A LEGO-based Approach to Introducing Test-Driven Development

Stan Kurkovsky
Central Connecticut State University
kurkovsky@ccsu.edu

ABSTRACT
Test-driven development (TDD) is an important software engineering technique that requires writing tests before writing the code to be tested. We describe an approach to introduce the main ideas and practices of TDD using an engaging hands-on activity where students write test cases and construct testable tangible objects with LEGO bricks.

Keywords
Test-driven development, refactoring, software engineering, active learning, LEGO.

1. INTRODUCTION
TDD is a software development strategy that requires writing test cases for every functional unit of the program before it is created. Typical units are the smallest testable components; usually these are individual functions, procedures, or class methods. Automated unit testing frameworks, such as JUnit for Java, help streamline the testing process and minimize the required amount of effort. The main emphasis of TDD is that the developer needs to write the tests before the code that will be tested. Consequently, the *test-driven* aspect of TDD has a significant impact on how software engineers approach their analysis, design, and programming decisions [8]. TDD aligns well with agile methods, which assume that the product requirements are incomplete or will be changing soon.

Refactoring is an integral part of TDD, which promotes testing as an integral part of the analysis and design steps. Refactoring aims to modify the existing internal structure of the code without modifying its external behavior. As the new tests are written and new features are implemented by adding new code, the resulting program structure may become too complex or inflexible. Refactoring aims to reduce that complexity without affecting the outcomes of the corresponding unit tests.

Recent studies indicate that TDD can improve code quality and productivity [2]. Writing tests before writing code forces developers to make better design decisions because they must be able to "distinguish between the functionality to implement and the base conditions under which the implementation has to work" [11]. From the educational perspective, using TDD in the curriculum may help improve students' analytical skills, increase their confidence levels [5], and offer them a practical experience that can become very useful in their professional careers.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
ITiCSE '16, July 09-13, 2016, Arequipa, Peru
ACM 978-1-4503-4231-5/16/07.
http://dx.doi.org/10.1145/2899415.2925500

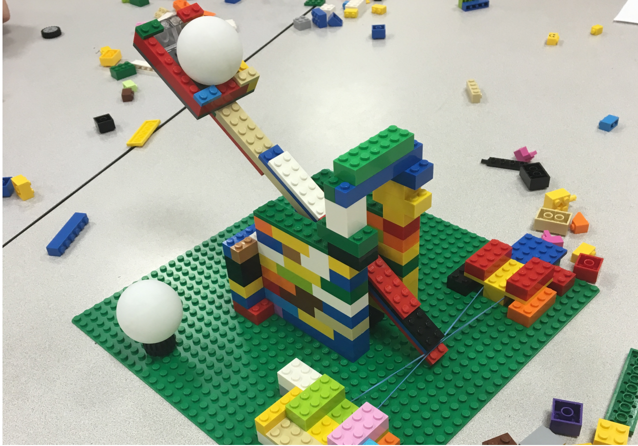
For many students, as well as for seasoned developers, it may be a radical departure from their existing practices to have a test case written before the code that is to be tested. It is the opposite of the traditional test-last software development approach where code is written to design specifications, and most of test cases are written after the bulk of the code has been produced. Consequently, to minimize the amount of confusion that students may have with conflicting testing and development practices, many educators choose to introduce TDD very early in the curriculum and continue using it for an extended period of time, at least throughout a single semester [10]. In a survey of 18 studies, Desai et al [5] describe the applications of TDD at various points in academic programs (from freshmen, to upper level, to graduate courses) and suggests that in order to be most effective, TDD should be introduced sooner in the curriculum. Although many studies report overall positive experiences, some concerns remain. In particular, TDD places a high cognitive and technical load, especially on novice programmers, which may serve as a significant deterrent to making it a persisting practice [8,9].

TDD is very simple to describe, but is rather difficult to utilize as a sustained practice, which requires the programmer to be well-disciplined and methodical in applying the technique [6]. Furthermore, TDD cannot be adequately covered in a typical standalone lecture [12]. For novice programmers, additional hurdles include the difficulty of learning a new framework, which can be overcome using student-friendly tools like WebCAT [3] and Marmoset [12]. Although using these tools forces students to use TDD, they may still not be convinced from the outset about the value of the test-first approach. Some reports indicate that students may not view writing tests as a relevant or helpful task [7]. In fact, students may prefer debugging or just running the program to writing test cases. Some researchers link poor students attitudes towards testing as a reason for not adhering to TDD practices [3], which may have far reaching effects. Furthermore, a systematic literature review of industrial practices found that adoption of TDD is hindered by the insufficient adherence to TDD protocol and inadequate testing skills [4].

2. Learning TDD with LEGO
In courses where instructors want to have coverage of TDD, but may not want to use a dedicated software tool, it is important to have an active learning technique to demonstrate and engage students in TDD. Since 2014, the author has been using a hands-on activity where students are introduced to the main ideas and practices of TDD by building LEGO models. It was inspired by the presentations by the work of Beecham and Bowler [1] who conducted numerous training workshops for professional software developers. The technique presented here retains the playful elements of LEGO construction, but offers a number of substantial differences: it is more *student-centric* with a focus on a tangible product; it demonstrates that *refactoring* is an integral activity of TDD; it allows students to build *functional models* that provide a rich foundation for experimenting with the models and

writing test cases; it helps students learn from the *collective effort* and refine their unit tests based on the experience of the entire class.

Below is a detailed description of this activity focusing on building a functional model of a catapult using the TDD approach. It has been piloted in several offerings of an upper level software engineering course with several teams consisting of 4-5 students.



Materials and supplies. Each team needs one or two LEGO sets with some long plates to build the arm of the catapult and axles to allow the arm to swing. Useful LEGO sets include many of the larger LEGO Classic boxes or LEGO Serious Play Starter Kit. Ping pong balls are useful as projectiles. Rubber bands can be used to spring-load the catapult's arm.

Introduction. Typically, a the basics of TDD are covered during the previous class. Each student receives a one-page worksheet that includes a simple diagram summarizing TDD as a refresher. Students are told that they will be building a catapult, but no specific details are given except that it will have a base platform, a frame, and a moving arm.

Building. Student teams are asked to build one element/feature of the catapult at a time. Teams are asked to create the catapult elements following this sequence of steps: build the main platform on a large LEGO baseplate; build a frame on the platform; build and attach an arm to the frame; build and attach a payload bucket to the arm; use a rubber band to spring load the arm; launch a ping pong ball at least 10 feet forward. Before building the element of each step, the teams are asked to formulate and write down the test cases that they would use to verify the correctness of their construction. Typical test cases may refer to minimal/maximal or relative sizes (e.g., the frame must be three times taller than the base) of the catapult elements or about their functional properties (e.g., the arm must be able to swing). Once the test cases are written for the given step, teams build the corresponding element of the catapult and apply the tests. If one or more tests fail, teams would need to adjust their models and run the tests again. Following each step, teams are asked to refactor their entire models by reviewing and, if necessary, modifying what has been constructed so far to make sure that the element added at the last step is well-integrated well and that *all* of the test cases written thus far are satisfied.

Review. Once the teams finish all construction steps, they are asked to revisit the test cases they wrote. The teams are asked whether they would write some tests differently for each step given the experience with TDD they have gained by completing this exercise. If the team decides to make changes, students are asked to write down the revised tests for each step in a separate column of the worksheet. By verifying the meaningfulness of the test cases, the instructor can see how well students understand the principles of TDD. Comparing the original tests with the revised ones would enable the instructor to determine whether this exercise helped students understand TDD better. Finally, if time permits, it's always fun to have teams compete against each other to launch a projectile the farthest distance or at a specific target.

3. REFERENCES

[1] Beecham, B. and Bowler, M. 2014. *TDD and Refactoring with LEGO*. http://www.infoq.com/presentations/tdd-lego.

[2] Bissi, W., Neto, A., Emer, M. 2016. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology* 74, (2016), 45-54.

[3] Buffardi, K. and Edwards, S. 2012. Exploring influences on student adherence to test-driven development. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (ITiCSE '12), 105-110.

[4] Causevic, A., Sundmark, D. and Punnekkat, S. 2011. Factors Limiting industrial adoption of test driven development: a systematic review. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 337-346.

[5] Desai, C., Janzen, D. and Savage, K. 2008. A survey of evidence for test-driven development in academia. *SIGCSE Bull.* 40, 2 (June 2008), 97-101.

[6] Hammond, H. and Umphress, D. 2012. Test driven development: the state of the practice. In *Proceedings of the 50th Annual Southeast Regional Conference* (ACM-SE '12), 158-163.

[7] Isomöttönen, V. and Lappalainen, V. 2012. CSI with games and an emphasis on TDD and unit testing: piling a trend upon a trend. *ACM Inroads* 3, 3 (Sep. 2012), 62-68.

[8] Janzen, D. and Saiedian, H. 2005. Test-driven development: concepts, taxonomy, and future direction. *Computer* 38, 9 (Sep. 2005), 43-50.

[9] Kollanus, S. and Isomöttönen, V. 2008. Test-driven development in education: experiences with critical viewpoints. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (ITiCSE '08), 124-127.

[10] Marrero, W. and Settle, A. 2005. Testing first: emphasizing testing in early programming courses. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '05), 4-8.

[11] Müller, M.M. and Tichy, W.F. 2001. Case study: extreme programming in a university environment. *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, 537-544.

[12] Spacco. J. and Pugh, W. 2006. Helping students appreciate test-driven development (TDD). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (OOPSLA '06), 907-913.