Challenges in Teaching Test Driven Development

Rick Mugridge

University of Auckland, New Zealand
r.mugridge@auckland.ac.nz

Abstract. We identify two main challenges in the teaching of Test Driven Development (TDD) over the last two years. The first challenge is to get students to rethink learning and design, and to really engage with this new approach.
The second challenge is to explicitly develop their skills in testing, design and refactoring, given that they have little experience in these areas. This requires that fast and effective feedback be provided.

1 Introduction

Test Driven Development (TDD) is a powerful technique that has evolved from test-first programming [2], one of the practices of XP [1]. Programmer tests are designed to drive the evolutionary development of a system in small steps. TDD is likely to have as much impact on programming practice in the 2000s as structured programming had in the 1970s and object-orientation had in the 1980s and 1990s.

TDD has been taught for the last two years in a second year software engineering course, SOFTENG 251, as a part of an engineering degree at the University of Auckland. SOFTENG 251 teaches advanced object-oriented programming and design and iterative development. The development of a drawing package over several steps was used as an example, with refactorings and design rationale provided.

In 2001, JUnit and unit testing were introduced, along with test-first programming [1]. The drawing package was redeveloped test-first with a larger number of steps. Assignment work involved making a series of smaller changes using a test-first approach. However, it became clear that the students needed to build their skills further.

In 2002, experience with JUnit and unit testing were provided in a prior year two course, so that the students could develop further with TDD in SOFTENG 251. A new example was used: digital circuit simulation. Refactoring of methods in a simple class was covered in labs before introducing subclassing, so that the students could better build experience with refactorings at the method level.

From this experience, two challenges in teaching TDD have been identified[1]. The first challenge is to get students to reconsider their models of learning and design, and really learn to apply TDD. With only 12 months experience in programming before this course, some students are unhappy about "starting again" with TDD when they had found learning to program the first time a demanding exercise.

The second challenge is to explicitly develop the various skills that they need to utilise TDD well, including skills in testing, design and refactoring. This requires that students are provided with fast and effective feedback on their progress.

[1] For an extended version of this paper, with three challenges, see [5]

M. Marchesi and G. Succi (Eds.): XP 2003, LNCS 2675, pp. 410–413, 2003.

We follow with a brief introduction to TDD. Sections 3 and 4 describe the challenges and discuss what we have done to meet them. Section 5 concludes.

2 TDD

With TDD [2], frequent micro-iterations are used to develop software, in which each micro-iteration may take only a few minutes. There are generally three steps in each micro-iteration (the third is not necessary on every cycle):

- A concrete example, as an executable test, is designed, written and shown to fail
- The code is extended to pass the test while continuing to pass all other existing tests
- The code and tests are refactored [3] to eliminate redundancy

Thus abstractions are developed from concrete examples, and so the programmer doesn't have to think about all of the cases, in all their generality, all at once. A common difficulty in learning the techniques of TDD is in designing the next test, as this has a strong impact on how the program develops. Beginners especially find it difficult.

The tests act as a mirror of the developing system. Some tests are designed to expand the interface of the program and/or its components; such a test is used to design that interface. Other tests are examples that are designed to extend the capability (or boundary) of the current system, and thus lead to code expansion and generalisation.

Tests are often chosen to tackle design issues that are not too difficult and not too easy, so that something can be learned from each step. An information theoretic view of TDD would focus on the information-gathering potential of each new test [6].

This approach reflects the larger-scale cycle of XP iterations, where customer tests drive the development of the design of the whole system. This enables the customer to get feedback from the evolving system, so that the overall system design can be altered to meet the currently-understood needs of the system in an organisational context.

3 Challenge: Rethinking Learning and Design

Students inevitably bring preconceptions about learning, design and software development to a course, and these can restrict their ability to learn to apply TDD. Hence we have found it useful to explicitly address these preconceptions.

Programming is a creative act, requiring the development of a range of design and problem-solving skills. Active learning is essential, with ongoing reflection by the learner on how well they have done and how to improve.

Many students find it challenging to learn to program, but reach a level of competence that they find satisfactory. This may result in their being resistant when they have to unlearn their current approach for a new way, regardless of the perceived value of the new approach. This happened with some students in SOFTENG 251 with TDD. There is no way to avoid the pain of such change; students need to expect it as programmers and become agile learners. Maslow [4] distinguishes four stages of learning:

- Unconscious incompetence, where you are not aware that you don't know much and you don't mind

- Conscious incompetence, where you become aware of your lack of skills and don't enjoy it
- Conscious competence, where you are skillful but need to continue to put conscious effort into developing your skills
- Unconscious competence, where your skills work well without regular reflection

Design is difficult to teach and to learn, and there is considerable discomfort associated with the second and third of Maslow's stages. This is not helped by a common over-simplified model of design [6]. Often in examples of program design, a problem is clearly posed and a completed solution given. This does not take account of two issues:

- A problem may well not be clearly articulated or well understood. The design process helps to uncover a clearer view of the problem, aided by concrete examples.
- A complete solution for a realistic problem may be difficult to derive without experimentation. It may appear that an overall approach will work, but "the devil is in the detail". Appropriate abstractions and modularity may only become clear through the evolutionary development of a design.

We addressed these issues in a number of ways. Several examples of applying TDD were covered in class. The tests and code for a simple animation system (where shapes bounce around within a window) and a digital circuit simulator (with explicit gate delays) were covered in 2002, with other examples being available. Test design, refactoring and the principle of "Do the simplest thing that could possibly work" were stressed.

Assignment work involved making changes to these two systems through a sequence of explicit steps. Each of these steps required several micro-iterations and led to interesting design changes of the two systems. Students were required to hand in all of their micro-iteration steps, to show that they were applying TDD.

4 Challenge: Developing Skills

To practise TDD, a student needs to develop a range of interdependent skills:

- Write tests for an existing system, to better understand testing
- Use a testing framework such as JUnit
- Refactor application code
- Refactor tests
- Decide on areas of the application that can be extended, in order to choose a test
- Choose the next test to design, based on its intuitive merit (or economic value, in an information-theoretic sense [6])
- Design a suitable test, given the general aim of pushing the development
- Reflect on the impact of the sequence of tests that were chosen
- Make the simplest possible changes to the code (ie, don't speculate about the future)
- Understand why the code failed to pass a test (problem solving)

Many of these skills apply more generally; they are relevant to approaches other than XP. However, they tend not to be taught so thoroughly in other software development approaches.

The students had met JUnit in an earlier course in 2002, and had used it for traditional testing of several provided programs. The students had also carried out refactoring of a program where a single class was involved and all changes were to methods.

Once subtyping and subclassing were introduced, they applied refactoring in the lab to a program involving several interrelated classes. This program was a variant of the movie rental program used by Martin Fowler to explain refactorings [3]. Most of the students' practical learning of the other skills occurred when they carried out the assignment work. Simpler tasks need to be designed so that students can independently practice the various skills.

We have tried to build the needed skills incrementally, aiming to apply the ideas of TDD to the process of learning TDD. We are lacking suitable (automated and manual) tests/assessments that could be used to drive the process and to provide quick feedback to students on their progress. The correctness of the assignments was successfully assessed using the programmer tests that I had used to develop my assignment solution.

However, it's hard to assess some aspects automatically, such as the quality of the tests and the code. We have started to explore the use of simple metrics to provide automated feedback. We can compare the relative (compiled) size of the tests and code supplied by the students with the sample solution. Programs where the tests or other code are much larger than expected can be flagged, as well as those in which the size of the tests is small compared to the size of the code. We have also considered determining the code coverage of the supplied programmer tests, but this has limitations.

5 Conclusion

TDD is a powerful technique that is having a significant impact on software development [2]. It needs to be incorporated early into software curricula. TDD has been taught for the last two years in a second year software engineering course. Two challenges in teaching TDD have been identified. It is necessary to get students to reconsider their models of learning and design, and really engage with TDD. Many are reluctant to set aside the way they currently program, and "start again" with TDD when they had found it demanding to learn how to program the first time round.

The second challenge is to explicitly develop the various skills that they need in order to utilise TDD well, including skills in testing, design and refactoring. This requires that fast and effective feedback be provided to them on their progress, analogous to the role of programmer tests in TDD. A new challenge is still being planned: teaching TDD in the first introductory programming course in software engineering.

References

1. K. Beck. *eXtreme Programming Explained*. Addison Wesley, 2000.
2. K. Beck. *Test Driven Development: By Example*. Addison Wesley, 2002.
3. M. Fowler. *Refactoring*. Addison Wesley, 1999.
4. A. Maslow. *Motivation and Personality*. Harper and Row, 1970.
5. R. Mugridge. Agile learning of test driven development. Technical Report 2, Software Engineering, University of Auckland, www.se.auckland.ac.nz/tr/UA-SE-2003-2.pdf, 2003.
6. D.G. Reinertsen. *Managing the Design Factory*. The Free Press, 1997.