ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming

Vesa Lappalainen
Department of Mathematical Information Technology, University of Jyväskylä
Finland
vesa.t.lappalainen@jyu.fi

Jonne Itkonen
Department of Mathematical Information Technology, University of Jyväskylä
Finland
jonne.itkonen@jyu.fi

Ville Isomöttönen
Department of Mathematical Information Technology, University of Jyväskylä
Finland
ville.isomottonen@jyu.fi

Sami Kollanus
Department of Computer Science and Information Systems, University of Jyväskylä
Finland
sami.kollanus@jyu.fi

ABSTRACT

Research has noticed that imparting TDD-like testing to an early computing curriculum is challenging because it increases technical and cognitive load for the students. This paper addresses the challenge with a software-based solution constructed to facilitate the process of writing tests. The solution allows using a compact while efficient syntax for formulating tests, writing tests into JavaDoc comments, thus close to the source code that implements intended functionalities, and automates the generation of actual test code. The constructed solution — the ComTest tool — has now been used in four introductory level programming course offerings. The paper presents the tool and concludes with initial lessons learned.

Categories and Subject Descriptors

K.3.2. [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Design, Languages, Documentation

Keywords

TDD, Unit Testing, Tools, Education

1. INTRODUCTION

Test first programming has been practiced for decades in software engineering [15], but it was popularized as one of the key practices of Extreme Programming (XP) [2], called Test-Driven Development (TDD). In TDD, the developer writes a test, implements a functionality just enough to pass the test, and finally does refactoring if needed [3]. These steps are repeated in short cycles. According to Beck, TDD is primarily not a testing but a design practice.

In the academic environment, students have been introduced with XP project courses, e.g. [7] [8], and the courses employing particularly the test-driven development, e.g. [16] [14]. A part of the studies have investigated TDD in early programming courses, and in this context, TDD has been suggested to be a useful tool in injecting testing into the students' programming habit [11]. However, experiences from imparting TDD-like testing to introductory level courses differ. While some of the studies report positive experiences suggesting that it is possible to introduce a TDD kind of testing to a novice programming course [6], some report on difficulties due to increased load for the novice students [12]. Our research interest is based on the problem of increased technical and cognitive load for the novices.

We approach the research problem with a software based solution — ComTest — that was constructed to ease the students' test writing process. Supporting learning with tools is a widely adopted research approach in computing education, and the approach to promote particularly the students' testing habit with a tool is also acknowledged. For example, Briggs and Girard [4] implemented a tool designed to make systematic testing a more transparent effort in introductory programming. Moreover, environments designed for educational purpose also include testing tools. For example, BlueJ allows the recording of user actions to automatically generate test code.

In this paper, we want to bring our contribution to this research track. ComTest tool is introduced, and it is available at `https://trac.cc.jyu.fi/projects/comtest/wiki/ComTestInEnglish`. A plugin for Eclipse has also been written to ease writing and execution of tests, available at the same location. ComTest has now been used in four course offerings. This allows us to conclude with initial lessons learned.

ITiCSE'10, June 26–30, 2010, Bilkent, Ankara, Turkey.

2. TDD AT INTRODUCTORY LEVEL

Janzen and Saiedian [9] found that mature programmers had clearly more approving attitude towards TDD than beginner programmers in the introductory programming courses (CS1, CS2). In another paper Janzen and Saiedian [10] report that most of the CS1 students were not willing to use TDD after the course, and only few of the CS2 students used TDD when it was voluntary. Melnik and Maurer [17] also found some correlation between age and attitude towards TDD so that more experienced appreciated the method more. It seems to be a challenging task to motivate especially beginning programmers to understand the importance of TDD or unit testing overall.

In each of the Janzen and Saiedian's [9] respondent categories, the attitude towards TDD was more positive than the respondents' willingness to use the method in the future. For example, while almost 40 % of the beginner programmers believed that TDD produces fewer defects compared to test last approach, only about 10 % of the students were willing to use it in the future. Melnik and Maurer [17] also found a gap between positive beliefs and actual practice. In their study, most of the respondents believed the expected benefits of TDD but minority of them had really used it in their course work when it was not compulsory. The same gap is identifiable from beginner programmers to experienced professionals. Abrahamsson et al. [1] found in an industrial case study that employees regarded TDD as a useful practice but were not so willing to try it in practice.

Janzen and Saiedian [11] have managed to implement a TDD kind of practice at CS1 level, and they have reported positive experiences [10]. Wellington et al. [23] report that they managed to include both student written tests and refactoring at introductory level without leaving out any of the former course content. In the studies reporting positive experiences, TDD is seen not only as a design and testing practice but as an educational tool that facilitates learning.

Regardless of some positive experiences, several studies conclude that it might be too challenging to introduce TDD among the variety of new concepts for the students at CS1 level. Marrero and Settle [16] tried to include unit testing in a CS1 course, but found that this hindered students' performance compared to the former course offerings. Regarding increase in students' cognitive load, Sanders [20] reports on an observation that CS1 students, who do not yet understand what a computer can do by programming, have problems designing tests.

TDD has been found to be a difficult practice not only for beginners but also for advanced students . Several studies employing XP methodology in software engineering courses have discovered TDD as the most difficult XP practice [19] [22] [12]. The difficulties have been similarly encountered by master's level students [14] and even the experienced professionals [21]. The difficulty of learning TDD arises from the demand to handle several different skills, such as test writing, testing tools, and refactoring [18].

Keefe et al. [12] concluded that TDD-like testing with a tool such as JUnit adds too much technical load for the novice students. They recommend that at the introductory level students should be first exposed to idea of testing in general without introducing any extra load with a technical environment. Carlsson [5] also report difficulties with JUnit in CS1.

Based on the referenced literature, imparting TDD and unit testing to early programming courses is considered challenging but important. One of the major concerns is to make it simple enough in order to avoid a too high technical and cognitive load. This was the original motivation for constructing the ComTest tool presented in this paper.

3. COMTEST

3.1 Background

The ComTest tool has been used in CS1 and CS2 contexts. The objective of the CS1 is to learn procedural programming, which principally means learning to write a program using simple conditional and looping clauses, and methods as procedures. This is regarded as a fertile ground to introduce students with unit testing, because tests can be associated with simple procedures. The course uses Java programming language and is inevitably involved with JDK objects and some utility objects developed particularly for the course. However, students are not expected to design and implement their own objects. The learning objective of the CS2 is to adopt basics of object paradigm, modularity, and abstraction. Inheritance is shown in examples but not included in the obligatory objectives of the course. Both of the courses are completed by returning weekly programming assignments, a course assignment, and passing an exam.

Main methods in Java are usually used as a starting point of the program execution. In former CS2 offerings, unit level tests were written into main methods that were not used for execution of the program but for testing. As a consequence, the tests were separated from the code to be tested. In the spring 2007, JUnit was introduced in the course, due to its *de facto* position in the industry. This moved the tests into separate test classes, further away from the code to be tested. The lecturer of the course found that this complicated the teaching of testing to students. Even though the students were able to comprehend the significance of testing, writing JUnit tests was perceived as burdensome. They felt they could not write the tests under given schedule, and demonstrated a clear resistance to test writing. The number of assignments could not be reduced, so the number of obligatory tests in the weekly assignments was reduced from 100 % to 25 % per assignment. This led to the research question of how to make writing tests more reasonable in the educational context.

3.2 Basic usage

ComTest was developed to make writing of unit tests easier and to make the tests more readable. Basic idea of ComTest is writing unit tests into JavaDoc comments using a specific macro language with simple and short syntax. In the lectures the tool is used with an Eclipse plugin. Both ComTest and the associated Eclipse plugin are available as open source software licenced under GNU Lesser Generic Public License[1] (ComTest) and Eclipse Public License[2] (plugin).

With Eclipse IDE, installing ComTest is a standard Eclipse plugin installation. Eclipse's automatic translation of apostrophes to HTML coding in JavaDoc comments has to be disabled, as this interferes with ComTest. Also, JUnit 4 libraries have to be included in the project, as the test code

[1] `http://www.gnu.org/copyleft/lesser.html`
[2] `http://www.eclipse.org/org/documents/epl-v10.php`

generated by ComTest heavily relies on these. Further deployment details can be found at ComTest pages[3].

ComTest Eclipse plugin provides a code template called 'comtest' to help in writing tests. This template generates simple skeleton code for writing a test into a JavaDoc comment. Actual usage is about adding tests with the ComTest macro language to JavaDoc comments. ComTest transforms written tests into JUnit tests. Running these tests is identical to running JUnit tests. The plugin adds useful tools to Eclipse's target popup menu. The menu includes tools for generating and running the corresponding JUnit tests. Assigning user preferred shortcuts to these tools makes running ComTest tests effortless.

3.3 Syntax

A simple ComTest test is given in Example 1. The example shows how ComTest tests are written inside an *example*-tag in JavaDoc comments. The *pre*-tags of HTML allow generation of readable Java source code into API documentation. The test is named with the attribute *name*.

Example 1: Simple usage of ComTest

```
public class SmallestDivisor {

   /**
      * Function finds the smallest divisor for given
      * number
      *
      * @param n number to be studied
      * @return smallest divisor for n, 1 if n is a prime
      * @example
      * <pre name="test">
      *   smallestDivisor(1) === 1;
      *   smallestDivisor(2) === 1;
      *   smallestDivisor(3) === 1;
      *   smallestDivisor(4) === 2;
      *   smallestDivisor(5) === 1;
      *   smallestDivisor(6) === 2;
      * </pre>
      */
   public static int smallestDivisor(int n) {
      for (int i=2; i<=n/2; i++)
         if ( n % i == 0 ) return i;
      return 1;
   }

   // The traditional way of students to test a procedure
   // or module, indicating restricted test coverage.
   public static void main(String[] args) {
      int n = 25;
      int divisor = smallestDivisor(n);
      System.out.println(divisor);
   }
}
```

The ComTest macro language resembles Java extended with syntactic sugar to ease in writing tests. For example, the previous example uses the operator *===* to mark an equality test. This operator was chosen instead of Java's ordinary equality operator *==* because the latter is needed in some test sentences in its ordinary meaning, for example, when testing reference equality *a == b === true* instead of testing equality of the states of the objects *a === b*. The macro language also enables using alias *'=>'* for the *'==='* operator. This could be used for example in testing a function's outcome like *f(4) => 2* instead of *f(4) === 2*. As a matter of fact, the macro language allows to redefine the operator to any other string literal.

To further shorten the test case, the equality tests can also be written in a tabular form as in Example 2. The variables *arg* and *result* are replaced with the values from the corresponding columns.

Example 2: Simple tabular usage of ComTest

```
/**
 * Function finds the smallest divisor for given
 * number
 *
 * @param n number to be studied
 * @return smallest divisor for n,
 *         1 if n is a prime
 * @example
 * <pre name="test">
 *   smallestDivisor($arg) === $result;
 *
 *   ----------------
 *    $arg | $result
 *   ----------------
 *      1  |    1
 *      2  |    1
 *      3  |    1
 *      4  |    2
 *      5  |    1
 *      6  |    2
 * </pre>
 */
```

Example 3 shows a test for a more complicated case. The example shows a test for the method *add*, that adds a new entry into a *Counter* class, that calculates sum, minimum, and maximum of the inserted entries. This example shows how ComTest macro variables like *sum*, *max* and *min* are used, and values for these are given in the tabular format. The double line generates a new independent test case.

Example 3: Extensive tabular usage of ComTest

```
/**
 * Adds i to current counter
 * if not already too many added.
 *
 * @param i value to add
 * @example
 * <pre name="test">
 * Counter cnt = new Counter(3);
 * cnt.add($add);
 * cnt.getCount() === $count;
 * cnt.getSum() === $sum;
 * cnt.getMax() === $max;
 * cnt.getMin() === $min;
 *
 *  ---------------------------------------
 *   $add | $count | $sum   | $max | $min
 *  ---------------------------------------
 *    -   |   0    |  0     |   0  |   0
 *    1   |   1    |  1     |   1  |   1
 *    2   |   2    |  1+2   |   2  |   1
 *    3   |   3    |  1+2+3 |   3  |   1
 *    4   |   3    |  6     |   3  |   1
 *  =======================================
 *    5   |   1    |  5     |   5  |   5
 *    2   |   2    |  5+2   |   5  |   2
 *    3   |   3    |  5+2+3 |   5  |   2
 *  =======================================
 *   -1   |   1    |  -1    |  -1  |  -1
```

[3] https://trac.cc.jyu.fi/projects/comtest/wiki/ComTestInEnglish

```
 *    2   |   2    | -1+2   |   2  |  -1
 *    9   |   3    | -1+2+9 |   9  |  -1
 *
 * </pre>
 */
public void add(int i) {
  // ...
}
```

The ComTest tool has been successfully used in all CS1 and CS2 level programming assignments while the above examples characterize more of the syntax than the applicability of the macro language. Because the macro language translates into ordinary JUnit, one can write any Java and JUnit structure into a ComTest test. ComTest is equally expressive to JUnit and interoperates with test coverage tools. An advanced user can also write more complicated test code directly into the JUnit counterpart of the ComTest test. The macro language is discussed in detail at `https://trac.cc.jyu.fi/projects/comtest/browser/proto/vesa/trunk/comtest/ComTest.java`.

3.4 Benefits

We identify the following benefits in writing ComTest unit tests.

- ComTest tests are shorter and easier to write than their JUnit counterparts. JUnit counterpart for Example 3 is found at `https://trac.cc.jyu.fi/projects/comtest/browser/proto/vesa/trunk/example/test/CounterTest.java`. The JUnit counterpart takes 50 lines not including imports or package definition, nor whitespace or comments, whereas ComTest takes 22 lines.
- Writing tests into JavaDoc comments keeps the tests close to the implementations, so the cognitive load in browsing between tests and implementations diminishes.
- ComTest is a tool for designing classes and methods, particularly interfaces, see Example 3 (cf. test-first). A ComTest test specifies how a unit to be implemented works.
- ComTest enables writing tests inside JavaDoc comments, where tests also serve as examples of method usage. This slightly resembles Knuth's literate programming [13] which focuses on writing source code that explains the logic of a program, instead of source code that only instructs a compiler.

The first three of these particularly address our research concern on testing at introductory level programming courses.

4. LESSONS LEARNED

ComTest has now been used in two CS2 offerings (spring 2008, spring 2009) and two CS1 offerings (autumn 2008, autumn 2009).

In the CS2 offerings, teaching testing started from using traditional printline commands. Deficiency of such approach was demonstrated to the students and the need for a more advanced approach stated. ComTest was then presented as an alternative for automated unit testing and the relation between ComTest and JUnit was explained. The students could choose whether they used ComTest or JUnit in the weekly assignments and the course assignment. Unit tests must be included in the course assignment, and test-first was the recommended approach, even though not forced. Test-first approach was used in the lectures from which the students could mimic it. In CS1, the students were not expected to write tests in the beginning of the course because of their insufficient understanding of the structure of a program. During the latter half of the course students were introduced to testing while the lecture had used ComTest examples early in the lectures. The students were first encouraged to mimic the style of the lecture and use existing tests in the weekly assignments, and then write their own tests. Writing own tests was not compulsory but included in the bonus tasks of the weekly assignments during the latter half of the course.

CS2	N	Tool users		AVG of tests	
		ComTest	JUnit	ComTest	JUnit
2007	93	-	s=57,t=150	-	2.6
2008	95	s=75,t=258	s=23,t=69	3.4	3.0
2009	99	s=80,t=264	s=15,t=37	3.3	2.85
s=number of students, t=number of tests					

Table 1: The number of ComTest and JUnit users and their test writing in a CS2 weekly assignment

CS1	N	ComTest users	AVG of tests
2008	124	s=27,t=71	2.6
2009	153	s=26,t=72	2.8
s=number of students, t=number of tests			

Table 2: The number of ComTest users who wrote their own tests and their test writing in a CS1 weekly assignment

In Table 1, the number of ComTest and JUnit users and their test writing are given from a representative weekly assignment round over the three year period in CS2. The assingment was very similar each year. The number of JUnit users declined after introducing ComTest in the 2008 offering and the ComTest users were able to write more tests than the JUnit users. The rest of the data follows the same tendency[4]. The lecturer also noticed that the students' resistance to test writing, initially raised by using JUnit in the 2007 offering, disappeared after introducing the ComTest tool.

In Table 2, an example is given from the CS1 weekly assignment data. Only minority of the students are able to write their own tests but the average number of tests those students wrote in a single weekly assignment round is promising. Overall, the weekly assignment data indicates that the CS1 students felt insecure writing their own tests but took advantage of pre-written tests.

5. CONCLUSIONS AND FUTURE WORK

In this paper, a tool to ease students' test writing process was introduced. The initial observations indicate that the tool is useful at both CS1 and CS2 contexts. Most of the

[4]Both the CS1 and CS2 data is found at `https://trac.cc.jyu.fi/projects/comtest/wiki/ComTestInEnglish/Observations`.

students chose to use ComTest at CS2 level, and ComTest users wrote more tests than JUnit users. CS1 students took advantage of pre-written tests but only some of them wrote their own tests. This is why we plan to make test writing a compulsory part of the CS1 course in the future. This allows us to better observe CS1 students' capability of test writing.

ComTest and the Eclipse plugin are finished enough for teaching but require future development to become a software engineering tool. For example, automatic syntax checking for ComTest tests is not yet implemented. Another interesting future goal is to make the testing of private methods possible. This could be implemented by generating a duplicate of the source code in which *private* keywords are replaced with *public* keywords and then running tests on such a manipulated version of the code. We aim to address these issues in the near future and design a controlled experiment to evaluate the tool.

6. REFERENCES

[1] P. Abrahamsson, A. Hanhineva, and J. Jäälinoja. Improving business agility through technical solutions: A case study on test-driven development in mobile software development. In R. .Baskerville, L. Mathiassen, J. Pries-Heje, and J. DeGross, editors, *Business Agility and Information Technology Diffusion*, pages 227–243. Springer, 2005.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, first edition, 1999.

[3] K. Beck. *Test-Driven Development: By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[4] T. Briggs and C. D. Girard. Tools and techniques for test-driven learning in CS1. *J. Comput. Small Coll.*, 22(3):37–43, 2007.

[5] B. Carlson. An agile classroom experience: Teaching TDD and refactoring. In *AGILE '08: Proceedings of the Agile 2008*, pages 465–469, Washington, DC, USA, 2008. IEEE Computer Society.

[6] C. Desai, D. S. Janzen, and J. Clements. Implications of integrating test-driven development into CS1/CS2 curricula. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 148–152, New York, NY, USA, 2009. ACM.

[7] Y. Dubinsky and O. Hazzan. Improvement of software quality: Introducing extreme programming into a project-based course. In *14th International Conference of the Israel Society for Quality*, pages 8–12, Jerusalem, Israel, 2002.

[8] G. Hedin, L. Bendix, and B. Magnusson. Teaching extreme programming to large groups of students. *Journal of Systems and Software*, 74(2):133–146, 2005.

[9] D. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 719–722, May 2007.

[10] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, 2008.

[11] D. S. Janzen and H. Saiedian. Test-driven learning: Intrinsic integration of testing into the CS/SE curriculum. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 254–258, New York, NY, USA, 2006. ACM.

[12] K. Keefe, J. Sheard, and M. Dick. Adopting XP practices for teaching object oriented programming. In *ACE '06: Proceedings of the 8th Austalian conference on Computing education*, pages 91–100, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[13] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[14] S. Kollanus and V. Isomöttönen. Test-driven development in education: Experiences with critical viewpoints. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 124–127, New York, NY, USA, 2008. ACM.

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

[16] W. Marrero and A. Settle. Testing first: Emphasizing testing in early programming courses. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 4–8, New York, NY, USA, 2005. ACM.

[17] G. Melnik and F. Maurer. A cross-program investigation of students' perceptions of agile methods. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 481–488, New York, NY, USA, 2005. ACM.

[18] R. Mugridge. Challenges in teaching test driven development. In *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, page 1015, Berlin Heidelberg, 2003. Springer.

[19] M. M. Müller, J. Link, R. Sand, and G. Malpohl. Extreme programming in curriculum: Experiences from academia and industry. In *Extreme Programming and Agile Processes in Software Engineering*, volume 3092 of *Lecture Notes in Computer Science*, Berlin Heidelberg, 2004. Springer.

[20] D. Sanders. Extreme programming: The student view. *Computer Science Education*, 12(3):235–250, 2002.

[21] O. P. N. Slyngstad, J. Li, R. Conradi, H. Ronneberg, E. Landre, and H. Wesenberg. The impact of test driven development on the evolution of a reusable framework of components - an industrial case study. In *ICSEA '08: Proceedings of the 2008 The Third International Conference on Software Engineering Advances*, pages 214–223, Washington, DC, USA, 2008. IEEE Computer Society.

[22] C. Wellington. Managing a project course using extreme programming. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, pages T3G–1, Oct. 2005.

[23] C. A. Wellington, T. H. Briggs, and C. D. Girard. Experiences using automated tests and test driven development in computer science I. In *AGILE '07: Proceedings of the AGILE 2007*, pages 106–112, Washington, DC, USA, 2007. IEEE Computer Society.