The TDD-Guide Training and Guidance Tool for Test-Driven Development

Oren Mishali[1], Yael Dubinsky[2], and Shmuel Katz[1]

[1] Computer Science Department
The Technion, Haifa, Israel
{omishali,katz}@cs.technion.ac.il
[2] IBM Haifa Research Lab
31905 Haifa, Israel
dubinsky@il.ibm.com

Abstract. A tool is presented for guiding Test-Driven Development (TDD), called TDD-Guide. The tool is integrated into an existing development environment and guides the developer *during* the development by providing notifications that encourage use of TDD. The TDD practice is defined through rules that can easily be changed and are used to generate code incorporated to a development environment using an aspect-based framework, so that the development of the tool has agile characteristics. Feedback from user experiments both validates the rules and suggests refinements to improve TDD-Guide, as is shown in descriptions of two user experiments.

Keywords: Rule-based framework, test driven development (TDD), software process support, user evaluation.

1 Introduction

Test-Driven Development (TDD) is widely considered both one of the central contributions of Extreme Programming to general agile techniques [1], and one of the most difficult practices to internalize [2, 3]. In this paper the TDD-Guide tool is shown both to effectively encourage use of test-driven development, and to allow incremental and flexible integration into an existing general development environment. The tool can detect conformance or deviation from test-driven practice as coding or testing steps are being developed, and provide valuable notifications to the developer. Some of the notifications provide the developer with positive feedback when the practice is followed, while others identify deviations from TDD. When deviations are detected, the tool can guide the developer to correct the deviation or even strictly enforce TDD by not allowing the developer to perform an operation deviating from the practice.

TDD-Guide is an application of the Aspect-Oriented Process Support (AOPS) framework. This framework, whose concepts were first introduced in [4], facilitates the definition and deployment of support for a variety of software processes in the form of rules and here the framework is used to define rules to support TDD. As its name suggests, the framework is based on aspect-oriented technology [5]. Using the

P. Abrahamsson et al. (Eds.): XP 2008, LNBIP 9, pp. 63–72, 2008.

support definition, code containing aspects and classes is automatically generated, ready to be integrated into the target development environment. Such integration guarantees the customization of the environment according to the defined rules. This aspect-based integration approach is used here on the Eclipse platform and thus the generated types are in AspectJ[1] and Java. The rules for TDD, code generated from this set of rules, a repository of key TDD events and their connection to the environment, together with a user-interface common to all framework products, all integrated into Eclipse, comprise the TDD-Guide tool.

The rules defined using the framework are simple to express, and it is relatively easy to add, remove or modify rules. The framework is especially appropriate for defining development practices that are flexible, may need frequent adjustment, and can be seamlessly integrated with an existing, familiar, development environment. This differs from previous Process Centered Engineering Environments (PCE's), such as [6, 7], that generally replace existing environments and are oriented to a fully detailed process model.

The current version of TDD-Guide is the result of ongoing research whose goal is to define practical and effective TDD rules. Given that goal and the flexible nature of the framework, we chose to define the rules in an agile fashion, starting from a basic and simple set of rules that is iteratively refined. In each iteration, the existing set of rules is tested on real developers and the gathered user feedback is used to refine the set toward the next iteration. In this paper, two such iterations are described, focusing on the experiments within them. In each experiment, student developers with novice TDD skills were given a Java development task, and were asked to develop the task using TDD. TDD-Guide was integrated in advance into the users' development environment (Eclipse), and significant development steps were logged. Based on the logs and questionnaires, we searched for and developed possible rule refinements. We were also interested in examining the reaction of the developers to this kind of on-line guidance.

We present results showing that TDD-Guide is in general perceived by the users to be helpful and that the tool indeed is effective in guiding TDD. More importantly, we show how the experiments provide important user feedback that helps both to improve the rules themselves and to refine the user-interface. In the next Section we present the user-interface and rules of TDD-Guide while explaining how rules are defined using the framework. The experiments' goals, description, and results are presented in Section 3, and conclusions and future directions are provided in Section 4.

2 TDD-Guide and the AOPS Framework

We metaphorically view a software development process as a trail defined by the process methods and practices; the developers are considered as hikers who are supposed to follow the trail but, for various reasons, once in a while deviate from it. Accordingly, an AOPS rule can be of kind *deviation* or *on-track*; a rule of kind on-track when triggered denotes that the developer is following the trail, and encourages the developer by providing positive feedback. Similarly, a rule of kind deviation is activated when the developer deviates from the desired trail; here, the rule may force the developer to return to the trail, or alternatively provide the developer with the choice to deviate while presenting negative feedback with different severity levels.

[1] The AspectJ Project, http://www.eclipse.org/aspectj/

2.1 TDD-Guide User-Interface

Upon activation of an AOPS rule, its message is presented to the developer in the AOPS view (Figure 1), where an appropriate icon denotes the type of the message. In addition, the AOPS bar (Figure 2) updates its color according the kind of the activated rule and also supplies a tooltip to quickly observe the rule's message.

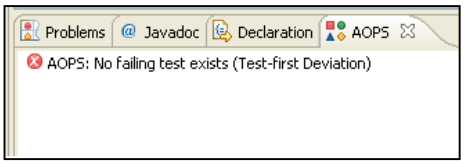

Fig. 1. The AOPS view

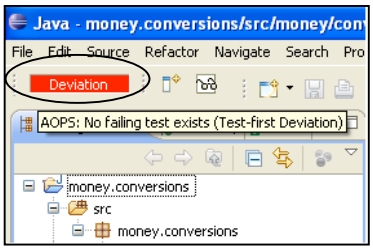

Fig. 2. The AOPS bar

Rule messages may also be presented to the developer within Eclipse dialogs and wizards. In Figure 3, for instance, we see the same "No failing test exists" message, but presented within the Java class creation wizard. This tight integration with Eclipse allows natural enforcement of the rule by simply disabling the 'Finish' button. However, in editing mode the same rule is not mandatory and can be overridden.

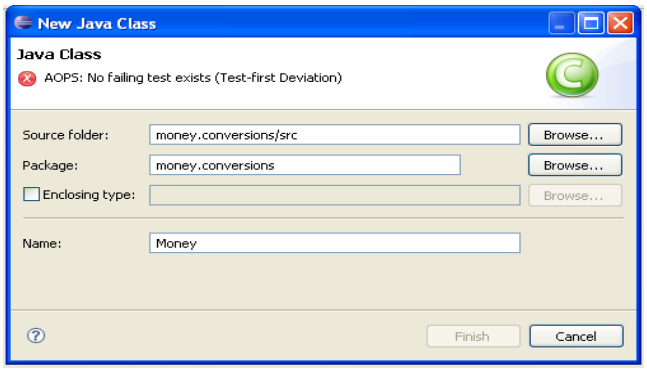

Fig. 3. Java class creation wizard augmented with an AOPS message

2.2 Rule Definition

To define AOPS rules, the manager/governor (the one who defines the rules) should first define an abstraction of the underlying development process, namely a set of entities that represent important elements in the process. Then, rules are defined that operate on the entities.

In Figure 4 we see some of the entities and the rules that constitute TDD-Guide. Two entities are defined, *CodingSpace* and *TestingSpace*, representing the space where the functional code is developed and where the unit tests are developed, respectively. Each entity can have *key-events* and *attributes*; key-events represent abstract events that occur during the development related to the modeled process element, and attributes are meant to hold related important values. Both entities have two key-events representing creation and modification of types in the space, and in addition, *TestingSpace* has two attributes *numOfFailingTests* and *numOfBrokenTests*; both attributes are of type Integer and hold the current number of failing tests and broken tests (that do not compile), respectively. An AOPS rule is activated when one of the key-events defined in its *condition* part is activated and its predicate, also defined there, holds. The rule *NeverWriteCodeWithoutFailingTest* is therefore activated when the developer creates a new type/class or modifies an existing one and neither a failing test nor a broken test exists. Activation of this rule is the most severe deviation from TDD and therefore a *strategy* of type error is defined (a strategy describes the general course of action taken upon rule activation). The second rule *ChallengeExistingCode* is of kind *on-track* and is activated when the developer is modifying a test and no failing test exists. The rule encourages the developer by clarifying the task ahead: writing a test that is not passed by existing code.

Four more rules are defined. Two of them enforce coding standards that distinguish between coding and testing elements, and another one *HaveOneActiveTest* recognizes deviations from the TDD recommended guideline of not trying to fix several things at a time[2]. The last rule *MakeExistingCodePass* encourages the developer to fix the code when in the coding space and having one failing test.

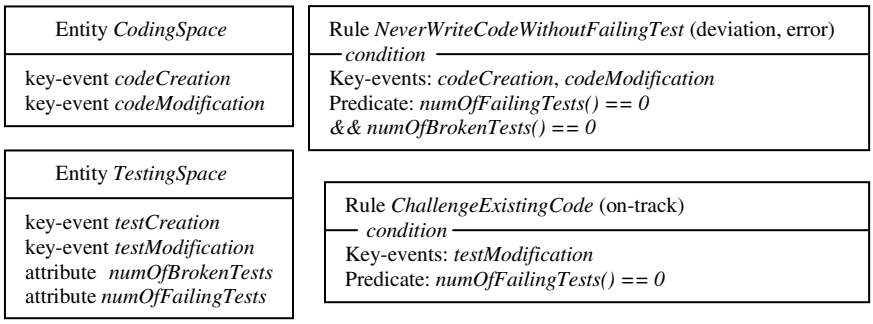

Fig. 4. Sample TDD entities and rules

The defined entities are just declarations and thus should be connected to the underlying development environment. This process of connecting the entities to the environment is called entity-mapping and uses a repository of concrete Eclipse method calls not elaborated here. After the mapping, during development of an application the entities are continuously updated to reflect the state and behavior of the underlying process elements that they represent. The entities, their mapping, and the rules, are all defined using the framework's graphical interface. A public release of

[2] http://c2.com/cgi/wiki?OneUnitTestAtaTime

the AOPS framework is expected within several months. An Eclipse plug-in of TDD-Guide is available upon request from the authors.

3 Evaluating TDD-Guide

A support tool to guide TDD can be best validated through user experiments. The validation should involve user experience that is rigorously planned and executed aiming at refining the tool to be more effective [8, 9]. The users in our case are developers in software development teams who work to produce code according to some predefined functionality and need to produce unit tests to support the code.

In this section we describe the two user experiments of the first two development iterations of TDD-Guide. The first experiment was a spike whose major purpose was to examine the initial set of the rules of the first iteration in a real development setup. Based on this spike, several changes were introduced to the tool. The second version of the tool was experimented with in a larger setting, more focused on rule refinement, namely examining the effectiveness of the rules in supporting TDD, and searching for unanticipated development states.

3.1 First Experiment

Six experienced programmers familiar with Java and Eclipse, and less familiar with TDD, were given a simple development task and asked to develop the task while using TDD. The initial feedback was encouraging: all of the participants showed positive reactions to the idea of accompanying the development with messages and alerts, and four participants reported that the messages helped them to develop test-first. No change was noticed in Eclipse performance due to the addition of aspects into it.

The experiment led to changes in the user-interface, in the rules, and in the logging. Four participants reported that paying attention to the AOPS view did not disrupt their concentration in developing the task. The other two felt that it sometimes was a burden. Accordingly, we decided to add the AOPS bar (Figure 2), hoping that colored feedback would be more intuitive than a purely textual one. The rule *NeverWriteCodeWithoutFailingTest* in its first version attempted to treat a special case: when the developer had a broken test, the rule allowed moving to the code without requiring to execute the test (assuming that the developer is interested in creating, e.g., a missing declaration and then returning to the test). However, we observed that two participants did not act according to our assumption. They indeed created a missing declaration, but instead of then returning to the test they continued to develop the code without first running JUnit. We changed the rule so that test execution is required before each move to the code, to make the TDD cycle simpler and more uniform. Since execution of broken tests is also reported by JUnit with red indication, we also added a warning to the rule to remind the developer that the JUnit bar is red due to a compilation error and not because of a failing test. We also added time-stamps to the logs to facilitate better reasoning.

3.2 Second Experiment

The participants in the second experiment were 34 CS-major fourth-year students in an advanced Software Engineering project course. The experiment had three phases:

1. Pre-experiment phase in which participants filled in a questionnaire about the level of their familiarity and experience with programming concepts and tools in general and with TDD in particular. Then, they heard a one-hour lecture about unit testing and specifically about TDD.
2. Experiment experience phase in which participants moved to the computer lab where they were guided in groups of 2-4 to perform a specific programming task. After completing the task they filled in a personal reflection.
3. Post-experiment phase in which participants were asked one week after the experiment to indicate two features of TDD-Guide that they perceive as most significant and two possible improvements or extensions. This feedback was obtained using a web-based feedback mechanism familiar to the students.

Twenty seven of the participants filled in the questionnaire of the pre-experiment phase. The results show that participants felt knowledgeable with Java programming and object-oriented design, less knowledgeable with Eclipse IDE and unit testing, and beginners in JUnit and TDD. Regarding the development process, participants were less experienced with measuring the development process and product, but felt knowledgeable and even expert with working in pairs.

Given a project named `money.conversions` that contains classes and conversion utilities[3], participants were asked to define a class `Money` that represents a certain amount of money in a specific currency. In addition, the class should have the method `Money add(Money m, String currency)` where the returned `Money` represents the addition of the called `Money` object and the given `Money` argument, in the given `currency` argument. Participants were asked to develop according to the TDD technique (within 35 minutes) and to take notice of AOPS messages. As in the first experiment, aspects were also added to Eclipse to log actual behavior and timing information.

3.2.1 Experiment Outcomes

We illustrate the experiment findings for the *NeverWriteCodeWithoutFailingTest* rule of TDD-Guide that detects a deviation as aforementioned. We considered recurrences of series of events in the logs that show a specific behavior of the developers either before or after the deviation. The logs of twelve groups that completed their task were considered and the following four findings were formulated:

- The first finding deals with the intuitive tendency of developers to start programming with coding rather than with testing. An expected behavior in the beginning of the log is *Test - TestFailed - Code* where *Test* means writing test lines, *TestFailed* means that running JUnit causes a failure, and *Code* means writing code lines. Four logs out of twelve include *Code - DeviationMessage - Test* at the beginning of the log (meaning that they start directly with code as they used to, noticed the AOPS deviation

[3] The given task is a simplified version of a well-known TDD example by Kent Beck and Erich Gamma (http://junit.sourceforge.net/doc/testinfected/testing.htm).

message that appears and responded by starting to test). This tendency to start with coding was also found in the middle of the task when instead of the expected *Code - TestSucceeded - Test* we found *Code - TestSucceeded - Code - DeviationMessage - Test.* These cases show that novice developers can benefit from TDD-Guide messages and by following them they overcome their tendency to start coding without testing, and thus adhere to the TDD practice. Since this rule can be overruled in *edit* mode, we found two cases of *Code - DeviationMessage - Code* meaning the deviation message was ignored by the developers who continued to work on the code although there was no failed test. This can be also explained as a refactoring activity and was marked by us for further investigation.

- The second finding concerns getting used to actually run the tests before moving to code. We found eight cases in six logs where developers did *Test - Code - DeviationMessage - TestFailed - Code* meaning they worked on the test and switched to code without receiving the feedback of running JUnit. Following the deviation message they ran the test, causing a test failure, and went back to code.
- The third finding relates to the learning curve that can be observed especially when adding the time measure of the different activities. The following series of events was found starting at the beginning of a specific log:
    - *Code - DeviationMessage* for 1 minute; no work for 2 minutes;
    - *Test* for 15 seconds;
    - *Code - DeviationMessage* for 4 seconds; no work for 7 seconds;
    - *Test* for 8 minutes;
    - *Code - DeviationMessage* for 1 second;
    - *TestFailed - Code*

  The deviation message was used three times to correct the development in this trace. We observed here and elsewhere that the time to respond to the deviation messages decreased while the time invested in testing increased.
- The fourth finding reveals strong emotions against testing and can be seen as anecdotal: one group used “i dont want to test” as part of their test file name.

3.2.2 Participants’ Reflection on the Experiment

After completing the task, participants filled in their level of agreement with statements related to the experiment. Table 1 summarizes their answers; a clear majority is marked in grey. As can be observed most participants felt that the Eclipse IDE works as usual (statement 1) and that TDD-Guide helped them in working according to the TDD technique (e.g., statement 6). However, statements for which no clear majority exists reveal issues that may suggest rule refinement. For instance, statements 2 and 4 reveal usability issues, and statements 8 and 13 disagreement with the TDD guiding rules (we refer to these issues in Section 4). Statements 7 and 16 uncover resistance to the TDD concept. We believe this only emphasizes the necessity of the guidance, in particular for novices who are not yet familiar with the advantages of TDD.

To assess the longer-term impact of this experience, we asked for feedback one week after the experiment. As noted, participants were asked to indicate two features of TDD-Guide that they perceived as most significant and two possible improvements or extensions to the tool. Thirty two participants responded to this phase.

Table 1. Reflecting on the experiment activity

#	Statement	Agree	Tend to agree	Tend to disagree	Dis-agree	No answer
1	During development, I felt that the Eclipse interface responded as usual	9	15	8	2	
2	Paying attention to the AOPS messages was a burden	3	14	14	3	
3	I have hardly had any AOPS Deviation (Error) messages	1	12	11	8	2
4	Some of the AOPS messages were not comprehensible	5	12	11	5	1
5	Sometimes I didn't agree with what an AOPS message was saying	2	4	15	12	1
6	The AOPS messages helped me to develop test-first	11	13	8	1	1
7	I find test-first an annoying technique	5	14	13	2	
8	Sometimes, I just ignored an AOPS message	9	9	8	8	
9	Sometimes, I felt that an AOPS message was needed but it didn't show up	3	5	18	8	
10	I think that accompanying the development with messages and alerts is not a good idea and just interferes with the fluent work	1	8	17	8	
11	Several times, AOPS messages led to a change in my behavior	4	16	10	4	
12	I looked several times at the reference page to figure out how to develop test-first	3	6	10	15	
13	When a failing test does not exist, the AOPS system should always disallow any coding	5	13	13	3	
14	I got several "false alarms" (incorrect AOPS messages)	2	3	14	14	1
15	The AOPS view was more useful than the AOPS bar	2	18	11	3	
16	I will definitely develop test-first in the future	2	15	10	6	1
17	Most of the AOPS Warning messages were justified	6	24	5	-	

Most of them indicate the main tool features, though some mixed the TDD technique itself with the features of the guiding tool. Following are some of their suggestions for improvements: "A feature can be added to mark code that is already covered by tests thus help with the testing management"; "Better indication of the current stage in the development process, sometimes it was difficult to understand what the environment expects us to do"; "Introduce development tasks into the environment in order to enable the planning of the product roadmap according to the list of tests that should be written"; "An error should not always be created in order to go forward, there can be an option to skip the obvious errors in the beginning or at least to mark

them for example as 'preliminary development remarks'"; "In my opinion no significant improvements/extensions are needed"; "I suggest to emphasize the status marker"; "Possible extension is an automatic correction offer when a problem is diagnosed"; "Add voice alert when there is a warning".

4 Conclusion and Future Work

We conclude this paper by describing the implications of the outcomes presented on the TDD-Guide rules, the user-interface, and the log used to gather information.

As previously mentioned (Section 3.1), after the first experiment we added a warning to the rule *NeverWriteCodeWithoutFailingTest* that is activated when the developers write code while having only broken tests. Its purpose was to remind them that the JUnit bar is red due to a compilation error and not because of a failing test. The logs show that although the warning was presented, usually the developers did not execute the test again after the missing declarations were created but continued to code in that state, without knowing for sure that the test fails. One possible remedy could be to activate the warning again after some time. We should also examine whether the warning message is clear enough.

The addition of time-stamps to the logs discovered that a significant aspect of a correct TDD trail is related to time. For instance, we found several cases where tests were developed (for the first time) for more than ten minutes before moving to the code, and cases where the first successful test execution took place only after fifteen minutes, both indicating that the initial TDD steps are too complex. A new story defined for the third development iteration of TDD-Guide is to provide timing alerts, e.g., if the developer stays too long in the testing space.

Although Section 3.2.1 discussed the *NeverWriteCodeWithoutFailingTest* rule, of course the other rules were examined and guidelines for their refinements exist. The rule *HaveOneActiveTest* was defined to be activated when the developer is in the coding space and has more than one failing test. However, one log revealed that coding while having several failing tests is not always a deviation; that may happen when coding indeed starts with one failing test however changes made in the code cause the failure of others. Although it may indicate tests that are not reasonably independent, it should not be classified as a deviation. Thus, the first improvement is to distinguish between that case and the case where the deviation is certain, that is, where coding *starts* with several failing tests. The logs report on three occurrences of the latter and show that the rule's notification was ignored, i.e., the developers continued to code. The reason may be that the guidance was applied in retrospect, i.e., when the developers already had the tests written. The lesson learned here is that a deviation should be reported as early as possible, when its correction is practical.

As noted in Section 3.2.1, the pattern *Code - DeviationMessage – Code* could also be a sign for a refactoring activity and in that case the TDD rules should not report a deviation. Therefore, another new story for the third iteration is to define refactoring as a new state where modified TDD rules apply.

There is a need to emphasize the user interface indications (see statement 2 in Table 1 and the last feedback in Section 3.2.2). In the next iteration we plan to add vocal indications. Another point to consider is the use of interactive communication with

the developer, e.g., pop-ups asking for real-time developer feedback or a button whose pressing indicates moving to a refactoring state.

The performance logs were the primary aid for reasoning on the development and the effectiveness of TDD-Guide. During their analysis, we noted that different views of the logged data are needed, e.g., to identify recurrent patterns and to analyze all activations of an individual rule. These views were created manually and we plan to add their automatic creation. We are also considering the use of a relational database that will store the data and allow queries and reports.

As confirmed by the experiments, after two iterations TDD-Guide is already an effective tool for guiding test-driven development. Its flexibility and light-weight integration into the Eclipse IDE, provided by the AOPS framework, increases the potential of widespread adoption for this tool and its extensions to additional agile software processes.

References

1. Beck, K.: Test-Driven Development By Example. Addison-Wesley, Reading (2003)
2. Dubinsky, Y., Hazzan, O.: Measured Test-Driven Development: Using Measures to Monitor and Control the Unit Development. Journal of Computer Science, Science Publication 3, 335–344 (2007)
3. George, B., Williams, L.A.: A structured experiment of test-driven development. Information & Software Technology 46, 337–342 (2004)
4. Mishali, O., Katz, S.: Using aspects to support the software process: XP over Eclipse. In: International Conference on Aspect-Oriented Software Development, pp. 169–179. ACM, Bonn, Germany (2006)
5. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming, pp. 220–242. Springer, Heidelberg (1997)
6. Bandinelli, S., Braga, M., Fuggetta, A., Lavazza, L.: The Architecture of SPADE-1-Process-Centered SEE. In: Third European Workshop on Software Process Technology, pp. 15–30. Springer, Heidelberg (1994)
7. Junkermann, G., Peuschel, B., Schafer, W., Wolf, S.: MERLIN: Supporting Cooperation in Software Development Through a Knowledge Based Environment. In: Software Process Modelling and Technology, pp. 103–129. John Wiley and Sons, Chichester (1994)
8. Dix, A., Finlay, J., Abowd, G.D., Beale, R.: Human-Computer-Interaction, 3rd edn. Prentice Hall, Englewood Cliffs (2003)
9. Sharp, H., Rogers, Y., Preece, J.: Interaction Design: Beyond Human-Computer Interaction, 2nd edn. John Wiley & Sons, Chichester (2007)