side, programmatic access to the Document Object Model (DOM) for dynamic interaction with the contents and objects in HTML documents. Event handling through JavaScript enables dynamic and sometimes interactive behavior, usually in response to user actions and control-based events (*e.g.*, keyboard, mouse, focus, and content change). Cascading Style Sheets provide the capability to modify the "Look and Feel" of markup-styled documents and enables separation of document content from document presentation. Common uses of CSS include, but not limited to, styling of font, color, size, and alignment of text and modification of layout and appearance of elements contained within the page. When applied in unison, JavaScript and CSS provide the ideal rich, dynamic client-side behaviors and appearances for authoring of source code in web-based, online software development environments. For example, as a user types characters into an HTML TextArea control, JavaScript event handlers can be invoked to examine the control's text entered thus far and if language-specific keywords are found, their style can be modified by applying corresponding CSS rules that modify the keyword font weight and color.

Once source code has been authored, compilation is the next step. In a web-based, online environment, compilation is usually delegated back to or another known server, which contains all of the necessary environment settings and tools (*e.g.*, Java source code compiler "javac.exe" and Java Development Kit (JDK) for system libraries). The source code authored by the user is uploaded to the server, compiled, and the results are sent back to the user, which typically include any output generated from the compiler during the compilation process, such as logging, warnings, and errors (if any are found). Usually at this point, systems differ in how they handle execution of compiled code. Some only provide download links to compiled source code (*e.g.*, .class files for Java source code) while others run code that only performs output only (*e.g.*, non-interactive. Systems including the Online C/C++ Compiler (bOtskOOl, 2011), Ideone (Sphere Research Labs, 2011), and Youjavait (Youjavait, 2011) execute Java code online, but they are limited by allowing full interaction

with the end user. Only output is generated with these tools and attempts to read user input from a command line are either ignored and compiled out or is stopped by the compiler before an executable can be generated, with the user being subsequently notified of such a limitation. Of these systems, none appear to allow full user interaction at the command line for end users. In addition, these systems lack any remote collaborative capabilities or any type of bi-directional communication between two parties. Usage of any of these tools for the JavaTutor program would not be sufficient due to these limitations.

Going beyond the simple web-based single TextArea control, coupled with options to build and execute the code, are those systems that attempt to replicate complete IDEs. Such systems provide rich graphical user interfaces, are highly dynamic and responsive, and simulate as much as possible the traditional thick-client, desktop application IDE. Features typically found in traditional IDEs and replicated in most online systems include, but not limited to, project, folder, and file management, compiler settings, windowing dock /float capability, runtime information such as call stack and watch views, and source code authoring information such as cursor position, row and column line counts. The level of effort required to develop such systems are vast and usually have one goal in mind: to mimic an IDE such that the online version operates just as seamlessly and thoroughly as its client-side-only counterpart. For users who are comfortable operating in a traditional client-side IDE, this type of online environment may be more suitable to their needs. This type of user should quickly adapt to the online environment and be able to identify all of the parts of the IDE such as menus and toolbar buttons corresponding to various commands including file-related, source code compilation and execution, tools and settings, windowing-based, and system help. Project and file views are quite common with these systems, with each typically providing a variety of views for editing documents, navigating content, and execution of compilation and execution of various build configurations (*e.g.*, debug versus release). Examples of such systems include Compilr (Ninja Otter Inc., 2011) and the web-based version of Eclipse named WWWorkspace (Ryan, 2007). At the time of this writing, Compilr is expanding and growing in capabilities and user base. However, despite providing a thorough development environment for the web, Compilr appears to run within and applet

and provides downloadable hyperlinks for running compiled Java code on the local user machine. However, no real-time remote collaboration or text-based messaging are present, except for the ability to make projects either public or private, which could be picked up throughout a search. WWWorkspace (Ryan, 2007) leverages the plug-in based architecture of Eclipse to provide a Web-IDE plug-in for mimicking the behavior of the Eclipse development platform and is coupled with additional technologies including Eclipse Equinox, Jetty Web Server, the Dojo Toolkit, JavaScript, Comet, JSON, and HTML. This system has shown to be one of the front runners in providing an extremely rich and dynamic user interface, providing file sharing to other users, and textual communication through a customized chat client. However, this system lacks the capability to provide real-time remote collaboration, a necessity of JavaTutor in that tutors must be able to view what the student has typed and what actions have taken place (*e.g.*, initiation of compilation or code execution).

2.3.3 Integrated development environments (IDE)

Of the three types of software development environments reviewed here in this work, IDEs are by far the most extensive and are the most thorough with respect to included features, capabilities, robustness, extensibility, and scalability. In the most basic sense, an IDE is typically comprised of tools already found in disjoint/isolated systems, namely a source code editor, a compiler, and a debugger, but with all pieces tied together in an "integrated" manner. However, IDEs have made much progress over the past decade and have yielded tremendous strides in available functionality and capabilities, all which have only strengthened their popularity and expanded their target audience. Source code editors now enjoy code auto-completion (*e.g.*, Microsoft IntelliSense (Microsoft, 2011), Eclipse Content Assist), syntax highlighting, and many other document editing features such as custom paste/insertion, multi-row tab indentation, and automatic brace matching. Practically all IDEs provide debugging capabilities that enable developers to improve their productivity and to more thoroughly examine the runtime and behavior of their applications. Watch windows, call stack navigation, code breakpoints, and runtime code stepping are among several of the available features to the user, providing a wealth of invaluable tools to facilitate application

development and testing. Being an integrated environment, some IDEs are extensible and thus allow for third-party, external systems to be integrated back into the main IDE, thus providing even more functionality to the user. The Eclipse development environment is comprised of features and plug-ins and make up smaller components of the overall Eclipse workbench. Such features and plug-ins may introduce a new or alter an existing perspective, view, or editor or provide the capability to integrate external tools and libraries but in an Eclipse-structured way. As evidenced in the work of (Ho *et al.*, 2004), an Eclipse plug-in named Sangam was developed for enabling real-time, remote collaboration between a tutor and a student. Microsoft Visual Studio allows users to leverage macros and add-ins for customization of the hosting IDE and offers users the ability to introduce automation, customized project systems, debugger, editor, and user interface extensions, authoring and integration for generation of help content, and host of many other capabilities.

While all of these features and capabilities provided by IDEs yield many benefits to the user, IDEs do come at a cost. First and foremost is users of said IDEs must first install them on their computer, which in and of itself can be quite expensive in terms of size, scope, time, monetary costs, licensing, and level of effort. As IDEs have grown over the years, so has their required footprint for installation. Not only have hard disk space requirements increased, so has the minimum amount of available RAM and type of processor (*i.e.*, processor speed), as well. Minimum requirements or recommendations even exist for peripheral devices such as display resolution for monitors, which lead back to having a sufficient video card and/or adapter. Some IDEs may require certain operating systems and architectures (*e.g.*, 32 bit, 64 bit) while others are considered platform independent (*e.g.*, Eclipse). In terms of scope, some IDEs can operate completely isolated within a dedicated file directory while others may have dependencies on existing system resources and integrate with the hosting environment. For example, Eclipse comes packaged as compressed archive files, ready to run following decompression on a hosting machine. Minus the requirement of having Java installed, Eclipse is fairly straightforward to install. Microsoft Visual Studio is a bit more invasive and takes advantage of Windows-based system registry settings and

operating system related content (*e.g.*, Global Assembly Cache (GAC), WinSxS, user-account specific solution file directories, etc.).

As is the case with learning any new software application, there exists an associated learning curve. IDEs are not immune to this, particularly due to the level of complexity and vast number of features present. Users who have little to no experience at all developing software in an IDE may initially find IDE user interfaces to be overcoming and may negatively contribute to their understanding of the system. Time required learning an IDE and the capabilities it offers can be substantial, so much so that it can often lead to critical decisions being made as to whether or not to even use the IDE to begin with. Therefore, IDE designers and developers must be careful as to how best to organize an IDE, with respect to menu layout, toolbar grouping and categorization, and window placement. For those users who have been exposed to and/or previously operated and developed with an IDE, they will find many common threads between IDEs such as standard menus and commands, project/file views, editor windows, source compilation, and debugging facilities. Such commonly shared, IDE-agnostic features allow for knowledge transfer between different IDEs and enables developers to more rapidly acclimate to new IDEs.

Monetary costs and product licensing can severely impact one's ability to acquire and legally use an IDE. Some systems are open source and freely available on the internet and come with an appropriate license (*e.g.*, Apache License, GNU General Public License, etc.) while others may be retail and come with a proprietary license and potentially limit the number of users who may use the software. Volume and/or site licensing is sometimes used to allow flexibility in group-level usage while in other cases, per-user, per-machine restrictions may be present.

For all types of systems identified in this work, they each serve their intended purpose but fail to simultaneously provide key functionality, namely real-time, remote collaboration and bidirectional communication, in a cohesive, unified architecture.

CHAPTER 3

System Research, Design, and Development

The JavaTutor prototype system follows the standard client/server model for distributed computing. An Apache Tomcat server was used to host a customized J2EE web application, which handled all server-side processing and integrated Adobe Flash technology for processing client-side events and processes. A J2EE web application was used for compiling all Java source code and management of spawned processes dedicated to the execution of student programs. Any user, regardless of role (*e.g.*, student, or tutor) would access the JavaTutor server through a standard web browser with having Adobe Flash being installed locally. **Figure 1** depicts a general, high-level overview of the JavaTutor prototype system, with discussion of each component in the following sections.

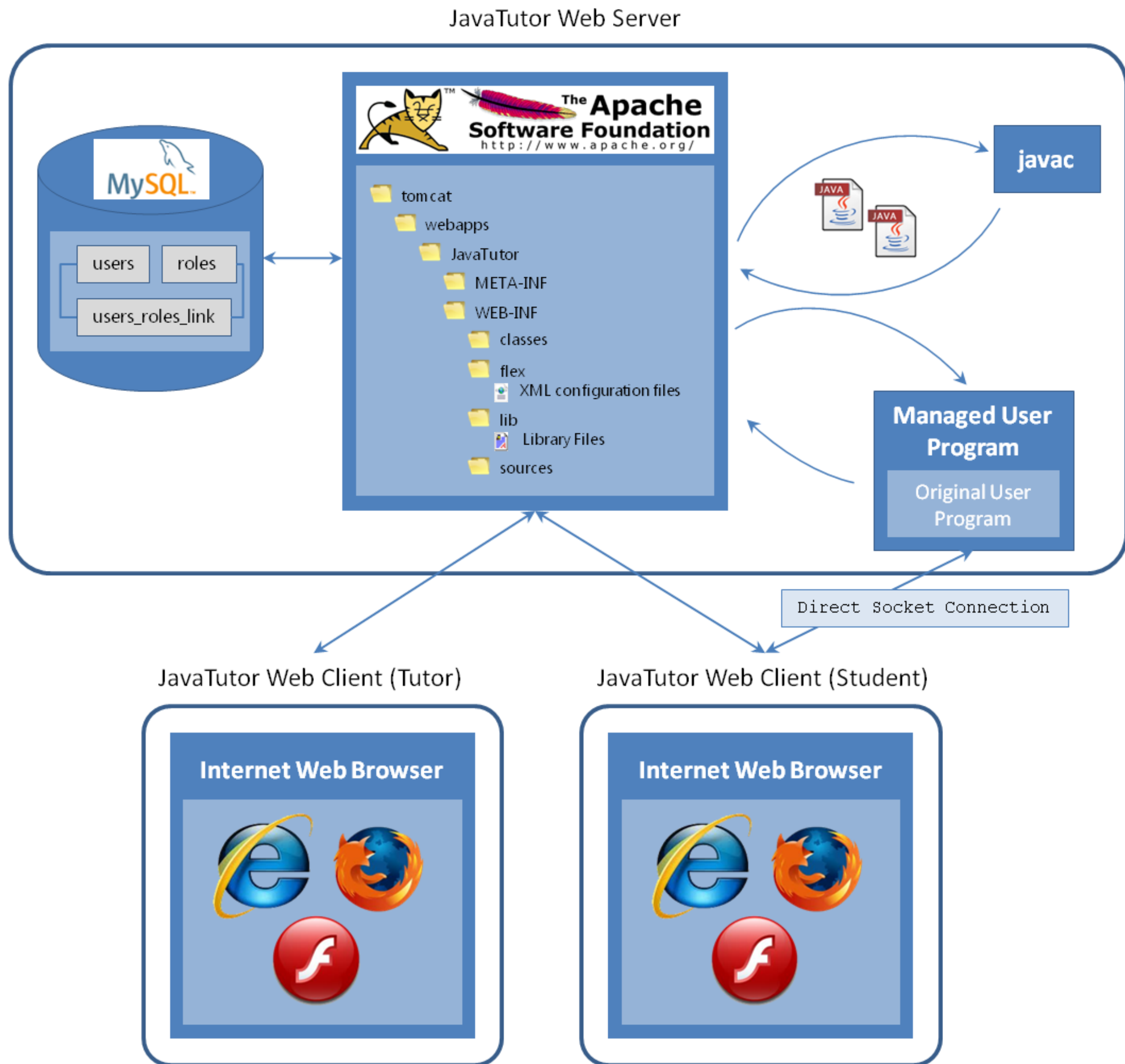

Figure 1. High-Level JavaTutor System Architecture

3.1 JavaTutor Server

The JavaTutor server component is by far the largest and most diverse of all of the pieces comprising the JavaTutor system. Responsibilities include user authentication and authorization, code compilation and execution, process management, and event handling among JavaTutor users. Several technologies have been integrated for supporting this work

including Apache Tomcat, Adobe BlazeDS, J2EE, Adobe Flash, Adobe Flex, MySQL, and JBoss Hibernate and are discussed in details in the sections that follow.

3.1.1 Apache Tomcat

Apache Tomcat is an open source technology that provides a server container implementation of Sun Microsystem's Java Servlet and JavaServer Pages (JSP) specifications (named Catalina) and contains an HTTP connector component (named Coyote) for hosting Java web applications. Catalina is at the core of the Tomcat system and provides the Java engine (*e.g.*, JVM, JRE) that is central to the entire web server. Coyote listens for incoming connections on a given TCP port and delegates handling the request onto the Tomcat engine for further processing, followed by subsequently issuing a response back to the requesting client. Tomcat was chosen for the JavaTutor prototype because it is lightweight and highly responsive, flexible and scalable, secure, and best of all, free and 100% open source.

3.1.2 Adobe BlazeDS

Adobe BlazeDS, a Java remoting and web messaging system supporting bi-directional, asynchronous data communication for Adobe Flex and Adobe AIR, was chosen as part of the server technology for the JavaTutor prototype. BlazeDS offers many desirable features including improved data transfer rates for improved application performance, allowing data to be loaded up to 10 times faster than with XML or SOAP, remoting with automatic data marshaling between Flash clients and Java methods on a server, real time data pushing over HTTP, publish/subscribe mechanisms that extend current messaging infrastructure (*e.g.*, Java Message Service (JMS)), and is open source. The Adobe Action Message Format (AMF), a binary format that is used to serialize ActionScript objects, was one of the driving factors in the decision for adopting Adobe Flash over other RIA technologies. Using AMF yielded an extremely responsive data transfer for all of the Java code and textual communication being transferred between the student and tutor, a vital part of having a usable system.

To facilitate easier installation and setup, Adobe made available a customized distribution of the Apache Tomcat (version 6) web server with BlazeDS pre-installed and

configured. Consumers of the BlazeDS technology could download, extract, and install, immediately yielding a working, "out-of-the-box" web server system that could host web applications that include Adobe Flash-based content. This distribution of Apache Tomcat and BlazeDS was used for the JavaTutor prototype system.

3.1.3 J2EE Web Application

The J2EE web component of the JavaTutor system was created using two disparate tools for development. These tools were the Eclipse IDE for Java EE Developers, Helios SR2 distribution and the Adobe Flex Builder 4.0 distribution, and are discussed in detail in the following sections. The resulting JavaTutor web application consisted of two deployments with one deployment coming from the Eclipse IDE and the other from Adobe Flex Builder. With the functionality of these two tools were combined, a fully integrated, Flash/J2EE-based web application was created.

The Eclipse IDE for Java EE Developers provides tools for software developers who are creating Java EE and web applications and provides the infrastructure needed to create and deploy web applications into a variety of web servers including Apache Tomcat, JBoss Application Server, and IBM WebSphere. A traditional, J2EE web application was created using a dynamic web application project template provided by the Eclipse IDE, which consisted of Java code corresponding to all backend server functionality and web-based content including Flash-based files (*e.g.*, HTML web page hosting the JavaTutor SWF file, Flash setup files, etc.) and startup XML configuration files for initialization of the BlazeDS system. The main body of Java code consisted of a "Service" class, which directly responded to and handled all client-side, Flash-based events including user authentication and authorization via the initial login screen and IDE-based controls events such as button clicks for compiling or execution of source code. A "Compiler" class was created for handling submitted Java code to be compiled with the results being sent back down to the Flash-client. The service class actually handles the compilation button click event and delegate event handling to this compiler class for further processing.

Adobe Flex Builder provides the environment for developing Flash-based solutions using ActionScript and the open source Flex framework. A new Flash-based project was

created that was embedded within the JavaTutor J2EE web application already deployed on the Tomcat web server. Within this project, MXML and ActionScript files were developed and grouped into a single SWF file for deployment to the Tomcat server. MXML is the extensible markup language first used by Macromedia and is used to model layout, container, and control-based content in a declarative mode with ties back to ActionScript for initialization, event handling, and further customization.

This initial JavaTutor prototype consisted of only two states that the application could be in, which corresponded to either "user logged in" or "user not logged in." This Boolean state indicator drove which screen would be shown to the user. Following startup of Tomcat, initial visits to the JavaTutor server would route unauthenticated users to a Flash-based login screen where each user would enter their account credentials, which included a login identifier and password. For the purposes of the JavaTutor tutoring studies, generic login identifiers were used (*e.g.*, student1, student2, …, and tutor1, tutor2, …), which help to shielded participants from exposing their true identity. Upon successful authentication steps, users would be redirected to the JavaTutor IDE component. Authentication was performed by accepting a user's submitted credentials and verifying them against the contents of the user's login credentials recorded in an underlying MySQL database. User passwords in the JavaTutor database were stored in an MD5 encrypted file format (one-way-hash algorithm) and would later be compared to an MD5 hashed representation of submitted user login credentials. Upon a successful match, the JavaTutor system would redirect the user to the JavaTutor IDE component. Otherwise, the user would be notified about the error and given another opportunity to login.

3.1.4 DBMS: MySQL

A MySQL database was used for containing user account information, role definitions, and the roles a user may be associated with. Three relations were defined for this data with minimal key constraints specified, thus yielding a very small and trivial database schema. No other content was modeled or stored in the database. All database connections were made available by using the JDBC Driver for MySQL (Connector/J) connector driver, which enabled database storage and retrieval from the JavaTutor server.

Hibernate, the relational persistence tool for Java and .NET, was used to model the JavaTutor database schema via POJO-style domain-specific types. This technology enabled first class data types to be created and operated upon dynamically in a DBMS-agnostic manner. No knowledge of the underlying database or technology making up the database or the DBMS was needed and thus, provided a clear separation between the JavaTutor server business logic and the data layer. Any changes to the database schema would require manual intervention to update the POJO-types and would be affected by the scope and impact of changes made to the underlying JavaTutor database. However, using a relational persistence tool yields tremendous benefits and does scale with respect to the size and complexity of the database being modeled.

3.2 JavaTutor Client

Being a web-based system, clients of the JavaTutor system only required a web browser configured with Adobe Flash 10 installed. No other technologies were needed. To use the JavaTutor prototype, users would be required to open their web browser and enter the URL address of the JavaTutor web application. Upon initial arrival, users would be shown a standard login form where they would provide their credentials, followed by being routed to the IDE component. From here, the interface was fairly limited in the available functionality as the prototype system was intended to provide the minimum required functionality to achieve a working IDE environment where Java code would be authored, compiled, and executed, while communicating with another individual (*e.g.*, tutor). **Figure 2** and **Figure 3** show screen shots taken of the login and IDE screens respectively.

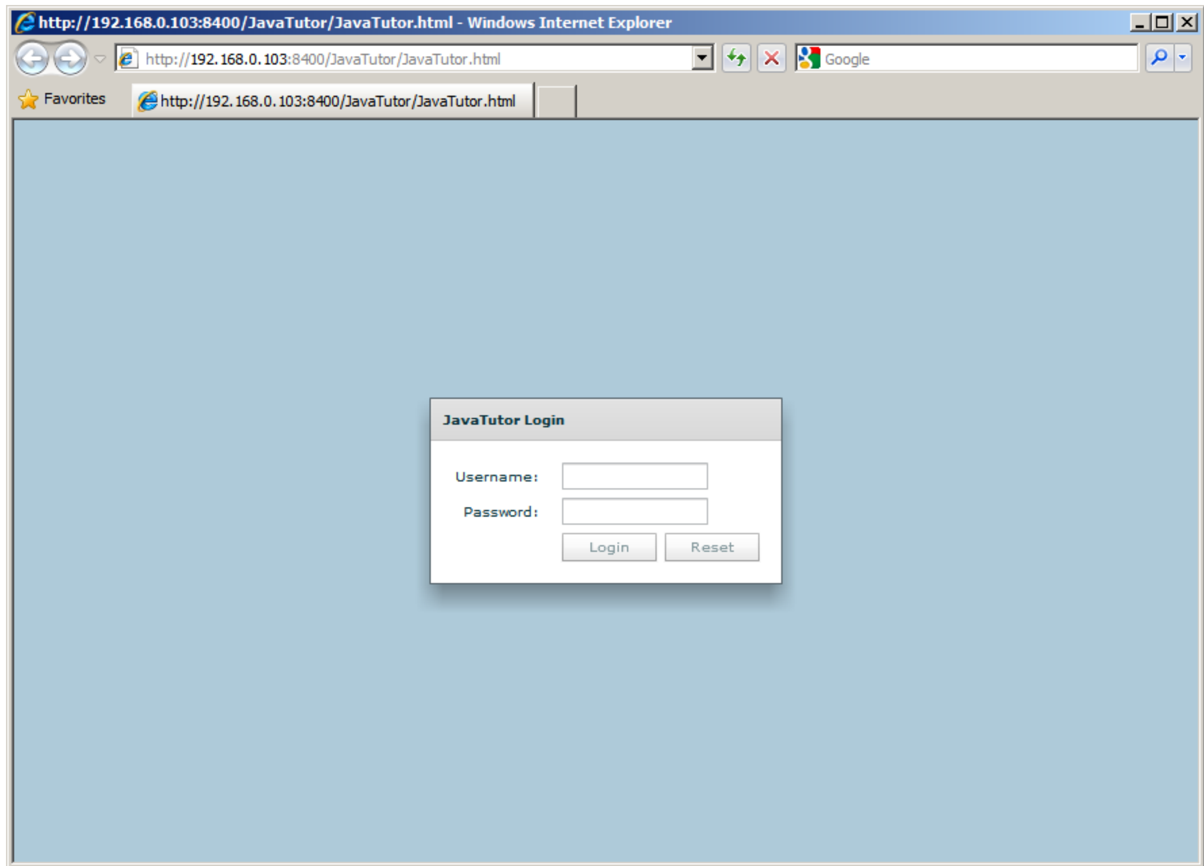

Figure 2. JavaTutor Login Screen

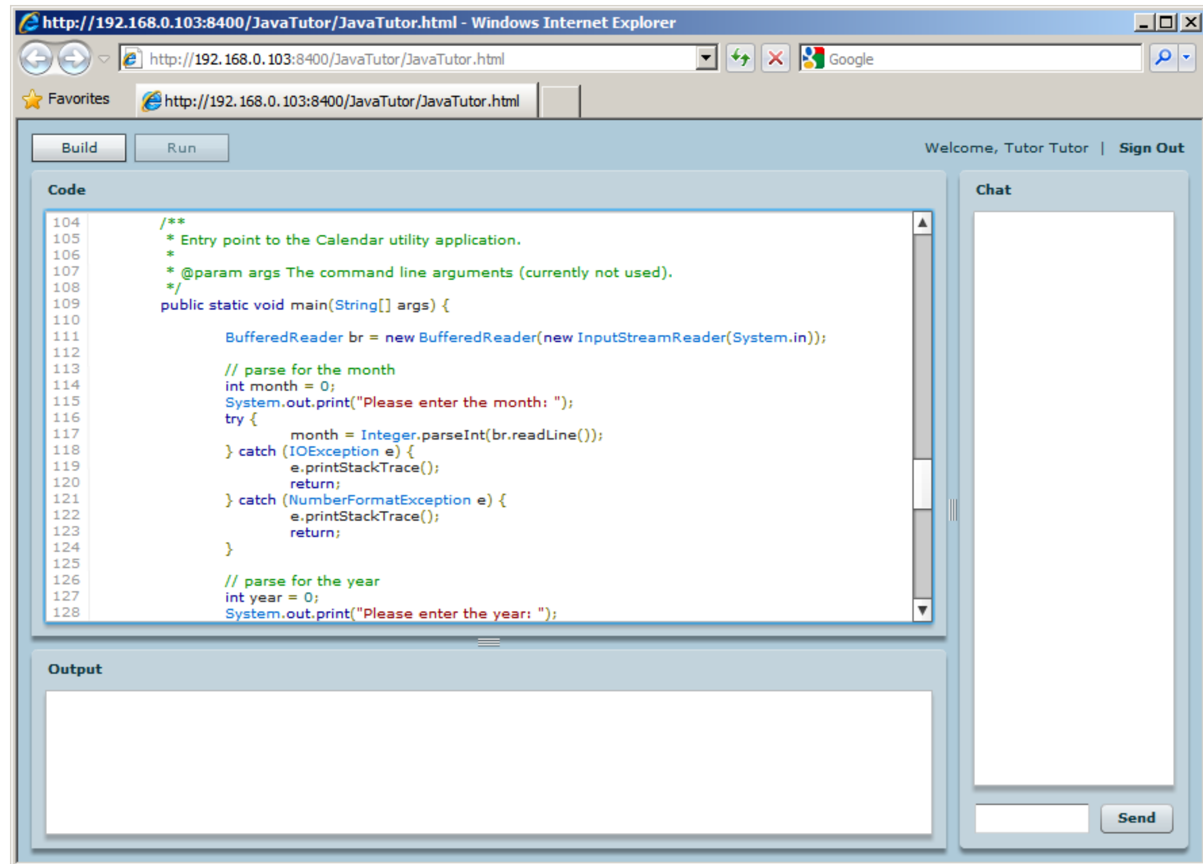

Figure 3. JavaTutor IDE

In development, Microsoft Internet Explorer and Mozilla Firefox were targeted and used extensively, with both web browsers yielding near similar results. Internet Explorer was the only browser used during the tutorial sessions due to a communication failure found to be present in Mozilla Firefox 3.x.

3.3 Authoring Java Code

Central to authoring code using the JavaTutor code editor was the Adobe Flex and ActionScript technologies. Using the Spark architecture in Adobe Flex 4.0, a Spark TextArea control was used as the Java code editor and was customized via component behaviors and skinning, a major strength of the Spark architecture. Several open-source solutions were adopted and modified for the purposes of this work. The contributions of Anirudh Sasikumar and his work on *AS3SyntaxHighlight* consisted of a port of Google's *google-code-prettify* to

ActionScript 3.0 provided the initial implementation of the syntax highlighting. Anirudh's work included applying a lexer to HTML to extract tags and convert source code into plain text. Extracted tokens are then processed for "text decorations," which decorate the associated token for customized appearance, followed by reassembling of said tokens for redisplaying purposes. This resulting decorated content served as input to the control making use of the syntax highlighting, in JavaTutor's case a Spark TextArea control.

To achieve a non-blocking, interactive experience for the user when authoring Java code, *AS3SyntaxHighlight* makes use of Alex Harui's open source `PseudoThread` Action Script class, which simulates multi-threading behavior found in traditional programming languages (*e.g.*, Java, C, C++, etc.). Since ActionScript does not allow true multi-threading, Alex's approach was to provide a pseudo version by dividing computationally expensive functionality into smaller segments and iteratively executing each segment until processing is complete. The `PseudoThread` class is actually an extension of the `flash.events.EventDispatcher` and attaches event listeners (*e.g.*, frame, mouse, and keyboard) to the main drawing area (technically named "Stage") via use of the application's `mx.managers.ISystemManager` instance, which manages the "Application Window" of a Flash-based application. By adding a generic `mx.core.UIComponent` as a child to this `ISystemManager` instance, the `PseudoThread` registers itself as an event listener to this child for responding to rendering event changes and uses this time to invoke the callback function provided by the caller. In the case of JavaTutor, this caller would be the *AS3SyntaxHighlight* component, where it is given time to execute text decoration functionality.

3.4 Code Compilation

Java 6.0 SE brought forth the Compiler API, which enabled software developers to use the javac Java source code compiler programmatically by using the `javax.tools.JavaCompiler` class. By using this class's `getTask()` method, a new `javax.tools.JavaCompiler.CompilationTask` can be created from supplying various inputs including, but not limited to, an IO-based writer to capture output from the *javac* compiler and compilation units that are to be compiled. These compilation units correspond

to the submitted user code, which are embed into `javax.tools.SimpleFileJavaObject` derivative types (*i.e.*, wraps the actual user code). The compilation task resulting from the compilation process is then used to invoke the actual process via the `JavaCompiler.CompilationTask.call()` method, which returns a Boolean status of whether or not the process failed or succeeded. From this invocation, the output sent to the writer is then routed back to the client for displaying to the end user.

3.5 Student Program Execution

Console-based programming applications, regardless of the language used for implementation, reside and execute on the hosting machine. Since JavaTutor is a distributed application running a web server with clients connected via a web browser, program execution could not be delegated to client machines. In order to simulate client-side console program behavior, several enhancements had to be developed, which are described in detail in the following sections.

3.5.1 Student Code Extraction

Submitted Java code by the student participant required additional parsing and transformation into a format that would be best suited to run within the JavaTutor system. This differs from the "build-only" operation which performs compilation on submitted Java code only and not on any post-processed version. Upon each "run" attempt by the student participant, their Java code is submitted to the JavaTutor server, where it is first parsed for specific content including Java import declarations, Java classes, and a main method, where they are later re-inserted into a statically defined Java server template file (see section 3.5.2) to yield a server-based class that is later executed for carrying out actual student program execution.

Code extraction was achieved by applying regular expressions for locating patterns within the Java source code. The `java.util.regex.Pattern` and `java.util.regex.Matcher` types were used to first compile a pattern to search for (*i.e.*, the regular expression), which produced a new `Matcher` object to use for performing subsequent matching. This matcher allowed for iterative processing over each of the matched segments

for easing extraction without having to maintain any type of supporting indexing or positioning information. Since regular expressions were used to find matching content, an understanding of the Java programming language grammar was required. The grammar referenced for this work was from Oracle and provided the rules to follow when attempting to locate content of interest (*e.g.*, Java import declarations, start of class definitions, etc.). **Figure 4** and **Figure 5** show the grammar rules applied.

```
ImportDeclaration:
      import Identifier { . Identifier } [ . * ] ;
```

Figure 4. Java Import Declaration

```
TypeDeclaration:
    ClassOrInterfaceDeclaration
    ;

ClassOrInterfaceDeclaration:
    {Modifier} (ClassDeclaration | InterfaceDeclaration)

ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration

NormalClassDeclaration:
    class Identifier [TypeParameters] [extends Type] [implements TypeList]
    ClassBody
```

Figure 5. Java Class Declaration

From these BNF-style entries, regular expressions were formulated and applied where required. Location of a potential a main method, while not explicitly identified in the grammar, was also identified via a regular expression. **Table 1** shows all of the regular expressions applied for aiding in user code extraction.

Table 1. Regular Expressions for User Code Extraction

Search/Match Item	Regular Expression
Java Import Declaration	\bimport\s[A-Za-z]+[.A-Za-z]*\b
Java Class Declaration	\bclass\s[A-Za-z]+\b
Java main Method	\bpublic\sstatic\svoid\smain\b

3.5.2 Server Code Template

User-submitted code was assumed to be console-based and thus potentially interactive (*e.g.*, not just unidirectional). To simulate this, user code was extracted as described in Section 3.5.1 and later inserted into a statically defined, server template Java source code file. This new template instantiation had to be compiled using the code compilation described in Section 3.4 and subsequently farmed out to a dedicated process for actual execution.

In this template file resided two markers that served as placeholders for the port number to bind to when the instantiated server was actually started and where the main method body in the user's submitted file should be inserted (**Figure 6** shows the marker for the main method body insertion). The port number to be inserted was selected using an internal port management class for open ports (*e.g.*, ports not in use by other running user code). For the main method body to be inserted, if no main method existed in the user submitted code, then the marker was simply removed. Otherwise, a static function call was inserted that corresponded to invoking the original main method but under a different name. Since the main method represents the entry point into the application and contains all of the business logic authored by the user, preserving the original main method was performed by simply renaming the main method to `mainMethod`. This newly renamed main method is what is invoked statically in place of the corresponding marker in the server template file. **Figure 7** shows the main method body marker replacement.

3.5.3 Communication: Sockets

The server template is actually a socket-based server implementation that creates a new instance of the `java.net.ServerSocket` class that is bound to a supplied port number and subsequently begins to listen for client connections. Upon accepting a client connection, a new `java.net.Socket` instance is obtained and is where a key part of this design takes shape. This newly created socket provides accessor methods for obtaining the input and output streams, which are later set as the system-level streams to read from and write to respectively in the running server. For output coming from the server, a new `java.io.PrintStream` object is created to wrap the client socket output stream. The

`java.io.PrintStream` class adds functionality to other output streams and provides several print functions covering various data types. This class can also be configured for automatic data flushing following the writing of a byte array, the invocation of a `println()` method, or if a new line character or byte (`'\n'`) is written. Automatic data flushing is enabled in the JavaTutor server code template by specifying a `true` Boolean value as the second argument to the `java.io.PrintStream` constructor. This causes all `System.out.println()` statements in the student-submitted code to automatically flush upon invocation. **Figure 7** is an excerpt from the server template class and shows how the socket input and output streams are configured.

```
ServerSocket server = new ServerSocket(port);
. . .
Socket client = server.accept();
System.setIn(client.getInputStream());
System.setOut(new PrintStream(client.getOutputStream(), true));
System.setErr(new PrintStream(client.getOutputStream(), true));

______@@@_REPLACE_HERE_MAIN_BODY_@@@______
```

Figure 6. JavaTutor Server Template Excerpt

```
ServerSocket server = new ServerSocket(port);

Socket client = server.accept();
System.setIn(client.getInputStream());
System.setOut(new PrintStream(client.getOutputStream(), true));
System.setErr(new PrintStream(client.getOutputStream(), true));

Calendar.mainMethod(new String[] {});
```

Figure 7. JavaTutor Server Template Excerpt - Post Main Body Replacement

3.5.4 Server Socket Policy File/Cross-Domain Permissions

Adobe Flash clients are commonly hosted entirely within the same domain assumed by the server. This works well for most applications but when the server instructs Flash clients to redirect their lines of communication and data access to another domain, another level of

security is required and is achieve through the use of a socket policy file, which specifies site-level permissions and which ports can be accessed from certain domains. The security vulnerability presented by this approach is the potential for files from other domains to assume a user's credentials when accessing content on a domain in question. Thus, Adobe recommends administration oversight on the usage and content of cross domain policy files.

As to why a cross domain policy file was needed for the JavaTutor prototype system, when the JavaTutor web application would spawn a new process for running client code, it would be configured with a port that was different from the port assumed by the JavaTutor web application. For example, a port of 8400 could have been used to host the JavaTutor web application and would correspond to the JavaTutor login and IDE components. Another port of say 4223 could have been used for running the client code, which corresponds to a different domain. The cross domain file used in shown in **Figure 8**. This file was placed into the Tomcat `/webapps/ROOT` directory, which was all that was required for clients to access their programs.

```
<?xml version="1.0" encoding="UTF-8"?>
<cross-domain-policy
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://www.adobe.com/xml/schemas/PolicyFile.xsd">
  <site-control permitted-cross-domain-policies="all"/>
  <allow-access-from domain="*" to-ports="4000-8000"/>
</cross-domain-policy>
```

Figure 8. Cross Domain Socket Policy

3.5.5 Limitations

Arguments normally provided on a command line as input into a program were not accounted for in this initial prototype system. However, standard input and output could be performed via direct code implementation. This allowed for students to develop code that could elicit and obtain user input in a program-specific manner. Command line arguments for a program being executed within the context of an IDE are typically handled through data-persistent means and recalled at later times. For example, Microsoft Visual Studio and Eclipse both couple command line arguments to different configurations (*e.g.*, debug, release, etc.).

An additional limitation to interactive standard input and output was the type of Java IO readers and writers that could be used from the Java IO package. Buffered readers and writers proved to be troublesome in that explicit flush attempts were required for pushing data over streams, which could not easily be inserted or modified on submitted user code. Thus, operating upon the client socket input stream in its raw form and by wrapping the client socket output stream with the `java.io.PrintStream` type configured with auto flushing enabled proved to be the best solution while avoiding having to make adjustments to client code.

3.6 Adobe Messaging Services

Textual communication between the tutor and the student was done by using the Messaging Service, an extension of the core messaging framework, specifically for using the publish-subscribe support over multiple Adobe Flex clients. Message Producers create and send (*i.e.*, Publish) messages to the server, where they are routed to subscribed message Consumers (**Figure 9**). Tutors and students simultaneously acted as a message producer and consumer. Whenever the student would add or remove text in the code editor or type in text to the chat message followed by a pressing of the send button (or pressing of the keyboard enter button), a new message would be created and published to the server. The server would then asynchronously notify the subscribed tutor and student message consumers, where they would add the received message content to their respective code editor or chat message window. The following sections describe in detail Adobe Flex messaging and how it was used in JavaTutor.

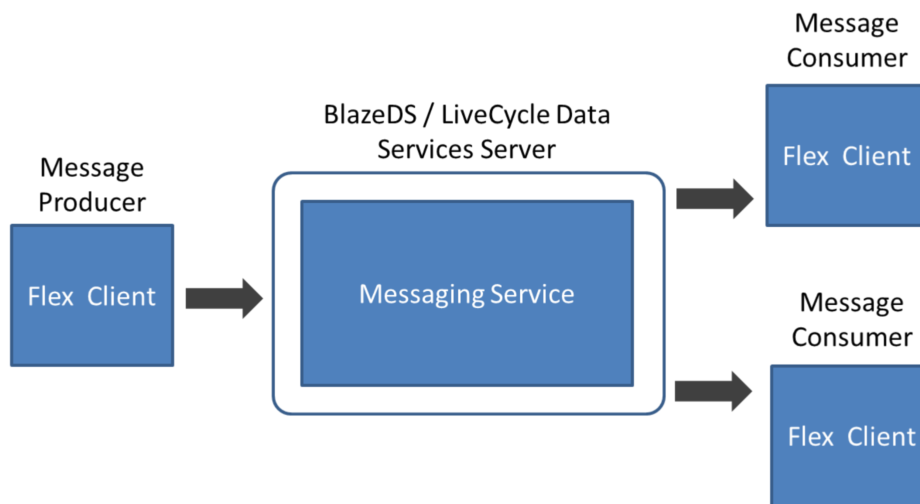

Figure 9. Adobe BlazeDS/LiveCycle Flex Message Flow - Producer to Consumer(s)[3]

3.6.1 Adobe Flex Messaging Architecture

The messaging architecture is comprised of channels, destinations, data adapters, producers, and consumers. **Figure 10** provides a high-level overview of the messaging architecture with each of these components discussed in detail in the sections immediately following.

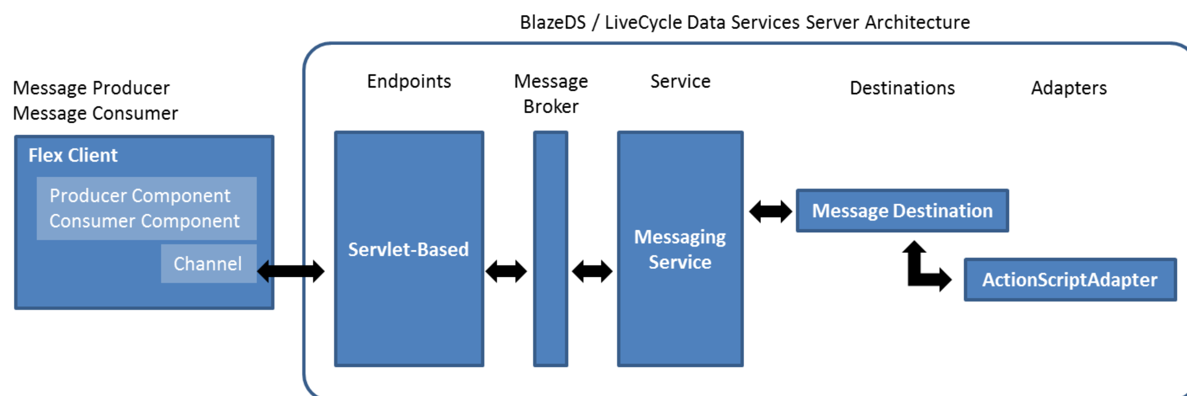

Figure 10. BlazeDS/LiveCycle Data Services Server Architecture[4]

---

[3] Modified from
http://help.adobe.com/en_US/LiveCycleDataServicesES/3.1/Developing/images/lm_basic_message_arch.png

[4] Modified from
http://help.adobe.com/en_US/LiveCycleDataServicesES/3.1/Developing/images/lm_message_arch_popup.png

3.6.2 Message Protocols

Adobe messaging services support three messaging protocols, namely the Real Time Messaging Protocol (RTMP), Hypertext Transfer Protocol (HTTP), and the Action Message Format (AMF). RTMP uses a dedicated socket connection between the server and a client and is used in the push-model where a server pushes data directly to clients. HTTP is the networking protocol used for the World Wide Web and follows a request/response model for data transfer. AMF is a binary protocol developed by Adobe System Inc. that serializes ActionScript object graphs. AMF encoded objects are representative of the state of an application and are used to persist such state between two endpoints for communicating in a strongly-typed manner. HTTP and AMF channels, when configured with polling enabled, continuously poll the messaging server for new messages when one or more consumers (*e.g.*, tutors and students) are subscribed for receiving messages. Real-time messaging is achieved by using streaming protocols or non-streaming protocols with polling enabled.

JavaTutor used the AMF and HTTP protocols for all forms of data transfer and communication. AMF was chosen over RTMP due to the fact that AMF is sent over HTTP whereas RTMP is sent over TCP. In some environments, RTMP will be blocked (*e.g.*, firewalls) where AMF will not due to it being treated as HTTP traffic. Availability of existing HTTP-based authentication methods also played a part in why JavaTutor used the AMF networking protocol.

3.6.3 Message Channels

Messages sent and received by tutors and students to and from the messaging server are sent over "Channels." Channels enable communication between Adobe Flex clients and the messaging server and correspond one-to-one with server-side endpoints. Different types of channels support the protocols discussed in Section 3.6.2 and can be configured for enabling polling, streaming, and secure data transfer. Channels are configured in an XML file named *services-config.xml* that is read at startup of an application and contains an ordered listing of channel entries to be used for communication. A top-down ordering of channels is assumed and each channel is tried for availability, up until the first channel becomes available or until

all channels have been tried. JavaTutor declared six channels in its services-config.xml file and were a combination of streaming and non-streaming channels, AMF and HTTP channels, and secure and non-secure channels. **Table 2** lists all of the channels specified in JavaTutor *services-config.xml* file, specifically the channel identifier and type.

Table 2. JavaTutor Top-Down Prioritized Channels

Channel ID	Channel Type
javatutor-streaming-amf	mx.messaging.channels.StreamingAMFChannel
javatutor-amf	mx.messaging.channels.AMFChannel
javatutor-secure-amf	mx.messaging.channels.SecureAMFChannel
javatutor-polling-amf	mx.messaging.channels.AMFChannel
javatutor-http	mx.messaging.channels.HTTPChannel
javatutor-secure-http	mx.messaging.channels.SecureHTTPChannel

3.6.4 Message Adapters

BlazeDS allows clients to access a variety of persistent data storage systems including, but not limited to, the Java Message Service (JMS), Enterprise Java Beans (EJB), ColdFusion components, and applications. Messaging adapters provide the connectivity to these storage systems and can be extended to handle other data sources not covered under the existing adapters. The `ActionScriptAdapter` type is the default data adapter used in BlazeDS and is used specifically for allowing interaction between Adobe Flex clients using the producer/consumer model. Since JavaTutor used only the producer/consumer model, the default `ActionScriptAdapter` was sufficient for all data persistence needs.

3.6.5 Message Destinations

Message destinations represent the location on the messaging server where messages are sent and transmitted. Message producers publish messages to destinations that are subsequently sent out to consumers who are subscribed to receive those messages. It is common practice for message publishers not to be aware of specific consumers that may be interested in the published message. Destinations are defined and configured in either the services-config.xml

file or the messaging-config.xml file and may include settings specifying which data adapter and channel(s) to be used by the destination.

JavaTutor used two message destinations to isolate the different types of message content exchanged between a tutor and a student. One destination was used for code editor messages while another destination was used for textual chat messages. This simplified message event handling for both the message producer and consumer by having event handling code separate from one another. For example, in the code editor, every key press event had to be immediately processed whereas textual chat messages only required processing when the user pressed the "Send" button (or *enter* button on the keyboard).

3.6.6 Message Service

Expanding on the core messaging framework is the message service, where two messaging models are supported, namely the publish/subscribe model and the message push model. Under the publish/subscribe model, multiple Adobe Flex clients are subscribed as consumers to message destinations that are published to by other message producers. The push model allows for the message server to push data out to clients without having explicit requests come in from those clients. The message service bridges clients and destinations together by maintaining a list of available message destinations and which clients are subscribed to those destinations. Configuration of the message service is done through the XML configuration files where messages can be configured to be transported over certain message channels.

CHAPTER 4

Tutorial Sessions

A qualitative study and analysis of an initial JavaTutor prototype system was the central goal of this work. Having user input and feedback was therefore essential for identifying functionality, behavior, and specific components of the JavaTutor IDE that were either missing or required further improvement. To support this effort, a pilot study consisting of four tutorial sessions was held in order to field test the prototype JavaTutor system. While not an adequate sampling for performing any statistical research, the tutorial sessions were meant to elicit user responses. The remainder of this chapter further describes the tutorial sessions.

4.1 Participants

Each tutorial session was comprised of four participants who assumed one of three roles: a tutor, a student, or a researcher. Two researchers were present in each tutorial session with each being paired with either the tutor participant or the student participant. The tutor and student roles required their corresponding participants to engage one another in an attempt to solve a given problem-solving task (see Section 4.4), with the tutor providing instructional support to the student. The researcher role required administering the tutorial session by providing all materials, answering any questions the participant may have, and observing and documenting any notable findings. All participants were drawn from the NCSU CSC IntelliMedia research group. No compensation was offered and participation was completely voluntary.

4.2 Participant Interaction

Interaction between participants was dependent upon respective roles assumed. Researchers were allowed to communicate openly with their paired tutor or student, specifically to address questions regarding the tutorial session, relevant documents, the procedure, or the programming problem. Questions intended to elicit any information about or leading to the

identification of either the student or tutor were not permitted. Both researchers followed the same administrative procedures and were nearly identical in the material reviewed, with the exception of having additional instruction provided by the researcher paired with the tutor for the purposes of discussing access and usage of a programming problem solution. Aside from serving as a guide and an information resource when needed, researchers were directed to observe and document student and tutor interaction with the JavaTutor system.

Interaction between tutor and student participants was constrained to two-way textual chat and real-time, remote collaboration with the JavaTutor IDE. Tutor and student participants were instructed to communicate with one another through the textual chat component, specifically for the purposes of initial acknowledgement and subsequent dialogue regarding the programming problem. Tutor participants were instructed to provide assistance on the usage of the Java programming language and on how to go about formulating a solution to the programming problem. Role-specific user names and time stamps accompanied each textual chat entry entered by the tutor and student participants. The JavaTutor IDE provided a real-time view of the student authoring Java code to the tutor, which provided instant feedback regarding student progress and provided insight as into how the student was attempting to solve the programming problem. Every character entered or removed from the code editor was immediately replicated in the corresponding participant view, regardless of assumed role. This included individual key press events, copy and paste events, and caret positioning and automatic scrolling into view events. For the purposes of this study, tutors were not allowed to take control of the Java source code editor to provide their own programming input or solution. Only students were allowed to type into the code editor with the tutor only viewing what was entered or removed by the student.

4.3 Problem-Solving Task

Because the goal of each tutorial session was to have each feature of the JavaTutor web-based system utilized as much as possible, the problem-solving task was designed to have the student and tutor participants actively engaged, remotely collaborating in real time on developing a solution to the programming problem. The problem used in the tutorial sessions

was modeled after the Unix/Linux-based *cal* command line utility program, which prints a Gregorian calendar given a numerical month and year input (Appendix A). Only a subset of the actual *cal* program was modeled, which included printing a Gregorian calendar to a console window and did not recognize any of the actual *cal*-documented command line arguments (*e.g.*, -1, -3, -s, -m, -j, and -y). Having this limited scope of functionality provided enough material for the student to experience authoring a solution in JavaTutor in the allotted amount of time. Such a process included repeated cycles of code writing, building, execution, and testing, coupled with engaging a tutor via the textual communication component.

At the beginning of each tutorial session, tutors were provided with two Java files that corresponded to a solution version and a partially-complete version that would be provided later to the student as a starting point for development. Since tutors were not expected to be advanced in their understanding of the Java programming language, the solution file was provided in order to guide the tutor in assisting the student in their solution development. Given the limited amount of time to complete the task and after having initially seen the programming problem following the start of the tutorial session, having tutors diverted into developing their own solutions would detract from the interaction with the students and would lead to loss of control over how the students were to complete the task.

Three functions were explicitly identified in the problem statement for the student to complete, which corresponded to implementing Zeller's algorithm for determining the day of the week provided a year, month, and day as input, determining if a given four digit year is considered to be a leap year, and to print out a Gregorian calendar year to the standard console prompt. Following the initial tutorial session, it was found too much time was being spent on how to implement Zeller's algorithm and thus detracted from using certain JavaTutor IDE features (*e.g.*, execution, console input and output) that would be used later in the programming exercise. Subsequent tutorial sessions had their problem statement modified to remove the requirement for implementing Zeller's algorithm, along with having the partially-complete version (initially provided by the tutor to the student) pre-filled with an appropriate algorithm implementation.

4.4 Procedure

Prior to the actual tutorial sessions taking place, potential participants were polled for their availability using the web-based Doodle system to broadcast a polling schedule showing open dates and times. After gathering the dates and times provided by each potential participant, a schedule was formed to accommodate having the exact required number of participants be present, which amounted to four (1 student, 1 tutor, and 2 researchers). Each participant was notified via email about the date, time, and location for their assigned tutorial session. By having two separate rooms for each tutorial session, the student participant was directed to appear in the first room while the tutor participant was directed to appear in the second room. This kept the two participants physically separated and thus, did not bias the study by having knowledge of either participant's identity be known to each other. The two remaining researcher participants were assigned to either room with only having a requirement of serving as either a student or a tutor in a previous tutorial session. Assignment of student and tutor roles were random and the corresponding participants were not screened ahead of time for their background or knowledge of the Java programming language.

4.5 System Configuration

Two computers were used in each tutorial session, one for the student participant and the other for the tutor participant. Each computer was configured with Microsoft Internet Explorer web browser (version 8.0) and Adobe Flash Player (version 10). The computer used by the tutor hosted the JavaTutor server, which was only done as a result of having two computers present (*i.e.*, a third computer would have been preferred to host and run the JavaTutor server while having both tutor and student participants running on their own computers).

CHAPTER 5

Results and Analysis

5.1 User Interface and System Behavior

5.1.1 Tutor versus Student Environments

As a prototype system, a single client-side development environment was created that would enable a student to author, compile, and execute Java code. This environment was mirrored and led to an exact replica for the tutor to view in real time. Thus, no distinction was made on how the environment should be rendered regardless of the participant being either a student or tutor. Identifying participant role(s) was possible for customizing the environment as each user login was associated with zero or more roles in the database. This led to having certain user interface features being displayed in the tutor environment that were not applicable in the tutorial sessions. Two buttons were displayed for the student to press, one labeled “Build” for compiling their code and another labeled “Run” for executing their code (see **Figure 11**). Since tutors were directed to provide instruction only and never assume control over the student’s code either through direct editing, compilation, or execution, there was no need for either of the buttons to be displayed. Valuable screen real estate was thus occupied and did not serve to benefit the tutors.

Despite the extraneous displaying of these two buttons, in some cases, this may prove to be a useful feature. Communicating instructional content using textual chat alone may be insufficient and could benefit from direct coding. A tutor may prefer to “take over” control from a student in order to express an idea or example by authoring, compiling, and running while not providing an exact solution for a programming task. To do this, invoking the process for compilation and execution requires some form of activation. JavaTutor provides these command buttons to fulfill this need while attempting to be as non-intrusive as possible. Regardless of which buttons were displayed to whom, having role-based functionality dynamically provided (and rendered) to end users is preferred.

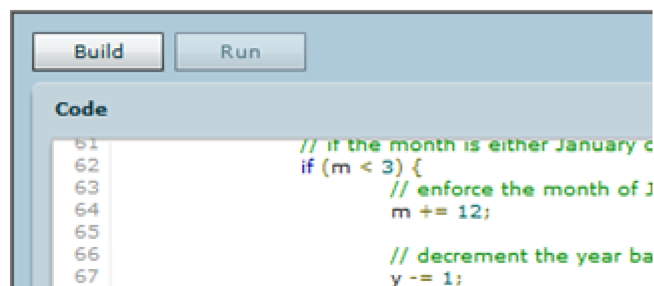

Figure 11. JavaTutor IDE Command Buttons

5.1.2 Syntax Highlighting Responsiveness

Tutors provided the "shell" of the *cal* program to be completed by the student. Shortly after students began filling in the missing code sections, it was discovered that the syntax highlighting feature was slow to respond. This was first observed when large amounts of text would be inserted or removed, as in the case when the tutors would paste the *cal* program template to the students in the code editor. Upon first receiving the code template from the tutors, students would initially see the code but with a standard black/grey font (*i.e.*, in non-syntax highlighted format). The code editor would then immediately follow with creating a pseudo-thread to perform the syntax highlighting. While this approach worked for smaller amounts of text, it was less than ideal for code that expanded beyond the visible area of the code editor. ActionScript does not support multi-threading.

At the time of development, a timer-based, pseudo-thread implementation was the only known solution for simulating multi-threading in a Flash-based environment. Harui's `PseudoThread` class offered a solution that relied on an event-based model for thread management and behavior. Handling these events helped to control runtime state and operation of the simulated thread, specifically for invocation of the function registered at pseudo-thread creation time. In the context of JavaTutor, this function corresponded to the syntax highlighting and thus was directly affected by the runtime performance of the pseudo-thread. The `PseudoThread` class defined the amount of time the function callback should be allowed to run as the time elapsed (in milliseconds) since the start of the ActionScript Virtual

Machine (AVM) via the `flex.utils.getTimer()` method, plus an offset computed from the frame rate of the display stage of the system manager,[5] as shown in **Figure 12**.

$$\alpha = getTimer() + \frac{1000}{\beta}$$

Figure 12. PseudoThread Elapsed Time Computation

Where β is the display stage frame rate (in seconds), ranging from 0.01s to 1000s, and α represents the number of milliseconds that should elapse before terminating the simulated thread. This resulting time was inversely proportional to the display stage frame rate and allows for systems that offer lower frame rates more time for thread invocation. The larger the frame rate, the smaller the elapsed time, and vice versa. While this approach attempts to give equal time to function callbacks under a dynamically set frame rate, it does not help to boost performance of the code executed in the function callback.

A true, multi-threading library is believed to offer a better solution to this problem while continuing to follow the grammar-driven lexing approach. Another possibility is to apply syntax highlighting to the visible area only. In this prototype system, the entire text contained in the code editor was processed for syntax highlighting, despite a portion of the text not be visible in the client windows. This would require finding the starting and ending indices of the visible area on a per-client basis (*e.g.*, tutor and student) and extracting that text portion for further syntax highlighting.

5.1.3 Live Viewing, Scrolling, and Visibility of Work

Tutors were notified about any code editor changes made by students as they occurred. Aside from the content that changed in the students' code editors, the location of where the changes took place also had to be communicated. This helped to set the vertical and horizontal cursor and scrollbar positions. A problem discovered during the tutorial sessions was scrolling tutors' code editor windows into a viewable range that contained the students' cursor

[5] The system manager "manages" the application and the display stage represents the main drawing area of the application window.

position, specifically at the bottom of the viewable range. Students would typically have their cursor located at a position that enabled several lines of code to be visible immediately above and below the cursor position.

Students did not type code at the top and bottom boundaries of the viewable range. Tutors' viewable range was set dynamically via the `spark.components.scrollToRange()` method of the code editor `spark.components.TextArea` class, which was the only solution found at the time of development to bring the code being edited into a viewable range for the tutors. This solution has shown that it is not ideal for future development as tutors would have to always look at the bottom of their code editor to see the student's changes. Tutors could not view surrounding lines of code for establishing context. A better approach should be developed to account for both the students' cursor position and their vertical and horizontal scrollbar positions. Scrollbar positioning is directly affected by window dimensions and would also need to be communicated from students to tutors for proper scrolling into viewable range.

5.1.4 Student/Tutor Code Editor Window Dimensions and Line Numbering

The JavaTutor IDE layout was designed to be grouped into four separate spaces for functional components to reside. With this layout, students and tutors could adjust the divider bars between components as they saw fit. Disallowing this functionality was contemplated as the `spark.components.TextArea` control used for the code editor wrapped the text located at the end of each line (*i.e.*, text did not continue into the non-visible range on the same line). Text expanding beyond the horizontal viewable area of the TextArea control wrapped but was reported as a single line in the line numbering code in the code editor TextArea skin[6]. Visually, a user would see this displayed as a line continuation with the line number on the left only being present for the top line and nothing for the continuation line. On its own, the line number skin worked as expected. When coupled with other position-dependent functionality, problems began to surface. Cursor and scrollbar positioning does not treat customized, single line continuations over multiple physical lines as a single line. Instead,

---

[6] Skins provide clean separation of logic and visual elements of a component.

each line occupied is treated as a line in use. Thus, a single line spanning two lines of space would be treated as one line from the point-of-view of the skin but as two lines from the point-of-view of the cursor positioning. When communicating reference locations in their code editor windows between one another, student and tutors could ultimately refer to different locations. To alleviate this, the divider bar positions could be communicated so as to allow opposite users the opportunity to have their environments change in response. This option is rather stringent and could lead to difficulty for the users to keep track of layout and content. This approach would be best suited for view-only/read-only JavaTutor IDEs and not really ideal for more collaborative, distributed development. A more preferred approach would be to update the `spark.components.TextArea` class to automatically horizontally scroll to the right on the same line, allowing the viewable range to move along in the same direction as the user is typing (*e.g.*, left-to-right) and include a horizontal scroll bar when needed. The information produced by this approach would be communicated to the opposite end user for incorporating into their environments and adjusting accordingly.

5.1.5 Data Input and Output

The JavaTutor IDE was prototyped to work only within a command-line environment, where students would see output from the Java compiler following compilation of their code to the actual execution of their code with all input and output prompts being directed through the console window. This window was defined as a TextArea control and handled all input and output for the programming task. Two issues emerged from the tutorial sessions that were noted by the students. First was the cursor position in the console window when the program prompted the student to enter data. When the console window would gain input focus from the student following the prompt for data, the cursor would be positioned at the beginning of the line containing the message prompt instead of being positioned at the end of the message prompt. This occurred for all prompts for input and caused and extra step to be taken by the students when entering data on the command line. A solution to this problem would be to track the length of text displayed in the console window and position the cursor accordingly,

with respect to any explicit code references for carriage returns and/or line feeds (*e.g.*, `System.out.print()`, `System.out.println()`).

The second problem was tied to having a single line of text wrapped into subsequent line(s) and the resulting line numbers that followed. When students submitted code with one or more wrapping lines to the compiler, any errors found in the code would be reported back to the student but with line numbers that differed from those in the code editor. This problem would be alleviated by the correction recommended above to allow for text to span horizontally beyond the visible range.

5.1.6 Chat Window Rendering

Text-based chat messaging between students and tutors was quite simple to build using the Adobe Flex messaging service. Rendering the chat messages that move through the messaging service was a bit more challenging. Custom text formatting and layout was needed for differentiating between senders of chat messages and for proper vertical scrolling of the list control in the chat window. Following the first tutorial session, which in fact served as a pilot test, it was discovered that chat messages would not scroll into a viewable area once the chat window had vertically filled up with chat messages. To address this issue, a custom rendering approach was needed for drawing list-based items in the correct location of a hosting list-based control.

Adobe Flex provides the infrastructure for customized rendering of list-based items by using the `spark.components.supportClasses.ItemRenderer` class. This class was used to provide a customized `ItemRenderer` that helped to support scrolling of chat message entries into the viewable area. It contained text formatting (*e.g.*, color, boldness) and line spacing for increasing the padding to the contained text message, in order to assist in separation between adjacent chat message entries in the list control for improved readability. Additional customized rendering included applying a border around the chat entry and nesting visual components for positioning content.

5.1.7 Adobe Flash Player Debugger and Runtime Notifications

Any machine that has the Adobe Flash Player debugger version installed will prompt users with messages about warnings and errors that occur at runtime. In the context of the tutorial sessions, this occurred and was found to be an intrusive pop-op notification that distracted a tutor from the task at hand. The solution to this would be to simply have the non-debugger version of the Adobe Flash Player installed on all of the machines that would be running the JavaTutor client.

5.2 Participant Actions and Programming Task Logistics

5.2.1 Starting and Acknowledgements

Participants noted that after reading through the programming task document, actually starting the process of interacting with the other participant using the JavaTutor IDE was a bit unclear. Participants were logged into the system prior to their arrival, which positioned the participants to come into the tutorial session with an empty development environment view already set to use. When a user logged into the system, a chat message was sent automatically on his/her behalf to announce to other users that he/she had just logged into the system. Since JavaTutor had two users (*e.g.*, student and tutor), whichever user logged in first would see their automatic message, plus the automatic message from the other user (when they came online). The user that logged in last would see their automatic message but not that of the user already logged in. This proved troublesome for users starting their interaction with one another as there were no separate indicators showing which users were online. Once both users acknowledged one another via the chat component, both user names appeared in the chat message history and helped to establish the presence of both users. Following this, the tutorial sessions progressed as expected.

5.2.2 Programming Task Code Template

Tutors were allowed to view a solution to the programming task for helping them guide students to a working solution. This included a single Java source code file that was the same as what the students started out with (*i.e.*, a partially complete version of the programming

solution) but already complete. Since the goal of the tutorial sessions was to elicit feedback from participants to improve the JavaTutor system, having tutors spend their limited amount of time developing a solution on their own would have detracted from their time with the students and the JavaTutor environment. The solution file was already opened in a text editor for tutor participants to reference prior to them starting their tutorial session. This was not ideal in that tutor participants had to leave the JavaTutor environment for referencing this material. It would have been preferred to have a single encompassing project to house all relevant files. As a prototype system, JavaTutor was developed to work on only a single file and could be expanded to handle multiple files.

5.2.3 Scope of Student Code Modifications

Students were free to develop their solutions to the programming task and were provided a partially complete Java source code file as a starting point. The programming task was designed to be completely self-contained and did not require any additional resources, libraries, or classes. Only completing the existing empty methods were required of the students. What was not expected and did occur were students going outside of these functions to leverage additional functionality. For example, students would look to import additional classes to help in arriving at a solution. Another example was a student looking to use the Java API and delegated their request to their tutor. This tutor then questioned the researcher present about going outside of the JavaTutor environment for fulfilling this request. External aids could be integrated to alleviate the need for referencing beyond the limits of JavaTutor. In some end-user environments, access to such aids may be limited or unavailable entirely, thus leaving JavaTutor to fulfill this need.

5.3 Participant Knowledge

Participants were not formally screened for prior knowledge of the Java programming language, except to discover self-described levels of confidence in programming with Java. Tutors were required to have some amount of prior knowledge whereas students did not. This yielded varied levels of interaction between students and tutors, especially in the amount and type of interaction and in the programming task completion rate. In one tutorial session, the

student was discovered to be advanced in programming in Java and interacted with the tutor for acknowledgement and Java API information requests. Beyond this, the student was quite autonomous and did not require programming assistance from the tutor. As a result, very little interaction took place and led to the chat component to not be used as much as desired. The student progressed through the programming task in the smallest amount of time taken by any other student participant and was able to use the execution feature of the environment to run his/her code, a step some participants failed to reach. In another tutorial session, it was discovered the student had very little knowledge of the Java programming language and ultimately led to very little progress being made. At the time of scheduling the tutorial sessions, the intent was for students who had at least some basic knowledge of the Java programming language would be able to work through the programming task with some assistance. It was not expected that students would sign up to take part without having any programming experience. This level of knowledge in the Java programming language yielded more textual chat messages being exchanged between the student and tutor but failed to get the student to the point of performing any compilation attempts over different parts of the programming task, much less initiating any program execution attempts.

5.4 Programming Task

Feedback from students indicated the programming task used during the tutorial sessions was ideal in that its level of difficulty was appropriate and the amount of prior knowledge required by each participant was in line with what would be expected of first year introductory programming students. Participants noted the use of certain programming constructs such as functions (*e.g.*, parameter passing, invocation, etc.), for loops, array access and iteration, and input and output, which represents what introductory Java programming classes usually cover. What was also noted was the lack of using any object-oriented development methodologies to solve the problem, which was attributed to the size and scope of the programming task. Since this prototype version of JavaTutor operated only on a single “file,” it was not possible for users to logically separate content into other files, which is commonly done for representing data types in an object-oriented fashion.

5.5 Recommended Improvements

5.5.1 External Aids and Information Sources

Students and tutors indicated having access to additional "help" resources would be beneficial while working in JavaTutor including an integrated code-assist feature for dynamically examining and inserting the contents of a data, such as what is currently found in the Eclipse IDE (*e.g.*, Content Assist) and in Microsoft Visual Studio (*e.g.*, IntelliSense). These tools enable rapid auto-completion, documentation, and disambiguation between variable, function, and class names. Since JavaTutor has been developed using Java, a natural extension would be to use the Eclipse Content Assistant and have it operate over the network and streamed into and out of the JavaTutor client.

Another feature mentioned several times by participants was the need to browse the Java API, particularly for finding information describing data types and their available interface methods. In many different types of development environments, the practice of resourcing materials found outside of the environment (*e.g.*, Internet web sites, searches, forums, blogs, tutorials, etc.) is common. In fact, many development resources are web-based and are best left provided by their supporters and maintainers. Having the Java API simply replicated within JavaTutor would not be an efficient use of resources as valuable screen real estate would be further partitioned, network channels would incur additional data transmission costs, and the overhead of additional processing would detract from the JavaTutor IDE performance.

5.5.2 Additional Development Tools

Some students expressed their desire for additional editing and development tools that would have allowed them to be more productive during their tutorial sessions. The only actions that could be taken in JavaTutor with respect to Java code authoring was typing, text cut & paste, source code compilation, and execution. Additional tools that could enhance JavaTutor include code formatting, error and warning markers along with suggested corrections, runtime configuration management (*e.g.*, debug versus release), integrated debugging and support for runtime variable, memory, and stack tracing. The initial version of JavaTutor is a

prototype system and contains a very limited set of actions that can be taken. Absence of additional development tools was not intentional as the initial version of JavaTutor was to show the possibilities of a remotely collaborative, distributed programming environment. Other suggested improvements included integration of a "Scratch Pad" for keeping notes, code, or other relevant content available for later use.

5.5.3 Multiple File Management and Organization

The JavaTutor prototype system contained a single TextArea HTML control for authoring Java code within the context of a single file. To be considered a viable system, JavaTutor must be able to scale for handling more than a single file. Development environments usually handle many files, logically grouped into variable levels of depth. Even most modern text editors provide tabbing capability for editing multiple files. This is an essential part of any development environment and would help scale JavaTutor for larger, more complex systems. Most development environments allow for some type of "project" or "solution" to be created for containing all relevant files and directories. For example, Eclipse creates Java projects that contain source directories, which correspond to the Java package structure of the source code. Microsoft Visual Studio groups C++ header and source files into their own project-sub directories. Not only must JavaTutor provide support for multiple files, multiple nested directories should also be provided. JavaTutor could adopt this approach and build upon it by providing customized programming task document organization and delivery to students and tutors.

CHAPTER 6

Conclusion

One-on-one human tutoring is an exceptionally effective pedagogical approach for effective student learning. Intelligent tutoring research and relevant supporting systems have attempted to deliver a level of instruction equal to or beyond that found in traditional human tutoring. The area of introductory computer programming has received much attention over recent years and has witnessed the emergence of programming-centric intelligent tutoring systems, many of which bring together a student with some form of intelligent tutor, be that a simulated or actual human tutor. The presentation and delivery of instructional material presented in these systems vary in their approach. From learning environments that include customized user interfaces that navigate a student along a controlled sequence of steps (*e.g.*, filling in missing code, function tracing, resulting values) to those that are entirely open-ended where students are free to author their own solution, they all seek to provide an effective computer-based learning experience.

6.1 Summary

Development and evaluation of a remotely collaborative, distributed learning system for learning to program using the Java programming language is the goal of this thesis. Research and exploration of current technologies was performed to determine what would meet the requirements of such a system. Primary areas of consideration that drove the choice in which technologies to adopt included what would provide asynchronous data transfers (*e.g.*, text messaging, layout and position data, code editor content), which rich internet application technologies (*e.g.*, Adobe Flash, Microsoft Silverlight, Eclipse Rich Ajax Platform (RAP)) would be most supported, available, and extensible for integration of other technology, ability to provide remote collaboration in near-to-actual real time, and program execution in a distributed environment. Through human-to-human tutoring studies, the prototype system was evaluated and participant feedback was obtained. Results from these studies yielded behavioral, functionality, and visual items to be addressed for improvement and/or correction

to the prototype system. Upon completion of the tutoring studies, it was shown that a real-time, remotely collaborative system for learning to program using the Java programming language was indeed possible and under the employed technologies, could exist as an effective remote tutoring system.

6.2 Limitations

Evaluation of the JavaTutor prototype system was done using feedback obtained from a low number of tutorial session participants. The extent to which the findings obtained from these participants could generalize to other systems for establishing any significant, commonly occurring areas of interest could not be established. Participant-specified items to be addressed for correction and/or improvement were noted for further system improvements. The background of each of the participants was not screened prior to taking part in the tutorial sessions. This could have affected a tutor's ability to provide effective tutorial instruction and could have affected a how a student progressed using the system and the level of engagement with the tutor.

6.3 Future Work

Following the work performed in this thesis, future work should address issues of scalability in several different areas of the system. First, the speed of syntax highlighting was greatly affected as a result of operating on larger source code files. Designing a viewable-area-only approach would be recommend, coupled with either improvements to or replacement of the existing simulated threading behavior. Second, operating on more than one source code file is quite common in nearly all areas of software development. Design and development for improved code management and configuration should be addressed, specifically for providing project-type organization for grouping artifacts (*e.g.*, multiple files, resources) and project-level settings (*e.g.*, classpath, dependencies). Third, for improvements in extraction of Java code segments, it is recommended to employ a Java code lexer and parser, which would adhere to a well-defined and finite grammar. Isolation of parsing has already been achieved and would easily be integrated into the current JavaTutor system. Fourth, student program execution requires a more robust and scalable solution for handling larger numbers of

simultaneously running programs. Security issues remain as student code is not isolated in a per-student sandbox environment. Finally, users of different roles use JavaTutor and should have adaptive individual learner environments. Users would have features that are relevant to their work only and not be hampered by unnecessary or irrelevant parts of the system.

6.4 Concluding Remarks

This work reported in this thesis was driven by the objective of helping to create a tutorial environment that would support remote collaboration between two users in real time and to demonstrate the feasibility of the approach investigated. This work represents a meaningful step toward the design of a viable, robust, and scalable software development environment for learning to program using the Java programming language.

GLOSSARY

AJAX – Asynchronous JavaScript and XML

AMF – Action Message Format

CSS – Cascading Style Sheets

DBMS – Database Management System

DOM – Document Object Model

HTML – Hypertext Markup Language

IDE – Integrated Development Environment

ITS – Intelligent Tutoring System

J2EE – Java 2 Platform, Enterprise Edition

JDK – Java Development Kit

JMS – Java Message Service

JRE – Java Runtime Environment

JSON – JavaScript Object Notation

JSP – JavaServer Pages

JVM – Java Virtual Machine

MXML – Macromedia XML

POJO – Plain Old Java Object

RIA – Rich Internet Applications

RMI – Remote Method Invocation

Remoting – Used for symmetric and asymmetric communication over a network (*e.g.*, one-way messaging, asynchronous method callbacks, method invocations)

SOAP – Simple Object Access Protocol

Skin – A visual mechanism for specifying the visual aspects of an Adobe Flex component.

TCP – Transmission Control Protocol

XML – Extensible Markup Language

REFERENCES

Adobe. "Cross-domain Policy File Specification | Adobe Developer Connection." *Adobe.com*. Adobe. Web. 23 Sept. 2011. <http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html>.

Ai, H., Tetreault, J. R., & Litman, D. J. (2007). Comparing user simulation models for dialog strategy learning. *Proceedings of the North American Chapter of the Association for Computational Linguistics Human Technology Conference, Companion Volume,* Rochester, New York. (pp. 1-4).

Aleven, V., Popescu, O. & Koedinger, K. R. (2003). A Tutorial Dialog System to Support Self-Explanation: Evaluation and Open Questions. In U. Hoppe, F. Verdejo, & J. Kay (Eds.), *Proceedings of the 11th International Conference on Artificial Intelligence in Education, AI-ED 2003* (pp. 39-46). Amsterdam, The Netherlands: IOS Press.

Aleven, V., McLaren, B., Roll, I., & Koedinger, K. (2004). Toward Tutoring Help Seeking: Applying Cognitive Modeling to Meta-Cognitive Skills. *Proceedings of the 7th International Conference on Intelligent Tutoring Systems,* (pp. 227-239).

Bloom, B. S. (1984). The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher, 13*(6), (pp. 4-16).

bOtskOOl. *Online C/C++ Compiler*. Computer Software. Web. 23 Sept. 2011. <http://www.botskool.com/online-compiler>.

Boyer, K. E., Dwight, A. A., Fondren, R. T., Vouk, M. A., & Lester, J. C. (2008). A Development Environment for Distributed Synchronous Collaborative Programming. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education,* Madrid, Spain. (pp. 158-162).

Boyer, K. E., Ha, E. Y., Phillips, R., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2010). Dialogue Act Modeling in a Complex Task-Oriented Domain. *Proceedings of the 11th Annual SIGDIAL Meeting on Discourse and Dialogue,* Tokyo, Japan. (pp. 297-305).

Boyer, K. E., Lahti, W., Phillips, R., Wallis, M., Vouk, M. A., & Lester, J. C. (2010). Principles of Asking Effective Questions to Improve Student Problem Solving. *41st SIGCSE Technical Symposium on Computer Science Education,* Milwaukee, Wisconsin. (pp. 460-464).

Boyer, K. E., Phillips, R., Ha, E. Y., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2010). Leveraging Hidden Dialogue State to Select Tutorial Moves. *Proceedings of the NAACL Workshop on Innovative use of NLP for Building Educational Applications,* Los Angeles, California. (pp. 66-73).

Boyer, K. E., Phillips, R., Ingram, A., Ha, E. Y., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2010). Characterizing the Effectiveness of Tutorial Dialogue with Hidden Markov Models. *Proceedings of the 10th International Conference on Intelligent Tutoring Systems,* Pittsburgh, Pennsylvania. (pp. 55-64).

Boyer, K. E., Phillips, R., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2009). Investigating the Role of Student Motivation in Computer Science Education through One-on-One Tutoring. *Computer Science Education, 19*(2), (pp. 111-135).

Boyer, K. E., Phillips, R., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2009). The Impact of Instructor Initiative on Student Learning: A Tutoring Study. *Proceedings of the 40th SIGCSE Technical Symposium on Computer Science Education,* Chattanooga, Tennessee. (pp. 14-18).

Boyer, K. E., Phillips, R., Wallis, M. D., Vouk, M. A., & Lester, J. C. (2008). Learner Characteristics and Feedback in Tutorial Dialogue. *Proceedings of the Third ACL Workshop on Innovative use of NLP for Building Educational Applications,* (pp. 53-61).

Boyer, K. E., Ha, E. Y., Wallis, M. D., Phillips, R., Vouk, M. A., & Lester, J. C. (2009). Discovering Tutorial Dialogue Strategies with Hidden Markov Models. *Proceedings of the 14th International Conference on Artificial Intelligence in Education,* Brighton, U. K. (pp. 141-148).

Comet Daily. *Comet*. Web. 23 Sept. 2011. <http://cometdaily.com/>.

Evens, M. W., Brandle, S., Chang, R., Freedman, R., Glass, M., Lee, Y. H., Shim, L. S., Woo, C. W., Zhang, Y., Zhou, Y., Michael, J. A., & Rovick, A. A. (2001). CIRCSIM-Tutor: An Intelligent Tutoring System Using Natural Language Dialogue. *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference* Miami University, Miami, OH, March 30-April 11, 2001 (pp. 16-23).

Fossati, D., Di Eugenio, B., Brown, C., Ohlsson, S., Cosejo, D., & Chen, L. (2009). Supporting Computer Science Curriculum: Exploring and Learning Linked Lists with iList. *IEEE Transactions on Learning Technologies, 2*(2), (pp. 107-120).

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005). Java Language Specification, Third Edition. *Oracle Technology Network for Java Developers*. Web. 23 Sept. 2011. <http://java.sun.com/docs/books/jls/third_edition/html/syntax.html>. Addison Wesley.

Graesser, A. C., Chipman, P., Haynes, B. C., & Olney, A. (2005). AutoTutor: An Intelligent Tutoring System with Mixed-Initiative Dialogue. *IEEE Transactions on Education, 48*(4), (pp. 612-618).

Graesser, A. C., Jackson, G. T., Mathews, E. C., Mitchell, H. H., Olney, A.,Ventura, M., Chipman, P., Franceschetti, D., Hu, X., Louwerse, M. M., Person, N. K., & TRG (2003). Why/AutoTutor: A Test of Learning Gains from a Physics Tutor with Natural Language Dialog. In R. Alterman, & D. Hirsh (Eds.), *Proceedings of the 25rd Annual Conference of the Cognitive Science Society*. (pp. 1-5). Mahwah: Erlbaum.

Harui, A. Threads in Actionscript 3. Adobe. Web. 23 Sept. 2011. <http://blogs.adobe.com/aharui/2008/01/threads_in_actionscript_3.html>.

Hazel, S. *codepad*. Computer Software. Web. 23 Sept. 2011. <http://codepad.org/>.

Ho, C. W., Raha, S., Gehringer, E., & Williams, L. (2004). Sangam: A Distributed Pair Programming Plug-in for Eclipse. *Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange,* (pp. 73-77).

Jetty - Jetty WebServer. Program Documentation. *Jetty WebServer*. Mort Bay Consulting. Web. 23 Sept. 2011. <http://jetty.codehaus.org/jetty/>.

JSON. Program Documentation. *JSON*. Web. 23 Sept. 2011. <http://json.org/>.

Jurado, F., Molina, A. I., Redondo, M. A. & Ortega, M. (2009). Learning to Program with COALA, a Distributed Computer Assisted Environment. *Journal of Universal Computer Science*, 15(7), (pp. 1472-1485).

Khen, G., Khen, D. E. & Koubi, G. *CodeRun.* Web. 5 April 2011. <http://www.coderun.com>.

Lane, H. C. & VanLehn, K. (2005). Teaching the tacit knowledge of programming to novices with natural language tutoring. *In S. Fitzgerald and M. Guzdial (Eds.) Computer Science Education, Special issue on doctoral research in CS Education*, Swets & Zeitlinger, 15(3), (pp. 183-201).

Lane, H. C. & VanLehn, K. (2004). A Dialogue-based tutoring system for beginning programming, In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*, AAAI Press, (pp. 449-454), Miami Beach, FL.

Lane, H. C. (2004). Natural Language Tutoring and the Novice Programmer, *dissertation, University of Pittsburgh, Dept. of Computer Science*.

Litman, D., & Silliman, S. (2004). ITSPOKE: An intelligent tutoring spoken dialogue system. In *Proceedings of the North American Association for Computational Linguistics and Human Language Technologies Conference (NAACL HLT),* (pp. 5-8).

Microsoft. *Using IntelliSense*. Program Documentation. Web. 23 Sept. 2011. <http://msdn.microsoft.com/en-us/library/hcw1s69b.aspx>.

Natsu, H., Favela, J., Moran, A. L., Decouchant, D. & Martinez-Enriquez, A. M. (2003). Distributed Pair Programming on the Web. In *Proceedings of the Fourth Mexican International Conference on Computer Science (ENC'03)*, IEEE Computer Society, (pp. 81-83).

Ninja Otter Inc. *Compilr. Online C#, PHP, C, C++, Ruby, VB, Java IDE & Compiler*. Computer Software. Web. 23 Sept. 2011. <http://compilr.com/>.

Oracle. *JDK 6 Java Compiler (javac)-related APIs & Developer Guides*. Computer software. *The Java Programming Language Compiler - Javac*. Version 1.6. Oracle. Web. 23 Sept. 2011. <http://download.oracle.com/javase/6/docs/technotes/guides/javac/index.html>.

Oracle. *NetBeans*. Computer Software. Web. 23 Sept. 2011. <http://netbeans.org>.

Nokia. Qt Creator IDE and tools. Computer Software. Web. 5 April 2011. <http://qt.nokia.com/products/developer-tools>

Ryan, W. (2007) Web-Based Java Integrated Development Environment, *dissertation, University of Edinburgh Division of Informatics*. Web. 23 Sept. 2011. <http://www.willryan.co.uk/dissertation.pdf>.

Sasikumar, A. AS3SyntaxHighlight - ActionScript Library That Can Syntax Highlight Code in Many Languages - Google Project Hosting. *Google Code*. Google. Web. 23 Sept. 2011. <http://code.google.com/p/as3syntaxhighlight/>.

Sasikumar, A. Syntax Highlighting in AS3. Web. 23 Sept. 2011. <http://anirudhs.chaosnet.org/blog/2009.01.12.html>.

Schulze, K. G., Shelby, R. N., Treacy, D. J., Wintersgill, M. C., VanLehn, K., Gertner, A. (2000). Andes: An intelligent tutor for classical physics. *The Journal of Electronic Publishing*, University of Michigan Press, Ann Arbor, MI, 6:1, http://www.press.umich.edu/jep/06-01/schulze.html.

ShiftCreate. (2010) ShiftEdit, The Online IDE. Computer Software. Web. 23 Sept. 2011 <http://shiftedit.net>.

Silveria, R. A. & Viccari, R. M. (2002). Developing Distributed Intelligent Learning Environment with JADE – Java Agents for Distance Education Framework. In *Intelligent Tutoring Systems: 6th International Conference, ITS 2002*, Biarritz, France and San Sebastian, Spain, (p. 105). Berlin, Heidelberg: Springer.

Sykes, E. R., & Franek, F. (2003). A Prototype for an Intelligent Tutoring System for Students Learning to Program in Java. In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education* (pp. 78-83).

Sykes, E. R., & Franek, F. (2004). Inside the Java intelligent tutoring system prototype: parsing student code submissions with intent recognition. In *Proceedings of the IASTED International Conference on Computers and Advanced Technology in Education* (pp. 613-618). Innsbruck, Austria, January 2004.

Sykes, E. R. (2006). Design, Development and Assessment of the Java Intelligent Tutoring System, *dissertation, Brock University*.

Sykes, E. R. (2005). Qualitative Evaluation of the Java Intelligent Tutoring System. In *Journal of Systemics, Cybernetics, and Informatics*, 3(5), (pp. 49-60).

Sphere Research Labs. *Ideone*. Computer software. *Online IDE & Debugging Tool C/C++, Java, PHP, Python, Perl and 40 Compilers and Interpreters*. Sphere Research Labs. Web. 23 Sept. 2011. <http://ideone.com/>.

The Dojo Foundation. *The Dojo Toolkit*. Program Documentation. *Unbeatable JavaScript Tools - The Dojo Toolkit*. Version 1.6. Web. 23 Sept. 2011. <http://dojotoolkit.org/>.

The Eclipse Foundation. *Eclipse*. Computer Software. Web. Wed 23 Sept. 2011. <http://www.eclipse.org>.

Tillman, N., Halleux, P., Schulte, W., Bjørner, N., Gvervo, N., Thummalapenta, S., Nepomnyashchaya, K., Lakhotia, K., Kong, S., Clark, A., Vanoverberghe, D., Anand, S., Xie, T., Csallner, C. & Schütt, T. *Pex for Fun*. Computer Software. Web. 5 April 2011. <http://www.pex4fun.com>

Tschalär, R. *JXXX Compiler Service*. Computer software. *JXXX Compiler Service*. Web. 23 Sept. 2011. <http://www.innovation.ch/java/java_compile.html>.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R. H., Taylor, L., Treacy, D. J., Weinstein, A., and Wintersgill, M. C. (2005). The Andes physics tutoring system: Five years of evaluations. In: G. I. McCalla and C.-K. Looi (Eds.), *Proceedings of the Artificial Intelligence in Education Conference*. Amsterdam: IOS.

VanLehn, K. (2008). The Interaction Plateau. Keynote talk presented at the *Proceedings of the 9th International Conference on Intelligent Tutoring Systems,* Montreal, Canada.

APPENDICES

APPENDIX A: Select Materials for Study

Table 3. Programming exercise (Version 1) for tutorial sessions

JavaTutor Programming Exercise Study

cal is the command to display a simple ASCII-based calendar on UNIX and Unix-like operating systems. (See figure below). When provided with a month and year, the cal program displays a monthly calendar with the days of the week properly aligned in a "day of the week" column format. In this exercise, you will be implementing a Gregorian calendar that mimics a portion of the *cal* calendar utility features in the Java programming language, for a single month/year combination.

```
     April 2011
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

An empty "Calendar" class has already been created for you containing three empty methods for determining the day of the week, determining if a year is or is not a leap year, and a main method that will accept user input for the month and year and print out the corresponding monthly calendar. You will need to complete these three methods in order to complete this project.

Here is list of items that you must resolve:

1. In the Calendar class, there exists a method named dayOfWeek that accepts a month, day, and year and calculates the day of the week, where 0 = Saturday, 1 = Sunday, 2 = Monday, and so on. You must complete this method to assist in the displaying of the monthly calendar. This method is actually an implementation of Zeller's Congruence, which is as follows:

$$h = \left(q + \left\lfloor \frac{(m+1)26}{10} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor + 5J \right) mod\ 7$$

where

h is the day of the week
q is the day of the month

m is the month with 3 = March, 4 = April, …, 14 = February
$K$is the year of the century
J is the century

2. Being a Gregorian calendar, leap years must be recognized and handled accordingly. In a leap year, one additional day is added for the calendar year (*e.g.*, 366 days). February 29th is the date assumed every four years for this additional day and is what will be assumed in this programming exercise. You must complete the function named isLeapYear that will determine whether or not a given year is considered to be a leap year. As a reminder, a leap year is recognized as being evenly divisible by 4 and not by 100, as well as being evenly divisible by 400.

3. To glue the application together, you will complete the main method that will accept two integers as input that correspond to a month and year to display. Upon accepting these input parameters from the user, you should determine whether or not the year submitted is a leap year by using the isLeapYear method. If you have a leap year, the number of days in February will go from 28 to 29.

   Hint: Use the days array in the Calendar class to keep track of this additional day.

4. Printing a formatted calendar will involve three parts:

   1. Printing the month and year header
   2. Printing the names of each day of the week
   3. Printing the days by number (*e.g.*, integer starting at 1)

   You may use the month array in the Calendar class to print out the month name, followed by the year entered by the user. Names of days should be abbreviated to the first two letters of the day name (see above figure). To print each of the days, you must iterate over the days array for the month and print each value in the correct "day of the week" column. Leveraging the value obtained from the dayOfWeek method would be recommended in doing so as it will aid you in determining where to being printing the calendar and to know when to being a new week (*hint: think modulus*).

Table 4. Programming exercise (Version 2) for tutorial sessions

JavaTutor Programming Exercise Study

cal is the command to display a simple ASCII-based calendar on UNIX and Unix-like operating systems. (See figure below). When provided with a month and year, the cal program displays a monthly calendar with the days of the week properly aligned in a "day of the week" column format. In this exercise, you will be implementing a Gregorian calendar that mimics a portion of the *cal* calendar utility features in the Java programming language, for a single month/year combination.

```
     April 2011
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

A partially completed "Calendar" class has already been created for you containing two empty methods for determining if a year is or is not a leap year and a main method that will accept user input for the month and year and print out the corresponding monthly calendar. You will need to complete these two methods in order to complete this project.

Here is list of items that you must resolve:

1. Being a Gregorian calendar, leap years must be recognized and handled accordingly. In a leap year, one additional day is added for the calendar year (*e.g.*, 366 days). February 29th is the date assumed every four years for this additional day and is what will be assumed in this programming exercise. You must complete the function named isLeapYear that will determine whether or not a given year is considered to be a leap year. As a reminder, a leap year is recognized as being evenly divisible by 4 and not by 100, as well as being evenly divisible by 400.

2. To glue the application together, you will complete the main method that will accept two integers as input that correspond to a month and year to display. Upon accepting these input parameters from the user, you should determine whether or not the year submitted is a leap year by using the isLeapYear method. If you have a leap year, the number of days in February will go from 28 to 29.

Hint: Use the days array in the Calendar class to keep track of this additional day.

3. Printing a formatted calendar will involve three parts:

    1. Printing the month and year header
    2. Printing the names of each day of the week
    3. Printing the days by number (*e.g.*, integer starting at 1)

    You may use the month array in the Calendar class to print out the month name, followed by the year entered by the user. Names of days should be abbreviated to the first two letters of the day name (see above figure). To print each of the days, you must iterate over the days array for the month and print each value in the correct "day of the week" column. Leveraging the value obtained from the dayOfWeek method would be recommended in doing so as it will aid you in determining where to being printing the calendar and to know when to being a new week (*hint: think modulus*).

Table 5. Participant post-tutorial session questionnaire

JavaTutor Post-Tutorial

Session Questionnaire

Tutorial Session ID: ___________
Date: ______________________
Time: ______________________
User Role: __________________
Researcher: _________________

Instructions

After the participant has completed their Java programming exercise, please ask the questions below to elicit user feedback regarding their experience with the JavaTutor system. Please document as much of the user's response as possible. Please write IN THE USER'S OWN WORDS.

Questions

1. What did you like best about the system?
2. What did you like least about the system?
3. How would you describe your experience developing Java code using this system?
4. What existing functionality would you like to see improved upon?
5. What was the one thing you wanted to do with the system that you were not able to do?
6. How would you describe your overall experience with this system?
7. Do you have any additional comments?