SUPPORTING INTRODUCTORY TEST-DRIVEN LABS WITH WEBIDE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Thomas Dvornik

December 2010

© 2010

Thomas Dvornik

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Supporting Introductory Test-Driven Labs
with WebIDE

AUTHOR: Thomas Dvornik

DATE SUBMITTED: December 2010

COMMITTEE CHAIR: David Janzen, Ph.D.

COMMITTEE MEMBER: John Clements, Ph.D.

COMMITTEE MEMBER: Alex Dekhtyar, Ph.D.

Abstract

Supporting Introductory Test-Driven Labs with WebIDE

Thomas Dvornik

WebIDE is a new web-based development environment for entry-level programmers with two primary goals: minimize tool barriers to writing computer programs and introduce software engineering best practices early in a student's educational career. Currently, WebIDE focuses on Test-Driven Learning (TDL) by using small iterative examples and introducing lock-step labs, which prevent the student from moving forward until they finish the current step. An initial set of labs and evaluators were created as examples of how to use WebIDE and were used in a pilot study in a CS0 course where students were split into two groups, one that used WebIDE and one that didn't. The WebIDE group showed a significant improvement in performance when writing a simple Android application. Additionally, among students with some programming experience, the WebIDE group was more proficient in writing unit tests.

Acknowledgements

I would like to thank Olga Dekhtyar for her help in planning and assessing the pilot study, and Halli Meth for developing early labs and giving initial feedback to the functionality of WebIDE.

I would like to extend thanks to Dr. John Clements for his involvement and great criticism throughout the entire project. Also, to my last committee member, Dr. Alex Dekhtyar, for joining my committee and being willing to use WebIDE in Winter 2011.

Special thanks goes to my advisor, Dr. David Janzen, whose continuous support and enthusiasm kept me motivated, whose constant feedback kept me focused, and whose countless hours spent working with me to perform the pilot study in his CS0 class during Fall 2010 quarter at Cal Poly, without which, my thesis would not be complete. But in all seriousness, the office was huge.

Last but certainly not least, I would like to thank my mother, Debbie, and father, Charlie, for their constant support. My sister Tanya, who ensured I never got homesick. And my brother Todd, without whom, I would not be at Cal Poly or in the Master's program; and who always went above and beyond to help me in every way possible.

Additionally, this material is based upon work supported by the National Science Foundation under Grant No. 0942488. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Tables	ix
List of Figures	x
1 Introduction	1
2 Related Work	5
2.1 Pedagogical Desktop Tools	6
2.1.1 BlueJ	6
2.2 Interactive Web-based Tools	7
2.2.1 Environment for Learning to Program	9
2.2.2 Turing Craft’s CodeLab	13
3 Background	15
3.1 Terminology	15
3.2 Android OS	18
3.3 Test Driven Learning	18
3.4 Web Technologies	21
3.4.1 Google Web Toolkit	21
3.4.2 The Cloud	22
4 Architecture	26
4.1 Services	26
4.1.1 Lab Service	27
4.1.2 User Service	32
4.1.3 Evaluator Service	32
4.1.4 File Servlet	36

4.2	Data Store and Twig	37
4.3	Client Side and SmartGWT	39
4.4	Course Management	39
5	Labs	41
5.1	TDL in WebIDE	42
5.2	Android: Tic Tac Toe	44
5.3	Labs in Pilot Study	51
5.3.1	Java Basics	51
5.3.2	Classes	51
5.3.3	Facebook: Selection	53
5.3.4	Facebook: Functions	53
5.3.5	Facebook: Iterations	55
6	Evaluators	57
6.1	WebIDE Evaluator Toolkit	58
6.1.1	Evaluator Class	58
6.1.2	JavaClass Class	62
6.2	Internal Evaluators	64
6.2.1	Regex	64
6.2.2	Arithmetic	66
6.3	External Evaluators	66
6.3.1	Java	67
6.3.2	Classes	68
6.3.3	Android	71
7	Results	72
7.1	Data Collection	73
7.1.1	Graded data	74
7.1.2	Surveys	74
7.1.3	Lab Logs	75
7.2	Hypothesis	77
7.3	Empirical Analysis	78

7.4	Lab Statistics	81
7.5	Qualitative Analysis	85
7.6	Threats to Validity	91
8	Future Work	92
9	Conclusion	94
A	WebIDE’s RelaxNG Lab Definition	96
	Bibliography	100

List of Tables

7.1	Student Grade Averages	75
7.2	Course Lab Completion Times	76
7.3	WebIDE Lab Evaluation Statistics	77
7.4	WebIDE Lab Completion Times	78

List of Figures

1.1	WebIDE home page	2
1.2	WebIDE lab with sample error feedback	3
2.1	BlueJ Environment	8
2.2	ELP Environment	12
2.3	WebToTeach Environment	14
3.1	Function implementation step in Tic Tac Toe	17
3.2	Java Basics Lab	17
3.3	WebIDE step requiring student-written examples	20
3.4	WebIDE step requiring student-written tests	21
4.1	WebIDE Architecture Overview	27
4.2	Lab Service Class Diagram	29
4.3	Lab Structure Class Diagram	30
4.4	Element to Object Mapping	31
4.5	User Data and Service Classes	33
4.6	Evaluator Service Classes	34
4.7	Evaluator Request and Response Classes	36
4.8	UserInfo Class	38
4.9	A professor's course management console in WebIDE	40
5.1	WebIDE step requiring students to write a full unit test class. . .	42
5.2	Possible TDL flow within WebIDE	43

5.3	QR code in the Tic Tac Toe Lab	45
5.4	Lab Example: Lab Element	46
5.5	Lab Example: Review Step	47
5.6	Lab Example: Segment and Evaluator	49
5.7	Lab Example: Lab Table	50
5.8	Java Basics Lab	52
5.9	Facebook: Selection Lab	54
5.10	Facebook: Iterations lab	56
6.1	WebIDE ETK getSegment() Example	59
6.2	WebIDE ETK getRequiredSegment() Example.	60
6.3	WebIDE ETK replaceSegmentValues() example	60
6.4	WebIDE ETK Java Scaffolding Example	61
6.5	WebIDE ETK: replaceClassName() example	63
6.6	WebIDE ETK: Running JUnit test	65
6.7	Using the regular expression evaluator.	66
6.8	A lab using the Arithmetic Evaluator	67
6.9	JavaCompiler: Inheritance Example	68
6.10	Multiple Evaluator Example	70
6.11	Custom Evaluator Example	70
7.1	Average Student Grades	79
7.2	The average reported time, in minutes, for students to complete each course lab.	80
7.3	The number of attempts, successes, errors, and tries after for all WebIDE labs	82
7.4	The average time, in minutes, it took students to complete each WebIDE lab.	83
7.5	Chart: Students found error messages helpful	86
7.6	Chart: Students liked lab content	87
7.7	Chart: Labs helped students learn Java	87
7.8	Chart: Students liked WebIDE	88
7.9	Chart: WebIDE helped students learn Java	88

7.10 Chart: WebIDE helped students learn Android	89
7.11 Chart: Students did not want to use WebIDE	89
7.12 Chart: WebIDE helped students learn Android	90

Chapter 1

Introduction

Students often struggle with the first few weeks of beginning computer science classes. In addition to learning programming concepts and syntax, students typically work in an unfamiliar computing environment, whether it be an integrated development environment (IDE), or a text editor with a command-line compiler. Students can quickly get behind in class material and have difficulty catching up, causing them to struggle for the rest of the course.

Instructors often respond to this problem by formulating one-off solutions that are targeted to a particular class. These take a variety of forms, and range from single-page HTML documents to full-fledged course management systems. After working with several of these tools, we decided to leverage their hard work to build a more general framework, WebIDE.

WebIDE helps students during the first few difficult weeks by offering a one-button interface (Run!) in a familiar context (a web browser). Figure [1.1](#) shows WebIDE’s home page, where students can type code, then press “Compile and Run” to see compile errors or execution results. This example shows the results

Figure 1.1: WebIDE home page

of a simple hello world program. Additionally, WebIDE focuses on applying test-driven learning (TDL)[18], a pedagogical approach for teaching with a test-driven development (TDD) approach. Previous studies indicate benefits from applying TDD, but note challenges of actually getting fledgling programmers to write code in a test-first manner[21, 9]. In our labs, WebIDE solves this by moving students through labs in a lock-step fashion, requiring them to write examples and tests before implementing solutions.

Additionally, WebIDE can help teach other software engineering (SE) techniques to students early in their education. For example, a plug-in could be

Figure 1.2: WebIDE lab with sample error feedback

created to allow for creation of UML diagrams within WebIDE. Studies have shown that TDL can be applied in entry level classes without removing current course material[19]. We hope this same principle can be applied and shown using other SE techniques using WebIDE.

WebIDE is built on top of Google Web Toolkit (GWT). The WebIDE framework is not chained to a particular target language or lab layout, and allows instructors to formulate their own labs, to modify those of others, and to contribute their own lab evaluators—written in any language they choose—to provide more helpful error messages to their students. This functionality was explored and reported in an early prototype[7].

Labs can be created for an individual class or shared between classes and instructors. We are creating a repository of labs for all educators to use, and anticipate that other faculty will contribute labs as well. Figure 1.2 demonstrates a step in a WebIDE lab. In this case, the student is asked to type a header for a Java function, but has forgotten the return type; an appropriate error message is shown in red.

WebIDE is a three year project sponsored by the National Science Foundation (NSF). Dr. David Janzen and Dr. John Clements are co-Principal Investigators

for the WebIDE project. Throughout this thesis, I refer to Dr. Janzen, Dr. Clements, and myself as “we.” This thesis focuses on the design and implementation of the WebIDE framework, as well as an initial set of labs and evaluators. We also performed a pilot study for an early assessment of the environment to determine the direction of WebIDE. I did the full implementation of WebIDE, the Android lab, evaluators, and most of the data analysis.

In section 2, we talk about related work and how WebIDE differs from other environments. Section 3 discusses the role of TDL, current web technologies, and some terminology used through this thesis. Section 4 gives a brief overview of the architecture design and its capabilities. Section 4.4 shows WebIDE’s course management capabilities. The initial set of labs and evaluators are discussed in sections 5 and 6. The pilot study and results are analyzed in section 7, then I finish with future work and conclusions in section 8 and 9.

Chapter 2

Related Work

Automated tutors exist for a variety of academic fields. Samples include Biology/Genetics¹, Mathematics², and Physics³. Many of these tools have been evaluated with promising results. For instance, Warnakulasooriya et al.[34] reports that their web-based automated Physics tutor improves student time to completion, reduces the need for hints, and improves the number of correct answers all by approximately 15%.

Not surprisingly, computing faculty and researchers have also built many software tools to support students as they learn to program. Valentine[32] reports that 22% of the CS1/CS2-related SIGCSE conference papers from 1984 to 2003 included software tools to aid learning. Some of the more popular tools include visualizations[16], Karel micro-worlds[28, 3, 2], automated assessment tools[10], and pedagogical development environments such as DrRacket[14], Alice⁴, and Scratch⁵.

¹<http://biologica.concord.org/>

²<http://www.assistments.org/>

³<http://www.masteringphysics.com/>

⁴<http://alice.org>

⁵<http://scratch.mit.edu>

2.1 Pedagogical Desktop Tools

Many pedagogical environments were developed for desktop computers, such as Greenfoot, DrRacket, JGrasp, and BlueJ. I will describe BlueJ in greater detail as an example of such systems.

2.1.1 BlueJ

BlueJ is a Java environment specifically for teaching object oriented programming to beginners. The classes and methods are displayed as objects in an UML diagram as seen in Figure 2.1. The diagram allows students to interact with objects. Students can click on objects and change parameters. They can also bring up the source code for a class. Students can create test objects for classes by directly calling the methods within the original object. BlueJ enables a good overview of all the classes, helping students get the general concept.⁶ [24, 25]

BlueJ is different than WebIDE in many ways. Specifically, BlueJ is not web-based and does not implement fill-in-the-blank labs. However, it does provide a simple interface with an abstraction over traditional methods of programming; in this case, a one button interface for compiling. BlueJ also provides many additional features without destroying the simplistic user interface. Some additional features include a fully functional Java debugger, integration with Java Docs, and direct feedback with individual objects. These additional features are something that WebIDE can learn from, and hopefully include in future releases.

BlueJ originated in 1999 and has been evaluated several times with positive results. The first study consisted of 26 students out of the 109 students taking a

⁶<http://www.bluej.org>

CS1 course [4]. Those 26 students were separated into two groups of 13 students each. The first group used BlueJ and the second group used the traditional method of a plain text editor with limited integration to the JDK. On average, group 1 outperformed group 2 by 9% on all quizzes and exams.

The study also got feedback from 43% of the participating students. Almost all of the feedback was positive. Students enjoyed the exercises and felt that BlueJ clearly presented the concepts associated with object oriented programming. There were also negative comments about the environment running slowly.

The next study evaluated the switch from C++ to BlueJ in a first quarter programming unit [15]. The quarter's failure rate decreased by 4% across 333 students. Students also evaluated BlueJ a 3.5 rating on a 5 point scale.

The last study consisted of 40 students taking a survey using a scale of 1 (high) to 7 (low) [33]. On average, students rated BlueJ a 3 on its effectiveness to help them learn. However, they rated the BlueJ environment a mean of 4.3.

2.2 Interactive Web-based Tools

Most web-based coding environments work on web-based scripting languages, such as PHP and Javascript. For example, W3's school⁷, Google's API Playground⁸, Cloud9⁹ and JSBin¹⁰ let users evaluate Javascript. However, several web-based systems can compile and run code. For example, Google's Go play-

⁷http://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_charat

⁸<http://interactivesampler.appspot.com/>

⁹<http://www.cloud9ide.com/>

¹⁰<http://jsbin.com/>

Figure 2.1: BlueJ Environment

ground¹¹ is specific to the Go programming language, while ideone¹² works with over 40 languages.

Coderun¹³ is a web-based IDE with all the features an IDE user would expect, such as syntax highlighting, code completion, and auto deployment. Unlike WebIDE, Coderun focuses on application development instead of educational labs. Users can create, run, and debug ASP.NET, Silverlight, and Facebook applications within Coderun. Additionally, Coderun supports PHP and Javascript.

A few excellent web-based systems are similar to WebIDE. Parlante’s CodingBat¹⁴ does adopt a somewhat test-driven approach, although students do not write tests and the system is limited to a set of small, focused exercises. Edwards’ Web-CAT[12] web-based automated grading tool assumes student creation of automated (presumably test-driven) unit tests, but it provides no support for interactive labs. More general web-based development environments have begun to emerge. Samples include WeScheme[35], ShiftCreate¹⁵, Lively Wiki[26], and a system by Azalov[1] that automatically generates lab exercises.

2.2.1 Environment for Learning to Program

The Environment for Learning to Program (ELP) is a web-based system with fill-in-the-gap exercises as seen in Figure 2.2. The ELP was developed by Nghi Truong, Peter Bancroft and Paul Roe from Queensland University of Technology in Australia. The authors wrote several papers that describe the idea, implementation, and results for the ELP [30, 31, 29].

¹¹<http://golang.org/>

¹²<http://ideone.com/>

¹³<http://www.coderun.com/>

¹⁴<http://codingbat.net>

¹⁵<http://edit.shiftcreate.com/>

The ELP addresses the high failure rates of the entry level computer science classes. It allows students to program in a simplistic development environment by eliminating the “difficult” parts. One example is interacting with the compiler. Students can click on the compile button, also seen in figure 2.2, instead of running the command-line compiler.

Like WebIDE, exercises are described in an easy XML language. The professor can specify a solution, and decide what code should be displayed or filled in. Comments and hints can also be specified in the document. However, this does not take a test-first approach or allow for third party content.

ELP does offer a wide range of analytic tools built right into the environment. These tools help determine when the student has written complex or invalid code. The analysis phase is split into two parts, a static analysis and dynamic analysis. The static analysis makes use of Software Engineering Metric Analysis and Structural Similarity Analysis. The dynamic analysis uses both black and white box testing. The complexity of the analysis works great for small programs but breaks down as the program or exercises get larger and more complex [31]. For example, structural similarity analysis works by turning the students program and the instructors solution into pseudocode, then comparing them and returning feedback. This becomes difficult because the amount of differences between the student’s and professor’s code grows exponentially as the size and complexity of the program grows. Additionally, this analysis assumes the instructor’s solution is “better” than the student’s solution, which may not always be the case.

Although WebIDE does not offer this level of analysis, anyone can add this feature by creating an evaluator that any lab author can plug into. This will also allow professors to analyze different parts of the lab in different ways.

Tutors can provide annotations on student programs through ELP, allowing them to help many students at once. At the same time, the students feel like they are still getting individual, instant feedback. Exercises and student code are stored online, however, the results of the student programs have to be downloaded and run locally. ELP does this by returning a Jar for the student to run. WebIDE allows the code to be run externally and then displayed in the browser.

There were two different experiments with the ELP to evaluate its effectiveness. The first study consisted of 30 students in the 5th week of Software Development 1 during part of a catch up course [30]. Only 12 students took a survey after using the system. The survey asked if the ELP helped them write and understand programs. It also asked if they would like to use the system for the remainder of the course. All of the survey results were positive, but not statistically valid.

The second experiment consisted of an evaluation on a course using ELP [31]. This survey was also given during the 5th week and consisted of 46 students; 63% whom evaluated the ELP positively. The evaluation included space for comments, which proved to be more useful. Several comments suggested that the “fill in the gap” exercises are great for struggling students. There were several comments that stated the anytime, anywhere characteristic of the ELP was the most useful feature. Some students said the ELP made it easy to write a program and that they could not compile a program before using ELP. This feedback is encouraging for WebIDE, since it shares similar principles.

Figure 2.2: ELP Environment

2.2.2 Turing Craft’s CodeLab

CodeLabs is a web based exercise system. The environment was previously referred to as WebToTeach, as seen in Figure 2.3, before it went commercial in 1999 [27]. CodeLabs claims to be used by over 100 institutions and more than 27,000 students¹⁶. The system focuses on short examples to help the students understand concepts, which CodeLabs calls “Graduating Complexity,” and uses a custom automated code-checking system that gives students instant feedback. CodeLabs’ interactive exercises allow multiple languages, including Python, C, C++, and Java.

CodeLabs automatically saves student progress and provides a custom analysis of individual student progress to the professor. CodeLabs provides over 300 exercises for professors to organize into custom course plans. The exercises range from basic to advanced concepts. Each exercise consists of multiple parts in several different programming languages.

Like ELP, CodeLabs also does not adopt a test-driven learning approach or allow for third party content contributed by individual faculty.

¹⁶<http://www.turingscraft.com>

WebToTeach: David's C
Display A Greeting: Write a C statement that displays a greeting.

General Actions: [Log Out](#) [Message of the Day](#) [View Exercise List](#)
[Change Password](#) [Get Help](#) [View Bug List](#)

Read the instructions below and write your solution in the text area(s) at the bottom of the page. When you are finished, click on the "SUBMIT YOUR SOLUTION" button. For additional help, you may consult one of the following:

[Get HELP for using this page.](#)
[Get a HINT on solving the problem.](#)
[Take a look at the SOLUTION.](#)

Exercise Instructions:

Write a C statement that displays a greeting. The greeting should be

Hello WebToTeach!

Make sure the greeting is exactly as you see it above, including capitalization and punctuation.

[LOAD PREVIOUS submission.](#)

Write your solution to the problem in the area below:

```
printf("Hello WebToTeach!\n");
```

Figure 2.3: WebToTeach Environment

Chapter 3

Background

First, I define terms that are used throughout this thesis in section 3.1. Next, I give a quick overview of the Android OS in section 3.2, which is the focus of a lab developed in WebIDE. Section 3.3 explains the importance of test-driven learning in education. Finally, section 3.4 describes several web technologies that WebIDE uses.

3.1 Terminology

I use several terms specific to WebIDE. Below is a list of these terms and their definitions. More detailed descriptions and examples are given in later sections.

- **Editor:** An editor determines how a user interacts with input. For example, an editor can be a HTML input tag, like a textbox or textarea, or a HTML 5 canvas with Javascript for complete syntax highlighting and auto-completion.
- **Segment:** A segment is a section in a lab where a student can give input,

usually code. Every segment has exactly one editor.

- **Step:** A page of visual elements—usually textual instructions—and segments. Figure 3.1 shows instructions to implement a function, and a segment where the student must implement the function code.
- **Lab:** A collection of steps represented as tabs. Usually, a lab focuses on one concept. For example, Figure 3.2 shows all the tabs in the Java Basics lab.
- **Evaluator:** A script or program that evaluates a segment. Evaluators can be run locally or externally, and can evaluate anything in written form. For example, an evaluator could check the grammar of a sentence. Multiple evaluators can be assigned to one step or segment.
- **Lab Author:** A lab author is a person who writes labs and/or evaluators. Anyone can create labs, however, we predict that lab authors will primarily be professors at universities. Currently, lab authors only consist of the WebIDE team at Cal Poly.
- **User or Student:** A user or student refers to a person who is completing a lab in WebIDE. I switch between student and user interchangeably. Although non-students can use WebIDE, the only users in this paper are students.

CellState has the following variables.

```
private GameState mGameState;  
private int mState;
```

We need a method that will set the state of CellState. It needs to do two things:

- Set the variable mState to EMPTY
- Set the CellState image resource by calling setImageResource with the image id (this will display the image in Android).

This method is implemented for you. Here is the specification for the method.

Method: setState

Visibility: public

Parameters: int state

Return: void

Description: Set the cell state to *state*.

Here is the completed implementation of the method.

```
public void setState(int state) {  
    mState = state;  
    setImageResource(mState);  
}
```

Figure 3.1: Function implementation step in Tic Tac Toe

Figure 3.2: Java Basics Lab

3.2 Android OS

Android ¹ is Google's open-source operating system created for mobile devices with limited screen real estate and hardware resources. The OS supports multiple hardware features including touch screens, accelerometers, magnetic field sensor, cameras, GPS, Wifi, and much more. Android has been extended for use on larger devices, such as tablets and televisions. In a recent study, the Android OS passed the iPhone4 in market share during the first quarter of 2010². Developers can create and distribute applications for Android, for free. That combined with the huge market share makes Android a hot topic. WebIDE's framework does not use Android directly, but I created an Android lab for WebIDE. Additionally, the CS0 course used in the pilot study focuses on Android development.

3.3 Test Driven Learning

One of WebIDE's goals is to enable labs to follow a Test-Driven Learning (TDL) approach. TDL is an effective method to teach Test-Driven Development (TDD), an extreme programming (XP) test-first technique [22]. The idea behind TDD is to create tests before writing any code. TDD can often be defined by the following steps [8].

1. Create a test for an unimplemented unit of functionality.
2. Run all tests and make sure the new unit test fails.
3. Write code for the unit of functionality.

¹<http://www.android.com/>

²<http://www.npd.com/press/releases/press.100510.html>

4. Run all tests and check that they now succeed.
5. Refactor.

The result is higher quality code and usually fewer lines of code (LOC). TDD is more than a testing method; it is a design method. Writing tests before writing code forces the implementer to think about the functionality and design of the system. In other words, programmers work more effectively when they focus on the results of functions—the tests—before thinking about *how* those results are computed.

Stephen Edwards, from Virginia Tech, explains that the lack of testing causes students to perform “trial and error.” [11] As a result, students obtain 4 misguided views that, as Edward says, some students will never lose.

1. If the compiler accepts my code, I removed all the errors.
2. My code will work all the time if it works for a couple values.
3. I’ll just switch a few things around to see if my problems will go away.
4. My program works for the instructor’s data so I am done.

TDD resolves these issues by teaching students how to write tests first [22, 23], removing the “trial and error” approach. TDD, however, is not regularly taught early in the Computer Science curriculum. Recent studies show empirical evidence that TDD makes students more productive, earn better grades, and write clean, concise, and well tested code [17, 8, 20].

TDL extends this claim to the corresponding pedagogic statement: students learn more quickly when they focus first on the set of possible inputs to a function,

Now let's write some examples for our function.
Enter values for the parameters, and corresponding expected return values to predict the output of our function.
The first row of the table has been completed for you as an example. Fill in the remaining rows.

Birth Year	Current Year	Expected Output
1990	2010	20
<input type="text" value="1992"/>	<input type="text" value="2010"/>	<input type="text" value="18"/>
<input type="text" value="2000"/>	<input type="text" value="2011"/>	<input type="text" value="11"/>
<input type="text" value="1999"/>	<input type="text" value="2010"/>	<input type="text" value="11"/>

Good Job!

Figure 3.3: WebIDE step requiring student-written examples

and the corresponding results. Best of all, TDL has been shown to have no extra cost [19]. In other words, students can use TDL and still learn all the concepts that were originally taught, in the same amount of time.

TDL can also improve the quality of instructor feedback. In TeachScheme! workshops[13], instructors using test-driven learning in lab settings repeatedly report that by examining students' test cases *before* looking at their code, they can diagnose problems more quickly. Many instructors found that students discover their own problems after written tests.

The same things that make TDL effective in one-on-one lab interaction are even more vital in an online setting. Writing a program that checks the correctness of a student's program and offers useful feedback is very difficult. By focusing on test cases, though, the task becomes vastly simpler. Additionally, correcting these errors before the students tackle the implementation of the corresponding function can save them time and stress. Figure 3.3 demonstrates how WebIDE can require students to create correct examples and even convert them into test cases (Figure 3.4) before moving on to implementing a solution.

Figure 3.4: WebIDE step requiring student-written tests

3.4 Web Technologies

The past several years, there has been an enormous growth in web technologies, including languages, frameworks, tools, and environments. Most of these technologies allow for developers to build web applications faster than they would be able to otherwise.

3.4.1 Google Web Toolkit

Google Web Toolkit (GWT) allows developers to write complete AJAX web applications using Java. Unlike traditional Java web applications, GWT compiles front-end Java to optimized JavaScript, CSS, and HTML instead of making the developer write JSP. Additionally, GWT abstracts Java servlets for the developer by using Remote Procedure Call (RPC). External servers can not access RPCs, but RPCs enable the developer to pass any serialized object without having to deal with the HTTP request or response. Developers can allow external access by using normal Java servlets within GWT.

GWT provides many other benefits. GWT is open-source and has an active,

thriving community. Developers have access to a library of widgets, Java objects that represent UI elements. For example, a Button widget corresponds to an HTML submit button. A Button object's methods allow for DOM manipulation with the additional benefit of cross-browser support. The JavaScript, CSS, and HTML created by GWT will work on all supported browsers, consisting of Firefox, Internet Explorer, Safari, and Chrome.

Many 3rd party libraries are built on top of GWT that provide a more extensive list of Widgets. WebIDE uses a library called SmartGWT³. SmartGWT provides more widgets with more customization than traditional GWT widgets. For example, widgets can be stacked, dragged, resized, minimized, maximized, and hidden, to just name a few.

3.4.2 The Cloud

Today, the amount of applications that are being built on the “cloud” is growing rapidly. The cloud is an overloaded term that can mean many things, including the internet as a whole. I define the cloud as a location to deploy web applications that are accessible to everyone with a web browser. More importantly, the developer does not have to maintain the hardware and, theoretically, has unlimited resources. This means that, as the application grows or gets more traffic, so does the resources to handle the space and bandwidth of the application.

Educational software is perfect for the cloud because of sparse usage. Usually, educational software experiences a lot of traffic when students are given an assignment during a course in the academic year. Otherwise, the environment is taking

³<http://code.google.com/p/smartgwt/>

up a whole server and barely being used. With the cloud, an environment can use virtually no resources when idle, and use several servers when under heavy usage. WebIDE uses two different cloud services: Google App Engine and Amazon EC2. Most educational software is hosted on the developer's or institution's servers, and do not currently take advantage of cloud services.

Google App Engine

Google App Engine (GAE) is Google's cloud service. They offer "fast development and deployment; simple administration, with no need to worry about hardware, patches or backups; and effortless scalability."⁴ Not surprisingly, the integration between GWT and GAE is seamless. In Eclipse, a developer just has to press a button, enter a password, and their application is deployed and ready for use. This quick cycle of development, along with GWT's features, enabled us to build and deploy WebIDE in a very short period of time with minimal configuration.

Naturally, the application must adhere to GAE's restrictions and limitations. This can include load times, CPU cycles, URL lengths, and no system access beyond the APIs. These restrictions are understandable for free cloud service, but limit the power of WebIDE. The WebIDE architecture may require long URL parameters, lots of URL fetches, and longer than normal requests, which are characteristics of most web applications.

GAE uses shared servers. Shared servers are great when hosting multiple applications that never hit a server's capacity. However, that means that your application also relies on the speed and load of others. The theoretical unlimited

⁴<http://code.google.com/appengine/>

resources still have a limit, especially on free cloud services. Because of this, we saw extremely fast load times and extremely slow load times with no changes to the user’s actions or changes to the code base. WebIDE also ran into load problems during our experiments, which we will talk about in section 7.

Amazon Elastic Computing Cloud

Amazon Web Services (AWS) provides storage (S3), databases (RDS), and much more. Another service is Amazon’s Elastic Computing Cloud (EC2), which provides server instances for users⁵. Like GAE, users do not have to worry about the underlying hardware. Unlike GAE, users receive virtual machine instances with complete system control. EC2 gives users the ability to create already configured instances or a blank slate, allowing the user to install whatever they want.

EC2 also allows for several types of instances, which gives a user high level control over the hardware. Several instances may run on the same server box, just like several GAE applications are running on the same machine. However, a user can get a dedicated instance that runs on its own server box. A user can also create several temporary instances and load balancers to help with high loads. For example, several instances can be created when a class is using WebIDE to improve the performance.

Many WebIDE labs rely on compiling and running Java. Since GAE does not give system access to users, we use Amazon’s EC2 service to host all evaluators that compile and run Java code. However, 30 students compiling and running Java programs on the same machine can cause significant latency problems. With

⁵<http://aws.amazon.com/ec2/>

EC2, we can create a load balancer to distribute the running Java programs on multiple instances without changing any of the code.

Just like GAE, Amazon's EC2 still has drawbacks. First, there is a lot more configuration, whether it is specifying the creation of instances or customizing the configuration of an instance to suit a specific need. More importantly, EC2 is not free. This could be a big problem for educational software, which operates on a minimal, often zero, budget. WebIDE can currently support the cost, however, this may not be the case in the future.

Chapter 4

Architecture

The WebIDE architecture is focused on extensibility in two main areas. First, the lab specifications are written in a well-defined XML language, so that labs may be edited and contributed by third parties. Secondly, I completely decouple the presentation of the lab from the evaluator using a service-oriented architecture (SOA) where URLs identify evaluators. The lab sources may be hosted on an external machine, and specified using a URL as well.

Figure [4.1](#) illustrates the high level architecture of the system. The solid lines represent HTTP connections. The dashed line indicates an implicit dependency. Specifically, the lab source (XML) uses URLs that allow WebIDE to locate the evaluators.

4.1 Services

GWT, as described in section [3.4.1](#), splits their Java code into a server and client package. I created several services to communicate between the server and

Figure 4.1: WebIDE Architecture Overview

the client that use remote procedure calls (RPC). A RPC is restricted to the Java application, whereas Java servlets will accept any HTTP request. Since RPCs are only internal, the communication is always between two Java classes. All objects sent via RPC must implement `Serializable` or extend `IsSerializable`.

Almost all services and servlets interact with the GAE's data store. However, instantiating the data store manager can be slow. I created a service utility class that acts as a singleton for the data store manager. This utility class also provides a helper method for getting the user's name out of their session. Any other helper methods that all services share can be added here.

First, I will describe WebIDE's RPC services in section [4.1.1](#), [4.1.2](#), and [4.1.3](#). Then I will go over WebIDE's only servlet in section [4.1.4](#).

4.1.1 Lab Service

The client will call the lab service with the lab's location—as a URL—when a user wants to process or save a lab. First, the service calls the `LabParser` class. This class will validate the lab using a `Validator` object. In order to enforce structure and prevent ad-hoc extension, labs are written using a XML language

defined using the Relax NG[6] specification language. So, for instance, labs are specified to contain a name, an optional description, and zero or more steps:

```
start = element lab {  
  attribute name { text },  
  element description { text }?,  
  step*  
}
```

Thus, I created a RelaxNGValidator class, as seen in Figure 4.2, that uses Jing¹ to validate a lab's XML. A new validator can be implemented in the case that the lab schema changes to a different definition language, e.g. Document Type Definitions (DTD). Currently, the Relax NG Specification file's (.rng) location is hardcoded in the RelaxNGValidator and should be moved to meta data in a future release.

After the XML is validated, the XML elements get mapped to an internal lab representation, serialized, and sent back to the client. Figure 4.3 shows the class diagram for the internal structure. At the top level, we have a Lab object which contains 0 or more Steps. A step contains 0 or more LabElements. Each element in a step must extend the LabElement. When the schema incorporates more lab elements, just create a new class that extends LabElement. Both the step and segment objects have 0 or more evaluators. The evaluator contains all the necessary information to build an evaluator request.

All XML elements that do not have a corresponding class are placed in a Text object. For example, we do not have to create a bold object for `<b>hey</b>`. Instead, we can put that line in a Text object, which gets rendered by the browser. It would be beneficial to make the distinction between HTML elements and text to help prevent confusion and cross-site scripting. However, the separation can

¹<http://www.thaiopensource.com/relaxng/jing.html>

Figure 4.2: Lab Service Class Diagram

Figure 4.3: Lab Structure Class Diagram

Figure 4.4: Element to Object Mapping

drastically increase the amount of objects created. The *bold* example above would only consist of two objects, but a HTML table with a lot of formatting would not. For example, a HTML table with 10 rows and 5 columns would be 111 objects, assuming there were no additional HTML tags in the cells. Figure 4.4 shows how the number of objects starts to increase rapidly. I attempted to minimize this number for the datastore, discussed in section 4.2. So in the lab table example, I only create one Text object instead of 111 objects.

Additionally, figure 4.3 shows two classes used for storing a user's lab state. If a lab does not have a StepManager then the user is viewing the lab for the first time or is not logged in. The StepManager is automatically created when a logged in user evaluates a segment or step. As its name suggests, the StepManager manages a number of StepStates equal to the number of steps in the lab. It also keeps track of the lab name, user, start date, completed date, step names, and all segment's values. The StepState keeps track of dependencies, segments within

the step, and whether those segments are complete. With this information, a student or professor can see their progress in a lab or load the lab in the last saved state. Section 5.2 gives some XML lab specification examples.

4.1.2 User Service

The user service deals with login, using Google accounts, and WebIDE registration. When a user loads WebIDE, the UserService asks Google if the current user is logged in and stores the appropriate LoginInfo, seen in figure 4.5. LoginInfo contains an URL to Google that will either log in or log out the user, depending on their current state. If the user is not known to WebIDE, a RPC will be sent to the user service to create a UserInfo object to store WebIDE related information. The user service also handles all course management, described in section 4.4, which primarily saves and retrieves course information to the data store.

4.1.3 Evaluator Service

The evaluator associated with a given lab step is responsible for determining the correctness of student entries. To evaluate a segment or step, the client side will send an RPC to the evaluator service. The service will read the request and send it off to the corresponding evaluator. Evaluators can live in the engine itself, or on an external server. If the evaluator is on an external server, the service will pass on the request and wait for a response before sending it back to the client. Figure 4.6 shows the evaluator service class, which simply has two methods: sendEvaluatorRequest() and logEvaluatorInformation().

Since the evaluators can be hosted on any server by any author, we supply an interface for communication between the engine and the evaluator. Both request

Figure 4.5: User Data and Service Classes

Figure 4.6: Evaluator Service Classes

and response are encoded using JSON. For a segment or step to be evaluated, we must tell the evaluator where the request is coming from (the lab id) and the segment's text (what the user entered). We also allow for arguments to be passed in to give the evaluator any additional information that it may need. The JSON definition for an evaluation request can be seen below.

```
request={
  "id": "<the id>",
  "args":{"<name>": "<value>", ...},
  "textfields":{"<name>":"<value>", ...}
}
```

The success of a tutor such as WebIDE depends crucially on the ability to deliver helpful error messages, and not a simple “success” or “failure.” Accordingly, the JSON format for the evaluator's response includes a message.

```
response={
  "status": ("success" | "failure" | "internal-error"),
  "message":"<some message>"
}
```

The client side code builds an `EvaluatorRequest` object, as seen in figure 4.7². This object is sent to the evaluator service, which then converts it to JSON and creates a HTTP request. An `EvaluatorResponse` object is created from the JSON retrieved from the HTTP response and sent back to the client.

Internal evaluators are written in Java, and can be seen in figure 4.6. The request and response object are sent directly to the evaluator without being converted to a HTTP request and sent to an external server. Additional evaluators can be added by simply extending the `Evaluator` class. However, internal eval-

²Only the private variables were shown since the classes only contain setter and getter methods.

Figure 4.7: The evaluator request and response classes used between the client and the evaluator service.

uators can only be written by the WebIDE team, have to be deployed with the environment, and must conform to the environment running WebIDE.

4.1.4 File Servlet

The file servlet extends the Java `HttpServlet` class (a regular Java servlet). The purpose of the file servlet is to generate one of two comma separated value (CSV) files for the instructor: a course progress report or a lab log. The course progress report includes each student’s progress on each lab. A lab log includes all evaluator request and response information for a given student and lab. Currently, the lab logs have to be downloaded individually but could be edited to combine all lab logs in a zip before sent to the instructor.

Additionally, I created a program that converts lab data to a more useful format, which includes the number of evaluation attempts, successes, failures, and errors. It also computes the number of evaluation tries after a student finishes the segment and estimates the time it took to do the lab. I talk more about this data in chapter 7. This source code can be integrated with the file servlet to

give instructors this information without having to download and run a separate program. I hope more statistic and analysis tools will be created and integrated to evaluate student performance.

4.2 Data Store and Twig

Since we use GAE to host WebIDE, we must use GAE's data store. The data store, which is not like a relational database, is built on top of Google's Big Table. Big Table can store and retrieve petabytes of data across thousands of servers[5]. Although the queries can look similar, the amount of query operations on the data store are limited. For example, a query can not contain any join operations or conditionally select on multiple properties. Additionally, the data store has limited support for inheritance, polymorphism, and generic types.

We decided to use a third party persistence layer on top of GAE's data store to overcome these issues. Twig-persist addresses all of the above problems. Twig also supports parallel asynchronous commands, plain old java objects (POJOs), and queries with multiple properties. All of our data models are POJOs and were shown in figure 4.3. Although, it may be necessary to use an annotation in the POJO for additional information, just like Java Data Objects (JDOs). For example, we use the Key annotation to specify a primary, unique key, as seen in figure 4.8³.

If our object has a primary key, we can store and access the object using the following code.

```
UserInfo user = ...  
datastore.store(user);
```

³Getter methods are not shown.

```

public class UserInfo implements Serializable {
    private String mName;
    @Key private String mEmail;
    private String mOrgnazationId;
    private String mTitle;
    private long mRegisteredDate;
    private List<String> mEnrolledCourses;

    private UserInfo() {}

    public UserInfo(String name, String email,
        String orgnazationId, String title, long registeredDate) {
        mName = name;
        mEmail = email;
        mOrgnazationId = orgnazationId;
        mTitle = title;
        mRegisteredDate = registeredDate;
    }

    //Getter methods are below

```

Figure 4.8: The UserInfo POJO used for data persistence.

```

UserInfo retrievedUser = datastore.load(UserInfo.class, user.getEmail());

```

If the POJO does not have a primary key, or we want to retrieve on a different attribute, we can retrieve the object(s) using the following code.

```

List<UserInfo> labs = datastore.find().type(UserInfo.class)
    .addFilter(UserInfo.FIELD_NAME, FilterOperator.EQUAL, userName)
    .returnAll().now();

```

Objects must be retrieved first to be updated or deleted.

4.3 Client Side and SmartGWT

GWT’s client side uses special Java objects—called widgets—to compile highly optimized JavaScript that manipulates the DOM. GWT offers a library of widgets that include buttons, date pickers, menus, trees, rich text areas, etc. Even with a long list of widgets, GWT’s library is under-developed and lacks the ability to be customized. For example, adding a “X” to the top right of a Window widget or enabling multiple tabs to scroll was extremely difficult and counter intuitive.

Luckily, several third-party libraries address this issue. We went with SmartGWT⁴ because of their extremely active community and extensive widget showcase. SmartGWT also offers a desktop-like UI and data binding widgets. However, WebIDE is not tied to SmartGWT in any way. The model (POJOs using Twig) and controller (RPCs) are separated from the view (SmartGWT). If we decide to lose SmartGWT, we only need to implement a new view using pure GWT widgets or another third party library.

4.4 Course Management

WebIDE offers simple course management that allows students to see specific labs and professors to keep track of those students. First, users log in to WebIDE using a Google account. First time users must register with WebIDE, entering their name, organization, and whether they are a student or professor. Students can enroll in a course, see labs for a specific course, and see their current progress for each lab. All labs are saved automatically when a user is logged in.

Professors can save labs, create courses, add labs to a course, and see all

⁴<http://www.smartclient.com/smartgwt>

Figure 4.9: A professor's course management console in WebIDE

students enrolled in a course. The professor can look at each student's progress, download student logs, or even load the student's lab within WebIDE to see exactly what the student sees. Figure 4.9 shows a professor's view of a course's lab list, student's lab list, their progress, and links to view or download student logs. A student would only see *their own* lab list and progress without links to their logs.

WebIDE could take a service-oriented approach here as well, funneling information on student performance to a dedicated course management system.

Chapter 5

Labs

Several Java labs were created for use in a pilot study in a CS0 course. Lab topics included basics (data types and variables), if statements, functions, iterations and classes. Each lab incorporates TDL in some form. For example, I showed a lab where students write input/outputs in figure 3.3. We will see students write full jUnit test files (figure 5.1) during an in-depth discussion of TDL in WebIDE labs in section 5.1.

The lab presented in section 5.2 introduces students to the core concepts and structure of the Android platform. This lab does not incorporate TDD, but forces the student to build an Android application in small iterations, helping students get a small part working before moving on. Students are meant to attempt the Android lab after they completed the other labs created for the pilot study, which are described in section 5.3.

Figure 5.1: WebIDE step requiring students to write a full unit test class.

5.1 TDL in WebIDE

One primary focus of WebIDE is to teach entry level students while focusing on test driven development. WebIDE's lock-step labs can force students to write a test then implement the corresponding code in small iterations, making it perfect for TDL. Additionally, WebIDE can slowly build JUnit class examples, as seen in figure 5.2. First, students determine inputs and outputs. Then, students can write parts of the assertion test before doing the full assertion test. Next, the students can write part of a unit test followed by the full unit test. Finally, the students can write the full JUnit test class.

After the student learns about unit testing—using the above or any other method—lab authors can have students write tests in a TDD fashion. Each step can have students perform the next iteration of test and code. With improve-

Figure 5.2: Possible TDL flow within WebIDE

ments to WebIDE, lab authors will be able to take TDL and TDD even further. For example, the most requested feature among users was the ability to have a “split” view. This additional view could be used for multiple reasons, such as displaying the instruction on one side and having the student code on the other. By allowing students to code in both views, authors can—very practically—teach the iterative nature of TDD. The student can write a small failing test on the left, then immediately implement the code on the right. In WebIDE’s current state, students would have to scroll or switch tabs, depending on how long the example gets.

5.2 Android: Tic Tac Toe

The Tic Tac Toe lab does not use TDL, however, it shows the power of WebIDE by allowing students to develop a full Android application right in the browser. At the end of the lab, students are presented with a QR code that an Android device can scan to start downloading and installing the application on the device, as seen in figure 5.3. Writing an Android application using WebIDE is similar to Google’s App Inventor¹, however, this lab allows users to develop with code instead of developing through a graphical interface. Additionally, the target of this WebIDE lab is for beginner programmers while App Inventor claims to target non-programmers as well.

All labs start with a Lab element, as seen in figure 5.4. A Lab element can also specify a default path for evaluators. Lab authors will most likely use evaluators from one source. In this case, the Tic Tac Toe lab uses all evaluators hosted on the Amazon EC2 instance. The Tic Tac Toe lab also defines a description—as

¹<http://appinventor.googlelabs.com>

Figure 5.3: The QR code step in the Tic Tac Toe lab. You can scan it with your Android device!

```

<lab name="Android: Tic Tac Toe"
    defaultpath="http://184.73.238.21/webide/evaluators/">
  <description>
    In this lab, you will incrementally implement an Android
    application of Tic Tac Toe. At the end of the lab, WebIDE
    will provide you with a QR Code that you can scan with your
    phone to download your application.
  </description>
  ...

```

Figure 5.4: Lab Example: Lab Element

most labs will—that gets displayed below the lab title.

After the lab and description, there can be zero or more steps². Figure 5.5 shows the first step in the Tic Tac Toe lab, which introduces the structure of an Android application. Basic HTML elements can be used in a step for additional formatting. Not all HTML elements are allowed due to a restriction with GWT, which is described below. Limited HTML elements also ensures that labs will look consistent across different authors. However, all HTML elements can be added at any time with no changes to the WebIDE base code by editing the lab RelaxNG definition. A full list of allowed HTML elements are mentioned in appendix A.

In addition to HTML elements, there are also several lab elements that the WebIDE engine interprets. These elements are also listed in appendix A. The two most important elements are *segment* and *evaluator*. Segment defines where a student can input text—usually code—whereas the evaluator evaluates the contents of the segment—and possibly others—when the student chooses to do

²A lab with zero steps would not be very useful!

```

<step name="Android Overview">

    First, we need to discuss the structure of an Android application.

    The most important file in an Android application is the
    AndroidManifest.xml. This file specifies the name, permissions, and
    activities of an application. An activity is a class where the
    application starts.

    <br /><br />

    Android projects consist of 3 different folders:

    <ul>
        <li>Src (source)</li>
        <li>Gen (generated)</li>
        <li>Res (resources)</li>
    </ul>

    The src folder contains all of your java classes. The gen folder
    contains java classes that Android generates. The res folder contains
    all resources that an application might use, such as images, layouts,
    and values.

    <br /><br />

    Android generates ids for all resources and places static final
    references in a generated file called "R.java." If you didn't get all
    that, don't worry because you only need to know how to use them. At
    any point in your code, just type the following:

    <br /><br />

    <code>R.&lt;type of resource>;.&lt;name>;</code>

    ...

```

Figure 5.5: Lab Example: Review Step

so. Figure 5.6 shows a segment where the student must implement the `clearState` method as described above the segment. Multiple regular expressions are used to provide good feedback to the student.

A segment's *editor* attribute can control how the student inputs text. The default is a HTML text-area input box. However, a lab author can use any other available editor. Currently, WebIDE provides a CodeMirror³ editor which can be used by using the following code.

```
<segment id="Heythere" editor="codemirror" width="600" height="300">
```

CodeMirror is a JavaScript editor that allows syntax highlighting of multiple languages. The above example does syntax highlighting for Scheme. Future WebIDE releases will allow lab authors to take advantage of the other languages by simply saying “`editor=“java”`”, or whatever language CodeMirror supports.

WebIDE also allows *labtable* element. Figure 5.7 shows a *labtable* element that indents a segment to match the surrounding static code, and places a description to the right of the segment. GWT does not allow a developer to add HTML code around a Widget. For example, I could not do the following in a lab.

```
<table><tr><td><segment></segment></td></tr></table>
```

Although the example is pointless, it clearly demonstrates the restriction within GWT. This restriction forced me to create a *labtable* element so that lab authors can place WebIDE elements in a table. The format in the lab's XML may seem strange, however, it can greatly increase readability and make it easy to parse. Additionally, the author will not deal with the XML once an authoring tool is created.

³<http://codemirror.net/>

```

<!-- INPUT: Implement the function clearState -->
<b>Method:</b> clearState<br />
<b>Visibility:</b> public<br />
<b>Parameters:</b> none<br />
<b>Return:</b> void<br />
<b>Description: </b>Clear the cell state by setting the state with the id of
the empty image.
<segment id="clearState" width="500" height="100" buttonName="Check">
  <evaluator name="clearStateMethod" labid="Android: Tic Tac Toe"
    href="evaluator://RegexEvaluator">
    <!-- Check they created the method -->
    <arg name="regex" value="clearState"/>
    <arg name="failed-message" value="Don't forget to declare the
      clearState method."/>

    <!-- Check the method is public -->
    <arg name="regex1" value="public"/>
    <arg name="failed-message1" value="Your method needs to be public."/>
    ...
    <!-- Check everything is put together right. -->
    <arg name="regex11"
      value="^\\s*public\\s+void\\s+clearState\\s
        *\\(\\s*\\)\\s*\\{\\s*setState\\s*\\(\\s*EMPTY\\s*\\)\\s*;\\s*\\}\\s*$"/>
    <arg name="failed-message11" value="Not sure what you did wrong.
      Check your function definition matches the description and the
      body properly clears the cell state. Also make sure all brackets
      and parenthesis match up."/>
    <arg name="success-message" value="One method done!"/>
    <segid id="clearState"/>
  </evaluator>
</segment>

```

Figure 5.6: Lab Example: Segment and Evaluator

```

<code>
@Override
public boolean onTouchEvent(MotionEvent event) {
</code>
<labtable rows="1" cols="2">
  <add width="30" />
  <add>
    <segment id="onTouchEvent" width="550" height="200">
      //Write the function code here
    </segment>
  </add>
</labtable>
<code>
  //Pass the event on to the parent
  return super.onTouchEvent(event);
}
</code>

```

Figure 5.7: Lab Example: Lab Table

5.3 Labs in Pilot Study

In addition to the Android lab, there were five other WebIDE labs created for the pilot study. The classes labsection 5.3.2 was created by Dr. David Janzen for his CSC 123 sections on Android development. The Java basics, if statements, functions, and iterations labs—sections [5.3.1](#), [5.3.3](#), [5.3.4](#), [5.3.5](#) respectively—were designed and written by Halli Meth and revised/edited by Dr. David Janzen. The last three have a Facebook theme and are intended to be used in the first few weeks of an entry-level computer science course. The Java basics lab is intended to prepare students for the Facebook-themed labs, while the classes lab is meant to transition students to the Android lab.

5.3.1 Java Basics

The Java Basics lab will, most likely, be the first lab students will complete if they have no programming experience. It focuses on data types, operations, expressions, and variables. Figure [5.8](#) shows a screen shot of the “Values, Operations, Expressions” step.

5.3.2 Classes

The Classes lab focused on the implementation and instantiation of classes, as well as inheritance. Students start by writing a Game class which would contain two team’s scores. The Game class contained three static methods to determine if the home team one, the visitor team one, or the game was a tie. Then, students implement a Match class that contained instantiations of the Game class. A match contained a five game series. The team who won three out of the five was

Figure 5.8: The Values, Operations, Expressions step on the Java Basics Lab.

determined as the winner. Finally, students extend the Match class to alter the rules of the match. On each step, students had to write unit tests to test the functionality of their class, as was seen in figure 5.1.

5.3.3 Facebook: Selection

This lab concentrates on if statements by having students group Facebook friends into categories. Students have to assign users to their corresponding group given different ages and name initial ranges. By the end of the lab, students should have a good understanding of if-else blocks and compound conditional statements. Figure 5.9 shows the second step in the selection lab where students demonstrate they understand the requirements of the next step.

5.3.4 Facebook: Functions

The Facebook: Functions lab has users calculate age given a birth date and the current date. Students started by calculating an approximate age with only the year, and worked their way up to the exact age with year, month, and day. TDL was heavily used in this lab by having students determine inputs and outputs and write unit tests before starting the function. At the end of this lab, students should understand a function signature, how to call a function, and how to write a function.

Figure 5.9: The Age Group - Input/Output step on the Selections Lab.

5.3.5 Facebook: Iterations

This lab had students plant seeds on a farm, which is based on the popular Facebook game, FarmVille⁴. Students have an array of “plots” and need to iterate through the array to plant seeds in different patterns, as seen in figure 5.10. At the end of this lab, students should understand how to do while loops, for loops, and nested loops.

⁴<http://www.farmville.com/>

Figure 5.10: The step shows the instruction and pattern of the “field” they have to farm. It also shows the students what a unit test might look like for this example.

Chapter 6

Evaluators

An initial set of internal and external evaluators were developed for the initial labs in section 5. Internal evaluators are stored in the WebIDE engine (GAE) and external evaluators are located on an external server (EC2). First, I will describe the Evaluator ToolKit (ETK), a framework for developing evaluators. Then I will talk about the internal and external evaluators, and how to use them.

The following sections show several code examples of PHP evaluators. However, the example below demonstrates that external evaluators may be written in a variety of languages. This evaluator is written in Racket. The boilerplate code for decoding and validating inputs was removed for brevity.

```
(define birth-year-example
  (make-java-header-checker
    `("int"
      "getApproxAge"
      (("int" "birthYear")
       ("int" "curYear")))))
```

6.1 WebIDE Evaluator Toolkit

The Evaluator ToolKit (ETK) is a framework for helping lab authors develop evaluators in PHP. The ETK is written in PHP and abstracts out the common functionality that most evaluators will need to perform. The main purpose of the ETK is to make writing evaluators faster and easier. Although we hope that there will be a wide variety of general evaluators for lab authors to use, there may come a time when an author needs to write their own. The current version of ETK consists of 2 classes, the Evaluator class and the JavaClass class.

6.1.1 Evaluator Class

The Evaluator class abstracts out the evaluator’s request and response and gives the lab author direct access to the evaluator arguments and segment values specified in the lab’s XML without having to deal with the JSON in the URL parameters. The class also sends a response back to the server, which creates the correct JSON response, sends it back, and stops the php execution.

To use the Evaluator class, an author must include the Evaluator.php file within his/her evaluator. This file creates a variable, `$e`, which is an instance of the Evaluator class. The instance will automatically get the request from the URL and decode the JSON. Figure 6.1 shows an evaluator example that receives and sends a request using the Evaluator instance. The evaluator gets the value of a segment with the name `input`, then returns that value back to WebIDE.

In figure 6.2, the code does the same thing as figure 6.1 but we tell the Evaluator instance that the segment is required. The method will automatically send back an invalid response with an appropriate error message if the “input”

```

<?php
    require_once ( "../WebIDE.ETK/Evaluator.php" );

    //Get the segment's value with name "input"
    $input = $e->getSegment('input');

    //Send a successful response with segment's value back to WebIDE
    $e->sendResponse(true, $input);

    //Won't get executed
    echo "AHHH" ;
?>

```

Figure 6.1: WebIDE ETK `getSegment()` Example

segment is missing. The same thing can be done with an evaluator argument using *getArgument()* and *getRequiredArgument()*.

The Evaluator class also provides two miscellaneous helper methods. The first method, *replaceSegmentValues()*, replaces segment names—marked with the @ symbol—within the argument’s values. Typically, an evaluator has to know the name of the segment in order to get the segment’s value. However, it is useful for an evaluator to be generic, i.e. allow dynamic segment names. With this method, a lab author can pass a segment name via an argument, then the evaluator can easily parse out the segment’s name from the argument and replace it with the segment’s value. Figure 6.3 shows an example of an evaluator that gets the value of a segment specified in an argument.

Additionally, authors can use the *replaceSegmentValues()* method for dynamic scaffolding within the lab’s XML. For example, let’s say an author wants a student to write a Java function. The author has the following two options. They can

```

<?php
    require_once (" ../WebIDE.ETK/Evaluator.php" );

    //Get the segment's value with name "input"
    $input = $e->getRequiredSegment('input');

    //Send a successful response with segment's value back to WebIDE
    $e->sendResponse(true , $input);

    //Won't get executed
    echo "AHHH";
?>

```

Figure 6.2: WebIDE ETK getRequiredSegment() Example.

```

<?php
    require_once (" ../WebIDE.ETK/Evaluator.php" );

    //Get the segment's value with segment name from the
    //argument segmentName.
    $input = $e->replaceSegmentValues($e->getRequiredArgument('segmentName'));

    //Send a successful response with segment's value back to WebIDE
    $e->sendResponse(true , $input);
?>

```

Figure 6.3: WebIDE ETK replaceSegmentValues() example


```

<evaluator name="CompileFunction" labid="Test Lab"
    href="JavaCompiler/JavaCompiler.php">
    <arg name="code"><value>
        class RunMethod {
            public static void main(String[] args) {
                System.out.println(theStudentFunction());
            }

            @studentFunction
        }
    </value></arg>
    <segId id="studentFunction"/>
</evaluator>

```

Figure 6.4: WebIDE ETK Java Scaffolding Example

write their own evaluator, which would most likely insert the function into a Java template, then compile and run that class. Instead, they could use the generic `JavaCompiler` evaluator, which will compile and run a given class using any text, segments, or a combination of both. This evaluator is explained in more detail in section 6.3.1, however, figure 6.4 shows an example of a lab using dynamic scaffolding with the `JavaCompiler` evaluator. The `@studentFunction` will get replaced using the segment *studentFunctions* value via the `replaceSegments` method.

The second helper function escapes special characters that will not render in the browser. For example, newlines (`\n`) and multiple spaces () in the response message will not get interpreted by the browser. This function will convert newlines to `<br />` and spaces to ` `; so the response will display correctly in WebIDE. The function also replaces other html characters, such as “<” with their corresponding ISO-8859-1 entity.

6.1.2 JavaClass Class

The JavaClass class allows evaluators to compile and run Java classes asynchronously. Compiling and running Java classes asynchronously becomes very important when multiple students are doing the same lab at the same time. For example, let's say an entry-level programming course is using a WebIDE lab that takes student input for a Game class, and compiles and runs the code. If the course has 30 students, the evaluator is potentially compiling and running all 30 Game classes at the same time, which would override student responses and could produce other race conditions. To solve these problems, the JavaClass gives the class a unique name. The evaluator will now compile and run each student's Game class independently. However, renaming the class can cause problems if another class depends on it. For example, imagine a class Tetris that extends the Game class. If the JavaClass renames the Game class, the Tetris class won't know what a "Game" is. The JavaClass lets an evaluator replace the old class name with the new class name within another class's code, as seen in figure 6.5.

A similar solution would give students their own directory to compile and run their code. However, since each JavaClass instance is independent, it would still have to let each other know which directory to find the compiled Game class.

An evaluator can also specify class paths and different executables for each JavaClass. For example, a lab author may want to run a class as a suite of unit tests. The evaluator passes in a constant variable called `Exec::JUNIT` to the run function, which will run the jUnit test runner against "this" java class, as seen in figure 6.6. The evaluator can also specify a custom Java executable and several other parameters, such as a custom security policy and the max execution time. By default, the class's main function is run with no parameters with a max

```

<?php
    require_once (" ../WebIDE.ETK/Evaluator.php");
    require_once (" ../WebIDE.ETK/JavaClass.php");

    //Create a Game Java class
    $game = new JavaClass($e->getSegment("GameCode"));

    //Get the tetris code
    $tetris = $e->getSegment("TetrisCode");

    //Replace the Game class name with the unique
    //name in the tetris code.
    $tetris = $game->replaceClassName($tetris);

    //Create the Tetris Java class
    $tetris = new JavaClass($tetris);

    //Compile the Game class first since Tetris relies on it
    $game->compile();
    $tetris->compile();

    //Run the Tetris class and return the response
    $response = $e->escapeHTMLChar($tetris->run());
    $e->sendResponse(true, $response);
?>

```

Figure 6.5: WebIDE ETK: replaceClassName() example

execution time of 1 second.

6.2 Internal Evaluators

Internal evaluators run within the WebIDE engine. By running internally, the evaluators become faster and more reliable by removing the extra http request to an external server. There are also a couple drawbacks to internal evaluators. First, the evaluator must conform to the restrictions of the environment that WebIDE runs on. For example, GAE does not allow system calls or file I/O. Therefore, evaluators like the JavaCompiler must run on an external server. Second, the WebIDE engine must be redeployed to add new evaluators or make changes to current evaluators. In contrast, modifications to external evaluators do not require WebIDE to be redeployed, allowing the changes to be seen immediately.

Currently, there are two internal evaluators: a regular expression evaluator and an arithmetic evaluator.

6.2.1 Regex

The Regex Evaluator uses Java's Pattern class to evaluate regular expressions. The evaluator allows multiple regular expressions and failure messages that run on each segment passed in. If a regular expression does not match, the corresponding error message will be returned. Figure 6.7 shows the lab XML for a regular expression evaluator that checks that the student input is not empty and contains the word monkey. Notice how there are two "regex" arguments and two error messages. The evaluator requires at least one argument called "regex" and "failed-message." After that, each argument must be appended with an increasing

```

<?php
    require_once (" ../WebIDE.ETK/Evaluator.php");
    require_once (" ../WebIDE.ETK/JavaClass.php");

    //Get the student test
    $test = $e->getSegment("Tests");

    //Create a JavaClass with the student test
    $test = new JavaClass($test);

    //Check the class compiled correctly using the junit Jar
    if (!$test->compile(Jars::JUNIT)) {
        //Close the class and deletes all created files.
        $code->close();

        //Get compile error
        $error = $e->escapeHTMLChar($test->getCompileError());

        //Send back compile error
        $e->sendResponse(false, $error);
    }

    //Run the test
    $result = $test->run(Exec::JUNIT);

    //Close the class and delete all created files.
    $test->close();

    $e->sendResponse(true, "You ran ".$test->getTotalTest()." test.");
?>

```

Figure 6.6: WebIDE ETK: Running JUnit test

```

<evaluator name="Write Monkey" labid="Monkeys"
  href="evaluator://RegexEvaluator">

  <arg name="regex" value="\S"/>
  <arg name="failed-message" value="Your input should not be empty."/>

  <arg name="regex1" value="monkey"/>
  <arg name="failed-message1" value="Try to include the word 'monkey'"/>

  <arg name="success-message" value="Good Job!"/>

  <segid id="studentInput"/>
</evaluator>

```

Figure 6.7: Using the regular expression evaluator.

integer starting at 1.

6.2.2 Arithmetic

The Arithmetic Evaluator can evaluate simple arithmetic expressions. It supports five operators: addition, subtraction, multiplication, division, and modulus. This evaluator is good for segments where a student needs to write a function that returns a value based on an equation. Figure 6.8 shows a lab XML where a student must give an expected output that is the difference of two inputs.

6.3 External Evaluators

External evaluators can be executed on any server. Currently, all the external evaluators that we built are hosted on our Amazon EC2 instance. First, I describe

```

<evaluator name="TestEval" labid="Difference"
  href="evaluator://ArithmeticEvaluator">
  <arg name="equation" value="test1b - test1a"/>
  <arg name="expected" value="test1c"/>
  <arg name="failed-message" value="I'm sorry , you input is incorrect."/>
  <arg name="success-message" value="Good Job"/>

  <segid id="test1a"/>
  <segid id="test1b"/>
  <segid id="test1c"/>
</evaluator>

```

Figure 6.8: A lab using the Arithmetic Evaluator

all the Java evaluators in section 6.3.1. In section 6.3.2, I discuss the strategy behind the evaluators used in the classes lab. Finally, I talk about the evaluators used in the Tic Tac Toe lab in section 6.3.3.

6.3.1 Java

There are three Java evaluators that all use the `JavaClass` in the ETK. The first evaluator, *JavaEvaluator*, only evaluates a single function and lets the author specify the segment with the student's function, the function call and the expected output. For example, let's say a student needs to write a function, *sum(int x, int y)*, that sums two numbers. The function call could be *sum(89231, 32133)* with an expected output of *121364*.

The second evaluator, *JavaCompiler*, just compiles a Java class and returns the results. The class must contain a *main* function. This evaluator will only fail if there is a compile error or the runtime exceeds the limit, which is 1 second by default. The playground on WebIDE's homepage, as seen in figure 1.1 uses this

```

<evaluator name="Inheritance" labid="inh" href="Java...???">
  <arg name="code" value="@Animal"/>
  <arg name="code1" value="@Dog"/>
  <arg name="test" value="@AnimalTest"/>
  <arg name="test1" value="@DogTest"/>

  <arg name="failed-message" value=""/>
  <arg name="success-message" value="Good Job!"/>

  <segid id="Animal"/>
  <segid id="Dog"/>
  <segid id="AnimalTest"/>
  <segid id="DogTest"/>
</evaluator>

```

Figure 6.9: JavaCompiler: Inheritance Example

evaluator.

The final evaluator, *classUnitTestEvaluator*, runs JUnit classes. The lab author can call this evaluator with multiple test and Java classes. For example, let's say a student is learning inheritance by writing a Dog class that inherits from an Animal class. The lab author can have the student write both course classes *and* unit test classes. Figure 6.9 shows what the code might look like inside the lab's XML. A lab author could also write a test suite of their own, which would fix the potential problem from the *JavaEvaluator*.

6.3.2 Classes

We created a separate evaluator for each segment in the Classes lab, which is an example of a different way to develop evaluators. The benefit is less code in the lab XML, fewer HTTP requests, and less bandwidth per request. For example,

the “Classes” lab asks a student to write a game class that determines which team won depending on a score. To provide additional feedback, the author may want to run several regular expressions before compiling and running the code to give detailed feedback. The response will also be faster because the WebIDE will not have to wait for the code to be compiled if one of the regular expressions fail. If the lab uses the generic evaluators, it has to call the Regex evaluator and the JavaCompiler evaluator. The lab also has to define several arguments that the evaluators need to run. However, all the arguments can be moved to a custom evaluator’s code instead of being defined in the lab. Additionally, imagine a segment that needs to use ten evaluators instead of two. In this case, WebIDE has to make ten different request with, potentially, a lot of information in each request. With a custom evaluator, only one request is made with only the contents of the segment. Figure 6.10 shows a lab’s xml that uses a generic evaluator multiple times, and figure 6.11 shows a lab’s xml for using a custom evaluator.

Initially, the Classes lab was using generic evaluators but WebIDE was sending all request information via HTTP GET. GET sends all parameters through the URL instead of the request header, which caused problems. For example, sending instructors test, evaluators arguments, and student test/code raised an exception because the URL was too long. By using custom evaluators, we were able to put all arguments and instructor tests in the evaluators. This temporarily prevents the exception until WebIDE switches to HTTP POST, which puts all information in the request header.

```

<segment id="seg1">
  <evaluator name="TestEval" labid="MultiEval" href="evaluator://genericEval">
    <arg name="argument1" value="value1"/>
    <arg name="argument2" value="value2"/>
    ...
    <segid id="seg1"/>
  </evaluator>
  <evaluator name="TestEval2" labid="MultiEval" href="evaluator://genericEval">
    <arg name="argument1" value="different value1"/>
    <arg name="argument2" value="different value2"/>
    ...
  <segid id="seg1"/>
</evaluator>
...
</segment>

```

Figure 6.10: Multiple Evaluator Example

```

<segment id="seg1">
  <evaluator name="TestEval" labid="MultiEval" href="http://customEval">
    <segid id="seg1"/>
  </evaluator>
</segment>

```

Figure 6.11: Custom Evaluator Example

6.3.3 Android

The Android lab uses only three evaluators: the Regex evaluator, JavaCompiler evaluator, and the Android evaluator. All the steps were able to provide detailed feedback by using the first two evaluators. The Android evaluator returns HTML to display a QR code image of the location of the compiled APK, as seen in figure 5.3. A student can use their Android device to scan the QR code which will download and install the application on their device. In this case, all the previous steps ensure that the student's code will produce a working app. So instead of compiling the Android application for every student, we use a precompiled APK of the application. For labs that might have students write customized Android applications, the Android evaluator can build and sign any Android application. It then takes the APK and puts it in a unique directory and returns a QR code that points to that APK. This will ensure that each student has their own APK.

Chapter 7

Results

CPE 101 was the first course in Computer Engineering, Computer Science, and Software Engineering at Cal Poly. However, the failure rates within CPE 101 are extremely high. Cal Poly hopes to solve this problem by introducing a new course, CSC123 (hereafter referred to generically as CS0), which focuses on current, interesting topics of computer science. In Fall 2010, the new CS0 course was offered with four different topics: Music, Robotics, Android Development, and Game Development. The goal is to increase interest and retention rate within the computer science department and see all students succeed in CPE 101 the following quarter.

We performed a pilot study on 51 students in two sections of CS0: Android Development during the Fall 2010 quarter at Cal Poly. For each section, we randomly assigned each student into one of two groups. We validated no significant difference in prior programming experience between the two groups. For Lab 4 and 5, group A (31 students) used WebIDE with evaluation, and group B (20 students) used WebIDE without evaluation. In other words, group A received immediate feedback on each lab segment, whereas group B received no feedback

until they completed the entire lab, submitted it, and received feedback with a grade from the instructor. Group B students were introduced to the Eclipse development environment, and they were encouraged to check their answers by compiling and running them with unit tests in Eclipse. This was deemed equivalent to a traditional lab where the student is given lab instructions in a static HTML page and uses a development environment to complete the lab.

For Lab 6, group A used WebIDE and group B used Eclipse with the Android SDK installed. Group B students were provided instructions in a static HTML page (equivalent to the WebIDE instructions) and a project stub that contained class definitions with method headers for each of the methods to be completed. Lab 6 lets us evaluate the usefulness of WebIDE as a simplified environment with no setup versus using a complex environment like Eclipse with the Android SDK.

After completing Lab 4 and 5, students were given a midterm exam with four Java programming questions. The questions asked them to 1) write a set of JUnit tests for a described method, 2) write a method that used if-then-else, 3) write a method that used nested for loops, and 4) implement two classes and two methods, where one class contained an array of instances of the second class. This last question was identical to a pre-experiment programming quiz that was given just prior to assigned Lab 4 and used to determine prior programming experience. In addition to the quantitative evaluations, qualitative surveys were administered at the end of each lab, and focus groups were conducted after Lab 6. All collected data is outlined in the next section.

7.1 Data Collection

We collected data in three different areas: graded data, lab logs, and surveys.

7.1.1 Graded data

Graded data consist of any material related to WebIDE that goes toward the student's final grade. This includes the following material:

1. Initial Java Quiz
2. Lab 4: Java Basics, Facebook: Functions, Facebook: Ifs.
3. Lab 5: Facebook: Iterations, Classes.
4. Lab 6: Android: Tic Tac Toe.
5. Midterm

The initial Java quiz was a baseline to determine what programming skill level the students were starting out at. This baseline is used to determine the student's—group A and B—progress throughout the lab. Some of the midterm questions were designed to show progress on certain topics based on the labs. For example, question 22 on the midterm deals with nested for loops, which was covered on Lab 5: Facebook: Iterations. Additionally, question 23 is the exact same question as the initial java quiz. The final exam and final project grades were not collected for this thesis. Table [7.1](#) shows an overview of the average grades for all material.

7.1.2 Surveys

Many surveys were given through out the course to capture personal information, like prior programming experience, and opinions on the labs, course, and WebIDE. The following surveys were given to both groups:

	Quiz	Lab4	Lab4-Q	Lab5	Lab6	Q20	Q21	Q22	Q23	Total
Min-A	0.0	0.0	0.0	0.0	0.0	0.0	25.0	5.0	7.7	42.2
Min-B	0.0	0.0	0.0	0.0	0.0	28.5	0.0	10.0	12.8	36.4
Max-A	90.5	100	88.9	100	100	100	100	100	97.4	95.6
Max-B	92.9	100.0	88.9	100	100	100	100	100	100	95.1
Avg:	25.2	69.0	68.6	77.5	79.2	67.9	72.9	72.2	63.1	72.8
Avg-A	23.3	69.2	70.3	83.2	89.0	67.5	75.2	71.6	58.5	71.7
Avg-B	28.1	68.8	66.1	68.6	64.0	68.6	69.5	73.0	70.3	74.7
StdDev:	34.5	31.6	32.5	38.3	33.6	28.8	25.5	30.3	32.6	18.3
StdDev-A	33.6	31.8	30.1	35.5	24.4	30.2	23.8	30.1	34.4	19.0
StdDev-B	36.4	31.9	36.5	41.5	40.3	27.1	28.3	31.2	29.1	17.4

Table 7.1: Student Grade Averages (%)

1. Initial Class Survey
2. Pre-WebIDE Survey
3. Lab 4 Survey
4. Lab 5 Survey
5. Lab 6 Survey
6. Post-WebIDE Survey

Students reported the estimated time—in minutes—to complete each lab, which is shown in table [7.2](#)

7.1.3 Lab Logs

WebIDE logs all interactions between students and evaluators. For example, whenever the student fills in a segment and presses the “Run” button, WebIDE records the time, evaluator request and evaluator response. The lab logs give us a lot more information on group A since they use the evaluators. In some WebIDE

	Lab1	Lab2	Lab3	Average	Total
Avg	108.83	206.40	246.50	188.53	495.98
StdDev	69.67	124.34	159.43	107.93	241.87
Total	5115.00	10320.00	9860.00	9615.00	25295.00
A-Avg	125.52	193.50	241.54	184.03	507.26
A-StdDev	69.67	124.34	159.43	107.93	241.87
A-Total	3640.00	5805.00	6280.00	5705.00	15725.00
B-Avg	81.94	225.75	255.71	195.50	478.50
B-StdDev	70.02	148.83	144.31	143.64	246.32
B-Total	1475.00	4515.00	3580.00	3910.00	9570.00
p-value	0.0441	0.4321	0.7465	0.7636	0.6870

Table 7.2: Time to complete WebIDE labs as reported by students.

labs, we added a button for group B that would send their input to an evaluator that always returns successfully. This way, we could save their progress and get the amount of time between steps. However, in the case of Lab 6, we have no logged information for group B.

A summary of lab statistics—computed from the lab logs—can be seen in table 7.3. *Steps* are the average number of completed steps per student. Each lab has an average number of evaluation *successes* and *failures* that each student attempted, which should equal the average number of *attempts*. *Error pct* is the percent of students that saw an error. *Tries after* is the average number of evaluation attempts a student tried after already completing the step. Finally, *time* is the average amount of time it took a student to complete the lab. Gaps in the evaluation timestamps of over 40 minutes are considered new “sessions,” with the idea that students will try to evaluate segments very frequently (less than 40 minutes). For example, if a student evaluated segments at 12:32pm, 12:35pm, 1:28pm, and 1:30pm, the amount of time computed would be 5 minutes.

The labs are labeled as the following.

Lab	Steps	Attempts	Success	Fail	Error %	After	Time
L1	5.8	20.0	9.9	8.8	19.4%	4.1	10.9
L2	7.0	25.9	10.7	13.0	30.0%	3.7	30.1
L3	11.5	67.2	15.2	50.2	61.3%	3.6	95.6
L4	5.9	61.8	32.7	28.0	45.2%	26.8	74.1
L5	3.6	91.7	5.8	27.1	100%	2.2	124.8
L6	8.6	144.3	39.5	101.1	32.3%	31.0	175.9
Total	42.8	378.3	110.4	210.8		67.6	472.1

Table 7.3: Evaluation statistics relating to each of the 6 WebIDE labs.

L1: Java Basics

L2: Facebook: Selection

L3: Facebook: Functions

L4: Facebook: Iterations

L5: Classes

L6: Android: Tic Tac Toe

In addition to the lab completion times reported in table 7.2, we also compiled the times for each lab via the logs, which is reported in 7.4. Note that Group B did not evaluate steps, resulting in a “time” of 0 unless there was an “save” evaluation. We can use this information to validate student estimates.

7.2 Hypothesis

The primary purpose of this pilot was to gain initial feedback on WebIDE to make improvements for future studies. However, the study was designed and run so that significant differences in the two groups could be discovered. We examined two hypotheses in this study: 1) students who used WebIDE would

	L1	L2	L3	L4	L5	L5B	L6	Avg	Total
Avg	6.6	18.0	56.2	53.4	120.0	25.5	170.2	70.8	321.4
StdDev	8.7	20.5	60.2	44.0	64.5	35.8	102.3	45.6	245.7
Total	339.3	903.4	2697.5	2455.0	3119.9	1276.2	5276.3	3612.9	16067.6
A-Avg	10.9	30.1	95.6	74.1	124.8	10.4	175.9	93.8	472.1
A-StdDev	8.7	20.5	60.2	44.0	64.5	35.8	102.3	45.6	245.7
A-Total	339.3	902.3	2675.7	2000.7	3119.9	321.6	5276.3	2908.4	14635.8
B-Avg	0.0	0.1	1.1	23.9	0.0	50.2	0.0	35.2	75.4
B-StdDev	0.0	0.2	3.1	19.8	0	45.8	0	28.2	55.1
B-Total	0.0	1.1	21.8	454.3	0.0	954.6	0.0	704.5	1431.8

Table 7.4: Time to complete WebIDE labs based on lab logs.

perform better on programming tasks than students who used traditional static labs, and 2) students who used WebIDE spend more time on labs (because of the lock-step aspect) than students who used traditional static labs.

7.3 Empirical Analysis

There were no statistically significant differences between group A and B on their lab scores or the midterm questions, with two exceptions. First, students who used WebIDE scored an average of 2.6 points (25%) higher on Lab 6 (Android), as seen in figure 7.1, with a p-value of 0.006. In other words, students were more likely to successfully complete their first Android app with WebIDE, than with a traditional development environment. The focus groups seemed to reflect this as well. Several students in group B wished they had been in group A on the Android lab because it stepped them through the solution and gave feedback on correctness of interim results. Students using Eclipse reported spending a large amount of time trying to debug small problems that were often a misunderstanding of the lab specification, whereas WebIDE would return an appropriate error message.

Figure 7.1: The average grades on student course work between group A and group B.

Secondly, among students who scored higher than zero on the pre-experiment Java quiz, students who used WebIDE scored an average of 2 points higher—with a p-value of 0.04—on the first midterm question that required them to write unit tests. In other words, among students with some prior programming experience, students who used WebIDE did better at writing automated unit tests than students who did not use WebIDE. This indicates that WebIDE may achieve one of our primary goals, which is to successfully integrate TDL into entry level courses.

It is important to note that using WebIDE did not harm students in terms of academic performance. There were no significant results showing a decrease in group A scores.

On the second hypothesis, group A (WebIDE) students reported spending

Figure 7.2: The average reported time, in minutes, for students to complete each course lab.

125.52 minutes on average on Lab 4, while group B reported 81.94 minutes on average, as seen in table 7.2. This difference was statistically significant with $p = 0.0441$. This matched observations that the lock-step aspect of WebIDE seemed to slow students down. However, on Lab 5 and Lab 6, group A students actually reported slightly lower average times than group B, although these results were not statistically significant. Figure 7.2 shows average lab times for each course lab.

Additionally, we created a decision tree using the C4.5 algorithm to see what the main contributing factors were for a student to pass the midterm. In other words, the decision tree predicts whether a student will pass the midterm based on quiz grades, lab grades, programming experience, programming confidence, and lab log statistics. We used a threshold of 0.02 when selecting splitting attributes.

The decision tree obtained an average accuracy of 80% using all-but-one cross-validation.

The decision tree showed that programming experience has the biggest impact on whether a student will pass the CS0 midterm. The other big contributors were programming confidence and lab grades. In two situations, the student's group had a very small effect, most likely an outlier. In one interesting situation among students with very little experience, students with some programming experience that had over 100 successful evaluations and rated their confidence as neutral passed the midterm while others didn't. Using this method, we confirmed the previous analysis that WebIDE did not, overall, increase student performance.

7.4 Lab Statistics

The ratio of successes to attempts can indicate the difficulty of a lab. For example, a lab would be too easy if the number of successful evaluations equaled the number of total evaluations since all the students got it on their first try. On the other end of the scale, if the ratio is 20 to 1 (failed evaluations for one single success), then the lab might be too hard. There is no correct ratio, instead, each professor will likely have their own ideal ratio. Figure 7.3 includes the access and attempts for each lab. The “Java Basics” lab may be interpreted as easy, since it has over a 50% success rate. The “Classes” Lab has hardly any success and could be interpreted as hard, however, notice the amount of errors in the lab. This implies a defective lab.

Additionally, the “tries after” can tell the lab author the student's interest in the course material. For example, let's say a student has 10 tries after success. This means a student kept playing with the code and trying new thing longer

Figure 7.3: The number of attempts, successes, errors, and tries after for all WebIDE labs

Figure 7.4: The average time, in minutes, it took students to complete each WebIDE lab.

than necessary, which could imply that the student was interested and engaged in the material. Using this model, figure 7.3 suggest that students were really interested and engaged in the iterations lab and Android Lab.

Let's do a quick review on how students did on each lab. All data is pulled from table 7.3. The time it takes to complete the lab uses the average time computed from the lab logs, which is summarized in figure 7.4.

Java Basics

This lab has 6 steps, and takes students an average of 11 minutes to complete. Students had a 50% success rate when evaluating their answers and only 15% of the students saw an error.

Facebook: Selection

This lab has 10 steps and took students an average of 30 minutes to complete. Students had a 41% average success rate when evaluating their answers and 75% of the students kept playing with a step even after they successfully completed it. Figure 5.9 shows a screenshot of the “The Age Group - Input/Output” step.

Facebook: Functions

The lab has 6 steps and took students an average of 95 minutes to complete. Students had a 22% success rate when evaluating and 89% of the students kept playing with a step after completion.

Facebook: Iterations

This lab had 6 steps and took students an average of 74 minutes to complete. Students had a 53% success rate when evaluating and 100% of the students kept evaluating a segment after they already completed it an average of 27 times! That is 4.5 times per step!

Classes

This lab had 5 steps that needed to be completed and the average student only got through 3 of them because of a bug that prevented students from continuing. Once the bug was fixed, only a portion of students finished it. It took students an average of 124 minutes to get through the number of steps they completed. However, this number is not reliable since a huge portion of the time was spent dealing with the bugs in this lab. Only 6% of the student evaluation attempts

resulted in a success.

Android: Tic Tac Toe

The Android lab had 12 completable steps, and the average student only got through 9 of them. WebIDE students had a success rate of 37.4%. The Android lab had fewer “tries after” than the Iterations lab by 1.1 times per step. That means that students were more interested in the Iterations lab, but the Android lab was more difficult.

7.5 Qualitative Analysis

After the pilot study, we conducted 3 focus groups of about 7 students each; two groups used WebIDE (group A) and one group did not (group B).

Students had some great discussions on the user interface of WebIDE. A number of students didn’t like the scrolling tabs across the top of the lab and would prefer it to be on the side, which would enable them to easily see the big picture. For example, one student said “In Lab 6, I wanted to move around a lot and using the tabs was really annoying.” Students also offered to solve this problem by having two screens side by side; one for instructions and the other for coding. They also suggested the split screen for multiple editing, where one is for the class code and the other is for unit testing. Several comments were about the windows in WebIDE being too small, forcing them to scroll in the lab and “scroll within really small text boxes.” When working on Lab 6, students wanted a more “eclipse feel,” with syntax highlighting and code completion.

Students seemed divided on the step size within the labs, although the major-

Figure 7.5: The percentage of students that felt the error messages were helpful, where 1 is strongly agree and 5 is strongly disagree.

ity seemed to like smaller steps. For example, many students suggested breaking up the class lab in more steps. They liked the step size on the Android lab—in particular, the steps implementing multiple methods—but felt the lab was too long and had too many steps. The Facebook theme was fine with students, although the majority of the students didn’t care one way or another and went straight to the code. One student said the Facebook theme seemed too “contrived.”

On the positive side, students felt the Android lab had really good error messages, whereas other lab’s error messages “just didn’t make sense”. Even still, only 25% of the students reported WebIDE’s error messages as not helpful, as seen in figure 7.5. Students really liked the tone of the error messages when responses asked if they forgot a piece of code, e.g. a function call or semicolon. They would think to themselves, “Oh yea, I did forget that.” They also liked the tone when the error gave a suggestion. For example, if the student forgot to call a function, the response would say “Why don’t you try to call the isEmpty() method.”

Figure 7.6: The percentage of students that liked the lab content, where 1 is strongly agree and 5 is strongly disagree.

Figure 7.7: The percentage of students that felt the labs helped them learn Java, where 1 is strongly agree and 5 is strongly disagree.

The surveys also provided a lot of data on whether the labs were useful, especially between group A and B. Over 65% of students from group A liked the lab content, compared to only 36% in group B, as shown in figure 7.6. However, 45% of group A felt like the lab content helped them learn Java compared to 53% in group B, as shown in figure 7.7. So group B felt they learned more from the labs even though they didn't like them as much as group A.

Group A and group B were both asked if they liked WebIDE. Unfortunately, figure 7.8 shows that the majority of both groups did not like WebIDE. The

Figure 7.8: The percentage of students that liked WebIDE, where 1 is strongly agree and 5 is strongly disagree.

Figure 7.9: The percentage of students that felt like WebIDE helped them learn Java, where 1 is strongly agree and 5 is strongly disagree.

focus groups suggest that this was primarily because of the problems students experienced with WebIDE, such as server errors. We also see similar results when asking students if they felt WebIDE helped them learn (figure 7.9). Although, some of the most encouraging feedback was when students in the focus group said they felt, in general, WebIDE helped them learn and that they liked the concepts and ideas behind it.

We also evaluated the content of the Android lab. Figure 7.10 shows that only 36% of group A agreed that Lab 6 helped them learn Android programming,

Figure 7.10: The percentage of students that felt the labs helped them learn the Android platform, where 1 is strongly agree and 5 is strongly disagree.

Figure 7.11: The percentage of students that asked if they would have preferred not to use WebIDE, where 1 is strongly agree and 5 is strongly disagree.

compared to 58% in group B. The focus groups reflected this. Students said the course jumped into Android too fast, and would have liked to see more overview and explanation. Students—in both group A and group B—reported they had trouble getting the overall picture in Lab 6.

At the same time, students felt that WebIDE helped them learn Android. Group B spent a lot of time debugging the application. Even though students felt they learned more using Eclipse, almost all students—both group A and B from the focus group—wanted to be in or stay in the WebIDE group. For

Figure 7.12: The percentage of students that felt like WebIDE helped them learn the Android platform, where 1 is strongly agree and 5 is strongly disagree.

example, one student said “When I went to office hours, everyone from group B was there asking why their code wasn’t working, especially the tic tac toe lab. People in group B were debugging for hours just because they missed one snippet of code. We didn’t get that in WebIDE.” Another student even moved over to group A because they heard WebIDE was easier to learn the Android platform. The survey data contradicts this, as the overall student rating for using WebIDE for Lab 6 (figure 7.12a) is still lower than completing Lab 6 in Eclipse(figure 7.10b). Additionally, the majority of students reported that they would have preferred not to use WebIDE (figure 7.11).

However, even with WebIDE’s negative rating, the focus group for group B regarded the system very positively. Students liked the steps of the lab, but wanted feedback. However, some students liked how they didn’t have to wait for evaluation and could look ahead. When asked if they would have liked to be in group A, most of them said yes. The rest of the students said they wanted to do the labs in Eclipse, but only because they were used to it or because they had to do the final project in Eclipse. Neither of those cases should be a problem for CS0 and CS1 students. Some students said they had trouble going to Eclipse

from Lab 6 and would prefer to use WebIDE.

7.6 Threats to Validity

There are three main threats to validity. First, Cal Poly implemented CS0 for the first time in Fall 2010. The expectation is for every student to succeed in CS1 in Winter 2011. Therefore, we don't know if the overall student success is, or will be, due to CS0 or WebIDE.

Second, we ran into major errors within the environment and labs during this pilot study. At one point, the steps on classes in Lab 5 returned numerous server errors, which caused students to be extremely frustrated and, of course, halted their progress on the labs. In addition, students sometimes received vague, invalid, and incorrect error messages on the evaluation attempts. For example, one lab evaluator initially contained a race condition that occasionally caused student's code to compile over other student's code, returning either an invalid compile error or one compiler error for both students. As a result, a student who submitted correct code could potentially still receive an error message.

Finally, a few students varied from instructions without notifying the instructor. We observed a student assigned to group A join group B (this was adjusted in the evaluation). We also heard of students copying code from other students just to pass the current step. In addition, a few students reported doing everything in Eclipse then pasting their code into WebIDE. Because the labs were not completely finished in the classroom, students may have changed instructions in different ways that weren't observed.

Chapter 8

Future Work

WebIDE is currently available to the public at <http://web-ide.org>. However, a number of additional features, additional labs, and a full assessment will occur in 2011. The full assessment will occur in CSC101 and CSC102 courses at Cal Poly in 2011. Controlled experiments are planned in these courses to empirically determine the effects of WebIDE. We also hope to track the students who use WebIDE in their 100 level series to see how these students perform. Ideally, we will see students who used WebIDE excel in future classes, especially classes that require testing.

Additional development is planned to ensure ease of use by students and professors. Possible features could include plugins to automated assessment applications, increases in performance, and better data collection. We also plan to add a web authoring tool for creating labs. This will prevent professors from writing pure XML, reducing the amount of time spent creating labs. Additionally, a professor will be able to “cut and paste” labs. In other words, if a lab is shared, professors can cut steps or segments of one lab and paste it in their own lab. This functionality will allow professors to quickly create and customize

labs for any class. We also want to make completing labs in WebIDE easier by including syntax highlighting, code completion, continuous evaluation as the user types, and conditional logic in the lab XML. Below is a simple outline of future work.

- Improvements
 - Speed and performance
 - Bug fixes
 - UI Enhancements
 - * Split view
 - * Vertical Step List instead of Tabs
 - * Resizable Windows and Segments
 - * Site Redesign
 - Features
 - * Syntax Highlighting
 - * Auto Completion
 - * Ratings and other Social Functionality
 - * Tutoring System (Allow tutors/lab mentors to see lab logs and student state.)
 - * Authoring Tool
- Labs and Evaluators
 - Additional languages
 - Detailed Error Messages
- Future Studies

Chapter 9

Conclusion

WebIDE, available at <http://web-ide.org>, is unique in its combination of features: a test-driven learning approach, completely web-based delivery, and intrinsic support for community-contributed content. Although WebIDE may be seen as an alternative to some systems such as Turings Craft, it is intended to complement many systems such as Web-CAT and BlueJ. The primary focus of WebIDE is on the first three to five weeks of an introductory course, after which we assume students will transition to a traditional development environment. It is possible that some faculty will use WebIDE throughout a course such as a CS0 for non-majors, or as a supplement in non-introductory courses where students must learn an unfamiliar language quickly.

WebIDE is built on top of Google Web Toolkit and Google App Engine. The framework allows anyone to build and host their own labs and evaluators—written in any language—to support any topic or pedagogical approach. As a demonstration, I built an Android lab where students develop a complete Android application in the browser. This application can then be downloaded and installed on the device. Labs can also support a range of topics outside the realm

of computer science, such as math, physics, and even English. All evaluators presented in this thesis are hosted on Amazon’s EC2 service, where we can leverage the power of cloud computing.

We performed a pilot study with in a CS0 course at Cal Poly with 51 students. Students who used WebIDE were more successful than non-WebIDE students in creating their first Android app. Also, among students with some programming experience, WebIDE students were significantly better than non-WebIDE students at writing automated unit tests. This confirms WebIDE’s primary goal, which is to successfully integrate TDL into entry level courses. Additionally, and possibly most importantly, WebIDE had no negative effect on students in terms of academic performance.

Appendix A

WebIDE's RelaxNG Lab

Definition

```
start = element lab {
  attribute name { text },
  attribute defaultpath { text }?,
  element description { text }?,
  step*
}

step = element step {
  (attribute name { text } &
  attribute buttonName { text }?),
  (dependency* & evaluator* & segment* & labtable* & hint* & (allowedFormatting)*)
}

segment = element segment {
  (attribute id { text } &
  attribute buttonName { text }? &
  attribute width { text }? &
  attribute height { text }? &
  attribute editor { text }?),
  (evaluator* & text)
}

dependency = element dependency {
```

```

    attribute stepName { text },
    empty
}

evaluator = element evaluator {
    (attribute name { text } &
    attribute labid { text } &
    attribute href { text } ),
    (segid* & arg*)
}

segid = element segid {
    (attribute id { text } | element id { text } ),
    empty
}

arg = element arg {
    (attribute name { text } | element name { text } ),
    (attribute value { text } | element value { text } ),
    empty
}

labtable = element labtable {
    (attribute width { text }? &
    attribute height { text }? &
    attribute cols { text } &
    attribute rows { text } &
    attribute border { text }? &
    attribute align { text }?),
    add*
}

add = element add {
    (attribute width { text }?) &
    segment* & labtable* & allowedFormatting*
}

hint = element hint {
    (attribute name { text }?),
    text
}

allowedFormatting =
    (text | b | i | code | pre | a | img | video | br | hr | ul | li | dl)

```

```

# Formatting tags
b = element b { allowedFormatting* }
i = element i { allowedFormatting* }
code = element code { allowedFormatting* }
pre = element pre { allowedFormatting* }
ul = element ul { allowedFormatting* }
li = element li { allowedFormatting* }
dl = element dl { dt* & dd* }
dt = element dt { allowedFormatting* }
dd = element dd { allowedFormatting* }

# Insert tags
a = element a {
  ((attribute href { text } | attribute lab { text }) &
   attribute class { text }? &
   attribute style { text }? &
   attribute name { text }? &
   attribute title { text }? &
   attribute target { text }?),
  (img* & video* & text)
}

img = element img {
  (attribute src { text } &
   attribute alt { text } &
   attribute border { text } &
   attribute height { text } &
   attribute width { text } &
   attribute class { text }? &
   attribute style { text }? &
   attribute title { text }? &
   attribute target { text }?),
  text
}

video = element video {
  (attribute src { text } &
   attribute controls { text } &
   attribute height { text } &
   attribute loop { text } &
   attribute preload { text } &
   attribute width { text }? &
   attribute style { text }? &

```

```
    attribute class { text }? &
    attribute title { text }?),
text
}
```

```
# Empty formatting tags
br = element br { empty }
hr = element hr { empty }
```

Bibliography

- [1] P. Azalov. A web-based environment for introductory programming courses. *J. Comput. Small Coll.*, 22(4):260–267, 2007.
- [2] B. W. Becker. Teaching cs1 with karel the robot in java. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 50–54, New York, NY, USA, 2001. ACM.
- [3] J. Bergin, J. Roberts, R. Pattis, and M. Stehlik. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [4] J. Borstler, T. Johansson, and M. Nordstrom. Teaching OO concepts—a case study using CRC-cards and BlueJ. In *FRONTIERS IN EDUCATION CONFERENCE*, volume 1. Citeseer, 2002.
- [5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
- [6] J. Clark, M. Murata, and Others. Relax NG specification. *OASIS Committee Specification*, 3, 2001.

- [7] J. Clements and D. Janzen. Overcoming obstacles to test-driven learning on day one. In *ICSTW '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 448–453, Washington, DC, USA, 2010. IEEE Computer Society.
- [8] C. Desai, D. Janzen, and K. Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, 2008.
- [9] C. Desai, D. Janzen, and K. Savage. A survey of evidence for test-driven development in academia. *ACM SIGCSE Bulletin*, 40(2):97–101, 2008.
- [10] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4, 2005.
- [11] S. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 26–30. ACM, 2004.
- [12] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3):1, 2003.
- [13] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The teach-scheme! project: Computing and programming for every student. *Computer Science Education*, 14, 2004.
- [14] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for Scheme. 12(2):159–182, 2002.
- [15] D. Hagan and S. Markham. Teaching Java with the BlueJ environment. In *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference ASCILITE 2000*. Citeseer, 2000.

- [16] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259 – 290, 2002.
- [17] D. Janzen and H. Saiedian. On the influence of test-driven development on software design. In *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*, pages 141–148, 2006.
- [18] D. Janzen and H. Saiedian. Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In *Proc. 37th Technical Symposium on Computer Science Education (SIGCSE)*, pages 254–258. ACM, 2006.
- [19] D. Janzen and H. Saiedian. Test-driven learning: intrinsic integration of testing into the CS/SE curriculum. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, page 258. ACM, 2006.
- [20] D. Janzen and H. Saiedian. Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*, 25(2):77–83, 2008.
- [21] D. Janzen and H. Saiedian. Test-Driven Learning in Early Programming Courses. In *Proc. 38th Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2008.
- [22] D. Janzen and H. Saiedian. Test-driven learning in early programming courses. *ACM SIGCSE Bulletin*, 40(1):532–536, 2008.
- [23] R. Kaufmann and D. Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 299. ACM, 2003.

- [24] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.
- [25] M. Kolling and J. Rosenberg. BlueJ-The Hitch-Hikers Guide to Object Orientation. *Journal of Object-Oriented Programming*, 2001.
- [26] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively wiki a development environment for creating and sharing active web content. In *WikiSym '09: Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, pages 1–10, New York, NY, USA, 2009. ACM.
- [27] D. Mirkovic and S. Johnsson. CODELAB: A Developers' Tool for Efficient Code Generation and Optimization. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 729–738, 2003.
- [28] R. E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1981.
- [29] N. Truong, P. Bancroft, and P. Roe. ELP—A Web Environment for Learning to Program. In *Proc. The 19th Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education, Auckland, New Zealand*, volume 19, pages 661–670, 2002.
- [30] N. Truong, P. Bancroft, and P. Roe. A web based environment for learning to program. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, page 264. Australian Computer Society, Inc., 2003.
- [31] N. Truong, P. Bancroft, and P. Roe. Learning to program through the web. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, page 13. ACM, 2005.

- [32] D. W. Valentine. Cs educational research: a meta-analysis of sigcse technical symposium proceedings. *SIGCSE Bull.*, 36(1):255–259, 2004.
- [33] K. Van Haaster and D. Hagan. Teaching and learning with BlueJ: an Evaluation of a Pedagogical Tool. In *Information Science+ Information Technology Education Joint Conference, Rockhampton, QLD, Australia*. Citeseer, 2004.
- [34] R. Warnakulasooriya, D. Palazzo, and D. E. Pritchard. Evidence of problem-solving transfer in web-based socratic tutor. In *Physics Education Research Conference 2005*, volume 818 of *PER Conference*, pages 41–44, Salt Lake City, Utah, August 10-11 2005.
- [35] D. Yoo, B. Hickey, E. Schanzer, and S. Krishnamurthi. Wescheme. online at <http://www.wescheme.org/>.