Teaching Unit Testing using Test-Driven Development

Workshop on Teaching Software Testing 2005

Patrick J. Schroeder, Darrin Rothe
Milwaukee School of Engineering
Department of EE/CS
Milwaukee, WI 53211
{schroedp | rothede}@msoe.edu

1. Introduction

Of all testing levels, the unit level of testing has undergone the most recent and most dramatic change. With the introduction of new agile (aka, “lightweight”) development methods, such as XP (eXtreme Programming) came the idea of Test-Driven Development (TDD). TDD is a software development technique that melds program design, implementation and testing in a series micro-iterations that focus on simplicity and feedback. Programmer tests are created using a unit testing framework and are 100% automated.

This paper is an experience report on incorporating TDD into an existing unit testing course at the Milwaukee School of Engineering (MSOE). At MSOE, we offer a 3 credit, 10-week, sophomore-level unit testing course as part of its Software Engineering curriculum. In this paper, we discuss the decision that lead up to offering TDD as part of the unit testing course and the results of our first attempt at presenting the material.

In section 2 of this paper we review TDD and unit testing; in section 3 we address adopting TDD in an academic course; in section 4 we discuss the unit testing course in which we introduced TDD; section 5 presents our conclusions and future work

2. TDD and Unit Testing

2.1 What is TDD?

Test-Driven Development (TDD) is one of the core practices of Extreme Programming (XP) [1, p. 54]. XP is one of several *agile* development processes. Agile development processes are designed to be responsive to the inevitable change that occurs in software development projects. They accomplish this through short iterative development cycles, close customer involvement, and lightweight documentation. Agile methods are sometimes viewed as a counter-culture movement (complete with its own manifesto [2]) that opposes to traditional, planned-based development techniques such as the Software Engineering Institute’s CMMI (Capability Maturity Model Integration) [3, p. 3].

TDD is also referred to as Test-First Design, Test-First Programming, and Test-Driven Design [4].

Astels characterizes TTD as such [5, p. 6]:

> *Test Driven Development (TDD) is a style of development where:*
> - *you maintain an exhaustive suite of Programmer Tests,*
> - *no code goes into production unless it has associated tests,*
> - *you write the tests first,*
> - *the tests determine what code you need to write.*

TDD activities are conducted by the developer of the software, so Astels and others refer to them as *Programmer Tests*. The tests are conducted at what is traditionally referred to as the *unit level* of testing [6, p. 136]. In the prominent object-oriented paradigm of software development the concept "unit testing" has come to mean low-level testing of software classes and components. Under TDD, developers test the classes they are developing method by method. Testing often requires the use of other classes in the system, so the boundary between unit testing and integration testing is not well defined.

Under TDD, all test cases are automated with the help of a Unit Testing Framework (UTF). The UTF assists developers in creating, executing and managing test cases (UTFs exist for almost every development environment, an indication of the popularity of this technique.). All test cases must be automated because they must be executed frequently in the TDD process with every small change to the code.

TDD uses a "test first" approach in which test cases are written before code is written. These test cases are written one-at-a-time and followed immediately by the generation of code required to get the test case to pass. Software development becomes a series of very short iterations in which test cases drive the creation of software and ultimately the design of the program. Grenning describes the TDD development cycle as in Figure 1 [7].

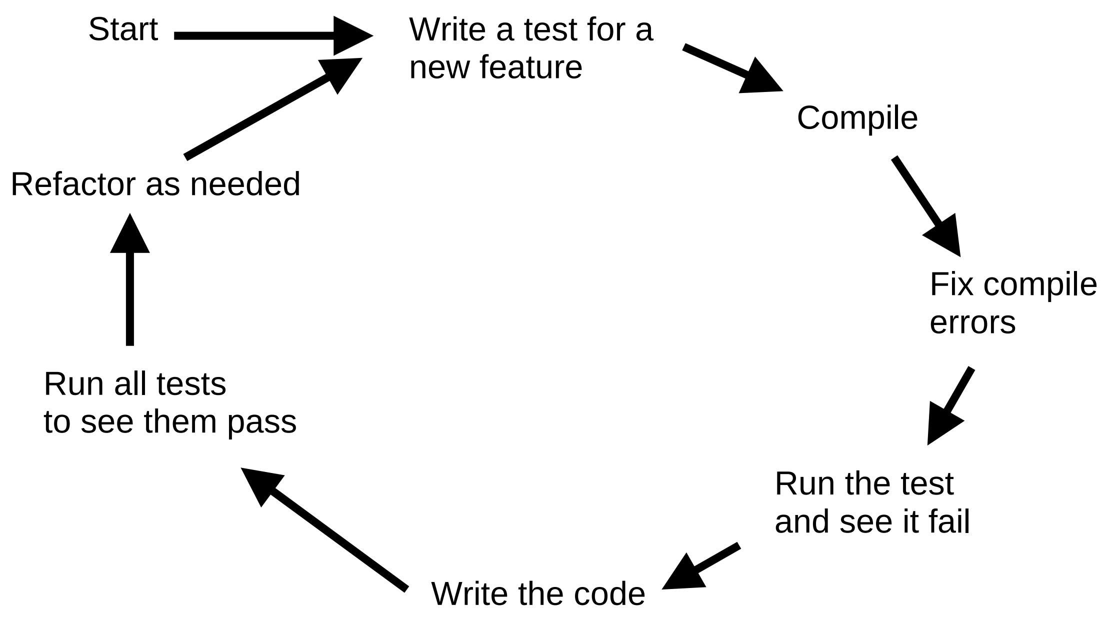
Start
Write a test for a new feature
Compile
Fix compile errors
Run the test and see it fail
Write the code
Run all tests to see them pass
Refactor as needed

Figure 1. TDD Development Cycle

Agile methods **do not** subscribe to the process of creating and documenting software design before beginning the coding phase; plan-based software development methods typically **do** subscribe to the Big Design Up Front (BDUF) approach in which software design is planned and documented before coding begins [3, p. 42]. On agile projects that use TDD, rather than conducting detailed design, developers begin creating test cases and code to support those test cases immediately. This process is executed in very short iterations in which code is generated rapidly and sometimes haphazardly to get the test cases to pass. As test cases and code are written, the developer comes to a better and better understanding of the problem and an effective solution. Restructuring haphazardly written code an effective solution is called *refactoring*. David Astels provides this definition of refactoring [5, p. 15]:

> *Refactoring is the process of making changes to existing, working code without changing its external behavior. In other words, changing* ***how*** *it does it, but not* ***what*** *it does. The goal is to improve the internal structure.*

In refactoring, one must achieve “observational equivalence,” where the behavior of the code remains the same before and after refactoring [8, p. 181]. The test cases determine if observational equivalence has been achieved in refactoring. (i.e., all test cases must pass before and after refactoring). To accomplish this, one must have a relatively comprehensive set of test cases. These test cases are said to act as a “safety net” for the developer during refactoring, giving them the confidence to change working software into a better design knowing that inadvertent changes will be detected.

2.2 Is TDD Testing?

Despite the fact that it is ***test***-driven development and involves the creation and execution of test cases, TDD's status as a testing technique is questionable. Ward Cunningham and others are quick to point out: "Test-first coding is not a testing technique [9]."

Because of its association with *testing* (referred to by Kent Beck as the "ugly step child of software development" [1, p. 115]), TDD has a marketing problem. Fowler laments that: "These days it's terribly unfashionable to talk about Test Driven Development (TDD) having anything to do with testing …." [10]. An excellent example of this is the first sentence of the preface of *Test-Driven Development: A Practical Guide* by David Astels. Astels assures potential buyers of the book that: "This isn't a book about testing." Astels goes on to describe tests as a "nice side-effect" of the TDD process [5, p. xv].

If TDD is not about testing, then what is it about? Design. Kent Beck, in his book Test-Driven Development: By Example, describes TDD: "TDD helps you pay attention to the right issues at the right time so you can make your designs cleaner, you can refine your designs as you learn." [8, p. 200]. In "*Test-Driven Development Isn't Testing*," Jeff Patton comments on how "unfortunate" it is that TDD has *test* in its title. He goes on to explain that TDD is actually a design technique where test cases are a description of the software design [11]. Astels lists TDD and refactoring as the main *design* tools of XP [5, p. xvi].

The idea that TDD is a design technique is referred to as *evolutionary* design [12]. Through the rapid development of test cases and code, the simplest most effective design evolves and the code is gradually refactored to achieve this design. This process stands in stark contract to the upfront, planned design of traditional software development processes.

Much of the early work on agile processes and TDD was done by software developers who focused the discussion on design and specification issues, not testing. Consequently, the information on testing was sparse, incomplete and in some cases simply wrong. From a traditional software development perspective, TDD appears to address only nominal test cases that achieve to achieve only functional demonstration, not a rigorous test of the program. Beck used TDD to testing the famous triangle program. He generated 6 test cases. Bob Binder, author of Testing Object-Oriented Systems: Modelss, Patterns and Tools, generated 65 test cases for the same problem [8, p. 196].

Kent Beck defends his position on testing and TDD [9]:

> *All you testing gurus are probably sharpening your knives. "That's not testing. Testing is equivalence classes and edge cases and statistical*

quality control and…and…a whole bunch of other stuff you don't understand."

Hold on there—I never said that test-first was a testing technique. In fact, if I remember correctly, I explicitly stated that it ***wasn't****.*

2.3 It Isn't Not Testing

The title of this section comes from Brett Pettichord's response to Patton's article, "*Test-Driven Development Isn't Testing*" [11]. While some may claim that TDD is *not* testing, it is often the ONLY programmer testing that is performed. A widely accepted software engineering principle is that programmer testing is essential in the creation of high quality software in a cost effective and timely manner. In many project contexts, we strive to implement programmer testing to some degree. Reducing programmer testing, or doing a poor job of it, may result in rework, delay, and a greater risk of field failure.

To reconcile the positions of designers and testers, Brain Marick has introduced the idea of tests that **support programming** and tests that **critique the product** [13]. The tests that support programming are those test cases used to support specification and design. These are the test cases created in the evolutionary design process. Other test cases are written to evaluate the product in terms of possible deficiencies. These test cases support the more traditional role of testing as a defect detection technique.

Patton describes another approach to reconciling the positions of designers and testers in TDD. He describes the work of Hunt and Thomas in Pragmatic Unit Testing. After the design-oriented test cases have been created and execute, developers must ask "What can go wrong?" and "Could this problem happen else where?" [11]. Patton describes this shift as moving from design to validating the design in a variety of tough conditions.

3. Adopting TDD in a Unit Testing Course

3.1 Should We Adopt?

Since, at least in the eyes of some, TDD isn't a testing technique, perhaps it should not be taught in a unit testing course. Perhaps it is more appropriately taught in a software development or design course. In addition to this, there are other reasons to question the adoption of TDD: it may not work.

Despite glowing reports from advocates of the idea, little data exists to indicate that TDD is effective as a software design or a software testing technique. Data from industry is sparse and contradictory. In one study at IBM, Maximilien and Williams found a 50% reduction in defects in using TDD over traditional development [14]. On the other hand, George and Williams report on a study using professional programmers, TDD and paired programming. In this study, those developers that used TDD created code that passed 18% more test cases than the code produced using traditional testing techniques; however, the TDD programmers required 16% more time on the project.

Another report from industry is particularly damaging. Keefer criticizes XP, which has TDD as a core practice [15]. He points out that the "legendary" project at the Chrysler Corporation, the C3 project, that sited as the archetype of XP success, was canceled in 2000, never reaching it full operational capacity. The second XP project in history, the VCAPS project, was also canceled, and XP has been retired as a software development method at Daimler Chrysler Corporation.

Mixed results have also been reported in academia. Jones reviews the literature for studies of TDD in academic setting [16]. In these studies, TDD is applied in programming courses with the expectation that TDD will lead to higher software quality, higher programmer confidence and productivity, and a better attitude toward testing. Three of the studies contained experiments in which students were separated into experiment and control groups to compare TDD and traditional software development. Two of the studies reported higher software quality and greater programmer confidence when using TDD; one of the studies reported higher programmer productivity. The third study, more methodologically sound than the first two, found no significant difference in software quality or programmer productivity when comparing TDD to traditional development.

3.2 The Positive Contributions of TDD

Cem Kaner refers to TDD as the "big news" in testing in this decade [17]. He sites the appeal of test cases as examples of how the code works for use in specification and maintenance of the code. He stresses the importance of automated unit-level regression testing TDD provides in support of refactoring. Kaner also cites the immediate feedback that TDD test cases provide to developers as a key to making bug finding and fixing a more efficient process.

Another reason to adopt TDD comes from Keefer. Despite his harsh critique of XP, Keefer offers this agile lesson learned [15]:

Any process that is actually applied is far superior than a perfect paper process that is not applied.

This statement gets to the heart of the problem with traditional unit testing practice: it is being done poorly, or not all. Rick Craig jokes that he owns the only copy of the IEEE Standard for Software Unit Testing (1008-1987) [6, p. 136]. While clearly there are some industries where unit testing is enforced (e.g., aerospace, medical device), in our experience, and in the experience of others, we find that unit testing is routinely ignored.

Beck and other practitioners had the insight to work *with* human nature instead of *against* it [1, p. 116]. In developing TDD, they took a task that was considered drudgery by developers and turned it into "fun." Many developers report that they *enjoy* TDD. Beck and Gamma, the creators of the JUnit testing framework, refer to this phenomenon, where programmers love writing tests, as being "test infected" [18]. While this development style is not for everyone, there continues to be substantial energy behind the idea. Several books on the topic exist, with more in process. Conferences and workshops on TDD and agile testing are common. A Google search on the exact phrase "Test Driven Development" generates 181,000 results. As of 01/29/2005, the TDD Yahoo! email group (*testdrivendevelopment@yahoogroups.com*) has 2425 members and has logged 8183 messages; there is no email group for unit testing.

Another breakthrough attributed to TDD and agile methods has to do with the automation of test cases. The testing community has for many years issued a warning: *Don't Automate Everything* [19]. Luckily, Beck and others ignored this advice. TDD insists that ALL tests be automated. They have to be automated because they must be executed frequently (every few minutes in some cases) as the code is enhanced and refactored. While automating *every* programmer test remains problematic, there have been significant advances in advances in number and type of tests that can be automated as a result of TDD. Ultimately, having developers focus on testing while creating the code will lead to systems with higher testability [5, p. 6]. Higher testability as been associated gains in software quality and productivity.

3.2 Decision to Adopt

Ultimately, we decided to adopt TDD as part of the unit testing course. Some of our reasons for this decision are listed here:

1) TDD has become decoupled form XP and has taken on a life of its own. There is significant momentum behind the technique and it will continue to grow and evolve.
2) Unit testing frameworks (UTFs) are being built into commercial products (e.g., Microsoft's Visual Studio) and open source development environments (e.g., Eclipse). There is very little doubt that a UTF will be a tool that today's student will encounter when entering the work place. The tool is likely to be adapted by developers in creating and executing tests regardless of the development process they are using.
3) Documented proof of the effectiveness of most software engineering techniques does not exist. History has taught us that techniques are often adopted and refined over time to meet initial unsupported claims. TDD appears to be a case in point.
4) TDD *seems* like it would be appealing to students. It is lightweight, simple in concept, possibly *fun*, and it has great tool support.
5) If TDD is added to earlier programming and design courses it is unlikely that the testing concepts will be covered in sufficient detail. Earlier

adoption of TDD in the curriculum is welcomed because it would allow the unit testing course to focus on more advanced testing concepts such as inheritance, polymorphism, and testing GUI applications.

4. SE283 Introduction to Software Verification

4.1 Course Description

At MSOE, in the sophomore year, students are required to take the *Introduction to Software Verification* course. The course description can be found in Appendix A. Appendix A also lists detailed course objectives. These objectives were written as a study guide for students and reflect the course as it was taught, rather than as it was planned. Course materials, including lab assignments can be found at: http://people.mose.edu/~schroedp/se283.

Despite the title of the course, the course is viewed by the faculty as a unit testing course. Verification activities such as reviews and formal proofs-of-correctness are covered elsewhere in the curriculum. This course covers code inspections and walkthroughs in lecture, but we currently have no lab exercises for these topics.

In designing the course, our main concern was to ensure that the students understood that testing under TDD can and should be rigorous within the project context. TDD should go beyond the functional demonstration that supports evolutionary design. Boehm points out that testing to stated (nominal-case) requirements is largely inadequate in many situations. His data on software rework distributions indicates that the 20% of the problems causing 80% of the rework are off-nominal requirements [3]. To account for this, the course was design to expose students to traditional unit testing techniques first, and then to introduce TDD. Ideally, students exposed to more rigorous testing techniques early will consider applying them later when executing TDD.

In this section we discuss two lab multi-week lab sessions where the students developed and tested their own programs. The first lab session was based on traditional testing techniques; the second lab session introduced TDD.

4.2 Test-first Structured Unit Testing Lab

Description

In this lab, students were given an assignment in which they were to use “test-first” structured unit testing to develop a small program as individuals. “Test first” in this instance refers to writing a manual set of test cases before the program is implemented as advocated by Hetzel [20], and more recently by Craig and

Jaskiel [6]. “Structured” unit testing refers to Myers process of black-box testing with coverage analysis followed by white-box testing of unexecuted code [21].

In this assignment, students followed this procedure:

1) Create a manual set of black-box test cases required for the program from the specification (i.e., test first).
2) Design and code a solution to the problem; update the set of test cases as necessary.
3) Execute the set of black-box test cases on the solution code; create a test log for the session; report and fix errors; update the set of test cases as necessary.
4) Measure statement coverage of the set of black-box test cases; analyze coverage; add white-box test cases to the set of test cases to achieve 100% statement coverage; update the set of test cases as necessary.
5) Execute the set of test cases on the programs of two other students. Create a test log for each program; report errors.

Objectives and Outcomes

1) An objective of the lab is to have students “grow” a set of test cases. This growth occurs because their understanding of the problem grows as they are immersed in the detail of writing code to solve the problem. The test-first approach is a great way to start a set of test cases; immersion into the problem leads to a deeper understanding and, ideally, more and better test cases. All students reported growth in the initial set of black-box test cases during the project (very few additional test cases appeared to be completely contrived).
2) Another objective of the lab is to have the students understand how code coverage measurement and analysis can alert them to additional testing ideas. This objective was not met satisfactorily. The assigned program (the CQuad class) is too algorithmic in nature; many students achieved 100% statement coverage with their black-box test cases.
3) In retrospect, an objective of the lab should be to recognize the role of exploration in testing. The students implemented the user interface in a variety of different ways. All interfaces were console applications, and all provided the required functionality; however, almost all programs were ever so slightly different. When students were assigned to test other student’s programs they first had to explore the system to figure out how it worked. This exploration lead to test cases that were blocked due to the difference in the interface and to ideas for new test cases that were added to the test set during testing. This generated a discussion of brief discussion of exploratory testing, which was not an original course objective.
4) The final, most important, and undocumented objective of this lab is to have the students design, code, and test a program to a specification, and then have bugs reported against it. Because the students get their own

test cases to pass, they believe the program is correct (a common notion among inexperienced developers). Simply giving the program to another student to test results in problems.

In the final part of this lab, every student had their program tested by two other students in the class. ALL students had at least one bug reported against their implementation (one student had exactly one bug reported, most students had several reported, a very few students had a large number of bugs reported). The exercise lead to a discussion of the ambiguity of natural language documents, the importance of deeply questioning all assumptions when coding, and the mind set of a tester vs. the mindset of a developer. All students received the test logs of the other two students as part of the evaluation of their project. Many of the bugs reported were invalid due to misinterpretation of the specification and/or bad test data. The bug reports did not directly affect the student's grade for the project, largely so that we did not have to resolve the large number of problems reported to determine the lab grade.

4.3 TDD Lab

Description

In this lab, the students used TDD to implement a class taken from an industrial project that one of the authors was involved with. This lab was the final lab of the term and was given after lectures covering basic testing concepts, domain analysis, error catalogs, and the previous lab on structured unit testing.

The problem was to implement the *Sequence* class. The class consists of simple mutators and accessors, but it also implements a more complex "bracketing" algorithm. The bracketing algorithm requires an instance of the Sequence class to maintain state between messages. This complexity allowed us to introduce the UML state chart as a software design technique and as a model-based testing technique.

The students were instructed to implement the class using paired programming and TDD. Students were given and overview of XP and TDD. Students were given a warm-up lab to become accustomed to the unit testing framework, CppUnit. MSOE does not have paired programming workstations. No effort was made to enforce compliance with paired programming and TDD. Some students that live on campus did work in pairs exclusively; other students living off campus found it impossible to schedule enough project time as a pair to complete the assignment. Similarly, there was a mix of students that followed the TDD process.

The lab assignment culminated in a testing competition. The students used the test cases they developed during the course of the lab and executed them on 35 mutant classes that we created. Only 30 of the classes had seeded faults. Each seeded fault was created using some realistic theory of error for the problem at hand (e.g., off-by-one, state corruption), rather than by arbitrarily changes operator and variables as is done in mutation testing [22]. This process produced a reasonable set of mutants; each mutant created was found by at least one student program. As part of the competition, the students had to document at least two of the bug detected using the open-source Mantis bug tracker (www.mantisbt.org).

Objectives and Outcomes

1) An objective of the lab is to have students develop a significant piece of code using a UTF. CppUnit was difficult to setup and use; however, most students appreciated the automated testing after the extensive manual testing in the previous lab. One student remarked that he “got it” when he saw how easy it was to re-run the test cases.
2) Another objective of the lab was to have students try TDD. We don’t believe any of the student teams were able to maintain the TDD process through the entire project. Some clearly tried the process with the simple accessors and mutators. The more complex bracketing algorithm caused problems. We exacerbated the problem by showing them how a UML state chart is used in the design and testing of this type of problem. We expected more students to enjoy the TDD method and were surprised by a small but vocal group of students that complained about using TDD.
3) The final objective of the lab was to conduct a “testing competition,” similar to the much more popular programming competitions. A testing competition is possible at this level of testing because the class interfaces and behavior are well defined. The idea behind the competition was to add enliven what is sometimes to be considered a boring topic. Due to technical difficulties with the setup of tools on the student’s laptop computers, we were not convinced that running the competition was going to be technically possible. As a result, we did not promote the competition as much as we could have. In the end, the competition did work perfectly. The students were able to download the mutant files, insert them into the project on their IDE and execute their test cases against them.

   Table 1 lists the results of the competition. The winning team (team 5) detected 27 out of 30 possible faults. The team found 3 faults that no other team found. Part of their success was in generating test ideas from the data files required for processing. The team did not use TDD. The did write the test for the most complex part of the system first, in order to get a better understanding of what the class was required to do.

Table 1. Testing Competition Results

Team	Number of Test Cases	Number of Test Data Files	Number of Faults Detected (30 possible)
1	22	1	16
2	9	12	22
3	31	6	23
4	33	10	23
5	42	20	27
6	14	1	21
7	11	7	23
8	13	2	18
9	13	1	14
10	14	2	19

5. Conclusion and Future Work

Adopting TDD in a unit testing course allows you to put the *test* back in TDD. In the production of high-quality, cost-effective software, we need developers that are capable of testing their own code. The depth to which they test will be project context specific. In teaching unit testing we want to expose students to rigorous, as well as, lightweight testing methods. In the end, it will be their experience and judgment that determines the depth of testing for a given situation.

Our unit testing course is designed to help developers think and act like testers when required. The course taught the traditional structured unit testing process and the more modern TDD process. The subtleties of evolutionary design are difficult for students to grasp at this level because of a lack of design experience; however, the efficiency and productivity of automated programmer testing using a UTF is fully realized, especially when contrasted with manual unit testing.

We will continue to follow the extension and refinement of TDD, and will continue to evaluate *if* and *how* to use TDD in our curriculum. We are considering introducing a UTF in our programming course sequence. Even if this is accomplished, the unit testing course will continue to use a UTF. This extension may allow the unit testing course to address more complex testing problems such as mock objects, GUIs, and databases.

References

[1] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison-Wesley, 2000.

[2] Agile_Alliance, "Manifesto for Agile Software Development," [01/10/2005], http://www.agilealliance.org.
[3] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesly, 2004.
[4] B. George and L. Williams, "An Initial Investigation of Test-Driven Development in Industry," presented at ACM Symposium on Applied Computing (SAC), Melbourne, FL, 2004.
[5] D. Astels, *Test-Driven Development: A Practical Guide*. Upper Saddle River, NJ.: Pearson Education, Inc., 2003.
[6] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Boston: Artech House, 2002.
[7] J. W. Grenning, "Test-Driven Development and Acceptance Testing," [10/05/2003], www.uniforum.chi.il.us/slides/testdrivendevel/TDD.pdf.
[8] K. Beck, *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.
[9] K. Beck, "Aim, Fire," [12/14/2004], http://www.computer.org/software/homepage/2001/05Design/.
[10] M. Fowler, "Specification by Example," [01/10/2005], http://martinfowler.com/bliki/SpecificationByExample.html.
[11] J. Patton, "Test-Driven Development Isn't Testing," [01/25/2005], http://www.stickyminds.com/sitewide.asp?Function=WEEKLYCOLUMN&ObjectId=8497&ObjectType=ARTCOL&btntopic=artcol&tt=WEEKLYCOL_8497_title&tth=H.
[12] M. Fowler, "Is Design Dead?" in *Extreme Programming Examined*, Succi, Giancarlo, and M. Marchesi, Eds. Boston, MA: Addison-Wesley, 2000.
[13] B. Marick, "Exploration Throught Example," [01/10/2005], http://www.testing.com/cgi-bin/blog/2003/08/21.
[14] E. M. Maximilien and L. Williams, "Assessing Test-Driven Development at IBM," in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 564-569.
[15] G. Keefer, "Extreme Programming Considered Harmful for Reliable Software Develment 2.0," [01/12/2005], http://www.avoca-vsm.com/Dateien-Download/ExtremeProgramming.pdf.
[16] C. G. Jones, "Test-driven Development Goes to School," *Journal of Computing Sciences in Colleges*, vol. 20, pp. 220 - 231, 2004.
[17] C. Kaner, "The Ongoing Revolution in Software Testing," presented at Software Testing and Performance Conference, Baltimore, MD, December 7-9, 2004.
[18] K. Beck and E. Gamma, "Test Infected: Programmers Love Writing Tests," in *Java Report*, vol. 3, 1998.
[19] J. Gainer, "Automated Testing: Myth vs. Reality," [01/14/2005] http://www.jeffgainer.com/testmyth.html.
[20] B. Hetzel, *The Complete Guide to Software Testing*, 2nd ed. Wellesley, MA: QED Information Sciences, Inc., 1988.
[21] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.
[22] R. A. DeMillo, "Mutation Analysis as a Tool for Software Quality Assurance," presented at Fourth International Computer Software & Applications Conference, New York, NY, USA, 1980.

Appendix A

Course Description: SE283 Introduction to Software Verification

number	SE-2831 [Required course in Software Engineering V2.1]
Course title	Introduction to Software Verification
Course description	This course introduces students to software testing and its many roles in the software development life cycle. The course focuses on unit testing, or the testing of software components, classes, or modules in isolation. Both specification-based and program-based testing techniques are taught. These techniques are then used in traditional unit testing strategies and in agile test-driven development strategies. Topics covered include: testing process terminology and limitations; unit testing strategies; test design techniques; unit testing frameworks; test-driven development; unit testing planning and documentation; and software defect reporting. Laboratory assignments provide an opportunity to apply software verification techniques and tools.
Prerequisites	CS-1030
Corequisites	CS-2851
Required materials	• *Systematic Software Testing* by R. Craig and S. Jaskiel, Artech House Publishers, ISBN 1-58053-508-9
Course structure	2-2-3 (class hours/week, laboratory hours/week, credits)
Laboratory topics	• Unit testing techniques (4 sessions) • Unit testing framework (1 session) • Software Defect Reporting (1 sessions) • TDD project and testing competition (3 sessions)

Detail Objectives by Lesson

At the completion of these lessons the student should be able to:

Lesson 1. Introduction to Software Engineering

1) Define software engineering (IEEE definition).
2) Understand the need and motivation for software engineering.

3) Describe the relationship between computer science and software engineering.
4) State the goals of software engineering.
5) Identify and describe several different roles that a software engineer may play on a software project.
6) Differentiate between software process and software product and give examples of each.
7) Describe the waterfall model and give a high-level description for each phase of the model.
8) State the benefits and risks of using an evolutionary (i.e., incremental or iterative) life cycle model.

Lesson 2. Introduction to Software Testing

1) Define the verification and validation processes and give examples of the tasks executed in each process.
2) Define software testing.
3) State why complete testing is not possible.
4) Understand the role of context and risk is software testing.
5) Describe the contents of a test case. Define positive and negative test cases.
6) Describe different levels of software testing.
7) Describe the relationship of software testing and software quality assurance.

Lesson 3. Risk Analysis

1) Explain the relationship of risk and economics on a testing project.
2) List several factors that contribute to risk on a software project.
3) Explain how the quantify risk (using the technique described in the course textbook).

Lesson 4. The Unit Testing Process

1) Define unit testing.
2) List several of the obstacles to implementing a unit testing process.
3) Describe the advantages and disadvantages of "test first" unit testing.
4) Describe the advantages and disadvantages of "test last" unit testing.
5) Define *structured* unit testing.
6) Specify the type of software errors that will not be detected when using white-box testing only.

7) Specify the type of software errors that will not be detected when using black-box testing only.

Lesson 5. White-box Adequacy Criterion

1) Explain what an adequacy criterion is and how it is used in unit testing.
2) Define statement, branch, and path coverage and state their relative strengths as adequacy criterion.
3) Draw a control flow graph for a program.
4) Use a control flow graph of a program in creating a set of test data that satisfies a specified adequacy criterion.
5) Describe a loop testing strategy.
6) List the problems associated with path testing.
7) Describe the difference between branch coverage and multiple conditional coverage.
8) Describe the type of tool necessary to conduct testing to meet a white-box adequacy criterion.

Lesson 6. Black-box Testing: Equivalence Partitioning & BVA

1) Define black-box testing.
2) Describe how equivalence classes are used in testing.
3) Explain the theory of error behind boundary value analysis.
4) For a testing problem, layout a boundary table.
5) Generate test cases from a boundary table.

Lesson 7. Extreme Programming (XP)

1) Define XP.
2) Explain how code reviews are conducted under XP.
3) Explain how testing is conducted under XP.
4) Explain how program design is conducted under XP.

Lesson 8. Test-Driven Development

1) Explain why all test cases must be automated under XP.
2) Describe a unit testing framework.

3) List the advantages of a system whose modules are highly cohesive and loosely coupled.
4) List the steps in the TDD development cycle.
5) Explain the advantage to TDD when compared to traditional unit testing.
6) Explain the disadvantage of TDD when compared to traditional unit testing.
7) Use an assertion in creating an automated test case.
8) Describe refactoring.

Lesson 9. Combinatorial Testing

1) Define combinatorial testing.
2) Define pairwise testing.
3) Create a pairwise test set for a simple problem.
4) State the strengths and weaknesses of pairwise testing.

Lesson 10. Reviews, Inspections, and Walkthroughs

1) List defect detection techniques used in verification other than unit testing. Explain where they are used in the software development lifecycle.
2) Describe the “zone of chaos.”
3) List the characteristics of good requirements.
4) Describe the difference between code inspections and code walkthroughs.
5) State the advantages and disadvantages of code inspections.

Lesson 11. Model-Based Testing (MBT)

1) Explain how models are used in testing.
2) List several different models that are used in testing.
3) Draw a UML state chart for a simple problem.
4) Explain how a UML state chart is used in testing.

Lesson 12. Master Test Plan

1) Describe the goal of test planning.
2) List several important sections of a test plan.
3) Describe IEEE 829 and explain why it is controversial.
4) Explain how different levels of test planning work together.

Lesson 13. Inventory Tracking Matrix

1) Create test objectives and inventory items for a testing problem.
2) Create and inventory tracking matrix.

Lesson 14. Bug Reporting

1) Explain the motivation for writing good bug reports.
2) Describe several on the characteristics of a good bug report (Can Pig Ride?).
3) Identify problems with bug reports and suggest improvements.
4) Describe the features of a bug tracking tool.
5) List the steps in the bug life cycle.

Lesson 15. Software Configuration Management (SCM)

1) Explain the motivation for using SCM.
2) Describe the concept of a baseline.
3) List the steps in the change control process.
4) Describe the features of a SCM tool such as RCS or CVS.