# Parameterized Unit Testing with Pex: Tutorial

Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{nikolait,jhalleux,schulte}@microsoft.com

This hands-on tutorial teaches the principles of Parameterized Unit Testing in Visual Studio with Pex, an automatic test input generator.

A parameterized unit test (PUT) is simply a method that takes parameters, calls the code under test, and states assertions. Given a PUT written in a .NET language, Pex automatically produces a small test suite with high code and assertion coverage. Moreover, when a generated test fails, Pex can often suggest a bug fix. To do so, Pex performs a systematic program analysis, similar to path bounded model-checking. Pex learns the program behavior by monitoring execution traces, and uses a constraint solver to produce new test cases with different behavior. At Microsoft, this technique proved highly effective in testing even an extremely well-tested component.

From a specification, the developer (1) writes parameterized unit tests in C# to reflect the specification, and (2) develops code that implements the specification. The tutorial outlinea key aspects to make this methodology successful in practice, including how to write mock objects, as well as the theoretical foundations on which Pex is built.

This document is separated into two main parts. The first part provides detailed walk-through exercises on unit testing in Section 2, the methodology of Parameterized Unit Testing in Section 3, the usage of the Pex tool in Section 4, and ways to deal with the environment in Section 5. The second part is for the advanced reader. It provides a background on white box testing techniques in Section 6, and discusses in detail various aspects of the Pex tool in Section 7. Section 8 gives an overview of related work. Finally, Section 9 concludes.

*This tutorial refers to Pex version 0.18. The latest version of Pex can be obtained from* `http://research.microsoft.com/Pex/`*.*

## 1 Introduction

Unit tests are becoming increasingly popular. A recent survey at Microsoft indicated that 79% of developers use unit tests [325]. Unit tests are written to document customer requirements, to reflect design decisions, to protect against changes, but also, as part of the testing process, to produce a test suite with high code coverage that gives confidence in the correctness of the tested code.

The growing adoption of unit testing is due to the popularity of methods like XP ("extreme programming") [31], test-driven development (TDD) [30], and test execution frameworks like JUnit [186], NUnit [322] or MbUnit [90]. XP does not say how

P. Borba et al. (Eds.): PSSE 2007, LNCS 6153, pp. 141–202, 2010.

and which unit tests to write. Moreover, test execution frameworks automate only test execution; they do not automate the task of creating unit tests. Writing unit tests by hand can be a laborious undertaking. In many projects at Microsoft there are more lines of code for the unit tests than for the implementation being tested. Are there ways to automate the generation of good unit tests? We think that *Parameterized Unit Testing* is a possible answer, and this is the topic of the tutorial.

We describe how to design, implement and test software using the methodology of Parameterized Unit Testing [314,315], supported by the tool Pex [276,263]. Pex, an automated test input generator, leverages *dynamic [141] symbolic execution [194]* to test whether the software under test agrees with the specification. As a result, software development becomes more productive and the software quality increases. Pex produces a small test suite with high code coverage from Parameterized Unit Tests.

In effect, we combine two kinds of testing introduced in Chapter 2: 1) Testing for functional properties: Just as unit tests, Parameterized Unit Tests usually serve as specifications of functional properties. 2) Structural testing: We analyze such tests with dynamic symbolic execution, a structural testing technique.

To facilitate unit testing, mock objects are often used to isolate the test from the environment. We extend this notion to *parameterized mock objects*, which can be viewed as a model of the environment. Writing such parameterized mock objects, and generating tests with them, is in effect a form of model-based testing (see Chapter 3).

The effectiveness of a test suite, whether written by hand, or generated from parameterized unit tests, can be measured with mutation testing (see Chapter 8).

Parameterized Unit Tests are algebraic specifications [38] written as code. Another name for this concept is *theories* [291,293] in the JUnit test framework. They appear as *row tests* in MbUnit [90], and under other names in various other unit test frameworks. While Pex is a tool that can generate test inputs for .NET code, many other research and industrial tools [141,297,67,66,296,142] exist that can generate test inputs in a similar way for C code, Java code and x86 code.

We introduce the concepts and illustrate the techniques with some examples. We assume deterministic, single-threaded applications.

An earlier and shorter version of this tutorial on Parameterized Unit Testing with Pex can be found in [91]. More documentation can be found on the Pex website [276].

## 2 Unit Testing Today

A *unit test* is a self-contained program that checks an aspect of the implementation under test. A unit is the smallest testable part of the program. One can partition each unit test into three parts: (1) exemplary data, (2) a method sequence and (3) assertions.

– *exemplary data* can be considered as the test input that is passed to the methods as argument values,
– *method sequence* based on the data, the developer builds a scenario that usually involves several method calls to the code-under-test,
– *assertions* encode the test oracle of a unit test. The test fails if any assertion fails or an exception is thrown but not caught. Many unit test frameworks have special support for *expected exceptions*, which can often be annotated with custom attributes.