Test-Driven Learning in Early Programming Courses

Test-driven development (TDD) [5] gained significant attention from professional programmers in the last decade due to its inclusion as a core practice in Extreme Programming (XP) [4]. Despite the many controversies regarding XP and agile software processes, TDD is now emerging as an industry “best practice” that is beneficial in agile software processes as well as more traditional plan-driven software processes. Although TDD may have been applied narrowly and in various forms for several decades [13, 27], Jeffries proposes that TDD has reached “an adoption level for the practice beyond the visionary phase and into the early mainstream.” [22]

TDD is a novel software development strategy that requires that automated tests be written prior to writing functional code in small, rapid iterations. TDD is a highly disciplined yet agile development approach that focuses the programmer’s attention on detailed software design decisions while developing code that is clean, correct, and flexible. The TDD rhythm of write a test, write code to make the test pass, and refactor, is enabled by xUnit frameworks that allow the programmer to quickly check small changes against an accumulated suite of tests that serve to document the code and detect defects early. In fact the creation of the free and simple JUnit [12] testing framework is likely one of the most important factors in TDD’s rapid rise.

While many industrial programmers have embraced TDD, the academic community has been slow to become familiar with and understand TDD. In fact, TDD appears to be following a path not unlike that of object-oriented computing (OO). Despite its academic roots, OO was first adopted widely in industry, particularly with the introduction of C++ in the mid-1980’s. OO made its way slowly into academic courses, starting as an advanced graduate level topic, then eventually emerging in first year programming courses. OO is now the dominant paradigm taught in first year programming courses.

Similarly TDD is emerging as an industry “best practice.” TDD has made its way into graduate and upper-level software engineering courses [3,9,11,23,31,33,34]. More recently a number of early adopters have successfully incorporated TDD into first year programming courses [24,29]. However, the following four hurdles must be overcome before TDD is likely to achieve widespread adoption in first year programming courses:

1. Evidence of TDD’s efficacy in academia
2. Evidence that TDD can be incorporated into first year programming courses without removing other topics
3. Creation of a textbook that fully adopts TDD
4. Dissemination of TDD knowledge to first year programming educators

Much of the Principal Investigator’s (PI) recent research has focused on the first hurdle. The PI and others have conducted at least a dozen empirical studies evaluating TDD in academic settings. This work will briefly be discussed in the related work section of this proposal, and is reviewed more thoroughly in [17] and [22].

The PI developed a pedagogical approach called Test-Driven Learning (TDL) [19]. TDL is an approach to teaching computer programming that involves introducing and exploring new concepts by example in a test-first manner with automated tests. TDL will be discussed in more detail in the next section. The PI has had success using TDL to teach TDD in CS1 and CS2 without impacting coverage of existing course topics, indicating that the second hurdle to TDD adoption may be achievable.
Intellectual Merit:
This proposal outlines a plan to ease and speed the adoption of TDD in academia by developing and evaluating a textbook and associated curriculum materials for a first year programming course based on the Test-Driven Learning (TDL) approach. Numerous articles and several books [2,6,26,28] have been written describing how to apply TDD, but they all target industry professionals. A couple of CS1 textbooks claim to use a TDD approach, however they have assumed the common misconception that TDD is equivalent to writing automated tests such as in JUnit, and they relegate coverage of the topic to a small section on testing late in the textbook. This project would produce the first CS1 textbook to incorporate a TDD approach throughout.

This proposal aims to improve student design and testing skills and increase programmer confidence by incorporating a disciplined test-driven approach (TDL). The PI is uniquely qualified for this work based on creation of the TDL approach, completion of a series of funded and award winning TDD and TDL empirical studies, over a decade of undergraduate teaching experience, and access to a mix of educational environments.

Broader Impacts:
Creation of a TDD-based textbook will increase awareness of TDD and ease adoption among first-year programming teachers. Broadening TDD adoption is expected to improve student skill in software design and testing without degrading student mastery of existing CS1 topics. Improvements in foundational courses should result in improved software professionals and the products they produce. Although it is yet unstudied, the use of TDD may actually improve student retention to the second programming course based on increased student confidence.

In order to reach the widest possible audience of computer science, software engineering, and STEM students, the textbook will target Java which is the most popular first-year language choice. The textbook and lab materials will be evaluated in three contexts: a large public university, a two-year public community college, and two small four-year liberal arts colleges. Materials will be disseminated initially through a textbook publisher, email lists, and conferences such as the ACM SIGCSE Technical Symposium on Computer Science Education.

1 Test-Driven Learning

Programmers often learn new programming concepts and technologies through examples. Instructors and textbooks use examples to present syntax and explore semantics. Tutorials and software documentation regularly present examples to explain behaviors and proper use of particular software elements. Examples, however, typically focus on the use or the interface of the particular software element, without adequately addressing the behavior of the element.

Consider the following example:

```java
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(5);
al.add(-14);
al.add(29);
System.out.println("The ArrayList contains " + al.size() + " elements");
System.out.println(" Element 0: " + al.get(0));
System.out.println(" Element 1: " + al.get(1));
System.out.println(" Element 2: " + al.get(2));
```

This is a reasonable example of how to use an ArrayList in Java, but it only reveals the desired interface. It teaches nothing about the underlying behavior. To see behavior, one must compile
and execute the code. While it is desirable to encourage students to try things out on their own, this can be time consuming if done for every possible example, plus it significantly delays the presentation/feedback loop.

As an alternative, we can introduce a simple automated unit test that demonstrates both the interface and the expected behavior. For instance, we could replace the above example with the following that uses the JUnit framework.

```java
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(5);
al.add(-14);
al.add(29);
assertEquals(3, al.size());
assertEquals(5, al.get(0));
assertEquals(-14, al.get(1));
assertEquals(29, al.get(2));
```

This example shows not only the same interface information as the original example in roughly the same amount of space, but it also shows the behavior by documenting the expected results.

This example demonstrates the basic idea of test-driven learning (TDL):

- Teach by example
- Present examples with automated tests
- Start with tests

Teaching by example has a double meaning in TDL. First TDL encourages instructors to teach by presenting examples with automated tests. Second, by holding tests in high regard and by writing good tests, instructors model good practices that contribute to a number of positive results. Students tend to emulate what they see modeled. So as testing becomes a habit formed by example and repetition, students may begin to see the benefits of developing software with tests and be motivated to write tests voluntarily.

The third aspect of TDL suggests a test-first approach. With a test-first approach, the test would be written prior to implementing a concept. By writing a test before implementing the item under test, attention is focused on the item’s interface and observable behavior.

For example, suppose we want to develop a web application that has a user login and password. We want to validate that passwords meet the following criteria: passwords must be between six and ten characters long, contain at least one digit, and contain at least one capital (upper case) letter. We might start with the following test to check that a password of length six is valid. We will show the test code on the left and the source (production) code on the right. As we go, new code being added will be highlighted in yellow.
Unfortunately this test doesn’t pass. In fact it doesn’t even compile because we haven’t created the PasswordValidator class yet. However we have decided that we need a class called PasswordValidator and that it should have a default constructor and an isValid method that accepts one string parameter and returns a boolean result. Next we create the PasswordValidator class with the isValid method.

Now we can run JUnit and see if our test passes. Figure 1 demonstrates the green bar that JUnit gives in Eclipse when this test passes.
At this point, we should reflect on what we have written. Does it look good? Do we have any duplicate code that we can merge? Is there a better way to do anything in either the source or the test code? If not, we can move on to our next test.

The second test that we might write would ensure that our new method rejects an invalid password. Let’s give it a password that is too short.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
    @Test
    public void testValidLength () {
        PasswordValidator pv = new PasswordValidator ();
        assertEquals (true ,pv.isValid ("Abc123 ");
    }

    @Test
    public void testInvalidLength () {
        PasswordValidator pv = new PasswordValidator ();
        assertEquals ( false ,pv.isValid ("Abc12 ");
    }
}
```

Figure 1: Passing JUnit test in Eclipse for PasswordValidator
If we did everything correctly, then our tests will pass and we will get a green bar. If not, we will get a red bar and we need to fix what is wrong. Once the tests all pass, we can again look for any duplication. In this case we see duplication in our two tests where we construct a PasswordValidator object. We can refactor to consolidate the common code in a setUp method in the TestPasswordValidator class. We will make the PasswordValidator object available to all our test methods by declaring it as a private member in our test class. For brevity, we will show the results of this step in our next example.

What should we do next? How about a test to check that a password with no capital (upper-case) letters is invalid. We’ll do that in a new test method named testNoUpperCase.

```java
public class TestPasswordValidator {
    PasswordValidator pv;

    @Before
    public void setUp() {
        pv = new PasswordValidator();
    }

    @Test
    public void testValidLength() {
        assertEquals(true, pv.isValid("Abc123"));
    }

    @Test
    public void testInvalidLength() {
        assertEquals(false, pv.isValid("Abc12"));
    }

    @Test
    public void testNoUpperCase() {
        assertEquals(false, pv.isValid("abc123"));
    }
}
```

As you might have suspected, this new test fails because we haven’t yet added the code to check for upper case letters in isValid. We will leave that as an exercise for the reader.

### 1.1 TDL in later courses

TDL is applicable at all levels of learning. Advanced students and even professional programmers in training courses can benefit from the use of tests in explanations. The PI has used TDL in upper-level undergraduate, graduate, and even industry training courses for professional programmers.

The final example below demonstrates the use of TDL when exploring Java’s DefaultMutableTreeNode class. Such an example might surface when first introducing tree structures in a data
structures course, or perhaps when a programmer is learning to construct trees for use with Java’s
JTree class. Notice the use of the breadthFirstEnumeration() method and how the assert statements
demonstrate not just the interface to an enumeration, but also the behavior of a breadth first search.
A complementary test could be written to explore and explain depth first searches.

```java
import javax.swing.tree.DefaultMutableTreeNode;
import static org.junit.Assert.*;
import org.junit.Test;

public class TreeExploreTest {
    @Test
    public void testNodeCreation() {
        DefaultMutableTreeNode node1 = new DefaultMutableTreeNode("Node1");
        DefaultMutableTreeNode node2 = new DefaultMutableTreeNode("Node2");
        DefaultMutableTreeNode node3 = new DefaultMutableTreeNode("Node3");
        DefaultMutableTreeNode node4 = new DefaultMutableTreeNode("Node4");
        node1.add(node2);
        node2.add(node3);
        node1.add(node4);
        Enumeration e = node1.breadthFirstEnumeration();
        assertEquals(e.nextElement(), node1);
        assertEquals(e.nextElement(), node2);
        assertEquals(e.nextElement(), node4);
        assertEquals(e.nextElement(), node3);
    }
}
```

In this case, TDL is not meant to replace other powerful pedagogical tools such as data structure
and algorithm visualizations. The point is simply that TDL can be used throughout the curriculum
whenever software examples are used.

1.2 TDL Related Work

Test-driven learning is not a radical new approach to teaching computer programming. It is a
subtle, but potentially powerful way to improve teaching, both in terms of efficiency and quality of
student learning, while accomplishing several important goals.

TDL builds on the ideas in Meyer’s work on Design by Contract [30]. Automated unit tests
instantiate the assertions of invariants and pre- and post-conditions. While contracts provide im-
portant and rigorous information, they fail to communicate and implement the use of an interface
in the efficient manner of automated unit tests. Contracts have been suggested as an important
complement to TDD [16]. The same could be said regarding TDL and contracts.

TDL is expected to encourage adoption of TDD. Although its name implies that TDD is a
testing mechanism, TDD is as much or more about analysis and design as it is about testing, and
the combination of emphasis on all three stands to improve software quality.

TDL was inspired by the Explanation Test [6] and Learning Test [6] testing patterns proposed
by Kent Beck, Jim Newkirk, and Laurent Bossavit. These patterns were suggested as mechanisms
to coerce professional programmers to adopt test-driven development.

The Explanation Test pattern encourages developers to ask for and provide explanations in
terms of tests. The pattern even suggests that rather than explaining a sequence diagram, the
explanation could be provided by “a test case that contains all of the externally visible objects and
messages in the diagram.” [6]
The Learning Test pattern suggests that the best way to learn about a new facility in an externally produced package of software is by writing tests. If you want to use a new method, class, or API, first write tests to learn how it works and ensure it works as you expect.

TDL expands significantly on the Explanation and Learning Test ideas both in its approach and its audience. Novice programmers will be presented with unit tests as examples to demonstrate how programming concepts are implemented. Further, programmers will be taught to utilize automated unit tests to explore new concepts.

While the idea of using automated tests as a primary teaching mechanism is believed to be a new idea, the approach of requiring students to write tests in lab and project exercises has a number of predecessors. Barriocanal [3] documented an experiment in which students were asked to develop automated unit tests in programming assignments. Christensen [7] 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 [35] presents mechanisms incorporated into the BlueJ [25] environment to support automated unit testing in introductory programming courses.

Edwards [8] 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. TDL pushes automated testing even earlier, to the very beginning in fact.

2 Evidence of TDD and TDL Efficacy

Numerous empirical studies focused on TDD effects on defects (external quality) and productivity. Most of these are summarized in [17] and [22]. Nearly all studies report that students wrote significantly more tests when they applied TDD. Not surprisingly, most studies indicated that they found fewer defects in code produced with TDD. Yenduri and Perkins [36] reported a 35% reduction in defects when senior students used TDD and Edwards [10] reported a 45% reduction among juniors using TDD and an automated grading system. George [14] reported fewer defects but lower productivity. Erdogmus [11] reported minimal external quality differences but improvements in productivity, likely attributed to less time fixing late-emerging defects. Geras [15] reported no changes in productivity, but less frequent unplanned test failures among TDD programmers.

Despite some confounding data, the general consensus appears to be that TDD may reduce defects, but possibly with an increase in initial cost/time [22].

The emphasis on external quality is valid and beneficial. Yet focusing on external quality perhaps emphasizes the testing aspects of TDD while missing the fact that TDD is first and foremost a design mechanism.

The PI and colleagues recently conducted a series of leveled studies [18, 20, 21] examining how the TDD process affects internal design quality, focusing on the key artifacts of software architecture, source code, and automated unit tests. Controlled experiments were conducted in five undergraduate and graduate academic courses, in one professional training course, and with five in-house professional development projects in a Fortune 500 company. In all the experiments, both test-first (TDD) and test-last groups wrote automated unit tests in short, rapid iterations. The only difference was whether the tests were written just before or just after writing small amounts of functional code.

The undergraduate experiments included both early programmers (CS1 and CS2) and upper-level software engineering students. All together the experiments involved over 230 student and professional programmers working on almost five hundred software projects ranging in size from one hundred to over 30,000 lines of code. The research also included a case study of fifteen software projects developed over five years in a Fortune 500 corporation.
This research indicated that software developers applying a test-first (TDD) approach are likely to improve some software quality aspects at minimal cost over a comparable test-last approach. In particular, this research has shown that test-first developers develop software that has significantly lower code complexity (McCabe’s Cyclomatic Complexity and Weighted Methods per Class), significantly smaller size (method lines of code and methods per class), and significantly higher test coverage without significantly increasing software development effort [20].

Surveys conducted with study participants indicated that mature programmers who have used both the test-first and test-last development approaches prefer the test-first approach. In addition, students that used test-first (TDD) and test-last approaches were more likely to choose TDD in the future than students who only used a test-last approach [21]. Müller et al. observed that use of TDD resulted in an increase in program understanding [33] and confidence in making changes to the code and code correctness [32].

Finally, the PI and colleagues are currently completing analysis on a study sponsored by Lockheed Martin that demonstrated that TDD could be incorporated into a second quarter Java programming course without removing any course topics.

3 Project Plan

This section discusses the activities proposed to occur in this project, the personnel who will carry out these activities, and a timeline in which the activities will occur.

3.1 Creation of a textbook

This project proposes to create a Java-based textbook for the first programming course (CS1) that applies the TDL approach from beginning to end. In order to reach the widest possible audience of computer science, software engineering, and STEM students, the textbook will target Java which is the most popular first-year language choice. Java is a particularly good choice for TDD due to the simplicity and widespread use of the JUnit framework. JUnit integrates nicely with most all integrated development environments (IDE). JUnit even comes bundled with the widely used Eclipse IDE as well as the pedagogical IDE BlueJ.

In addition to JUnit, there are a wide variety of TDD-related tools for Java such as in-memory databases (e.g. HSQLDB) and mock object frameworks (e.g. EasyMock). While many of these tools would not be used in CS1, first year students will benefit from knowing what percent of their source code their tests are executing. Code coverage tools such as Emma (http://emma.sourceforge.net/) and Cobertura (http://cobertura.sourceforge.net/) are accessible to first year students and can even be integrated into the grading process. Significant advances have recently occurred in automated grading systems. Most notably, two web-based automated grading systems have emerged that support a variety of pedagogical objectives. Web-CAT (http://web-cat.cs.vt.edu/) was developed by Stephen Edwards of Virginia Tech, and Marmoset (http://marmoset.cs.umd.edu/) was developed by a team of researchers led by William Pugh of the University of Maryland. Web-CAT is actively under development with a growing list of more than thirty institutional users. Web-CAT is currently supported by an NSF CCLI Phase II grant.

Students benefit from early, rapid, and frequent feedback. Not only do these new automated grading systems provide this kind of fast feedback, but they also provide advanced features such as code style checking, software metric calculations, and mixed automated and manual grading. The systems are highly configurable and can be used to support many teaching styles, but they are particularly useful when incorporating instruction on software testing even in early programming courses. They have been shown to significantly improve student success in learning to program [8].
This textbook will include references and perhaps tutorials for these automated grading systems, and many adopters of the textbook are expected to also adopt an automated grading system.

Aside from the adoption of the TDL approach, the textbook will target a “mainstream” audience, focusing on traditional CS1 topics as defined in the Computing Curricula 2001 [1]. Although the topics in the textbook will be traditional (e.g. control structures, classes, methods, exceptions, inheritance, input/output, ...), TDL is expected to drive a slightly non-traditional order of coverage. Most notably, TDL works better with early coverage of functions and later coverage of input/output (I/O).

Traditional CS1 courses typically spend significant time on input/output as this is the primary mechanism by which students can write interesting programs. Whether it be standard I/O, file I/O, or I/O through a graphical user interface (GUI), I/O is often a frustrating and unnecessarily tedious topic for beginning students. One solution has been the creation of supplemental libraries such as that created by the ACM Java Task Force (http://jtf.acm.org/). TDL lessens the need for I/O early in the course, allowing students to exercise their programs through tests. A very recent and yet unpublished study by the PI and colleagues demonstrated that examples with tests (i.e. TDL) in first year courses take roughly the same effort to present as examples with I/O statements or explanations, and student mastery of I/O is not reduced with later and shorter coverage of I/O.

One final expected difference from “mainstream” textbooks will be an emphasis on a functional style of programming. By the very nature of TDD, programmers often tend to write code in a more functional style. Functions that perform mutation tend to be more cumbersome to test, requiring an extra step to sense the mutation. In contrast, functions that return a value or perhaps an object tend to be simpler to test. As a result, TDL and TDD tend to drive programmers toward more functional solutions. As discussed earlier, the textbook will present a relatively “mainstream” approach with computation through control structures and object-orientation, but the astute reader will no doubt notice the functional approach mixed in.

Supplemental materials will be created to accompany the textbook. A graduate student will be employed to assist in the development of chapter exercises, labs, and projects.

### 3.2 Evaluation and Assessment

The textbook and materials will be evaluated beginning at about the halfway point of the grant. A more complete schedule is provided in the final section of this proposal. The evaluation will be led by an evaluation administrator who is trained specifically in pedagogical evaluation. The evaluations will take place in four institutions. The institutions were chosen for their diversity and to demonstrate the broader impacts of the work.

Evaluations will take place at California Polytechnic State University (Cal Poly) in San Luis Obispo, CA, Cuesta College in San Luis Obispo, CA, Bethel College in North Newton, KS, and Tabor College in Hillsboro, KS. Cal Poly is a public university with approximately 18,000 students. Cal Poly’s College of Engineering has one of the ten largest undergraduate enrollments in the country. The computer science department has approximately 500 computer science, 80 software engineering, and 40 masters students. Cal Poly’s academic schedule is based on the quarter system. Cal Poly teaches the ‘C’ programming language in the first quarter, followed by Java in the second and third quarter. The new TDL-based textbook and materials will be evaluated in at least two sections of both the first and second quarter courses at Cal Poly.

Cuesta College is a two-year public community college with about 11,000 students. Cuesta’s academic schedule is based on the semester system. Cuesta teaches CS1 using the ‘C’ programming language for the first two-thirds of the course, followed by five weeks in Java. The text will be evaluated during this five week period in late fall.
Bethel College and Tabor College are both four-year private liberal arts colleges with enrollments of about 500 students each. Bethel and Tabor were selected together because they participate in a computer science cooperative in which mid- and upper-level courses are taught collectively. Bethel and Tabor’s academic schedule is based on the semester system. Both Bethel and Tabor teach CS1 using the ‘C++’ programming language currently in the spring semester. They will adopt the new text in CS1. This will provide the opportunity to evaluate the text both in the context of a small liberal arts college, as well as in the first year of a transition to using Java.

The evaluations will consist of both qualitative and quantitative evaluations. Qualitative evaluations will include peer review of the textbook and accompanying materials, student surveys, and student interviews. These qualitative evaluations are expected to glean specific recommendations and insights that will result in improvements in the curricular materials.

Quantitative evaluations will provide the statistical analysis of the project deliverables. Controlled experiments will be conducted at each of the four evaluating institutions. Control and experimental groups will be established with the independent variable being the textbook choice. Control groups will use the currently used text at each institution. Experimental groups will use the new TDL-based textbook. Pre-experiment surveys will be conducted to ensure that the student populations in each course are statistically similar in terms of previous programming experience and gpa. Students will be expected to report data on the time they spend completing programming projects for the course. Evaluating instructors will provide all student lab and project artifacts so they can be compared in terms of test coverage, defect density, and internal quality (e.g. LOC, Cyclomatic Complexity). Post-experiment surveys will be conducted to measure students’ experiences and changes in their opinions. Finally retention rates to the second programming course (CS2) will be measured and compared between the control and experimental groups. Differences in minority and women students will be examined for all dependent variables. Approval from Human Subjects or Institutional Review Boards will be obtained from all four institutions.

3.3 Personnel

The Principal Investigator is uniquely qualified for this project. Dr. Janzen has more than a decade of undergraduate teaching experience, five years of industry experience with Sprint Corporation, and has taught at least 25 short courses to more than four hundred industry professionals. Dr. Janzen taught for seven years at a small private liberal arts college (Bethel College), four years at a large research-oriented university while completing MS and PhD degrees, and now two years at a medium-sized, highly selective teaching-oriented university. His PhD dissertation which was completed in 2006 examined the effects of TDD on internal software design quality through a series of controlled experiments in industry and academia. This work produced the Test-Driven Learning approach described in this proposal and was awarded third place in ACM’s Student Research Competition Grand Finals and OOPSLA competitions.

The Evaluation Administrator will be Olga Dekhtyar. Mrs. Dekhtyar received an M.A. in Measurement, Statistics and Evaluation in 2000 from the Department of Measurement, Statistics and Evaluation (EDMS) in the College of Education at the University of Maryland, College Park. Mrs. Dekhtyar currently teaches Statistics at Cal Poly and has more than a decade of research and teaching experience. Mrs. Dekhtyar has taught introductory computer programming so she is familiar with the domain.

Randy Scovil will be the evaluating instructor at Cuesta College. Mr. Scovil received an M.S. in Computer Science from Cal Poly in 2005. Mr. Scovil currently teaches computer science courses at Cuesta College, and he has over a decade of teaching experience.

Karl Friesen will be the evaluating instructor at Bethel College. Mr. Friesen received an M.S.
in Computer Science from Purdue University in 1988. Mr. Friesen has extensive industry programming, system administration, and teaching experience. He was a co-founder of SouthWind Internet Access, Inc. and competed on the finals of two ACM National Scholastic Programming Contests. Mr. Friesen is currently an Assistant Professor of Computer Science at Bethel College and he has eight years of undergraduate teaching experience.

Glen Diener will be the evaluating instructor at Tabor College. Mr. Diener received an M.S. in Computer Science from Kansas State University in 1984. Mr. Diener is currently an Associate Professor of Computer Science at Tabor College. Mr. Diener holds numerous professional certifications and has more than twenty years of undergraduate teaching experience.

3.3.1 Institutional Fit

California Polytechnic State University, San Luis Obispo is a primarily undergraduate institution located on the central coast of California, about half way between San Jose and Los Angeles. It is the only 4-year institution in San Luis Obispo county. The closest public institutions are UCSB, 90 miles away, and San Jose State University, 150 miles away.

Cal Poly consistently ranks as one of the top public undergraduate institutions in U.S. News and World Report’s annual rankings. Cal Poly’s College of Engineering 2006 undergraduate enrollment was the eighth largest in the United States. There are approximately 500 computer science majors and 80 software engineering majors at any given time. We also offer a Master’s degree in computer science. The campus is undergoing a shift to place more emphasis on research on campus in an effort to build a more vibrant research environment.

Cal Poly’s motto is “Learn by Doing,” and it has served as a regional and national leader in hands-on, project- and problem-based learning. Because of its close proximity to Silicon Valley and Southern California, and its outstanding reputation, Cal Poly’s computer science department collaborates closely with many leading national and global software companies. Our students are highly recruited and we maintain high visibility with many companies including Adobe, Apple, Google, Intuit, Raytheon, and Sun Microsystems, among others.

Cal Poly is among the most selective colleges and universities in California and the country, so we receive very high aptitude students. In addition to our on-campus curriculum, all students are required to complete practical industry-based projects plus a senior project. Most students complete multiple internships. Many of our graduate students participate in our 4+1 program so they have completed their undergraduate program at Cal Poly.

In addition, Cal Poly’s focus on providing an outstanding applied undergraduate education creates an ideal environment for developing TDL-based pedagogical materials. The computer science and software engineering faculty are very interested in innovative pedagogical approaches, and the institution places strong emphasis on pedagogical innovation.

3.4 Dissemination

The textbook materials will be widely disseminated through a variety of avenues. It is anticipated that the textbook will be published with a major publisher, opening a variety of dissemination avenues immediately. An Executive Editor from one major computer science publisher (Pearson Addison-Wesley) has already expressed interest to the PI in seeing TDL-based CS1 materials. In addition, the PI intends to present at least one technical paper and conduct a TDL tutorial at the annual SIGCSE technical symposium. If possible, appearances will be made at the publisher’s vendor booth at SIGCSE and possibly other conferences such as ITiCSE and CCSC regional conferences. In addition, announcements will be made on a variety of email lists including the SIGCSE
members list and the APCS list for high-school advanced placement computer science courses.

### 3.5 Project Schedule

The following table summarizes the proposed activities. Further explanation is provided below.

<table>
<thead>
<tr>
<th>Year</th>
<th>Activity</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>Author textbook, lab, and project materials</td>
<td>PI and Graduate Student</td>
</tr>
<tr>
<td>Two</td>
<td>Complete textbook in summer, evaluate at four institutions</td>
<td>PI, Graduate Student, Evaluation Administrator, Three External Evaluating Instructors</td>
</tr>
<tr>
<td>Three</td>
<td>Analyze evaluation; improve textbook and supplementary materials based on evaluation feedback; publish textbook and disseminate results</td>
<td>PI and Evaluation Administrator</td>
</tr>
</tbody>
</table>

#### 3.5.1 Year 1

The primary focus of the first year is the creation of the CS1 textbook. The majority of this work will occur solely by the PI during the summers at the beginning of years one and two. During the academic year, the PI will continue to focus on writing the text, while advising one graduate student who will develop accompanying lab, exercise, and project materials. The PI’s department provides substantial flexibility when selecting courses to teach so the PI will request to teach primarily CS1 courses during this year in order to maintain focus on completing the textbook and to pilot test materials as they are completed.

#### 3.5.2 Year 2

The textbook will be completed during the summer and early fall of the second year. During this time, the Evaluation Administrator (EA) will prepare an evaluation plan. The first evaluation will occur at Cuesta College. Cuesta teaches CS1 using the ‘C’ programming language for the first two-thirds of the course, followed by five weeks in Java. The text will be used during this five week period in late fall. The remaining evaluations will occur in the winter/spring at Bethel College, Tabor College, and Cal Poly. Bethel and Tabor teach CS1 in the spring semester. Cal Poly is on the quarter system, so CS1 is split between the first two courses. Currently the first course is taught using the ‘C’ programming language and the second course uses Java. In this second quarter course, faculty use a traditional Java-based CS1 textbook which will be replaced with the new TDL-based textbook in at least two sections. The PI will provide assistance to the EA and the evaluators as needed. The graduate student will provide assistance to the EA in collecting survey, lab, project, and exam artifacts from students at the evaluating institutions. In addition, the PI will submit the textbook for publication.

#### 3.5.3 Year 3

During the summer at the beginning of year three, the EA will complete an analysis of the evaluations with the assistance of the graduate student. The PI will utilize this analysis and the experience of the evaluators in order to improve the TDL-based textbook. The PI will finalize the publication process and will conduct dissemination activities. In particular, the PI will attend the annual SIGCSE technical symposium where he will present a tutorial on TDL and make appearances as arranged with the publisher.
3.6 Future Work

Successful completion of this project will likely lead to future dissemination and expansion activities. A possible NSF CCLI Phase II proposal might aim to broaden participation in TDL-based CS1 courses through regional training sessions, conference activities, and collaborations with authors of CS1 texts in additional languages. Thinking even more broadly, an NSF CPATH proposal might explore improving course instruction and student skills through incorporation of TDD and TDL across an entire computing curriculum. We can imagine TDL and TDD in a range of courses from introductory programming through Operating Systems, Networks, Advanced Data Structures, and Databases to name a few.