Teaching and Training Developer-Testing Techniques and Tool Support

Tao Xie[1] Jonathan de Halleux[2] Nikolai Tillmann[2] Wolfram Schulte[2]

[1]North Carolina State University, [2]Microsoft Research

[1]xie@csc.ncsu.edu, [2]{jhalleux,nikolait,schulte}@microsoft.com

Abstract

Developer testing is a type of testing where developers test their code as they write it, as opposed to testing done by a separate quality assurance organization. Developer testing has been widely recognized as an important and valuable means of improving software reliability, as it exposes faults early in the software development life cycle. Effectively conducting developer testing requires both effective tool support by tools and developer-testing skills by developers. In this paper, we describe our experiences and lessons learned in teaching and training developer-testing techniques and tool support in both university and industrial settings. We highlight differences in teaching and training in these two settings, and observations from interacting with practitioners in our process of teaching and training.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic execution, Testing tools

General Terms Reliability, Verification

Keywords Testing, unit testing, parameterized unit testing, theories, symbolic execution, mock objects, Pex

1. Introduction

Developer testing, often in the form of unit testing, has been widely recognized as a valuable means of improving software reliability. In developer testing, developers test their code as they write it, as opposed to testing done by a separate quality assurance organization. The benefits of developer testing are two folds: (1) gain high confidence in the program unit under test (e.g., a class) while developers are writing it and (2) reduce fault-fixing cost by detecting faults early when they are freshly introduced in the program unit.

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0240-1/10/10. . . $10.00

The popularity and benefits of developer testing have been well witnessed in the industry [24]; however, manual developer testing is known to be labor intensive. In addition, manual testing is often insufficient in comprehensively exercising behaviors of the program unit under test to expose its hidden faults. To address the issue, one of the common ways is to use testing tools to automate activities in developer testing. Developer-testing activities typically include generating test inputs, creating expected outputs, running test inputs, and verifying actual outputs. Developers can use existing testing frameworks such as NUnit [4] for .NET and JUnit [1] for Java to write unit-test inputs and their expected outputs. Then these frameworks can automate running test inputs and verifying actual outputs against the expected outputs.

To reduce the burden of manually creating test inputs, developers can use test-generation tools to generate test inputs automatically. Although great research advances have been made in automatic test generation, it is still a long way towards satisfactorily accomplishing effective test generation when testing common real-world code bases. Then when using test-generation tools, developers need to have the skills to understand the challenges that these tools face and provide guidance to the tools in attempting to address these challenges.

After test inputs are generated automatically, expected outputs for these test inputs are still missing. Developers could choose to write no explicit expected outputs but rely on uncaught exceptions or crashes to focus on robustness checking. To check functional correctness, developers need to write assertions within the test code or the code under test for asserting the expected program behaviors. Most modern (testing) frameworks provide some kind of `Debug.Assert(bool)` methods for developers to use. However, in practice, these assertions are usually written in an ad-hoc way, and less senior or experienced developers have no clear idea on where and for what purpose to write assertions.

To address the issue, two major approaches have been proposed. First, developers could write assertions to encode specifications such as design by contract [18] (a form of axiomatic specifications [13]) with tool support such as Code

Contracts [2] for the code under test to check program behaviors. Second, developers write assertions in Parameterized Unit Tests (PUTs) [22], which are unit tests with parameters; these assertions often encode expected behaviors in the form of algebraic specifications [12]. In these two approaches of writing assertions, developers need to have good skills to write specifications to capture expected behaviors for the code under test.

As educators, we need to devise effective ways to teach students or practitioners to equip them with these preceding skills. In this paper, we describe our experiences in teaching and training developer-testing techniques and tool support in both university and industrial settings. We highlight differences in teaching and training in these two settings, and observations from interacting with practitioners in our process of teaching and training.

The rest of the paper is organized as follows. Section 2 presents the key developer-testing techniques and tool support covered in our teaching and training materials. Section 3 describes our teaching and training experiences in university settings. Section 3 describes our teaching and training experiences in industrial settings, including observations from interacting with practitioners in our process of teaching and training. Section 5 compares differences in the university and industrial settings for teaching and training. Section 6 discusses related issues. Section 7 concludes the paper.

2. Background

Parameterized unit testing [22] is a new methodology extending the current industry practice based on closed, traditional unit tests (i.e., test methods without input parameters). Test methods are generalized by allowing parameters to form Parameterized Unit Tests (PUTs). Below is an example PUT for testing one behavior of `List`'s `Add` method. In the PUT, the `Assume.IsTrue` method specifies an assumption: any test inputs violating the assumption are filtered out during test generation or execution; the `Assert.AreEqual` method specifies an assertion.

```
  void TestAdd(List list, int item) {
   Assume.IsTrue(list != null);
   var count = list.Count;
   list.Add(item);
   Assert.AreEqual(count + 1, list.Count);
 }
```

This generalization to form PUTs serves two main purposes. First, PUTs are specifications of the behaviors of the methods under test: they not only provide exemplary arguments to the methods under test, but ranges of such arguments. Second, PUTs describe a set of traditional unit tests that can be obtained by instantiating the parameterized test methods with given argument values. Instantiations via argument values should be chosen so that they exercise different code paths of the methods under test. Most unit testing framework have been extended to support parameterized unit testing, provided that relevant argument values are specified by the user.

Dynamic Symbolic Execution (DSE) [11] (also called concolic testing [20]) is a recent technique to automatically supply such argument values. DSE combines static and dynamic analysis to automatically generate test inputs, e.g., argument values. Given a program that takes inputs, the goal of DSE is to generate test inputs that, upon execution of the program, will exercise as many reachable statements as possible. DSE is based on observing actual executions of the program under test. By leveraging observed concrete input/output values, DSE can simply concretize those operations that interact with the environment, or that are difficult to reason about (e.g., floating-point arithmetic), while previous approaches based on symbolic execution [15] would lose precision. Various implementations of DSE exist, ranging from academic open-source projects to industrial tools. Our teaching and training use the Pex tool [3, 21] from Microsoft Research, which tests .NET programs such as C# programs.

Environment isolation is conducted to test individual software components in isolation when they interact with environments. It makes testing more robust and scalable. Especially in the context of unit testing, where the intention is to test a single unit of functionality, all irrelevant environment dependencies should be mocked [17, 23], or simulated, so that the unit tests run quickly and give deterministic results. In contrast, the goal of integration testing is to test an integrated (sub)system, including all environment dependencies, at the same time.

Ideally, the code under unit testing should be written in a way that allows to substitute its constituent components at testing time, in order to isolate a feature under test. In other words, it should be possible to treat all components as test parameters, so that mocked implementations or simulations can be used to instantiate parameterized tests. One solution to the problem is to refactor the code [10], introducing explicit interface boundaries and allowing different interface implementations. When refactoring is not an option, e.g., when dealing with legacy code, other approaches can be used to detour environment-facing calls at testing time. Various tools exist to enable automatic code isolation, ranging from academic open-source projects to industrial tools. One such tool is Moles [8] from Microsoft Research.

3. Teaching and Training in University Settings

We next present an overview of our teaching and training experiences in university settings and then discuss our lessons learned from our experiences.

3.1 Overview

The teaching and training of developer-testing techniques and tool support were conducted by the first author in a graduate software testing course (CSC 712) at North Carolina State University for the 2008 Fall semester (20 students) and the 2009 Fall semester (18 students). The course schedule, homework and project assignments, lecture slides, and reading materials for these two semesters can be found here[1]. We next describe key teaching materials covered during the 2009 Fall semester (which are mostly similar to the ones covered during the 2008 Fall semester).

Lectures. There were two 75-minute lectures (Mondays and Wednesdays) each week for the 16-week semester. The lectures were given in a lab where every two students shared the same desktop with two monitors. We designed the lectures on Mondays to be mostly on testing foundations, particularly coverage criteria, based on selected materials and slides from the textbook "Introduction to Software Testing" by Ammann and Offutt [6]. We designed the lectures on Wednesdays to be mostly on testing techniques and tools including Pex for 10 weeks, Code Contracts [2] for 2 weeks, and (only briefly) NModel [14] (a model-based testing tool for C#) for 1 week. The instructor (the first author) gave both slide presentation (being more heavily used in Monday lectures) and live tool demonstration (being more heavily used in Wednesday lectures) during lecturing.

Quizzes and homework assignments. We designed four quizzes taken by students throughout the semester to assess students' mastery of lecture topics on testing foundations. To assess students' mastery of lecture topics on testing techniques and tools as well as testing foundations, we designed four homework assignments. Homework 1 included student surveys and their personal homepages. Homework 2 included exercises for familiarizing students with code development and testing in C#. Homework 3 included students' submission of candidate open source code to be tested in the term project. Homework 4 included exercises on applying the instructed test generalization techniques [16].

Term project. Based on the list of preferred teammate candidates that each student submitted together with their Homework 2, the teaching staff (i.e., the instructor and the teaching assistant) assigned students into teams, each of which included two students. As described earlier, in Homework 3, each formed team was asked to submit candidate open source code to be tested in the term project. In midsemester, each team was asked to submit a midterm project report describing the team's experience following the guidelines described in the sample paper skeleton[2] distributed to the students. Basically, each team was asked to write PUTs by performing test generalization on existing traditional unit tests for the chosen open source code under test and document their experiences. In the second half of the semester, each team was asked to write additional new PUTs to augment the PUTs written for the midterm report to achieve higher block coverage and possibly higher fault-detection capability, and document their additional experiences by expanding their midterm report to produce their final report.

3.2 Lessons Learned

We next describe some observations and lessons learned during our teaching and training in the university settings.

Integration of teaching testing foundations and testing techniques/tools was desirable for a testing course but could be challenging. In the design of our lecture topics, we arranged Mondays' lecture topics to be on coverage criteria and Wednesdays' lecture topics on testing techniques and tools. One potential risk of such design was that these two types of lecture topics might be difficult to be well integrated and students could have perception that the two types of lecture topics were too separated and isolated. We intended to alleviate the issue by demonstrating how achieving specific logic coverage criteria could be formulated as problems of achieving branch or path coverage with Pex [19]. However, such ways of using practical tools to demonstrate tool-assisted satisfaction of coverage criteria are limited to only logic coverage so far. More recently, we extended Pex to support mutant killing for mutation testing [26] and we plan to incorporate this new extension in our future offerings of the course to demonstrate tool-assisted satisfaction of mutation killing.

Another direction for the integration of teaching coverage criteria and practical tools could be to use and demonstrate coverage measurement tools when lecturing topics of coverage criteria. There exist a number of industrial-strength tools for measuring and reporting statement/block or branch coverage; however, they generally lack measurement tools for other more advanced types of coverage criteria such as dataflow coverage.

In addition, more thoughts and work would be needed to investigate how to weave in coverage criteria or more generally testing foundations when lecturing topics of testing techniques and tools. We already briefly introduced some basic background on constraint solving and theorem proving when lecturing the Pex tool and its techniques. Some technique and tool topics such as writing PUTs fed to Pex and writing code contracts fed to Pex and Code Contracts have strong formal foundations of algebraic specifications [12] and axiomatic specifications [13], respectively. The lecture topics of writing specifications were not explicitly listed in the testing textbook or its accompanying slides that we used [6] and thus not included in Mondays' lecture topics. However, we plan to collect teaching materials on writing algebraic specifications and axiomatic specifications, and include them in Mondays' lecture topics in our future offerings of the course.

A term project on testing realistic open source code could give students opportunities to gain testing experiences close to the real world but such a term project also had lim-

[1] `http://research.csc.ncsu.edu/ase/courses/csc712/`

[2] `http://research.csc.ncsu.edu/ase/courses/csc712/2009fall/wrap/project/generalization/testgeneralization.pdf`

itations on training *real* developer testing, where developers test their code as they write it. Open source code is abundant for students to choose and test. However, few open source projects are well documented or equipped with sufficient information for students to understand the full scale and details of expected behaviors of the code under test. Therefore, students could face significant challenges in writing down high-quality assertions for their newly written PUTs for the open source code under test. To alleviate the issue, we designed the term project to heavily focus on *test generalization* [16], where students tried to understand and recover the intended behaviors tested by traditional unit tests written by the open source code developers, and generalize these traditional unit tests to be PUTs. Such a procedure allowed students to gain not only program/test understanding skills but also generalization/abstraction skills. The last part of the term project was on writing new PUTs to achieve higher code coverage and likely higher fault-detection capability, requiring students to write new PUTs, without relying on or referring to existing traditional unit tests.

Our design of the term project allowed the students to heavily invest their course efforts on testing instead of writing production code (which they supposedly already learned from past programming and software engineering courses). The term project in fact simulated situations where third-party developers tested code not written by themselves, strictly speaking, not falling into the activities of developer testing. An alternative type of term projects could be to ask students to develop some new features of a software project while testing their newly implemented features (where students could possibly be requested to practice test driven development [7]). However, in this way, students would spend significant time on feature implementation (and thus less time on feature testing). In addition, it could be difficult to find an appropriate open source code base (e.g., not too complicated but realistic enough) to use in the term project.

We allowed students to search and choose open source code (to be tested in their term project) that satisfied the specified characteristics (e.g., equipped with traditional unit tests), rather than designating the same open source code across all the student teams in the class. Advantages of doing so included that different student teams could encounter different interesting observations and lessons learned by testing different open source code bases, and later sharing their different experiences with the whole class via final project presentations could be more beneficial for other students outside of their team. Disadvantages of doing so included that some student teams might choose open source code that might be inherently not amenable to applying Pex. For example, during the course offering of the 2008 Fall semester, a student team chose Math.NET[3], a mathematical library for symbolic algebraic and numerical/scientific computations, and in later phases of the term project, the team found out that Pex could not be effectively applied on it because the library implementation involves intensive floating-point computation, which is currently not well supported by Pex's underlying constraint solver [9]. To alleviate the issue, we did request student teams to conduct an early try-out of Pex on part of the open source code under consideration to reduce the risk. In addition, we allowed a student team to change their open source code under test over the duration of the semester without imposing penalty grade points. Furthermore, the term project was designed to have multiple milestones and the submission of a later milestone was built upon the submissions from previous milestones so that students could incorporate feedback from the teaching staff on previous submissions to improve their later submissions.

Using industrial-strength tools and technologies not only reduced the "*debugging*" overhead imposed on both the students and teaching staff, but also gave students experiences that they could immediately benefit from when they took on their industrial jobs. In the homework and project assignments, we deliberately used industrial-strength tools, rather than academic research prototypes. Academic research prototypes often lack support for dealing with various types of code features frequently included in real-world code bases. Furthermore, these prototypes might often include faults and the prototype developers might be often too busy to provide timely technical support or fixing of reported faults. For our students asking questions via the Pex MSDN forums[4], the Pex developers (the second and third authors of this paper) provided timely technical support in using Pex, substantially reducing the support effort from the teaching staff.

Incorporating the training of research skills in the term project benefited students in their future research career as well as software development career. The major deliverables of the term project used for grading included the midterm project report and the final project report. To train students in technical writing, we gave a lecture on common technical writing issues[5]. To reduce barriers for students who were new to writing technical papers, we provided a detailed paper template, which describes the desired structure of the paper including what sections should be included and what contents should go to each section. In the second half of the 2008 Fall semester, we also distributed a sample midterm report that was the best among the student submissions and whose distribution permission was given by the authoring team. In the 2009 Fall semester, both a sample midterm report and final report were distributed to the class at the beginning of the semester. These mechanisms allowed students to learn from good example writing, reducing barriers for them to prepare their own reports. Such a term project including the research-oriented empirical study and its technical writ-

[3] `http://www.mathdotnet.com/`

[4] `http://social.msdn.microsoft.com/Forums/en-US/pex/threads/`

[5] `http://people.engr.ncsu.edu/txie/advice/`

ing also gave students first-hand experience on conducting empirical studies or empirical evaluations.

4. Teaching and Training in Industrial Settings

We next present an overview of our teaching and training experiences in industrial settings and then discuss our observations from interacting with practitioners and our lessons learned from the process of teaching and training.

4.1 Overview

The teaching and training of developer-testing techniques and tool support were conducted by the second and third authors in the form of one-day or half-day tutorials both within Microsoft (such as internal training of Microsoft developers) and outside Microsoft (such as invited tutorials at .NET user groups). The attendees of a tutorial could range approximately from 10 to 25 practitioners. For a tutorial outside Microsoft (normally with half-day duration), sometimes attendees might not have already installed Pex on their laptops while attending the tutorial, and therefore, the tutorial presentation was primarily the combination of slide presentation and live demonstration of Pex and Moles. However, for a tutorial within Microsoft (normally with full-day duration), the tutorial was given in a training lab at Microsoft, where each attendee was able to use a lab desktop computer installed with Pex and Moles. In this setting, the tutorial involved frequent hands-on exercises conducted by attendees, besides slide presentation and live demonstration of Pex and Moles. The tutorial slides on Pex and Moles can be found at the slide deck section of the Pex documentation web[6].

4.2 Observations

We next describe some observations while interacting with practitioners during our teaching and training, and other general occasions in promoting technology and tool adoption. These observations provide insights not only for teaching and training but also for design or improvement of testing techniques and tools. We illustrate our findings with conversations between developers and trainers; while these conversations are anecdotal in nature, they show quite typical developer mind sets that have been observed by the trainers frequently[7].

Assertion deficit syndrome conversations occurred between a developer and a trainer (i.e., one of the second and third authors) as below:

- Developer: “Your tool only finds null references.”
- Trainer: “Do you have any assertions?”
- Developer: “Assertion???”

[6] `http://research.microsoft.com/pex/documentation.aspx`

[7] While there are a non-trivial number of practitioners in industry that would match the profiles described in this section, there are far more practitioners with great interest and passion in learning techniques that could improve the effectiveness and quality of their work.

When a developer is equipped with a test-generation tool such as Pex, the developer is often attempted to click a button provided by the tool to run the tool to generate a large number of test inputs to test the code under test, and then wait for the testing results, which include the test failures reported by the tool. Without assertions written by the developer, either in the test code of PUTs or in the code under test as contracts, the test failures would be limited to uncaught exceptions or crashes. Developers need to be aware of what a test-generation tool could offer if no assertions are written to capture intended behaviors of the code under test. As a consequence, we suggest that training should emphasize the importance of assertions, including quizzes to study beneficial assertion patterns.

Hidden complexity of the code under test is often not realized by developers. Code similar to the following was actually brought to the attention of the trainers:

```
void Sum(int[] numbers) {
     string sum = "0";
     foreach(int number in numbers) {
       sum = (int.Parse(sum) + number).ToString()
     }
     if (sum == "123")
         throw new BugException();
}
```

The API method invocations of `int.Parse` and `int.ToString` could incur challenges for a test-generation tool that analyzes and explores code, since these API implementations could be very complicated, incurring hidden complexity for the tool. More and more convenient framework and library APIs (whose implementations hidden from API-client-code developers could be quite complicated though) are available and popularly used by developers. The hidden complexity of these invoked framework or library APIs causes a test-generation tool to take long to explore; even worse, when these framework or library API implementations are in native code (other than managed code in .NET or Java), a tool that analyzes and explores only managed code could not explore these API implementations. As a consequence, a portion of training should be devoted to the issue of hidden complexity, teaching how to interpret the tool feedback to identify such cases.

Unit testing utopia and **Test Driven Development (TDD) [7] dogma** was deeply established among some developers. A developer, being a unit testing enthusiast, stated that “I do not need test generation; I already practice unit testing (and/or TDD)”. A developer, being a TDD convert, stated that “Test generation does not fit into the TDD process”. It is not easy to change the philosophy of these developers. It should be emphasized during teaching and training that using a test-generation tool can complement manually writing traditional unit tests (without parameters) since manual generation of test inputs could be limited, missing important corner or extreme inputs, due to the inherent limitation of human-brain power. A longer-term ideal situation could be that developers write PUTs instead of traditional unit tests; if needed, developers could manually write test inputs to the written PUTs besides those automatically generated test inputs for the PUTs.

Writing PUTs and applying a test-generation tool could also be integrated into the TDD process. Developers could go through the iterations of (1) writing PUTs before writing the code implementation under test, (2) applying a test-generation tool to generate test inputs for the PUTs and inspecting the reported test failures, and (3) writing the code implementation under test to a just-enough extent to make the test failures disappear. One key difference between this new TDD process and the traditional TDD process is that, in contrast to written traditional unit tests, written PUTs of higher quality would often be much more difficult to "fool" with a naive code implementation under test. As a result, developers could spend more effort in writing code implementation under test and spend less effort in incrementally improving the quality of the test code. It remains an open question for future empirical studies whether such "bigger-jump" iterations of improving code implementation under test would compromise the originally acclaimed benefits of "more modularized, flexible, and extensible code" [7]. These benefits are supposedly provided through "taking small steps when required", but the new TDD process would incur larger steps than the traditional TDD process.

Interacting with generated tests triggered quite some questions from developers. First of all, a developer might not know what to do after tests are automatically generated. For example, a developer in front of 100 generated tests asked "What do we do with the generated tests?" It is important to teach developers on how to interact with the generated tests, e.g., inspecting the reported test failures, inspecting the coverage reports to understand the insufficiency of the generated test inputs (and/or PUTs if written), diagnosing causes for the insufficiency, and providing guidance to the tool to address the insufficiency.

Below are conversations between a developer and a trainer on desired naming of generated tests by a tool:

- Developer: "Your tool generated a test called Foo001. I don't like it."
- Trainer: "What did you expect?"
- Developer: "Foo_Should_Fail_When_The_Bar_Is_Negative."

When developers write traditional test methods manually, they use meaningful naming conventions to these test methods. It is natural for developers to expect to see meaningful naming for generated test methods. Note that if developers write PUTs, they have control on the naming of the PUTs and the developers would need to pay less attention to the naming of the generated traditional unit tests that invoke the PUTs. But if developers write no PUTs but rely on a tool to generate test inputs for robustness checking, e.g., throwing uncaught exceptions, the developers would pay attention to the naming of the traditional unit tests generated by the tool. In such cases, tool builders could improve the naming of the generated traditional unit tests.

Below are conversations between a developer and a trainer on desired representative values for generated test inputs by a tool:

- Developer: "Your tool generated "\0""
- Trainer: "What did you expect?"
- Developer: "Marc."

It is important to explain to developers why and how "\0" is generated by a tool instead of a *normal* string like "Marc". Basically, a tool such as Pex relies on an underlying constraint solver such as the SMT solver Z3 [9] to solve the constraints of a path in the code under test. Constraint solvers are often designed to provide the simplest solution to satisfy the constraints. Developers could provide guidance to the tool by supplying some default values for the tool to use as starting points.

Isolate first development is crucial to make test generation work in real-world code bases in practice. In real-world code bases, a component under test (such as a method or a class) could have non-trivial dependencies on external environments such as file systems. The environment API implementations could be very complex or be written in native code rather than managed code being amenable to code exploration. Developers need to isolate the environment dependencies, e.g., with the assistance of a tool such as Moles [8]. Solving the dependency-isolation problem is orthogonal to and facilitates solving the test-generation problem: developers could use Moles without using Pex while manually writing test inputs, but could face challenges when using Pex without using Moles on environment-dependent code.

4.3 Lessons Learned

We next describe lessons learned while interacting with practitioners during our teaching and training in industrial settings.

Setting realistic expectations right away is very important. While it is important in an industrial setting to showcase the potential benefits of a new technology, automated tools will always have limitations, and these limitations must be clearly communicated. The developers have to be taught what the limitations are, how the developers can detect them when they hit such limitations, and how they can act on them. With regards to the fault-detection capabilities, it is important to emphasize the concept of assertions as specifications to specify the intended functionality of the code – in order to find violations of such specifications. With regards to the abilities of any code analysis tool, it is important to define the scope of their applicability. When they do not apply, the developers must be prepared to act on them, e.g., by manually writing traditional unit tests instead of PUTs, or by using code isolation frameworks in order to ensure that unit tests can run without environment dependencies. Training on these skills should be included in training sessions.

Trying to change deeply ingrained beliefs all at once is futile. Especially in an industrial setting, developers have usually become accustomed to particular development styles. Convincing them to change is difficult. It is important to highlight how a new advanced technology relates to earlier approaches, emphasizing on complementary aspects instead of differences or total replacement. For example, if a developer has adopted an approach of TDD, it should be emphasized how parameterized unit testing is a natural generalization of this approach, and not a radically new one or replacement.

5. Comparison of Teaching and Training at University and Industrial Settings

We observed three main differences on teaching and training developer-testing techniques and tool support at university and industrial settings.

First, students at university settings often do not have substantial experiences of industrial software development (especially C# software development given that Java has so far remained a popular teaching language at various universities), whereas practitioners at industrial settings often have substantial experiences (including C# software development). In our teaching and training, we used Pex for testing C# code. However, a non-trivial portion of the graduate students in our graduate course did not have C# programming experiences (with primarily Java programming experiences). To alleviate the issue, we designed a homework exercise in the beginning of the course on asking students to convert JUnit test code in Java to C# test code, getting them a quick start in getting familiar with C# coding.

Second, students at university settings have the incentives of studying well the teaching materials to earn good course grades besides learning various valuable skills, whereas practitioners at industrial settings often "come and watch", learning what is going on. At the university settings, adoption of tools or technologies being taught (after the course finishes) may not be heavily emphasized as a teaching objective; instead, emphasis is put on learning a wide range of skills ranging from abstract thinking, rigorous thinking, to understanding of testing techniques, writing of specifications in the form of PUTs, and effective usage of tools. At the industrial settings, adoption of tools or technologies being taught (after the tutorial finishes) could be an important training objective. Therefore, in training materials, it is desirable to incorporate more realistic and complex enough illustrative examples for applying the presented techniques and tool support in the industrial settings so that practitioners could be more easily convinced the utility of the techniques and tool support. On the other hand, in the university settings, illustrative examples used in the beginning of a course should have sufficiently low levels of complexity for students to understand and digest.

Third, teaching duration at university settings (being one semester long such as 16 weeks) is much longer than training duration at industrial settings (about half-day or full-day duration). At the university settings, substantial after-lecture exercises and projects as well as in-class discussion could be possible for students to digest and master the presented materials besides the slide presentation and live tool demonstration during lectures. In contrast, at the industrial settings, limited duration allowed presentation of only important knowledge points and brief summaries of important tool features.

6. Discussion

It is desirable that developers do not need to master sophisticated theories or technical details underlying tools when using the tools. For example, rather than demanding developers to write algebraic specifications [12] in a formal way, Pex allows developers to write intended behavior of the code under test in the form of PUTs (simply test methods with parameters), which in fact encode algebraic specifications. The internal details of dynamic symbolic execution as well as its underlying constraint solving and theorem proving of Z3 [9] exploited by the Pex tool are also not exposed via the tool interface to developers who are using Pex.

However, no state-of-the-art tool including Pex can deal with all complicated situations in real-world code bases automatically without human intervention or guidance. For example, sometimes a tool could fail to generate test inputs for covering a branch for one or more reasons. Understanding these reasons by the developers is required before the developers could provide guidance to the tool such as carrying out environment isolation, instrumenting some framework or library code that the code under test invokes, and writing factory methods that encode method sequences for generating desirable objects. While we are actively researching how to automatically provide better explanations when encountering problems [25], exposing some internal technical details of the tool to developers would be still needed (at least in the near future) to allow developers and a tool to cooperate for effectively carrying out testing tasks. Therefore, training developers with such skills remains important future work.

In our teaching at the university settings, coverage criteria were major lecture topics for testing foundations. It still remains an open question on how understanding these various coverage criteria could directly assist developers in carrying out developer testing, especially in the context of applying a powerful tool such as Pex. For example, given that a tool could be enhanced or pre-configured (by tool authors or vendors) to automatically achieve specific advanced coverage criteria [19], we hypothesize that the awareness and deep understanding of advanced coverage criteria could be less necessary in developer-testing practice. With the adoption of tools and methodologies such as Pex and PUTs, we hypothesize that much more emphasis should be placed on training developers on how to write high-quality specifications (e.g., in the form of PUTs) than manually understanding and applying various coverage criteria, which were originally the basis for test generation and selection (currently automated with a tool). However, there still remains an open question on how to effectively train such specification-writing skills.

7. Conclusion

Effectively conducting developer testing requires both effective tool support by tools and developer-testing skills by developers. As educators, we need to devise effective ways to teach students or practitioners to equip them with these skills. In this paper, we have described our experiences in teaching and training developer-testing techniques and tool support in both university and industrial settings. We highlight differences in teaching and training in these two settings, and observations from interacting with practitioners in our process of teaching and training.

In future work, we plan to conduct quantitative studies such as comparing measurable data before and after a course offering or training session. We plan to conduct comparison of teaching skills of developer testing with teaching skills of other testing types, or with teaching other software development skills. In addition, we plan to develop a set of educational tools for developers to learn developer testing. We have already released a website called "Pex for Fun" [5]. It currently provides different types of programming puzzles such as coding-duel puzzles. For a coding-duel puzzle, the user is requested to write and iteratively improve an implementation that matches a hidden implementation based on feedback provided by Pex in showing the behavioral differences (i.e., different program outputs) of the two implementations. Coding-duel puzzles can be used to train developers' programming skills and problem solving skills. We plan to extend "Pex for Fun" to include puzzles for training testing skills.

Acknowledgments. Tao Xie's work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, and CNS-0958235, and Army Research Office grant W911NF-08-1-0443.

References

[1] JUnit. `http://www.junit.org`.

[2] Microsoft Research Code Contracts. `http://research.microsoft.com/en-us/projects/contracts`.

[3] Microsoft Research Pex. `http://research.microsoft.com/Pex`.

[4] NUnit. `http://www.nunit.org/`.

[5] Pex for fun. `http://www.pexforfun.com/`.

[6] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. `http://www.introsoftwaretesting.com/`.

[7] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[8] J. de Halleux and N. Tillmann. Moles: tool-assisted environment isolation with closures. In *Proc. TOOLS*, pages 253–270, 2010.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[12] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[14] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[15] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[16] M. R. Marri, S. Thummalapenta, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. Technical Report TR-2010-9, North Carolina State University Department of Computer Science, Raleigh, NC, March 2010.

[17] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proc. AST, Business and Industry Case Studies*, pages 149–153, 2009.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[19] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proc. ICSM*, 2010.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[21] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[22] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[23] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Proc. ASE*, pages 365–368, 2006.

[24] G. Venolia, R. DeLine, and T. LaToza. Software development at microsoft observed. Technical Report MSR-TR-2005-140, Microsoft Research, Redmond, WA, October 2005.

[25] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Issue analysis for residual structural coverage in dynamic symbolic execution. Technical Report TR-2010-7, North Carolina State University Department of Computer Science, Raleigh, NC, March 2010.

[26] L. Zhang, T. Xie, L. Zhang, , N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proc. ICSM*, 2010.