Overcoming Obstacles to Test-Driven Learning on Day One

John Clements
Department of Computer Science
Cal Poly State University
San Luis Obispo, USA
clements@brinckerhoff.org

David Janzen
Department of Computer Science
Cal Poly State University
San Luis Obispo, USA
djanzen@calpoly.edu

***Abstract*—We describe the preliminary construction of a web-based tool for test-driven learning in the first weeks of programming. We discuss obstacles to test-driven learning—both pragmatic and ideological—and describe the ways that we believe our tool overcomes these obstacles.**

I. Introduction

"Develop a function." We do it every day, and our students do, too. For many of them, though, this is a new experience, and they must learn this skill. What are the first steps in developing a function? *Test-driven learning*[1], [2], [3] suggests that a disciplined design process starts with a clear mental model, and that the best way to make this model concrete is to write down examples: examples of what the function accepts, and what the function would then produce.

Using a computer, though, we can do more than simply write down examples in a human language; we can express these examples as test cases, that can be checked mechanically. Running these test cases verifies that the program behaves as the test cases indicates, but test-driven learning (TDL) suggests that this is *not* their principal utility; rather, the process of formulating examples in a way that is sufficiently concrete to be testable requires the designer to think ahead of time about the structure of the function.

Furthermore, the structure that the students deduce during this process is not that of the code—what loops go where, which statements are evaluated first—but rather of the function itself. How are the inputs related to the outputs? What groups of inputs produce related outputs? In short, what is the structure of the function itself? TDL argues that this understanding of the problem moves students from an understanding of a program as a sequence of machine operations to an understanding of a program as a model for a desired function.

Unfortunately, TDL is difficult to deploy. There are a parade of obstacles in the path of a teacher who wishes to train his students to think this way. In order to simplify this process, we are in the process of building a tool that aims to eliminate nearly all of these obstacles. An early version is now running—tentatively named "Smootxes", at http://smootxes.org:8001/go.ss—and in this paper we describe what we believe to be the major obstacles to the deployment of TDL, and the ways in which our tool addresses these problems.

Work supported by NSF Grant DUE-0942488

The next section describes our reasons for deciding to build this tool for the web. Section III shows screenshots of the tool as the students move through the earliest stages. Sections IV through VII describe obstacles and potential solutions. We conclude with future and related work.

II. A Web-Based Tool

For a tool such as this one, a web-based design has many advantages.

A. Ease of Deployment

One of the most significant hurdles in any programming education is that of installation and maintenance of software. This is particularly true at the elementary- and secondary-school level (K-12), where the teachers have no time and minimal institutional support.

For these settings, web-based tools—those that require only a network connection and a browser—are clear winners. Installing an IDE for programming work is just barely feasible, and requiring additional installation work to add test-driven learning tools often is not. Curiously, web-based development environments have been slow to emerge, though they are now becoming more common.

Accordingly, our tool takes the form of a web application. Users interact with a server by entering text into text boxes in response to challenges. Currently, evaluation of the user's code is performed on the server side for simplicity. Naturally, this solution doesn't scale well, but it does have the advantage of placing minimal requirements on the client hardware, an advantage in classrooms with extraordinarily underpowered computers.

B. Ease of Use

Another problem with many IDE's is the complexity of their user interfaces. Environments such as Eclipse have panels within panels, and evaluating a piece of code may require choosing from a drop-down menu attached to a button that is easily overlooked. This environment is an excellent one, but it is not designed for first-time programmers. The simplest interface we're aware of is DrScheme's[4], which provides users with only four buttons: "Step", "Check Syntax", "Run", and "Stop".

Our tool follows this lead still further; in its current state, its only button is "submit" (see Figure 1 below). This is possible because—in the early stages of learning—user interaction follows the model of "answering a question," rather than "designing a program." Our aim is to carry this simplicity forward, always presenting users with the simplest interface that allows them to complete their tasks efficiently.

C. Persistence

Beginning students often struggle to transfer their programming work between "lab machines" and their home computers. A web-based tool eliminates this problem, making it possible simply to walk away from one computer and continue working on another one.

D. Logging

Another advantage of web-based programming tools is the ability to compile extensive statistics on student experience. What exercises are difficult for students? Are there common mistakes that a rewording would solve? Do students do their work ahead of time, or right at the last minute?

E. Instructor Collaboration

Web-based tools make it easier for instructors to collaborate. Specifically, a tool such as ours contains a repository of exercises. Instructors that can pick and choose from existing exercises to build a problem set and contribute their own to the pool will cause the set of available exercises to grow rapidly and be adopted widely.

F. Internet Connection

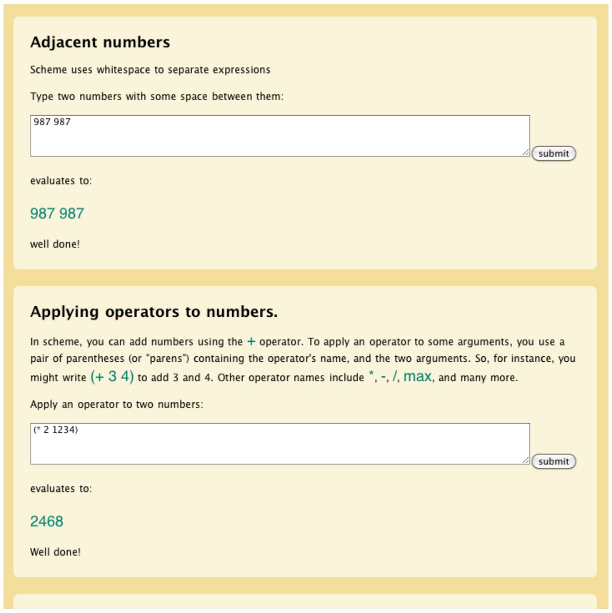
The primary disadvantage of deploying a web-based tool is the requirement for "always on" internet connectivity. In this age of ubiquitous computing and educational investment in technological infrastructure, we see this as a disappearing issue. However, we are careful to design for minimal bandwidth requirements.

III. An Example

In order to better understand the choices that we've made and to frame the discussion for the rest of the paper, we show a number of screenshots from our web application.

A. First Interaction

Figure 1 shows an early interaction, where students are learning to enter numbers and combine them into expressions. Observant readers will surely notice that the tool uses the PLT Scheme language. We will defer a discussion of language choice to later sections, but we should state at the outset that our goal is to provide exercises for a range of different languages.



Adjacent numbers

Scheme uses whitespace to separate expressions

Type two numbers with some space between them:

```
987 987
```

evaluates to:

987 987

well done!

Applying operators to numbers.

In scheme, you can add numbers using the + operator. To apply an operator to some arguments, you use a pair of parentheses (or "parens") containing the operator's name, and the two arguments. So, for instance, you might write (+ 3 4) to add 3 and 4. Other operator names include *, -, /, max, and many more.

Apply an operator to two numbers:

```
(* 2 1234)
```

evaluates to:

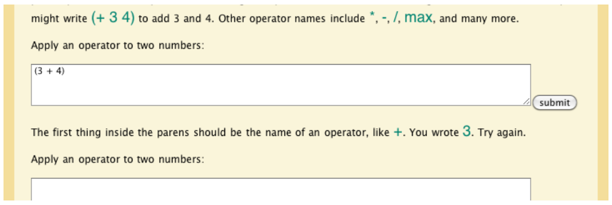
2468

Well done!

Figure 1. Learning to enter numbers and call numeric functions

We should also point out that the pedagogic orientation of this example is taken largely from *How To Design Programs*[5], which uses test-driven-learning extensively.

Smootxes uses an interactive question-and-answer format. Students complete each question before going on to the next one. However, they can return at any time to an earlier problem, and continue from that point instead.[1] Smootxes capitalizes on the tightly controlled environment to provide focused error feedback. In this particular example, for instance, placing the + in the middle elicits the message shown in Figure 2.



might write (+ 3 4) to add 3 and 4. Other operator names include *, -, /, max, and many more.

Apply an operator to two numbers:

```
(3 + 4)
```

The first thing inside the parens should be the name of an operator, like +. You wrote 3. Try again.

Apply an operator to two numbers:

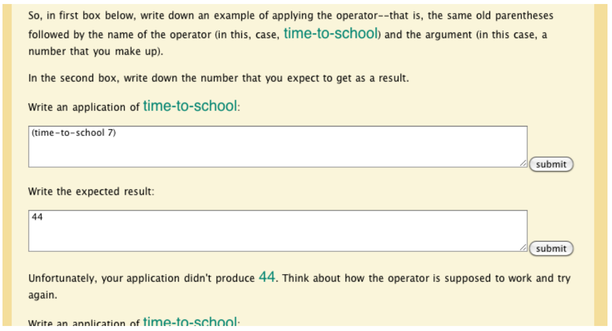
Figure 2. Focused error messages

Fortunately, since Smootxes is attached to an existing evaluator, the exercise-writer is not obliged to provide code to evaluate the student's expression. The evaluator's results may be passed directly to the student.

B. Developing Tests

Within four or five interactions, Smootxes requires students to start thinking in a test-driven way. Specifically,

[1]This is enabled by the use of a continuation-based web server[6]



So, in first box below, write down an example of applying the operator--that is, the same old parentheses followed by the name of the operator (in this, case, time-to-school) and the argument (in this case, a number that you make up).

In the second box, write down the number that you expect to get as a result.

Write an application of time-to-school:

```
(time-to-school 7)
```

Write the expected result:

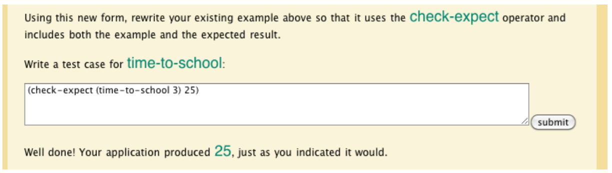
```
44
```

Unfortunately, your application didn't produce 44. Think about how the operator is supposed to work and try again.

Write an application of time-to-school:

Figure 3. Learning to enter numbers and call numeric functions

students must design an example for a simple *time-to-school* function that consumes a number and produces a number. Students enter the application and the expected value in separate boxes, as shown in Figure 3.



Using this new form, rewrite your existing example above so that it uses the check-expect operator and includes both the example and the expected result.

Write a test case for time-to-school:

```
(check-expect (time-to-school 3) 25)
```

Well done! Your application produced 25, just as you indicated it would.

Figure 4. Writing examples as test cases

Again, the tightly controlled environment makes it possible to provide focused and helpful feedback.

Next, students must learn how to represent an example and an expected value together as a test case. Figure 4 shows this transition.

From here, students will learn how to express the formula that computes the answer, and how to represent this formula as a program. For instance, a student might wind up with this definition:

;; time-to-school : number → number
;; computes the time required to get to school
(**define** (*time-to-school kms*)
 (+ 10 (∗ *kms* 5)))

After writing the definition itself, Smootxes will verify that the student's definition causes the test cases to pass.

IV. Test Syntax: simpler is better

Learning to program is hard. Why make it harder, by requiring students to learn additional syntax in order to express their test cases? This is the most commonly stated objection to test-driven learning—and to testing in general.

We agree completely with this sentiment, and we feel that the fault lies not with the idea of testing, but rather with the imposition of complex frameworks that require students to couch their test cases in several layers of syntax.

Our tool illustrates the idea that test cases need not be syntactically complex. At a bare minimum, a test case must include

1) Some indication that it's a test case,
2) an expression to be evaluated, and
3) a desired result.

These requirements are satisfied handily both by PLT Scheme's **check-expect** and by JUnit's *assertEquals*.

Unfortunately, using *assertEquals* in the context of a typical IDE requires lots of extra baggage. Here are the first 15 lines of the template provided by BlueJ for the creation of test cases:

```
//add import statements here
/**
* The test class MyTestCases.
*
* @author  (your name)
* @version (a version number or a date)
*/
public class MyTestCases extends junit.framework.TestCase
{
    /**
     * Default constructor for test class MyTestCases
     */
    public MyTestCases()
    {
    }
...
```

Adding a test case requires an additional layer of syntactic wrapping:

```
public void testAddition(){
    assertEquals(4,3+1);
}
```

All of this verbiage is justified and indeed vital for a large project that needs source tracking and javadoc. However, it will give a first-year student serious pause.

The problem, though, is not JUnit's design; rather, it is the one-size-fits all attempt to cast the relatively straightforward problem of writing test cases into the much more complex problem of formulating a class containing a library of testing methods. Any solution that requires students to read and edit complex blocks of code in order to write test cases will almost certainly wind up derelict.

By contrast, An IDE that allows students to write test cases in a straightforward way with negligible syntactic overhead has at least a fighting chance of making students "test-infected."

A. TDL in the classroom

Test-driven learning is a wonderful addition to the classroom, as well. This topic is beyond the scope of this paper, but it's worth noting that the syntactic simplification that makes it possible for students to formulate test cases is even more valuable when instructors use it in the classroom, where all of the code must fit on a whiteboard or a slide. Modeling test-first behavior in the classroom is one of the best ways to ensure that students can replicate the process at home.

V. TDL and I/O

Test-driven learning has an uneasy relationship with I/O. In many environments, it is difficult to formulate test cases for code that "crosses the process boundary" by interacting with ports.

A. *Test Forms for I/O*

In part, we have a solution to this problem. That is, we have developed a **check-with-input** form that allows users to formulate tests for code that reads from a port, or writes to a port.

Specifically, the **check-with-input** form evaluates its expression in a context where input is taken from the given string. For instance, we might test a *read-first-word* method by writing:

```
(check-with-input "abc def" (read-first-word) "abc")
```

By the same token, **check-with-output** checks to see whether its expression evaluates to the given value and produces the required output. For instance, we might test a method that prints "moo" and returns 3 by writing:

```
(check-with-output (print-moo) 3 "moo")
```

B. *The Larger Problem*

These test forms are fine for testing programs that read input and that produce output, but they are not adequate to model a sequence of interactions with a user. For instance, it may be that students are implementing a simple moon-lander game that shows the user the state of a moon lander and then requires him or her to indicate how much fuel to use in the next "turn."

The underlying problem with testing a program like this one is that it's extraordinarily hard to specify its behavior cleanly. For instance,

1) The program should always eventually respond to a user's input.
2) The program should not use part of the "last answer" to answer this turn's question (easy to do with input streams).
3) The program should not continue until the user has responded.

Programs with properties like these are not within the reach of simple testing, and it's because they're not modeling simple mathematical operators that map inputs to outputs in a fixed way.

Put differently, these programs make use of *effects*.

VI. Effectful Programming

Programs with effects are difficult to test, and (correspondingly) can contain subtle bugs. The prior section contained one example, that of a program using I/O to communicate with an outside user.

There is another large category of effects that lead to programs that are difficult to test, difficult to write, and difficult to debug: *mutation*. That is, programs that change the values of variables and/or class fields from one value to another.

Programs that perform mutation are a part of many first-year curricula, and it would likely be fruitless to argue that mutation does not belong there. Instead, we would simply like to draw attention to the difficulty of testing methods that perform mutation.

For instance, consider a queue, written in a traditional style, with an *enqueue* and a *dequeue* method. If the *enqueue* method returns nothing (PLT Scheme calls this (*void*), students might write this test case:

```
(check-expect (enqueue (new-queue empty) 14) (void))
```

... and that would be a valid test case, as far as it goes. However, it would fail to capture the important effect caused by the *enqueue* method, specifically the addition of 14 to the queue. Moreover, in the context of TDL, a student that has written this particular test case has probably not yet given much thought to the full meaning of the method.

A more exhaustive test case might read:

```
(define my-queue (new-queue empty))
(check-expect (enqueue my-queue 14) (void))
(check-expect my-queue (new-queue (list 14)))
```

This test ensures both that the result of enqueueing 14 is (*void*), and also that the resulting queue contains the element 14. Moreover, this test fits well with test-driven learning, and indicates that the student has given some thought to what the state of the queue should be after the call to *enqueue*.

So, effectful programs are more difficult to test: is this a problem with testing, or a problem with effectful programs? Perhaps the diplomatic answer is simply that teachers who wish to use test-driven learning might be well-advised to choose problems without effects in the early stages of learning.

VII. Extensional equality testing

The prior section's example highlights another important property of a testing framework: the ability to perform *extensional* equality testing. An extensional equality test is one that checks behavior, rather than identity.

Specifically, the final test case in the block above compares the student's queue, *my-queue*, with a fresh queue containing only 14, (*new-queue* (*list* 14)). In JUnit, for instance, we might write this test as:[2]

```
assertEquals(new Queue(Seq.sequence(14)),
             myQueue);
```

This test case will fail, unless the developer of the Queue class has equipped it with an *equals* method that checks the values of the fields for equality. The default *equals* method instead performs an *intensional* equality check, comparing the two pointers directly.

Many teachers gamely embrace this challenge, training students to write *equals* methods for all of their classes. However, in the context of early learning, the need to define

[2]We assume the existence of a `Seq.sequence` static method that constructs an ArrayList containing the desired elements.

equality methods in all classes greatly intensifies the pain of testing and programming in general.

Unsurprisingly, our web tool performs extensional equality testing in order to avoid this problem.

Perhaps more surprisingly, it turns out that extensional equality checking is not incompatible with Java. Viera Proulx[7] provides a `Tester` library that uses the reflection API to allow extensional equality testing on user-defined classes.

VIII. Future Work

Our work is just beginning, and the list of features that we're looking forward to adding is lengthy. Among these, though, are several that promise to be interesting. We comment on a number of these below.

A. Java Challenges

Our tool currently teaches students how to write Scheme expressions and tests. How would the tool be different if it were targeting Java?

Again, our goal is to keep things as simple as possible. That is, we can imagine translating the examples above as directly as possible. So, for instance, (+ 3 4) becomes 3 + 4, and (*time-to-school* 3) becomes *timeToSchool*(3). Aside from the minor overhead in showing two different syntactic conventions for infix and prefix functions, these examples are straightforward.

In a similar way, we anticipate translating uses of **check-expect** directly into uses of *assertEquals*, or Viera Proulx's *checkExpect*.

Other things will be more difficult. Should the students be required to develop constructors, or to provide *public* and *protected* annotations? Our hope is that a gentle introduction will ease the students into these requirements, though Java is not a language that lends itself to syntactic simplification.

A related problem is that of interpreting error messages. Any system that translates (*compiles*) user code into another language must somehow map that other language's error messages back into terms that the student can understand. Even when the "translation" consists only in placing the student's code within a class wrapper, it is common to find examples where the error messages refer to locations within the non-student code.

B. TDL and Coverage

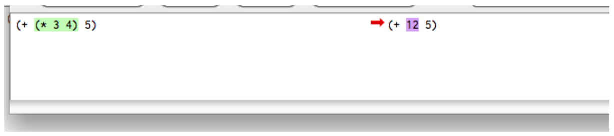
One of the golden opportunities of a TDL tool such as Smootxes is the ability to use coverage information to drive test case feedback. For instance, consider a function that merges two sorted lists. A student might supply this test case:

(**check-expect** (*merge-lists* (*list* 1 4 7) (*list* 6)))

By running this test case against an instructor-supplied definition, Smootxes might discover that this test case fails to cover the case of the first list being shorter than the second. This kind of feedback could potentially help students to better understand the domain of inputs, and how to make sure that they've considered all of the possible inputs.

C. Stepping

Another opportunity for Smootxes is a potential collaboration with DrScheme's stepper[8]. The stepper (one of us is the author of this tool) shows the evaluation of student code as a series of reduction steps. Figure 5 shows a simple step using this tool.



```
(+ (* 3 4) 5)                    (+ 12 5)
```

Figure 5. Reducing expressions to values

D. A Scripting Language

The success of a tool like Smootxes will hinge in large part upon its ability to allow teachers to design their own lessons and write their own exercises. This requires more than simply writing two or three blocks of text, though; an exercise must parse a user's code to provide feedback, rewrite it in some way, associate it with pre-written definitions, run it, and then analyze the result to provide feedback.

Our existing tool therefore contains a quantity of code for each exercise. As we develop more exercises, we expect to be able to discern patterns and discover a domain-specific language[9], [10] for expressing these exercises concisely.

E. Bridging the Gap

A tool such as Smootxes is merely a stepping stone. If students are to build larger programs and move beyond the sheltered waters of its one-button interface, they will need to transfer their knowledge to a traditional IDE, and to a setting where a program consists of source files, makefiles, scripts, and command-line tools. A tool like Smootxes will be most useful to students and teachers if it provides a smooth transition to existing development environments.

F. Evaluation

As our tool evolves, it will be critical to conduct experiments on students to validate the tool's utility.

IX. Related Work

TDL builds on the ideas in Meyer's work on Design by Contract [11], and was inspired by the Explanation Test [12] and Learning Test [12] testing patterns proposed by Kent Beck, Jim Newkirk, and Laurent Bossavit.

The approach of requiring students to write tests in lab and project exercises has a number of predecessors. Barriocanal [13] documented an experiment in which students were asked to develop automated unit tests in programming assignments. Christensen [14] proposes that software testing should be incorporated into all programming assignments in a course, but reports only on experiences in an upper-level course. Patterson [15] presents mechanisms incorporated into the BlueJ [16] environment to support automated unit testing in introductory programming courses. The *How To Design Programs* textbook [5] requires students to formulate examples before writing code. Edwards [17] has suggested an approach to motivate students to apply TDD that incorporates testing into project grades, and he provides an example of an automated grading system that provides useful feedback. A number of early adopters have successfully incorporated TDD into first year programming courses [18], [19].

A variety of web-based tools exist to support learning to program, from static lab exercises to applets that let you interact with data structures. Recently, more advanced web-based tools have begun to emerge, including WeScheme[20], JavaBat, ELP[21], and turingscraft. Smootxes brings an extra ingredient to this mix, in the form of test-driven learning.

Finally, outside of programming, Koedinger[22] has amazing results in the mathematical domain.

References

[1] D. S. 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*. New York, NY, USA: ACM, 2006, pp. 254–258.

[2] D. Janzen and H. Saiedian, "Test-driven learning in early programming courses," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2008, pp. 532–536.

[3] C. Desai, D. S. Janzen, and J. Clements, "Implications of integrating test-driven development into CS1/CS2 curricula," in *Proceedings of the 40th ACM technical symposium on Computer science education*. New York, NY, USA: ACM, 2009, pp. 148–152.

[4] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "Drscheme: A programming environment for Scheme," *Journal of Functional Programming*, vol. 12, no. 2, pp. 159–182, 2002.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How To Design Programs*. MIT Press, 2001.

[6] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen, "Implementation and use of the plt scheme web server," *Higher-Order and Symbolic Computing*, vol. 20, no. 4, pp. 431–460, 2007.

[7] V. K. Proulx, "Test-driven design for introductory OO programming," in *Proceedings of the 40th ACM technical symposium on Computer science education*, 2009, pp. 138–142.

[8] J. Clements, M. Flatt, and M. Felleisen, "Modeling an algebraic stepper," in *Proceedings of the 10th European Symposium on Programming*, ser. Lecture Notes in Computer Science, D. Sands, Ed., vol. 2028. Springer, 2001, pp. 320–334.

[9] J. Bentley, "Little languages," *Communications of the ACM*, pp. 711–721, August 1986.

[10] J. Clements, M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi, "Fostering little languages," *Dr. Dobb's Journal*, pp. 16–24, March 2004.

[11] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[12] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2003.

[13] E. Barriocanal, M. Urb'an, I. Cuevas, and P. P'erez, "An experience in integrating automated unit testing practices in an introductory programming course," *ACM SIGCSE Bulletin*, vol. 34, no. 4, pp. 125–128, December 2002.

[14] H. B. Christensen, "Systematic testing should not be a topic in the computer science curriculum!" in *Proceedings of the 8th Annual ITiCSE Conference*. ACM Press, 2003, pp. 7–10.

[15] A. Patterson, M. Kolling, and J. Rosenberg, "Introducing unit testing with BlueJ," in *Proceedings of the 8th Annual ITiCSE Conference*. ACM Press, 2003, pp. 11–15.

[16] M. Kolling and J. Rosenberg, "Guidelines for teaching object orientation with java," in *Proceedings of the 6th Annual ITiCSE Conference*. ACM Press, 2001, pp. 33–36.

[17] S. Edwards, "Rethinking computer science education from a test-first perspective," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications: Educators' Symposium*, 2003, pp. 148–155.

[18] K. Keefe, J. Sheard, and M. Dick, "Adopting XP practices for teaching object oriented programming," in *ACE '06: Proceedings of the 8th Austalian conference on Computing education*. Darlinghurst, Australia: Australian Computer Society, Inc., 2006, pp. 91–100.

[19] G. Melnik and F. Maurer, "A cross-program investigation of students' perceptions of agile methods," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 481–488.

[20] D. Yoo, B. Hickey, E. Schanzer, and S. Krishnamurthi. Wescheme. [Online]. Available: www.wescheme.org

[21] N. Truong, P. Bancroft, and P. Roe, "A web based environment for learning to program," in *ACSC*, ser. CRPIT, M. J. Oudshoorn, Ed., vol. 16. Australian Computer Society, 2003, pp. 255–264.

[22] K. Koedinger and V. Aleven, "Exploring the assistance dilemma in experiments with Cognitive Tutors," *Educational Psychology Review*, 2007.