Software Architecture Improvement through Test-Driven Development

David S. Janzen
University of Kansas
Electrical Engineering and Computer Science
Lawrence, Kansas USA
djanzen@ku.edu

ABSTRACT

This research involves empirical software engineering studies applied in academic and professional settings to assess the influence of test-driven development on software quality. Particular focus is given to internal software design quality. Pedagogical implications are also examined. Initial results and the study protocol and plans will be presented.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Evolutionary prototyping, object-oriented design methods*

General Terms

Design, Verification

Keywords

Test-driven development, agile methods

1. BACKGROUND AND PURPOSE

Despite a half century of advances, the software construction industry still shows signs of immaturity [1]. Professional software development organizations continue to struggle to produce reliable software in a predictable and repeatable manner. While a variety of development practices are advocated that might improve the situation, developers are often reluctant to adopt new, potentially better practices based on anecdotal evidence alone. Empirical evidence of a practice's efficacy are rarely available or conclusive and adopting new practices is time-consuming, expensive, and risky.

Test-driven development (TDD) is a new approach that offers the potential to significantly improve the state of software construction. TDD is a disciplined software development practice that focuses on software design by first writing automated unit-tests followed by production code in short, frequent iterations [2]. TDD focuses the developer's attention on a software's interface and behavior while growing the software architecture organically.

TDD has gained recent attention with the popularity of the Extreme Programming (XP) [2] agile software development methodology. Although TDD has been applied sporadically in various forms for several decades [7], possible definitions have only recently been proposed. While some XP practices like pair programming have enjoyed significant research [10], advocates of TDD rely primarily on anecdotal evidence of TDD's benefits. A few studies have looked at TDD as a testing practice to remove defects [5, 11, 3, 9, 8, 4]. However, there is no research on the broader efficacy of TDD, nor on its effects on internal design quality outside a pilot study for this work [6]. Further, no empirical research has examined the appropriate place or teaching techniques for introducing TDD in the undergraduate curriculum.

2. RESEARCH GOALS

This research will be the first comprehensive evaluation of how TDD effects overall software architecture quality beyond just defect density. Empirical software engineering techniques will be applied to evaluate the ability of TDD to produce better software designs than more traditional test-last approaches produce in terms of reusability, extensibility, and maintainability. Further, this research will examine defect density and whether TDD takes more effort than traditional test-last approaches.

In addition, this research will make important pedagogical contributions. The research will contribute a new approach to teaching that incorporates teaching with automated tests called "test-driven learning." The research will demonstrate whether undergraduate computer science students can learn to apply TDD, and it will examine at what point in the curriculum TDD is best introduced.

If TDD proves to improve software quality at minimal cost, and if this research shows that students can learn and benefit from TDD from early on, then this research can have a significant impact on the state of software construction. Software development organizations will recognize the benefits of TDD as both a design and testing approach, and they will be convinced to adopt TDD in appropriate situations. New textbooks and teaching materials can be written applying the test-driven learning approach. As students learn to take a more disciplined approach to software development with TDD, they will carry this into professional software organizations and improve the overall state of software construction.

3. APPROACH AND EVALUATION

This research will consist of designing and administering a series of longitudinal empirical studies with university

students and professional programmers. Controlled experiments will be conducted in a set of undergraduate courses from introductory programming through upper-level software engineering courses. A similar experiment will be conducted in one graduate course which consists largely of professional programmers. Finally a case study or controlled experiment will be conducted with more experienced programmers in a professional environment.

Undergraduate programmers will be taught to write automated unit-tests integrated with course topics using a new approach called test-driven learning (TDL). The TDL approach involves modeling regular unit-testing in lecture and lab instruction through examples with automated tests. Most commonly, output statements are replaced with automated unit tests to demonstrate both the interface and the behavior of the code under investigation. Graduate and professional programmers will be given more concentrated instruction on TDD and the use of automated unit-test frameworks.

Programmers will then be required to complete two programming assignments. The study group will be asked to use test-driven development techniques while the control group will be asked to use a more traditional test-last approach. The assignments will be as large as possible within the constraints of the course or project, and the second assignment will build on or reuse significant parts of the first.

At the beginning of the second project in lower-level academic settings, all programmers will be provided a solution to the first project that includes a full set of automated unit tests. In the second project, students may choose to build on either their own solution, or the solution provided.

Code samples will be gathered at multiple points in the development process to determine the degree of testing, the degree of reuse, and the overall quality of code. Unit-test quantification and coverage metrics will be calculated for each programmer or project team. Software design quality will be measured by calculating a set of static metrics. Code samples will be examined with available software metrics tools. Traditional and object-oriented metrics will be examined including code size, cyclomatic complexity, and coupling measures such as fan-in, fan-out, and information flow.

Reuse will be measured statically. Although many reuse metrics focus on reuse through inheritance, methods and classes reused with and without modification may be more useful measures particularly in the introductory courses. Such measures will be calculated between subsequent projects and when possible from one version to the next in the same project. This will help determine the degree to which the software evolves and the software's stability.

Final project submissions will be evaluated with a set of dynamic and static software metrics. Defect density will be measured through dynamic black-box acceptance tests. During the coding process, a random sample of programmers will be observed and interviewed regarding their use of test-driven development. Programmers will also be required to track the amount of time they spend on projects. Time spent extending the first assignment in the second assignment will be an indicator of design quality in terms of reuse and extensibility.

At the beginning and end of each study, programmers will be asked to complete a survey indicating their attitudes toward testing and test-driven development. Student exam and course grades will be compared to determine if any correlation exists between test-driven development and academic performance.

A sample of programmers from both the control and study groups will again be examined in subsequent courses or projects to determine voluntary use of test-driven development, long-term attitude changes, and effects on software design quality.

Results from all experiments will be compared and general conclusions may be drawn regarding the fit of TDD in the curriculum. Evidence of student ability to comprehend and apply TDD at certain levels, along with significant positive effects of TDD on software designs and student performance may provide strong motivation for introducing TDD in certain courses.

Data collected from the experiments will be reported and analyzed statistically. Tests such as the two-sample t-test will be employed to determine if differences between the control and experimental groups are statistically significant. Initial results from a summer 2005 study in an undergraduate software engineering course will be presented.

4. REFERENCES

[1] 2004 third quarter research report. Technical report, Standish Group International, Inc., 2004.

[2] K. Beck. Aim, fire. *Software*, 18(5):87–89, Sept.-Oct. 2001.

[3] S. Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA'03*, August 2003.

[4] H. Erdogmus. On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(1):1–12, January 2005.

[5] B. George and L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337–342, 2004.

[6] R. Kaufmann and D. Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299. ACM Press, 2003.

[7] C. Larman and V. R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, June 2003.

[8] M. Müller and O. Hagner. Experiment about test-first programming. *IEEE Proceedings-Software*, 149(5):131–136, 2002.

[9] M. Pančur, M. Ciglarič, M. Trampuš, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *Proceedings of EUROCON 2003. Computer as a Tool. The IEEE Region 8*, volume 2, pages 83–86, 2003.

[10] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Longman, Inc., 2002.

[11] L. Williams, E. Maximilien, and M. Vouk. Test-driven development as a defect-reduction practice. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 34–45, Nov. 2003.