On Using Test-Driven Development to Tutor Novice Engineering Students using Self-Assessment

J. Boydens
Professor Software Engineering
KU Leuven, Faculty of Engineering Technology
Ostend, Belgium
E-mail: Jeroen.Boydens@kuleuven.be

P. Cordemans
Assistant Software Development
KU Leuven, Faculty of Engineering Technology
Ostend, Belgium
E-mail: Piet.Cordemans@kuleuven.be

H. Hallez
Professor Software Engineering
KU Leuven, Faculty of Engineering Technology
Ostend, Belgium
E-mail: Hans.Hallez@kuleuven.be

Keywords: Programming, Test-Driven Development

INTRODUCTION

Test-Driven Development (TDD) is a software development practice, which has been popular and effective in industry. In academia the education of TDD principles has been examined. In this paper we want to explore this idea further, namely if teaching the TDD principles from the start benefits the student in learning programming. As we target the novice student with no programming experience, TDD as well as general software concepts has to be taught in a phased approach. We illustrate the approach to guide students in a test-driven manner in our introductory course and provide the feedback from these students.

1 TEST-DRIVEN DEVELOPMENT

Testing software has become a fundamental part of the software development cycle. Software maintainability depends on the availability of test suites of automated tests. Test automation is a key condition to regularly and timely execute the necessary tests in order to detect regression and mismatches between the specification and software implementation.

Test-Driven Development (TDD) is a software development methodology which promotes writing tests before the implementation. In TDD, tests define the interface of the software and its expected behaviour. This guides the developer to improve the

design of the software, document the code and to ensure the expected behaviour does not change unintentionally when changing or extending the program in the future.

The methodology of TDD has been proven useful in industry by Nagappan ea. [6] as well as in academia [2, 3]. Moreover, Miller and Smith [5] even introduce TDD to the embedded programming courses.

Within this section we will describe the TDD methodology in sections 1.1 through **Erreur ! Source du renvoi introuvable.** as the "Red-Green-Refactor" mantra. Furthermore we will motivate the usage of TDD in our programming courses in section 1.3.

1.1 Test as an instance of the specification

Specifications describe functional and non-functional requirements of the software to be developed. These specifications may be instantiated by a set of use cases. Use cases illustrate the specification of the software by delivering correct output values given a certain input. A use case captures the interface and the behaviour of the software component. Although use cases are not a complete description of the requirements, they are an adequate source to start writing tests.

A typical automated test corresponds with a use case. The environment is set up and a set of values are selected as input. Then, the software under test is executed and a set of assertions determines whether the expected behaviour has been accomplished.

As an example Listing 1 contains the specification of the addition operation for rational numbers.

```
Given two rational numbers, called lhs and rhs, the sum of lhs
and rhs is a rational number with a numerator which is the sum of
the cross-products of numerators and denominators of lhs and rhs.
  sum.numerator ==
    lhs.numerator * rhs.denominator
    + lhs.denominator * rhs.numerator
The denominator of the sum is the product of both denominators.
  sum.denominator == lhs.denominator * rhs.denominator
```

Listing 1. Specification of the addition operation of two rational numbers.

This specification gives both an informal textual as well as a formal mathematical definition. For instance, the use case in *Eq. (1),* is an instance of the specification in Listing 1.

$$\frac{3}{4}+\frac{7}{5}=\frac{3\times5+7\times4}{4\times5}=\frac{43}{20} \quad (1)$$

When a use case has been selected, it can be translated into a software test. This is illustrated in Listing 2 for Java using the JUnit test library. In this example the first two lines of code, setup the input values: two objects, `lhs` and `rhs`. On the third line the method under test `lhs.add(rhs)`is executed and returns the `sum` object. Finally in lines 4 and 5 assertions verify the state of the `sum` object with the expected state.

```
1.  Rational lhs = new Rational(3,4);
2.  Rational rhs = new Rational(7,5);
3.  Rational sum = lhs.add(rhs);
4.  assertEquals(43, sum.getNumerator());
5.  assertEquals(20, sum.getDenominator());
```

Listing 2. Test implementation for the addition of two rational numbers.

One important consequence of defining a test this way is that the interface becomes defined as the test is written. Since this test case is the first for the `add` method, no `add` method definition has been implemented yet in the `Rational` class. This allows to envision more carefully on how the software is used, rather than immediately diving into implementation details. To quote Kent Beck [1]: "*By writing a test before implementing the item under test, attention is focussed on the item's interface and observable behaviour."* Furthermore, this approach strives towards a testable design, as the code under test has a well-defined, loosely coupled interface. Tests also drive the design towards atomic methods, in such a method a single task is executed with no side effects. Otherwise test setup quickly becomes complex. When the test becomes too complicated the design has to be reconsidered.

1.2 Red bar/Green bar mantra

Red bar, implementing the skeleton: at the moment the test is created, the interface does not have to be implemented. This means the compiler will fail and point to missing definitions. In order to get the project to compile, a minimal skeleton implementation should be programmed. This skeleton can be generated by most Integrated Development Environments, such as Eclipse or Netbeans. It is important to define the interface to correspond with the test, however no working implementation should be given at this point. A test should fail at least once in order to prove its value. If a test has never failed before, it is impossible to know that it will effectively fail when something goes wrong. Generally this error occurs when the test contains incorrect or no assertions. Therefore a working implementation at this point is omitted and the test is run the first time in order to let it fail, tools illustrate this by showing a red bar.

Green bar, minimal implementation: next, a minimal implementation is provided to allow the test to pass. It is important to keep the implementation to a minimum, excluding trivial implementations which immediately return the asserted result. As a test typically asserts only a single path in code, a method requiring multiple branches should be covered by creating more tests and reiterating the first steps. For instance, a first test with a single focus will not introduce any branches in a method. However, when more tests are introduced, the method is tested in multiple scenarios, which leads to multiple branches.

Refactor, increasing maintainability: by iterating the previous steps the TDD cycle might lead to suboptimal code. Therefore when it is deemed as necessary and all tests are passing, the code should be refactored to produce clean [4] and maintainable code. Refactoring is changing the implementation of the code to improve its maintainability without changing its external observable behaviour. For instance, renaming variables and extracting methods to remove duplication are common refactoring techniques. Refactoring without tests asserting the external behaviour is dangerous since changing code might unintentionally change the behaviour of software. However, since TDD produces a test suite, any refactoring which changes behaviour will be quickly identified by running the tests and corrective measures can be taken. Therefore it is essential to frequently run the tests while refactoring. Note that both code under tests as well as the tests themselves should be refactored.

1.3 Test-Driven Development rationale

TDD promotes to think incrementally as it drives the design towards small testable functions, which allow to isolate problems. When dealing with a complex problem, start with a simple part and gradually add tests and functionality to solve a set of sub-

problems. Eventually the complex problem should be solved, while a set of tests will assert the behaviour in different cases.

A suite of tests gives a definition of the expected behaviour in the form of executable assertions. By frequently running the test suite, all previously defined behaviour is asserted. This allows to quickly detect bugs due to changes in software, i.e. regression. Consequently code can be safely refactored without fear of breaking the code.

Furthermore, writing tests defines the interface of code. The interface is a set of functions of a component. For instance, the methods of a class in an Object Oriented programming language. Tests define how objects are constructed, how methods are called and which parameters are expected. Additionally tests define how other objects interact, even before all objects are defined.

Moreover, these tests isolate a unit. TDD drives the design of the code towards a loosely coupled set of objects which interact with each other rather than a single monolithic set of statements. Designing software in terms of manageable objects is generally considered a good practice as it leads to maintainable software. Without a detailed technical design preceding programming, this is considered as difficult to achieve. With tests however TDD drives the design towards manageable software units with only a minimal upfront design.

2 TEST-DRIVEN DEVELOPMENT APPLIED IN PRACTICAL SESSIONS OF A PROGRAMMING COURSE

Analysing requirements to derive a set of isolated components with well-defined interfaces is hard and requires specific skills, which are mostly developed by practicing writing specifications and programming complex systems. We argue that adopting TDD principles in all programming courses allows to separate the definition of requirements from actual implementation of software. This allows students to focus on either designing interfaces or on the other hand implementing correctly functioning code.

2.1 Traditional approach

Deriving a set of requirements from the prose written in the instruction manual is challenging for students. Mapping the problem description to the implemented solution is difficult at first. To design a software program that solves the problem at hand, the student tries the gain insight in the problem. By himself, he designs a methodology on how he would solve the problem. Afterwards, he implements this methodology by using programming concepts and constructs as he has seen in the lectures or in example code.

Once implemented the student needs to compile the code, which mostly fails. When writing code, the students will definitely write errors against the rules of the programming language, so-called syntax errors. Hence, a lot of time is wasted on solving these errors in order to compile code.

Code which compiles does not imply correct functionality of the code. In this case, the student has to test his code manually. This is done by given some values as input. In the ideal case, the output matches the expectations of the student. Then the student has solved the problem and finished the assignment. However, this happens very rarely. Mostly, if the output does not match the correct one, the student has to start all over again. Starting by finding the flaw in his methodology, then implementing or correcting the code, correcting syntax errors and manually verifying the correctness.

2.2 TDD approach

As an alternative we suggest to adopt a TDD style of specifying the requirements for the exercises of students. Tests are concrete, they can be asserted at run-time and provide the expected interface of the implementation. However, a test is only a single instance of a requirement, it does not fully cover the requirement. Specifically corner cases require more tests. An approach using tests is proposed as follows:

First, does the student understand the problem of the exercise? If the problem is chosen too difficult, students might fail to understand it at all, thus not being able to advance. On the other hand, when the exercise is too trivial the student might develop a hack which solves the problem but does not address the learning opportunities of the exercise.

Then is the student able to derive use cases based on the problem specification? Namely is the student able to identify the sub-problems and describe an applicable use case for each of them. Additionally this proves to be difficult as a student might immediately try to think in terms of implementation constructs, i.e. how it should be programmed rather than how the software will behave.

When a use case has been defined can the student translate the expected behaviour into a viable test? A typical problem might occur as use cases are too broadly specified so corresponding implementations are too complex. Instead of recognizing the problem and reiterating the previous step, students keep extending their implementation instead.

Finally, when executing the implementation should be verified. Is the student able to setup a manual or automatic test and interpret the test results and make changes as deemed necessary? When a test fails either the implementation it is testing or the assertions in the test are wrong. Determining which case is applicable requires to fully understand the requirements.

3 GRADUALLY INTRODUCING TESTS OVER THE PROGRAMMMING COURSE

At first students actually receive the requirements as part of the exercise manual as well as an implemented test suite. Later on the specification of each use case is provided but the test implementation is omitted. This allows the students to write tests themselves. Finally use cases can be omitted and only a general specification should be given. They are expected to devise use cases in terms of tests themselves, implement the solution and evaluate the test. This requires a full understanding of the problem at hand, knowledge of test constructs and how values are asserted, as well as the effective implementation constructs. Feedback remains necessary in order to identify and correct trivial tests and solutions and anti-patterns in implementation, such as code duplication, cryptic names and violations against data encapsulation. At the end, students start from the written problem and have to design tests and code by themselves. This way, TDD is seen as an intermediate step between problem statement and software development. The students are given the incentive to think more in the problem space and less in the solution space.

3.1 Environment

Within the novice programming course the chosen language to teach is Java. The students use a user-friendly IDE, BlueJ, designed to introduce the object-oriented nature of the Java programming language. A popular framework in writing tests is JUnit, created by Kent Beck along with Erich Gamma. It is important to note that tests are not limited to JUnit and Java. The benefit of using JUnit lies in the fact that the

framework provides assertions to check expected results, preparations of a starting state and test runners to orchestrate execution of tests and make reports . From an educational point of view, the use of a starting situation is handy to provide each student with the same data and boundary conditions.
Test cases should be defined in the same programming language as the expected implementation. This allows students to read test code as the syntax is familiar. It allows them to understand the specific use case which has been implemented in the test rather than an unfamiliar use case requirement document, which is too abstract. Furthermore they can derive valuable techniques of how a software interface is used.

3.2 Phase 1: Students are given the test code

In order to learn how the implement tests, instructors design the test at the start of the exercise or during the lab session a completed test suite is handed out to the students. This way students have good learning examples of tests and they can use these tests to develop programming.

When tests are delivered, the students can start implementing code so the test should pass according to the “red bar/green bar” mantra explained in Section 1. The instructor aids the student in developing code, explaining programming concepts and illustrates this when the test has passed.

The student is posed with the problem of developing code so the test passes. In this phase the student will use programming concepts introduced in the lectures and apply them to solve the problem.

Furthermore, the tests can be considered as milestones in developing the whole software package. These milestones should be solved one at a time each time taking into account that the previous tests should always pass.

Milestones provide a measure for productivity for the student as well as for the instructor. The student can have an indication on how far he is from the complete solution of the program. This way, tests act as a work plan for the student and provide motivation to continue.

3.3 Phase 2: Students have to construct tests from use cases

In this phase, test code is not handed out to the students anymore, but the problem description is accompanied by several use cases. An example of such a description is as follows:
“*Given a word of unknown length, switch the last two characters of the word. If the word has less than 2 characters, no switching should be done.*
-Hint: `str.charAt(i)` *returns the character at position i+1*
Use cases: “” => “”, “A”=>”A”, “AB”=>”BA”,”RAIN”=>”RANI”

In this way the problem is illustrated by a set of use cases. The students are given a set of inputs and corresponding outputs, from which they can derive the functionality of the code.

These use cases are translated into tests in the unit test software component of their program. These tests use assertions that ensure the expected output of a method is

```
@Test
public void testUseCase4(){
     assertEquals("RANI",theMethod(“RAIN”));
}
```

Listing 3: Example of a test devised from one of the use cases of the problem

the same as its result. In the above example the student should devise a test which looks like Listing 3.

3.4 Phase 3: Students have to develop tests from the problem description

At the end of the semester students should be able to build their own test code from the problem description. This is done by encouraging the students to build use cases, where they find a set of expected output results with the corresponding input. We especially take care on the use cases that are situated at the boundary, e.g. the empty string in the above example. The building of the use cases relies deeply on the understanding and insight of the student in the problem. Hence, an adequate problem description with all boundary situations should be given. The second sentence in the problem description is very important to know the function of the program in these boundary situations.

4 BENEFITS AND LIABILITIES OF TDD AS A LEARNING INSTRUMENT FOR PROGRAMMING

4.1 Benefits

In our experience in teaching programming first year engineering students, a list of benefits, but not limited to, are:

- Students can self-evaluate their programming code
- Student gain more insight in the problem and know more what needs to be done
- Students can monitor their progress as the tests can be designed as milestones in the development of the whole program.
- Students are more productive as they initially know what the tests are. Later when they develop tests themselves, the productivity is increased as they are dividing the problem into sub-problems.
- Instructors can focus more on programming concepts rather than explaining the problem

4.2 Liabilities

However, there are also risks that should be taken into account:

- There is a risk that students will need the tests delivered from the instructor. The translation from problem to test is too big.
- Tests are not available or are available in another form as students learn new programming languages. This can be confusing and therefore there will be a risk that the student drops the TDD practice in later years.
- From the fact that a complete set of tests cover all program functionality is impossible, students may just implement a minimal functionality to pass the test. In this case it is up to the instructor to deliver a set of tests or to guide the student in developing tests which are sufficiently generalized.

4.3 Student comments

Students appreciate automatic validation of their code and a quantifiable technique to monitor their progress themselves.
However when presented with an open ended project or a new programming language students are not inclined to start developing in a TDD manner or even add tests afterwards. They quickly disregard testing, as projects typically have a strict deadline and little robustness is expected from the student’s code.

5 SUMMARY AND FUTURE WORK

5.1 Summary

Students appreciate receiving the software tests first and later writing the software tests, because this allows them to monitor their progress. The instructor has to put some effort in designing all necessary software tests and writing them so they are easily readable by students, but as these tests also deliver feedback to the student, time spent on correcting and delivering feedback is reduced during lab exercises. This way, the instructor gets more freedom to focus on programming concepts, rather than basic issues that students can solve themselves.

5.2 Future work

Do a full study and experiment on how the programming knowledge is on students with and without TDD. This can be done by comparing correctness of the code, time spent on the assignment, personal enquiry on the usability of TDD.
Furthermore, we will examine the possibility of grading the test program, in order to quantify the students' abilities to produce adequate tests and, hence, adequate software programs. We are not only interested in how the students have mastered programming concepts but also in which processes they follow.

REFERENCES

[1] Beck, K. (2003), Test-driven development: by example. Addison-Wesley Professional.

[2] Buffardi K., Edwards S.H. (2012), Exploring influences on Student adherence to test-driven development, *Proceedings of the 17th ACM annual conference on Innovation and Technology in Computer science Education*, pp. 105-110.

[3] Desai C., Janzen D., Savage K. (2008), A survey of evidence for test-driven development in academia, *ACM SIGCSE Bulletin*, Vol. 40, No. 2, pp. 97-101.

[4] Martin, R. C. (2008), Clean code: a handbook of agile software craftsmanship, Pearson Education.

[5] Miller J., James, Smith M. (2007), A TDD approach to introducing students to embedded programming, *ACM SIGCSE Bulletin*, Vol. 39, No. 3, pp. 33-37.

[6] Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams, *Empirical Software Engineering*, Vol. 13 No. 3, pp. 289-302.