Journal of Computer Science 3 (5): 335-344, 2007
ISSN 1549-3636
© 2007 Science Publications

Measured Test-Driven Development: Using Measures to Monitor and Control the Unit Development

[1]Y. Dubinsky and [2]O. Hazzan
[1] Department of Computer Science, Technion – Israel Institute of Technology
[2] Department of Education in Technology & Science, Technion – Israel Institute of Technology

Abstract: We analyze Test Driven Development (TDD) from cognitive and social perspectives. Based on our analysis, we suggest a technique for controlling and monitoring the TDD process by examining measures that relate to the size and complexity of both code and tests. We call this approach *Measured* TDD. The motivation for TDD arose from practitioners' tendency to rush into code production, skipping the required testing needed to manufacture quality products. The motivation for *Measured* TDD is based on difficulties encountered by practitioners in applying TDD. Specifically, with the need to frequently refactor the unit, after every few test and code steps have been performed. We found that the suggested technique enables developers to gain better control over the development process.

Key words: unit testing, Test Driven Development, measures

INTRODUCTION

Test Driven Development (TDD), an agile software development practice, aims to systematically overcome some of the characteristic problems of software development processes [1, 2, 3]. Though TDD has proven benefits, it is still one of the more difficult practices for implementation by software teams [4, 5]. In this paper, we analyze this phenomenon from a human perspective and argue that TDD can only partially solve the problems associated with traditional testing since additional conditions are needed in order to exhaust its benefits. This idea has already been mentioned in the literature. For example, in Extreme Programming development environments, TDD is strongly supported by other practices, like pair programming and simple design [6]; software teams are advised to apply TDD only when all teammates agree on its use, and in other cases, to better give it up [7]. Indeed, TDD requires a collaborative environment and additional supporting practices in order to be integrated successfully into software development processes.

Specifically, in this paper, we suggest that TDD be referred to as a process that, like other processes, should be monitored and controlled. For this purpose, we introduce a technique, named *Measured* TDD, that is based on size and complexity measures and that continuously monitors the TDD process. In addition, we illustrate both how this technique ensures the performance of TDD and how it provides ongoing quality measures.

The data presented in this paper were gathered as part of a comprehensive research, conducted over the past five years, on the introduction of agile software development methods into the work of software teams, both in academia and in industry [8, 9, 10]. In both cases, we introduced TDD as part of the agile approach and investigated its acceptance by developers, using several research tools for data collection and analysis.

In this work we analyze TDD from a human perspective and present the data that motivated the development of the Measured TDD technique. We then introduce the Measured TDD technique and present data that show how it supports development processes. We illustrate Measured TDD using a specific example.

TEST DRIVEN DEVELOPMENT

What is Test Driven Development?: Test Driven Development (TDD) is a programming technique that aims to provide clean, fault-free code [1]. TDD means that, first, we write a test case that fails and then we write the simplest code possible that enables the test to pass successfully. TDD implies that new code is added only if an automated test has failed. In addition, in order to improve our code, we perform refactoring activities [11], among other reasons, to eliminate duplications. Accordingly, the TDD guideline is *red / green / refactor*, where *red* means writing a simple test that fails; *green* means writing the minimal and simplest code that causes the test to pass (In graphical testing environments this is represented by a red/green bar

Corresponding Author: Y. Dubinsky, Department of Computer Science, Technion – Israel Institute of Technology

displayed when the test fails/passes); *refactor* means that code quality is improved without adding functionality. This guideline is iteratively implemented in small steps. The accumulative experience of the community is that TDD provides high-quality code [4], which usually means that the code is readable and includes fewer bugs as well. Furthermore, there is evidence that through this process, software developers improve their understanding with respect to the developed product [12].

How does TDD help overcome some of the problems inherent in testing?: In this sub-section, we analyze how TDD can help overcome some of the common problems associated with traditional testing that are encountered in software projects. The TDD analysis presented in this section further reinforces the importance attributed to human aspects of software engineering [13]. Thus, it is people-centered and addresses cognitive, social, affective and managerial elements. The analysis in this section is structured around arguments frequently offered to explain why, in many cases, traditional testing is skipped. Such arguments are accompanied by explanations on how TDD might help overcome these obstacles. In the subsequent sub-section, however, based on two data sets, we conclude that TDD processes should be more closely controlled in order to better exhaust their potential. This conclusion constitutes the motivation for the Measured TDD technique.

Not enough time to test: Traditionally, unit testing, if it exists, is performed after the code is written and usually under time pressure. Thus, according to Van Vliet, "the testing activity often does not get the attention it deserves. By the time the software has been written, we are often pressed for time, which does not encourage thorough testing" [14: p. 397]. However, "postponing test activities for too long is one of the most severe mistakes often made in software development projects. This postponement makes testing a rather costly affair" (ibid.). Since TDD introduces unit tests throughout the entire development process, this problem is eliminated in TDD processes.

Testing provides negative feedback: Traditional testing processes require the developer to find bugs in his or her own work; in other words, testing activities end in failure. Indeed, who would enjoy that? [15, 16]. In TDD, the rules of the game are reversed. TDD ends in success: after the test fails, code is written and the test passes – success! To illustrate this perspective, we quote the reflection of a practitioner, Michael Feathers: (http://c2.com/cgi/wiki?CodeUnitTestFirst) "Why don't people like testing? Well, the traditional way of testing is tough to take. You write what seems to be perfectly sensible code, then you write a test and the test tells you that you failed. No one wants to hear that. Let's turn it around. Write the test first; run it. Of course it fails.. You haven't written the code under test yet. Start writing code.. keep testing. Soon, the test will tell you that you've succeeded!"

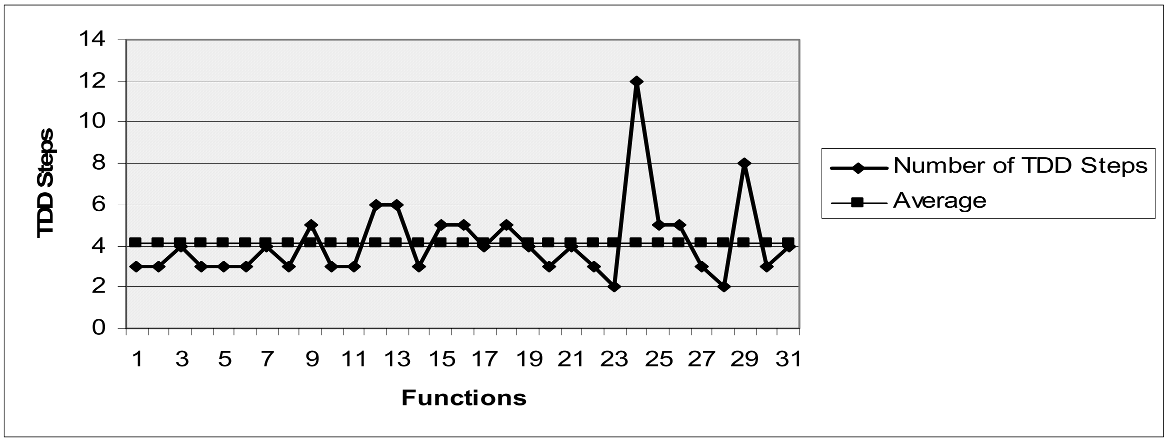
Responsibility for testing is transferred: In traditional environments, bugs are found and, in many cases, also fixed by other practitioners rather than by the developer who actually wrote the code. In TDD processes, the responsibility for testing is borne by the person who writes the code.

Testing is a low-status job: In traditional software development environments, testing is carried out at the end of the production line, and, inspired by traditional working class jobs, the task is attributed low status, which in turn leads to tension among different groups of employees. Cohen et al. [17] reported that "though most organizations recognize the need for high-quality testers and their specialized skill set, testers still struggle to win the respect they deserve. … The lack of status and support makes the tester's job more difficult and time consuming, as the struggle for recognition becomes part of the job itself" (p. 80). Since in TDD processes all developers test their own code, this negative feeling towards testing is eliminated.

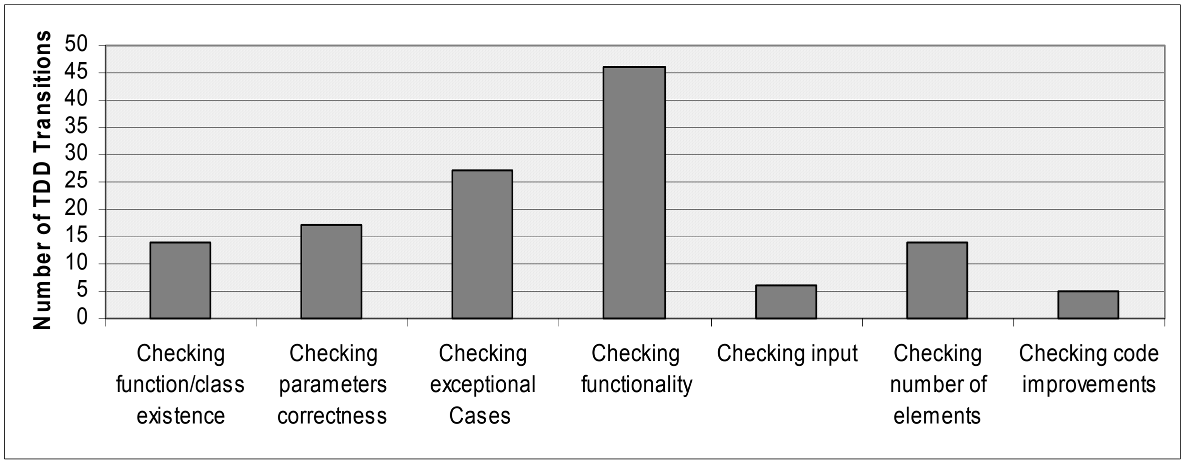
Testing is hard to manage: From a managerial perspective, it is sometimes claimed that in general, testing is a hard process to manage and, in particular, testing slows down the development process. Since TDD is firmly integrated throughout the entire software development process, it turns development and testing into controlled processes. Furthermore, the fact that TDD is done by writing automatic (not manual) tests, further increases the control level. Indeed, introducing TDD might slow down the development process in the short term simply because testing is actually performed. In the long run, however, it assists in shortening the integration period (especially when performing continuous integration).

Testing is hard: Testing is also difficult from a cognitive perspective mainly because it is not always clear what tests are suitable for a specific purpose and how much testing should be done. The following reflection of a practitioner, Ron Jeffries, explains how TDD supports the testing from the cognitive perspective (http://c2.com/cgi/wiki?RonJeffries): "A key aspect of this process: don't try to implement two things at a time, don't try to fix two things at a time. Just do one. When you get this right, development turns into a very pleasant cycle of testing, seeing a simple thing to fix, fixing it, testing, getting positive feedback all the way. Guaranteed flow.". Being a detail-oriented and explicit process, TDD improves one's understanding of what should be developed since the test must be written prior to the writing of the code.

Why is TDD not sufficient? Why it is still not performed at full scale?: Indeed, as illustrated in the previous section, TDD helps cope with traditional problems related to traditional testing. This section presents two illustrative data sets that show that practitioners still find it hard to use TDD. These data sets, as well as other data, motivated us to further elaborate on the TDD framework and to develop the Measured TDD technique.



(a)



(b)

Fig. 1: TDD steps and the reasons for the respective TDD transitions

Data Set 1. TDD Steps: An examination of 31 functions, developed using TDD, reflect a total of 129 TDD steps. The participants were asked to save the file that corresponds to each TDD step, to explain why they moved on to the next step, and to document their refactoring activities. By refactoring activities we refer to code refactoring activities that should be carried out frequently, on the level of the currently-developed unit, every time after several test and code steps have been developed.

Figure 1a presents the number of TDD steps for each of the 31 functions, as well as the average number of TDD steps per function (horizontal, dashed line), which was found to be 4.16. Figure 1b presents the reasons given for making TDD transitions to the next step in the development of the examined functions, as reported by the participants, as well as the number of times each reason was given. As can be observed, the two main reasons for moving to the next TDD step are checking the functionality of the feature to be implemented and checking exceptional cases. An examination of the participants' reports on their refactoring activities revealed that the participants did not perform code refactoring at all, i.e., they did not stop to improve the unit code each time after several test and code steps had been developed; rather, they continued developing till the unit coding was completed.

We note that we examine the refactoring activities performed during the development of specific functions developed by TDD as part of a large software project. Refactoring activities that are on the level of the project as a whole, such as improving class hierarchies are not included in the analysis presented in this paper.

Data set 2. Reflection on TDD: Participants were asked to reflect on the TDD activity. Following are some of the participants' expressions, categorized into pros and cons.

Developers described the *advantages* of TDD as follows:

- "It makes us think ahead";
- "There are less bugs. Developers are forced to produce high-quality software";
- "It helps us get acquainted with the software components";
- "It makes us think before coding";
- "It requires writing minimum code in order for the tests to pass";
- "It saves time that used to be dedicated to bug finding";
- "It helps in quality assurance!"

Developers described the *disadvantages* of TDD as follows:

- "Work is delayed because of relatively simple items";
- "It requires double the time to write code";
- "It increases development time";
- "There is no global view when dealing with complicated components";
- "Is hard to identify the critical cases";
- "Is not suitable for every kind of task";
- "Is a waste of time if the code is later not used".

An examination of these reflections reveals two main observations.

First, developers tend to refer to TDD as a thinking activity in general, and as a thinking-before-coding activity in particular. This observation means that when TDD guides the development process, coding is not perceived by developers as a spontaneous developer-computer interaction, but rather, developers perceive it as an activity that requires thinking before performing. This can be explained by the fact that unlike TDD, which forces the developer to think before coding, in many other cases developers tend to start coding intuitively.

The second observation is the contradictions and conflicts that TDD introduces. For example, one developer claimed that TDD ensures fewer bugs occur and consequently leads to shorter integration times. At the same time, however, this developer claimed that the time overhead that TDD introduces is a disadvantage. Another developer claimed that since she thinks before coding, she knows exactly what she is going to code. At the same time, however, this developer indicated a feeling of uncertainty when practicing the TDD approach. A third developer claimed that TDD disrupts the coding continuity, but acknowledged its convenience. These contradicting reflections may be explained by the fact that, traditionally, developers used to code first and test later; TDD forces them to perform these activities in a reversed order. Accordingly, their first TDD experiences cause mixed feelings and contradicting opinions.

Based on the above illustration, it is clear that TDD has many benefits and that it indeed might help cope with some of the cognitive, affective, social and managerial problems associated with traditional testing (as aforementioned). However, our Data Set 1 and the accumulated experience of the agile community tell us that though TDD does help overcome many of the problems associated with traditional testing processes by providing a tight and clear testing procedure to follow, it is not fully performed in agile projects and is still considered to be one of the more difficult practices to introduce when the decision to apply the agile approach in the organization is taken. Furthermore, even when TDD is applied, developers tend to reduce the number of TDD steps and to skip the refactoring phase, which is required repeatedly after every few TDD steps. In addition, according to Data Set 2, the new work habit that TDD introduces leads to some confusing feelings.

We propose that the reason for these phenomena is that, like other processes that must be measured, disciplined and controlled, TDD processes should also be measured and controlled. Specifically, we suggest that measures be taken alongside the TDD steps, to lead and guide this process. To that end, we present a technique whereby two measures are added to the TDD process, rendering it a more controlled process. We call this technique *Measured TDD.*

MEASURED TDD

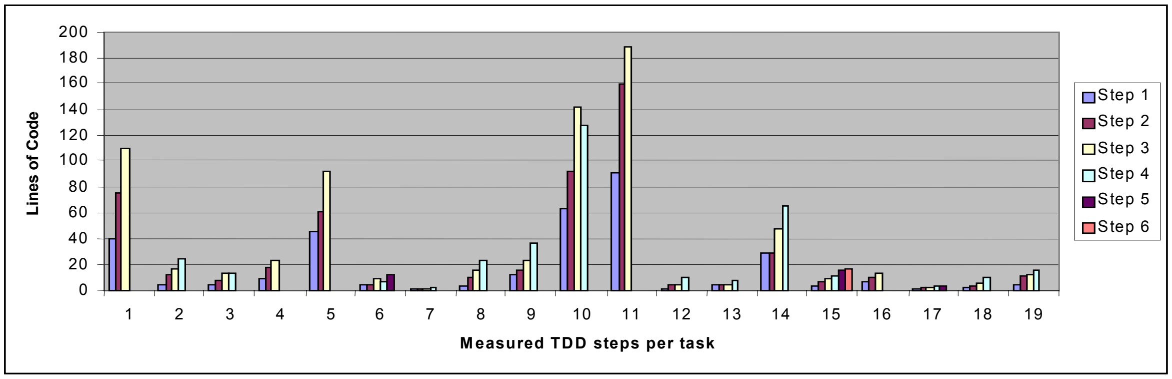
So far in this paper, we have analyzed problems associated with traditional testing and have presented data sets to illustrate why TDD, although it overcomes some of such problems, is still insufficient. In this section we introduce the Measured TDD technique, which deals with these yet unsolved problems and aims to improve the performance of TDD by incorporating measures and control elements into the TDD process itself.

Specifically, at the end of each TDD step, developers measure the size and complexity of the developed code. Size is determined by the number of lines and complexity is determined by calculating the cyclomatic complexity [18, 19], whereby a sequential method has a complexity of 1, and each decision that causes a binary split adds 1 to the complexity. The *Metrics* software for example provides Eclipse plug-in that automatically calculates McCabe cyclomatic complexity (http://metrics.sourceforge.net/). Size and complexity are measured also with respect to the evolved test. We note that other measures can be taken as well. However, we choose the aforementioned measures since they are simple, easy to use and can be taken automatically.

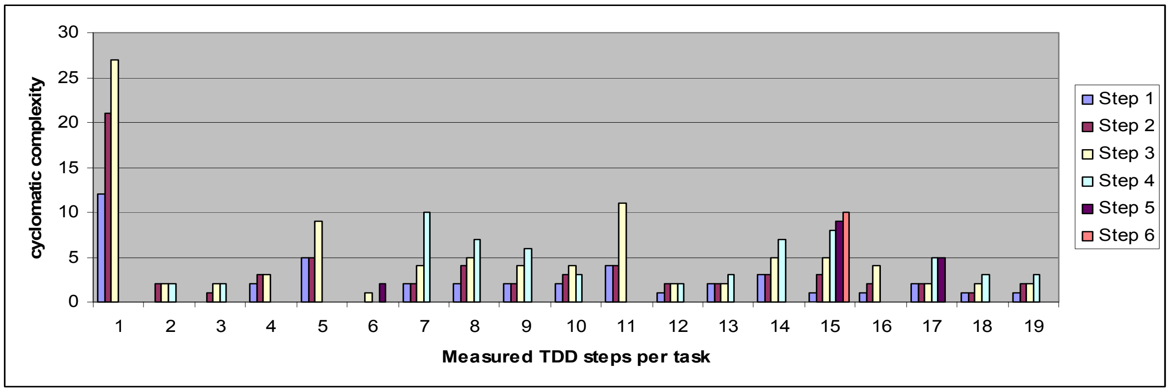
Measured TDD has the added value of measuring while developing. Specifically, the use of the size and complexity measures helps developers determine when, while implementing TDD steps of both the test and the code, they should refactor the code. This observation is reflected in Data Set 3.

Data Set 3. Size and Complexity Measures: An examination of the size and complexity measures of 19 different functions developed through a Measured TDD process and of the 75 TDD steps associated with these 19 developed functions reveals the following observations.

First, since in general each line of code is inspected by several tests (and each test is usually one line long), more lines of test are expected than lines of code. Indeed, the 19 different functions developed through



(a)



(b)

Fig. 2: Code size and complexity for Measured TDD steps

the Measured TDD process yielded about two times more lines of test (1582) than lines of code (800).

Second, as can be observed in Fig. 2, which presents the size (2a) and complexity (2b) of the code for each Measured TDD step, most of the functions were developed in three or four Measured TDD steps. Fig. 2a shows that most functions have less than 20 lines of code, which means that the functions are simple. Figure 2b, which presents the code complexity of each Measured TDD step in terms of cyclomatic complexity, further validates this assumption about the nature of the function developed through the Measured TDD process.

Third, though the number of TDD steps did not increase relative to Data Set 1 (the average number of Measured TDD steps for these 19 functions was 3.95), we will see later on that the added value of the Measured TDD is expressed mainly by the actual performance of refactoring activities that lead to simpler code. We note that when Data Set 3 is combined with another data set that refers to 16 functions developed through a Measured TDD for which the average number of Measured TDD steps was 5.5 steps (an increase of 32% relative to Data Set 1), the average number of Measured TDD steps for the 35 functions is 4.66, which indicates a 12% increase relative to Data Set 1.

Fourth, Fig. 2b reveals that the cyclomatic complexity in most cases is less than 5. As mentioned before, this means that indeed most of these functions are not complicated. In one case, for example, in which the cyclomatic complexity soared to 27, the code was checked and it was found that the task included nine hash table manipulation functions. When complexity was higher than 5, developers suggested improvements and in some cases also implemented them. In two of the 19 cases, in which the cyclomatic complexity was reduced (#6 and #10), the size of the code was also reduced (see Fig. 3). We conclude that since developers constantly monitor their work, Measured TDD keeps complexity, as well as size of code, low.

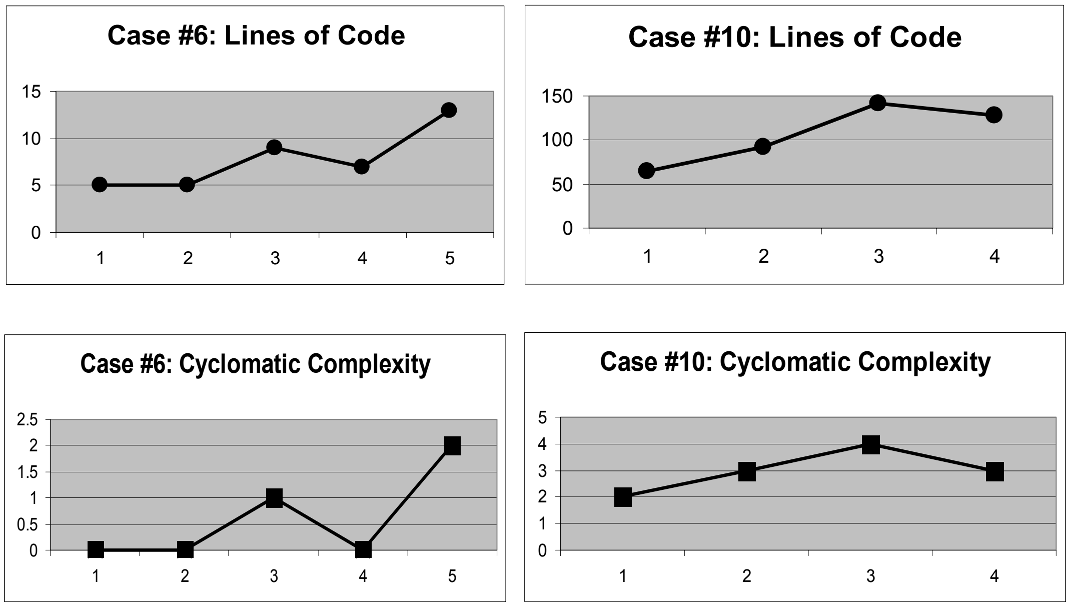
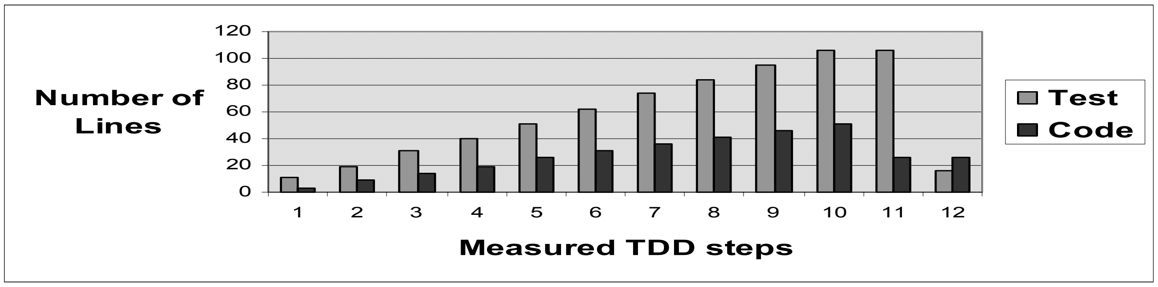


Fig. 3: Reduction in cyclomatic complexity and code size



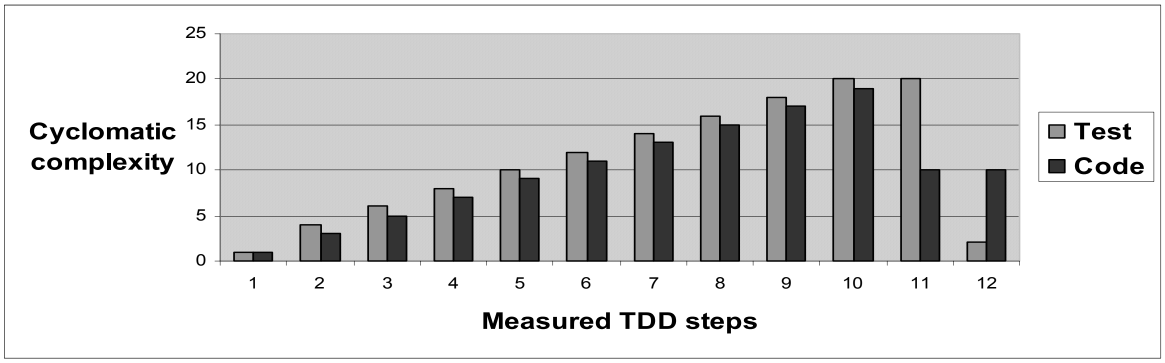


Fig. 4: Test and code measures of SearchCommand

ILLUSTRATING MEASURED TDD

This section illustrates the Measured TDD process by a Java class that was part of a project developed in the academia. The project was conducted in a 'Projects in Operating Systems' course that the first author teaches at the department of Computer Science of the Technion, Israel. The project deals with the development of a shell language that enables search in several digital libraries [20]. In order to increase awareness to measures as well as to their implications, the developers were asked to complete a tracking table containing the following information: number of step, test and code descriptions, size of test and code, cyclomatic complexity of test and code, and a description of the refactoring activities performed. The developers were told that they must complete all table columns for each TDD step and, specifically, a refactoring description must be written for all steps, even if they decide that a specific step requires no refactoring.

The name of the class was SearchCommand. Table 1 presents a tracking table for the class, as submitted by one of the developers. As can be observed, the table was indeed used by the developer to track the development process. The refactoring column was completed for all 12 steps and indicates the exact step (#7) where the developer became aware of the need to refactor. At this stage, the developer also started to make use of the measures, citing the high cyclomatic complexity as the rationale for her need to refactor. Though the need to refactor was detected in Step #7, the developer decided to continue with the class development before performing the refactoring (see the Comments column). The refactoring of both code and test was carried out in the last two steps (#11 and #12 respectively).

Figure 4 presents the test and code measures as reported in the tracking table (Table 1). It is clear that both refactoring activities (of the code and of the test) reduced the size and lowered the complexity of both code and test. It can be concluded that, as a result, the clarity and simplicity of both test and code increased.

Table 1: The Measured TDD tracking table for SearchCommand

#	Test	Code	Test lines#	Code lines#	Test CC	Code CC	Refactor	Comments
1	Sanity Check	Method signature	11	3	1	1	Not needed - I didn't start yet	Method signature and test's tearUp method
2	Set parameter – text	Handle text parameter	19	9	4	3	Not needed	
3	Set parameter – name	Handle name parameter	31	14	6	5	Not needed	
4	Set parameter -searchList	Handle searchList parameter	40	19	8	7	Not needed	
5	Set parameter -meta	Handle meta parameter	51	26	10	9	Not needed	
6	Set parameter –op	Handle op parameter	62	31	12	11	Not needed	
7	Set parameter –caseSensitive	Handle caseSensitive parameter	74	36	14	13	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
8	Set parameter –rank	Handle rank parameter	84	41	16	15	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
9	Set parameter - intoResultList	Handle intoResultList parameter	95	46	18	17	CC high -> need refactoring, later	Needed refactoring, I'll do it at end
10	Set parameter –popup	Handle popup parameter	106	51	20	19	CC high, code HIGH -> need refactoring	Finished method, now refactoring
11	Refactor existing code	Refactoring by introducing checkValue function, it will reduce both CC and CodeLines	106	26	20	10	Introduced a method that reduced code duplication	
12	Refactor test code	Refactor test code by introducing a help test method	16	26	2	10	Introduced a method that reduced duplication in test code	

To illustrate how Measured TDD is used, we present the code for three of the steps (#1, #7, and #12). The test of Step #1 consists of a sanity check. The developer checks that SearchCommand can be instantiated. Naturally, this test fails since no code exists at that time (note that the developed class extends the abstract class ACommand, and therefore three empty methods are created). Both measures are low and no refactoring action is needed.

The examination of Step #7 shows that the code indeed grew longer and now includes many repetitions. The code complexity measure for this step is 13 and increases to 19 before refactoring is performed (see Table 1).

Finally, in Steps #11 and #12, refactoring is performed. Both test and code are improved by introducing a method that eliminates code duplications. The measures indicate that the code is indeed more concise and simpler (see Table 1 and Fig. 4). Following is the refactored code.

Code of Step #1

```
package gsdl.command;
import gsdl.exception.*;
public class SearchCommand extends ACommand {
            public void setParameter(String name, String value) throws IllegalCommandException {
            }
            public void verifyParameters() throws IllegalCommandException {
            }
            public void execute() throws ScriptException {
            }
}
```

Code of Step #7

```
package gsdl.command;
import gsdl.exception.*;
public class SearchCommand extends ACommand {
  private static final String PARAM_TEXT = "-text";
  private … [variables declaration and initialization]
  public void setParameter(String name, String value) throws IllegalCommandException {
            if (PARAM_TEXT.equals(name)) {
                        if (value == null || value.length() == 0)
                          throw new IllegalCommandException(STR_GOT_NULL_VALUE + name);
                        } else if (PARAM_NAME.equals(name)) {
                        if (value == null || value.length() == 0)
                          throw new IllegalCommandException(STR_GOT_NULL_VALUE + name);
                        else
                          strName = value;
            } else if (PARAM_LIST.equals(name)) {
                        if (value == null || value.length() == 0)
                          throw new IllegalCommandException(STR_GOT_NULL_VALUE + name);
                        else
                                    strSearchList = value;
            } else if … [Same for more PARAM_'s]
                        ..
            }
            else   throw new IllegalCommandException("Got invalid parameter: " + name);
  }
  public void verifyParameters() throws IllegalCommandException { }
  public void execute() throws ScriptException { }
}
```

Code of Step #12

```
package gsdl.command;
import gsdl.exception.*;
public class SearchCommand extends ACommand {
  private static final String PARAM_TEXT = "-text";
  private … [variables declaration and initialization]
  public void setParameter(String name, String value) throws IllegalCommandException {
            checkValue(name,  value);
            if (PARAM_TEXT.equals(name)) {
            } else if (PARAM_NAME.equals(name)) {
                        strName = value;
            }else if (PARAM_LIST.equals(name)) {
                        strSearchList = value;
            }else if … [Same for more PARAM_'s]
                        ..
            }
```

```
        else   throw new IllegalCommandException("Got invalid parameter: " + name);
  }
  public void verifyParameters() throws IllegalCommandException { }
  public void execute() throws ScriptException { }
  private void checkValue(String name, String value) throws IllegalCommandException {
        if (value == null || value.length() == 0)
                throw new IllegalCommandException(STR_GOT_NULL_VALUE + name);
  }
}
```

CONCLUSION

In this paper we present a technique for function development that uses size and complexity measures for monitoring and controlling the TDD process. Though the use of these measures for the improvement of software development processes is already known, we suggest that the contribution of our work is expressed by the application of these measures in the context of TDD.

As mentioned above, measures are known to be beneficial for software development in general. In the case of measured TDD, we found that it overcomes difficulties developers face with applying and sustaining TDD and, specifically, encourages function refactoring; thus the TDD advantage of developing high quality software is gained. The simple and easy-to-automate measures ensure no significant overhead.

From a broader perspective, measured TDD provides us a means to promote automated unit tests which are considered to be the basis for the evolved software design.

REFERENCES

1. Beck, K., 2003. Test-Driven Development By Example, Addison Wesley.
2. Feathers, M., 2004. Working Effectively with Legacy Code, Prentice Hall.
3. Newkirk, JW and Vorontsov, AA., 2004. Test-Driven Development in Microsoft .NET, Microsoft Press.
4. George, B. and Williams, L., 2003. An initial investigation of test driven development in industry, Proceedings of the ACM Symposium on Applied Computing, March 09-12, Melbourne, Florida.
5. Meszaros, G., Smith, S. M and Andrea, J., 2003. The test automation manifesto, Proceedings of the XP/Agile Conference, pp. 73-81.
6. Beck, K., 2000. Extreme Programming Explained, Addison-Wesley.
7. Ambler, S.W., 2006. Introduction to Test Driven Development (TDD), http://www.agiledata.org/essays/tdd.html, Last updated: July 28, 2006.
8. Dubinsky, Y. and Hazzan, O., 2005. The construction process of a framework for teaching software development methods, Computer Science Education **15**(4), pp. 275–296.
9. Dubinsky, Y., Talby, D., Hazzan, O. and Keren, A., 2005. Agile Metrics at the Israeli Air Force, Agile Conference, Denver, Colorado.
10. Talby, D., Hazzan, O., Dubinsky, Y. and Keren, A., 2006. Agile software testing in a large-scale project, IEEE Software, Special Issue on Software Testing.
11. Fowler, M., 1999. Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional.
12. George, B., Williams, L. A., 2004. A structured experiment of test-driven development, Information & Software Technology **46**, pp. 337–342.
13. Tomayko, J. and Hazzan, O., 2004. Human Aspects of Software Engineering, Charles River Media.
14. Van Vliet, H., 2000. Software Engineering – Principles and Practices, Wiley.
15. Hamlet, D. and Maybee, J., 2001. The Engineering of Software, Addison Wesley.
16. Hazzan, O. and Leron, U., 2006. Why Do We Resist Testing?, System Design Frontier **3**(8), pp. 13-17.
17. Cohen, C. F., Birkin, S. J., Garfield, M. J. and Webb, H. W., 2004. Managing conflict in software testing, Communications of the ACM **47**(1), pp. 76-81
18. McCabe, T., 1976. A Complexity Measure, IEEE Transactions on Software Engineering, December, pp. 308- 320.
19. Watson, A.H. and McCabe, T.J., 1996. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-235.
20. Dubinsky, Y., Catarci, T., and Kimani, S., 2006. Active Data and the Digital Library Shell, The Joint Conference on Digital Libraries (JCDL), Workshop on Digital Libraries in the Context of Users' Broader Activities, Chapel Hill, NC, USA.