Parameterized Unit Testing in the Open Source Wild

Wing Lam[1], Siwakorn Srisakaokul[1], Blake Bassett[1], Peyman Mahdian[1], Tao Xie[1], Nikolai Tillmann[2], Jonathan de Halleux[2]

[1] University of Illinois at Urbana-Champaign, USA
[2] Microsoft Research, USA
{winglam2,srisaka2,rbasset2,mahdian2,taoxie}@illinois.edu
{nikolait,jhalleux}@microsoft.com

Abstract. With recent advances in test generation research, powerful test generation tools are now at the fingertips of developers in software industry. For example, Microsoft Research Pex, a state-of-the-art tool based on dynamic symbolic execution, has been shipped as IntelliTest in Visual Studio 2015. For test inputs automatically generated by such tool, to supply test oracles (beyond just uncaught runtime exceptions or crashes), developers can write formal specifications such as code contracts in the form of preconditions, postconditions, and class invariants. However, just like writing other types of formal specifications, writing code contracts, especially postconditions, is challenging. In the past decade, parameterized unit testing has emerged as a promising alternative to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests, without parameters. PUTs have been popularly supported by various unit testing frameworks for .NET along with the recent JUnit framework. However, there exists no study to offer insights on how PUTs are written by developers in either proprietary or open source development practices, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. To fill this gap, in this paper, we present the first empirical study of parameterized unit testing conducted on open source projects. We study hundreds of parameterized unit tests that open source developers wrote for these open source projects. Our study findings provide valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

1 Introduction

With recent advances in test generation research such as dynamic symbolic execution [15, 23], powerful test generation tools are now at the fingertips of developers in software industry. For example, Microsoft Research Pex [25, 27], a state-of-the-art tool based on dynamic symbolic execution, has been shipped as

IntelliTest [20] in Visual Studio 2015, benefiting numerous developers in software industry. Such test generation tools allow developers to automatically generate test inputs for the code under test, comprehensively covering various program behaviors to achieve high code coverage. These tools help alleviate the burden of extensive manual software testing, especially on test generation.

Although such tools provide powerful support for automatic test generation, by default only a predefined limited set of properties can be checked, serving as test oracles for these automatically generated test inputs. Violating these predefined properties leads to various runtime failures, such as null dereferencing or division by zero. Despite being valuable, these predefined properties are *weak test oracles*, which do not aim for checking functional correctness but focus on robustness of the code under test.

To supply strong test oracles for automatically generated test inputs, developers can write formal specifications such as code contracts [10, 17, 19] in the form of preconditions, postconditions, and class invariants. However, just like writing other types of formal specifications, writing code contracts, especially postconditions, is challenging. Consider an example method under test from the open source NUnit project in Figure 1. It is relatively easy to specify preconditions for the method as `(sn != null) && (sv != null)` but it is quite challenging to specify comprehensive postconditions for this method to capture its intended behaviors.

In the past decade, parameterized unit testing [26, 28] has emerged as a practical alternative to specify program behaviors under test in the form of unit tests. Developers can write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to conventional unit tests (CUTs), without parameters. Then developers can apply an automatic test generation tool such as Pex to generate input values for the PUT parameters. Note that algebraic specifications [16] can be naturally written in the form of PUTs but PUTs are not limited to being used to specify algebraic specifications.

```
//MSS=MemorySettingsStorage
00:public class SettingsGroup{
01: MSS storage; ...
02: public SettingsGroup(MSS storage){
03:  this.storage = storage; }
05: public void SaveSetting(string sn, object sv) {
06:  object ov = storage.GetSetting( sn );
07:  //Avoid change if there is no real change
08:  if(ov != null ) {
09:   if(ov is string && sv is string &&
             (string)ov==(string)sv ||
10:    ov is int&&sv is int&&(int)ov==(int)sv ||
11:    ov is bool&&sv is bool&&(bool)ov==(bool)sv ||
12:    ov is Enum&&sv is Enum&&ov.Equals(sv))
13:     return;
14:  }
15:  storage.SaveSetting(sn, sv);
16:  if (Changed != null)
17:   Changed(this, new SettingsEventArgs(sn));
18: }}
```

Fig. 1: **`SaveSetting`** method under test from the **`SettingsGroup`** class of NUnit

For the method under test in Figure 1, a real-world CUT written by the NUnit developers is shown in Figure 2. Despite seemingly comprehensive, the CUT is insufficient, not being able to cover Lines 9-13 of the method in Figure 1. The CUT's corresponding, and more powerful, PUT is shown in Figure 3. A PUT is annotated with `[PexMethod]`, sometimes attached with optional at-

```
//st is of type MemorySettingsStorage and
//instantiated in the init() method of the test class
01:public void SaveAndLoadSettings() {
02:  Assert.IsNull(st.GetSetting("X"));
03:  Assert.IsNull(st.GetSetting("NAME"));
04:  st.SaveSetting("X", 5);
05:  st.SaveSetting("NAME", "Charlie");
06:  Assert.AreEqual(5, st.GetSetting("X"));
07:  Assert.AreEqual("Charlie", st.GetSetting("NAME"));
08:}
```

Fig. 2: A real-world CUT for the method in Figure 1.

```
00:[PexMethod(MaxRuns = 200)]
01:public void TestSave1(MemorySettingsStorage st, string sn, object sv) {
02:    PexAssume.IsTrue(st != null);
03:    PexAssume.IsTrue(sn != null && sv != null);
04:    st.SaveSetting(sn, sv);
05:    PexAssert.AreEqual(sv, st.GetSetting(sn));
06:}
```

Fig. 3: The PUT corresponding to the CUT in Figure 2.

```
00:[PexFactoryMethod(typeof(MemorySettingsStorage))]
01:public static MemorySettingsStorage Create(string[] sn, object[] sv) {
02:    PexAssume.IsTrue(sn != null && sv != null);
03:    PexAssume.IsTrue(sn.Length == sv.Length);
04:    PexAssume.IsTrue(sn.Length > 0);
05:    MemorySettingsStorage mss = new MemorySettingsStorage();
06:    for (int count = 0; count < sn.Length; count++) {
07:      PexAssume.IsTrue(sv[count] is string || sv[count] is int
08:                       || sv[count] is bool || sv[count] is Enum);
09:      mss.SaveSetting(sn[count], sv[count]);
10:    }
11:    return mss;
12:}
```

Fig. 4: A factory method written to assist Pex to generate desirable objects for a non-primitive parameter of the PUT in Figure 3.

tributes to provide configurations for Pex's test generation. An example attribute is in `[PexMethod(MaxRuns = 200)]` shown in Figure 3. The `MaxRuns` attribute along with the attribute value `200` indicates the maximum number of runs/iterations allocated during Pex's path exploration for test generation. In the beginning of the PUT (Lines 2-3), the `PexAssume` statements are used as assumptions imposed on the three PUT parameters. During test generation, Pex filters out all the generated input values (for the PUT parameters) that violate the specified assumptions. The `PexAssert` statement in Line 5 is used as the assertion to be verified when running the generated input values.

For test generation, Pex can effectively handle primitive-type parameters such as `string` and `integer`. However, like any other state-of-the-art test generation tools, Pex faces challenges in generating input values for non-primitive parameters such as `MemorySettingsStorage` in the example PUT. These non-primitive parameters require desirable object states to be generated to verify different behaviors. However, a main challenge in constructing desirable object states for non-primitive parameters is to construct an effective sequence of method calls that create and mutate objects. Thus, for a non-primitive PUT parameter, developers typically need to write a factory method to supply to Pex an effective method-call sequence. Figure 4 shows an example factory method to assist Pex in generating desirable objects for `MemorySettingsStorage`, a non-primitive parameter of the example PUT. The factory method accepts two arrays of setting names and values, and adds those entries to the storage.

Since parameterized unit testing was first proposed in 2005 [28], PUTs have been popularly supported by various unit testing frameworks for .NET along with the recent JUnit framework (as parameterized test [5] and theories [9,21]). However, there exists no study to offer insights on how PUTs are written by developers in either proprietary or open source development practices, posing barriers for various stakeholders to bring PUTs to widely adopted practices in software industry. Example stakeholders are current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

To fill this gap of lacking studies of PUTs, in this paper, we present the first empirical study of parameterized unit testing conducted on open source projects. In particular, this paper makes the following major contributions:

- The first empirical study of parameterized unit testing in the open source wild, with focus on hundreds of real-world PUTs, producing study findings that provide valuable insights for various stakeholders.
- A collection of real-world open source projects equipped with developer-written parameterized tests (released on our project website [7]). These subjects can be used by the research community to conduct future empirical studies or to evaluate enhancements to automated test generation tools.
- A suite of analysis tools (released on our project website [7]) for the analysis of PUTs and associated code under test. These tools are based on the .NET Compiler Platform, and enable the collection of a variety of information and statistics from PUTs in a subject project.

The remainder of this paper is organized as follow. Section 2 discusses the setup of our study. Section 3 presents our study findings and discuss the implications to stakeholders. Section 4 discusses threats to validity of our study. Section 5 presents related work, and Section 6 concludes the paper.

2 Study Setup

This section describes our process for collecting subjects (i.e., open source projects including PUTs) and the tools that we developed to collect and process data from the subjects. The details of these subjects and our tools are released on our project website [7].

Subject-Collection Procedure. The subject-collection procedure (including subject sanitization) was a multi-stage process. At a coarse granularity, this process involved (1) comprehensive and extensive subject collection from searchable online source code repositories, (2) deduplication of subjects obtained multiple times from different repositories, (3) condensing multi-project subjects, (4) selecting developer-written parameterized tests (filtering out automatically generated parameterized test stubs), and (5) categorization of subjects into levels of parameterized test usage.

Table 1: Subjects collected for study.

Subject Name	**#PUTs**	**#CUTs**	**#LOC**	**Main purposes of PUTs**
Subjects with high number of PUTs				
~~PexFaultLocalization~~	~~168~~	~~1221~~	~~92144~~	~~Experimental subjects~~
QuickGraph	133	344	79922	Functional tests
linqtoroot	112	1212	55142	Functional tests
~~concolic-walk~~	~~87~~	~~1~~	~~2724~~	~~Experimental subjects~~
stringextensions	67	446	7382	Functional tests
isuka	54	300	18171	Functional tests
utilities-net	44	299	7446	Functional tests
henoch	27	170	18573	Functional tests
bbcode	26	16	1780	Functional tests
ConcurrentList	21	27	2171	Functional tests
HighTotal	**739**	**4036**	**285455**	
HighTotal(~~Noises~~)	**484**	**2814**	**190587**	
Subjects with low number of PUTs				
Scutex	13	206	33654	
PexMolesAndFakes	12	25	1472	
PurelyFunctionalDataStructures	10	40	2649	
binary-heap-pex	9	67	1341	
Moq	8	0	14729	
Pex-and-Moles-Overview	8	59	1496	
symb2	6	1	3332	
talks-pex-and-moles	6	35	765	
ReactiveUI	5	0	10177	
UnitTesting	5	47	597	
functionextensions	4	18	570	
tcrucon	4	33	2595	
PortableDeviceLib	2	4	4919	
raop	2	3	997	
robust	2	53	3035	
rss_insurcompaccountsystem	2	189	12328	
ScotAlt.Net-Pex-QuickCheck	2	0	347	
cardgameslib	1	3	3651	
daemaged.compression	1	1	2329	
AllTotal	**841**	**4820**	**386438**	
AllTotal(~~Noises~~)	**586**	**3598**	**291570**	

For comprehensive subject collection, we queried a set of widely known code search services. The used query was "`PexMethod`", which matched files containing parameterized unit tests specified with the Pex framework. The three code search services that returned results are Github [4], Black Duck Open Hub [2], and SearchCode [8]. For each code search service, we parsed the search results to extract the source code repository where each file in the search results was stored.

Analysis Tools. We developed a set of tools to collect metrics from the subjects. We used Roslyn, the .NET Compiler Platform, to build our tools. These tools parsed C# source files to produce an abstract syntax tree, which was traversed to collect information and statistics of interest.

Subject Misusing PUT Annotations and Assertions. After initial inspection of the metric values measured for the collected subjects, we showed high suspicion on one open source project named as AutomaTones [1]. For example, the project includes many PUTs (i.e., methods annotated with `[PexMethod]`); these PUTs typically are very long, including a lot of branch logics but no assumptions. After careful inspection of the project's source code, we found that the developers of this project misunderstood and misused PUT annotations (i.e., `[PexMethod]`) and PexAssert assertions (i.e., methods defined on the `PexAssert` class): adding them to the production code instead of test code by treating PexAssert assertions as code contract assertions. We excluded this subject out of our collected subjects.

Collected Subjects. The information on the collected subjects is shown in Table 1. Column 1 shows the name of each subject, and Columns 2 and 3 show the number of PUTs and the number of CUTs in each subject, respectively. In total, we identified 29 subjects and these subjects contain a total of 841 PUTs.

Based on the number of PUTs in a subject, we split the subjects into two categories: subjects with a high number of PUTs (i.e., >= 20) and subjects with a low number of PUTs (i.e., < 20). Our detailed study for research questions focus on the first category (including 10 subjects) because a subject in the second category often includes occasional tryouts of PUTs instead of serious use of PUTs for testing the functionalities of the open source project under test.

For the subjects in the first category, we also describe the main purposes of PUTs in each subject in the last column of Table 1. After inspection, we found that two subjects in the first category: PexFaultLocalization [6] and concolic-walk [3] use PUTs as experimental subjects (e.g., creating PUTs without assertions to experiment with Pex's capability of achieving high code coverage). We considered these two subjects as noises for our detailed study because the PUTs in these two subjects do not represent usage of PUTs for testing functionalities of the code under test. However, in our detailed study, we still include the statistics for these two subjects in order to shed light on how these noises could have negatively affected study observations and conclusions if they were not carefully identified and separated. To make it clear that these two subjects are noises, we put strike-through lines in the rows for these two subjects and calculate additional total or average statistics excluding these two subjects, postfixed with "(~~Noises~~)".

Implications. For testing researchers who conduct studies on open source projects, careful data sanitization and inspection are needed. Otherwise, duplicated data or noisy data (e.g., PUTs in the three subjects AutomaTones [1], PexFaultLocalization [6], and concolic-walk [3]) would have led the study to produce misleading observations or conclusions.

3 Study Results

Our study is primarily concerned with the characteristics of PUTs appearing in our subjects. Our study findings aim to benefit various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators. In particular, our study intends to address the following five main research questions:

- **RQ1**: What is the extent of PUTs being written in our subjects?
  - We address RQ1 because addressing it can help understand the current extent of PUTs being written and better inform stakeholders future directions on providing effective tool support or training to guide where (in the code base under test) to write PUTs for.
- **RQ2**: What are the relative complexity levels of assumptions vs. assertions written in PUTs? What are the types of these assumptions and assertions written in PUTs?
  - We address RQ2 because addressing it can help understand developers' current practice of writing assumptions vs. assertions in PUTs, and better inform stakeholders future directions on providing effective tool support or training on writing assumptions and assertions in PUTs.
- **RQ3**: What is the extent of having non-primitive parameters in PUTs? What is the extent of writing factory methods for those non-primitive parameter types?
  - We address RQ3 because addressing it can help understand the extent of writing sufficiently general PUTs (e.g., promoting an object produced by a method sequence hard-coded in a PUT to a non-primitive parameter of the PUT), and writing factory methods for those non-primitive parameter types to assist automatic test generation tools; in addition, addressing RQ3 helps better inform stakeholders future directions on providing effective tool support or training for addressing challenges related to non-primitive PUT parameters in terms of both PUT specification and test generation.
- **RQ4**: What is the extent of annotating a PUT with attributes to provide configurations for automatic test generation, and what types of attributes are written and how often?
  - We address RQ4 because addressing it can help understand the current extent of writing attributes to guide automatic test generation, and better inform stakeholders future directions on providing effective tool support or training for enabling developers to guide automatic test generation.
- **RQ5**: What are the characteristics of PUTs compared to CUTs written for the same subject?
  - We address RQ5 because addressing it can help understand the characteristics of PUTs compared to CUTs, and better inform stakeholders future directions on studies to compare PUTs and CUTs along with effective tool support and training for converting CUTs to PUTs and converting CUT writers to PUT writers.

Table 2: The number and percentage of public methods invoked by PUTs.

Subject Name	**#Public Methods**	**#Invoked in PUTs**	**%**
~~PexFaultLocalization~~	~~160~~	~~128~~	~~80.0~~
QuickGraph	118	23	19.5
linqtoroot	503	176	35.0
~~concolic-walk~~	~~21~~	~~5~~	~~23.8~~
stringextensions	68	63	92.7
isuka	450	61	13.6
utilities-net	207	102	49.3
henoch	175	66	37.7
bbcode	35	19	54.3
ConcurrentList	15	13	86.7
Average	**172.2**	**65.6**	**49.2**
Average(~~Noises~~)	**196.4**	**65.4**	**48.6**

3.1 RQ1. Usage of PUTs

We explored how highly used PUTs are in testing public methods from the project under test. Table 2 shows the number and percentage of public methods invoked by PUTs. Column "# Public Methods" shows the number of public methods that were declared in all projects except those that are in PUT projects. Column "# Invoked in PUTs" shows the number of unique public methods that are invoked directly inside at least one PUT. Column "%" shows the percentage of unique public methods invoked in PUTs over the total number of public methods in all projects.

As shown in Table 2, on average, 49.2% of all public methods are directly called inside PUTs. We expect this number to be even higher if we use dynamic analysis (e.g., running the generated input values for the PUTs) to additionally count the public methods that are indirectly called by PUTs during runtime. Nevertheless, according to the current results, developers rely on PUTs to test for approximately half of the public methods in their projects under test.

It is observed that the public methods that are not tested by PUTs fall into two main categories. The first category includes methods that are overloaded. Method overloading allows the method's argument list to differ in the following three ways: the number of arguments, data type of arguments, and sequence of the arguments' data types. Not testing these public methods can be due to the difference in the number of arguments. For example, developers may write PUTs to invoke only an overloaded method variant f_1 with fewer parameters, and f_1 can further invoke another overloaded method variant f_2 with more parameters (with additional `out` parameters, which the caller of f_1 may not care about). The opposite case can happen too: developers may write PUTs to invoke only an overloaded method variant f_1 with more parameters, and f_1 can further invoke another method overloaded variant f_2 with fewer parameters besides including some additional computations done in f_1 related to those additional parameters of f_1. For both cases, it is reasonable that developers write PUTs to test only one overloaded variant f_1 but not the other variant f_2 invoked by f_1 because the PUTs already indirectly test f_2.

The second category of public methods not tested with PUTs includes methods already equipped with code contracts. In this case, it is reasonable that developers do not write PUTs because writing PUTs may duplicate the checking power already accomplished by the equipped code contracts, and developers can apply Pex directly against the code under test equipped with the code contracts.
Implications. Knowing that a variant of an overloaded method can be indirectly tested by a PUT written for another variant, developers who are writing PUTs can avoid writing duplicate PUTs for multiple variants. In addition, knowing that the use of PUTs and the use of code contracts can complement with each other, developers can give priority to write PUTs for those code portions not equipped with PUTs or write code contracts for those code portions not equipped with PUTs. More future research is needed to investigate where to better use code contracts and where to better use PUTs.

3.2 RQ2. Assumptions and Assertions

Table 3: Different types of assumptions in subjects.

PexAssume Type	#
PexAssumeUnderTest	426
PexAssumeNotNull	216
IsTrue	203
IsNotNull	98
IsNotNullOrEmpty	24
TrueForAll	20
IsFalse	12
EnumIsDefined	9
InRange	7
TrueForAny	6
Fail	5
AreElementsNotNull	5
AreDistinctValues	1
AreEqual	1
Total	**1033**
Total w/o Null Check	**269**
Null Check Percentage	**73.97%**

Table 4: Different types of assertions in subjects.

PexAssert Type	#
AreEqual	110
IsTrue	88
IsNotNull	62
TrueForAll	16
Implies	16
Throws<>	15
AreElementsEqual	14
AreEqual<int>	10
IsFalse	9
AreBehaviorsEqual	6
Run	4
ImpliesIsTrue	4
Fail	3
AreEqual<bool>	1
TrueForAll<double[]>	1
ExpectExactlyOne	1
Inconclusive	1
Total	**361**
Total w/o Null Check	**299**
Null Check Percentage	**17.17%**

To understand developers' practices of writing assumptions and assertions in PUTs, we studied our subjects' common types of assumptions and assertions and compared the writing of assumptions and assertions to the writing of preconditions and postconditions in code contracts, respectively. As shown in Table 3, `PexAssumeUnderTest` is the most common type of assumption, used

Table 5: Number of PexAssumes and PexAsserts with Number of Clauses

Subject Name	**#Assumes**	**#Assume Clauses**	/	**#Asserts**	**#Assert Clauses**	/
~~PexFaultLocalization~~	~~483~~	~~336~~	~~0.70~~	~~21~~	~~22~~	~~1.05~~
QuickGraph	237	15	0.06	21	21	1.00
linqtoroot	123	0	0.00	0	0	N/A
~~concolic-walk~~	~~0~~	~~0~~	~~N/A~~	~~0~~	~~0~~	~~N/A~~
stringextensions	11	8	0.73	110	118	1.07
isuka	69	27	0.39	113	146	1.29
utilities-net	9	10	1.11	70	76	1.09
henoch	49	0	0.00	1	1	1.00
bbcode	30	21	0.70	5	5	1.00
ConcurrentList	22	11	0.50	20	20	1.00
Average			**0.45**			**1.04**
Average(~~Noises~~)			**0.44**			**1.06**

426 times, in our subjects. `PexAssumeUnderTest` is used to mark parameters as non-null and to be that precise type. The second most common type of assumption, `PexAssumeNotNull`, is used 216 times. Similar to `PexAssumeUnderTest`, `PexAssumeNotNull` is used to mark parameters as non-null except that it does not require the type to be precise. Since PUTs are commonly written to test the behavior of non-null objects as the class under test or use non-null objects as arguments to a method under test, it is reasonable that the most common assumption types used by PUT developers are ones that mark parameters as non-null. Furthermore, according to the last row for Tables 3 and 4, developers perform null checks much more frequently for assumptions than assertions. Since assertions are validated at the end of a PUT and it is less often that code before the assertions manipulates or produces a null object, it is reasonable that assumptions check for null more frequently than assertions do. For assumptions and assertions such as `TrueForAll`, we suspected that developers' low number of use may be due to the unawareness of such attribute's existence. `TrueForAll` checks whether a predicate holds over a collection of elements. In our subjects, we found cases where a collection was iterated over to check whether a predicate was true for all of the elements; instead, developers should have used the `TrueForAll` assumption or assertion.

Previous research on code contracts [22] shows that only 26% of code-contract clauses were postconditions implying that programmers tend to write preconditions more often than postconditions. As it is shown in Table 5, if we compare the ratio of #assumption clauses over #assumptions with the ratio of #assertion clauses over #assertions, 6 out of 7 of our subjects (excluding the two `noise` subjects and subjects with 0 assertion), have a higher number of clauses for assertions than assumptions. Based on previous research on Code Contracts and our findings, we hypothesize that developers refrain from writing postconditions because similar to assertions requiring a high number of clauses than assumptions, postconditions also require a high number of clauses than preconditions.

Implications. Knowing that certain types of assumptions and assertions are more (or less) common than others and how assumptions and assertions compare to preconditions and postconditions, respectively, researchers can be better informed about developers' current practices and thus can better focus their research efforts on addressing possible weaknesses in the practices. With the wide range of usage of assumption types and assertion types in Table 3 and 4, tool developers can incorporate this data with their tools to better infer assumptions and assertions to assist developers.

3.3 RQ3. Non-primitive Parameters

Table 6: Factory Methods for Non-primitive Parameters

Subject Name	**Non-prim Paras**	**Prim + Non-prim Paras**	**Non-prim / (Prim + Non-prim)**	**Non-prim Paras w/ Factory**	**w/ Factory / Paras**
~~PexFaultLocalization~~	~~190~~	~~313~~	~~60.70%~~	~~49~~	~~25.79%~~
QuickGraph	168	190	88.42%	36	21.43%
linqtoroot	183	248	73.79%	47	25.68%
~~concolic-walk~~	~~0~~	~~403~~	~~0.00%~~	~~0~~	~~N/A~~
stringextensions	35	187	18.72%	0	0.00%
isuka	4	88	4.55%	0	0.00%
utilities-net	15	66	22.73%	0	0.00%
henoch	48	54	88.89%	0	0.00%
bbcode	9	57	15.79%	0	0.00%
ConcurrentList	0	16	0.00%	0	N/A
Average			**37.36%**		**9.11%**
Average(~~Noises~~)			**39.11%**		**6.73%**

Typically developers are expected to avoid hard-coding a method sequence in a PUT to produce an object to be used for testing the method under test. Instead, the developers are expected to promote such object as a non-primitive parameter of the PUT. In this way, the PUT can be made more general, to capture the intended behavior and enable an automatic test generation tool such as Pex to generate objects of various states for the non-primitive parameter. To determine the extent of writing such sufficiently general PUTs, we studied how frequently developers write PUTs with non-primitive parameters. On the other hand, having non-primitive parameters for PUTs can pose challenges for an automatic test generation tool. Thus, developers are typically expected to write factory methods for those non-primitive parameter types to assist automatic test generation tools. We also studied how often developers provide factory methods for non-primitive parameter types of PUTs.

As shown in Table 6, developers, on average, write non-primitive parameters 39.11% of the time for our subjects excluding the two `noise` subjects. In other words, for every 10 parameters used by developers, 4 of those parameters are non-primitive. Yet developers, on average, write factory methods for their non-primitive parameters only 6.73% of the time for our subjects excluding the two

noise subjects. Ideally, developers should be writing factory methods close to 100% of the time when they use non-primitive parameters in their PUTs.
Implications. Knowing that the number of non-primitive parameters written by developers is nontrivial and how infrequent developers write factory methods for these parameters, educators can strongly communicate the importance of writing factory methods to developers and train developers with skills of writing factory methods. In addition, tool researchers and vendors can invest future research efforts for exploring effective tool support to assist developers to write factory methods.

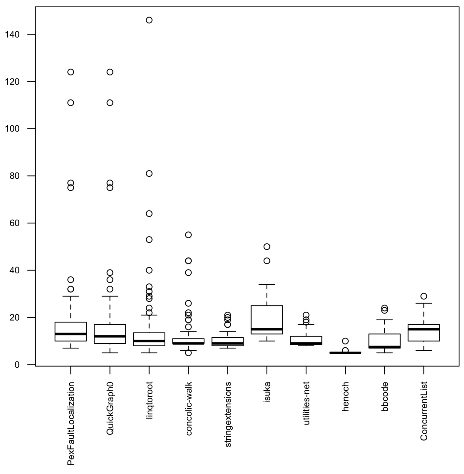
3.4 RQ4. Attributes

Table 7: PUT Counts

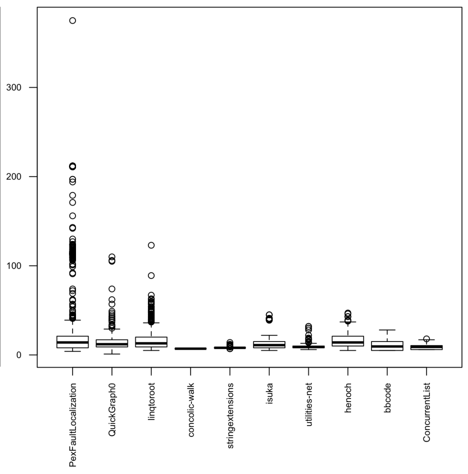
Subject Name	**Attrs**	**Attrs / PUTs**	**#PUTs with**				**%non-prim param methods**
			Assume	**Assert**	**Params**	**non-prim param**	
~~PexFaultLocalization~~	~~50~~	~~0.30~~	~~101~~	~~67~~	~~167~~	~~118~~	~~70.24%~~
QuickGraph	33	0.25	7	107	118	114	85.71%
linqtoroot	3	0.03	0	74	109	97	86.61%
~~concolic-walk~~	~~174~~	~~2.00~~	~~0~~	~~0~~	~~87~~	~~0~~	~~0.00%~~
stringextensions	0	0.00	4	57	67	33	49.25%
isuka	1	0.02	17	51	54	3	5.56%
utilities-net	0	0.00	3	42	44	11	25.00%
henoch	13	0.48	0	3	27	24	88.89%
bbcode	1	0.04	9	23	17	8	30.77%
ConcurrentList	6	0.29	11	21	14	0	0.00%
Average	**28.10**	**0.34**	**15.20**	**44.50**	**70.40**	**40.80**	**44.20%**
Average(~~Noises~~)	**7.13**	**0.14**	**6.38**	**47.25**	**56.25**	**36.25**	**46.47%**

To investigate developers' practices of configuring Pex via PUT attributes, we studied the number of attribute, as configuration options for running Pex, written by developers in PUTs. The third column of Table 7 shows the average number of attributes added per PUT. With the exception of one noise subject concolic-walk, which repeated the same two attributes for all 87 of its PUTs, the average number of attributes that developers added to PUTs is minimal. Including the two noise subjects, our data indicates that a developer added 3 attributes for every 10 PUTs; however, if we calculate this estimation exluding the two noise subjects, developers added only 1 attribute for every 10 PUTs.

Common attributes that developers added are MaxRunsWithoutNewTests and MaxConstraintSolverTime. MaxRunsWithoutNewTests is the maximum number of consecutive runs that do not generate a new test input before Pex terminates. Developers commonly set this attribute to be a maximum of 5 runs. We hypothesize that because most of these tests contain only one or two conditionals, developers felt that it was unlikely for Pex to require more than 5 runs and yet still be able to output any new test input. MaxConstraintSolverTime is the time limit restricting Pex as it explores the execution paths of a program for each test input that it tries to generate. Developers commonly set this attribute to be 5000



(a) PUT method length



(b) CUT method length

Fig. 5: Method length (#LOC) of PUTs and CUTs

seconds. Similar to why developers set the attribute for `MaxRunsWithoutNewTests`, tests with attribute `MaxConstraintSolverTime` contains few conditionals and developers felt that it is unlikely for Pex to require more time than 5000 seconds to generate a new test input.

Implications. Knowing that developers rarely add attributes to their PUTs, framework developers should be more motivated than before to enhance their feature set to contextually suggest or introduce attributes such as `MaxRunsWithoutNewTests` and `MaxConstraintSolverTime` to developers.

3.5 RQ5. Characteristics of PUTs compared to CUTs

To understand PUT developer's motivation for writing PUTs instead of CUTs, we studied the number of lines that PUTs and CUTs contain on average, respectively, the compatibility of these two types of tests with automatic test generation tools, and previous research related to this topic. According to the outliers in the plots from Figure 5, the length of PUTs is usually less than the length of CUTs (the length is measured by the number of lines of code in the test method). The average method length of PUTs is 12.9 lines while the one of CUTs is 13.9. One reason for why CUTs are longer is because developers usually write CUTs when they want to test with a specific input value. For example, a specific input value that we observed is a string value in the XML format. By hardcoding such a string value, developers might miss triggering failures due to non-XML format string values. Instead, what the developers should do is to use PUTs and specify the specific input value as one of the input values that Pex should generate.

Not only would writing PUTs assist developers in writing less code, doing so may also help the developers trigger failures by allowing tools such as Pex

to generate input values that developers did not think of. Previous research [24] on retrofitting CUTs into PUTs showed that by retrofitting 407 CUTs into 224 PUTs, developers detected 19 new faults that were not detected by existing CUTs and increased branch coverage by 4% on average.
Implications. Knowing that PUTs are shorter to write, are easier to generate input values for, detect more faults, and achieve higher branch coverage than CUTs, developers who have not started writing PUTs should consider writing PUTs in their future testing practices. Our findings on why developers chose to write CUTs can also provide insight to tool developers so that their tool can assist developers to convert CUTs to PUTs.

4 Threats to Validity

There are various threats to validity of the study, being broadly divided into threats to construct, internal, and external validity.
Construct Validity. Threats to construct validity are concerned with the validity of the use of our chosen metrics to model properties of interest. For example, we used the number of clauses contained in an assertion or assumption to model its strength. The metric, the number of clauses contained in an assertion or assumption, is an approximation to the actual strength of the assertion or assumption, and was primarily used for ease of comparison to related work [13].
Internal Validity. Threats to internal validity are concerned with the validity of our experimental procedure. Due to the complexity of software, faults in our analysis tools could have affected our results. However, our analysis tools were written with an associated suite of unit tests, and samples of the results were manually verified. Results from our manual analyses were confirmed by at least two of the authors.
External Validity. Threats to external validity are concerned with the validity of our conclusions when applied to subjects other than those that were studied. In other words, how well do our results generalize? We primarily focused on projects using PUTs in the context of automated test generation, so we may not generalize to situations outside of this setting (e.g., general usage of Theories [21] in Java). In addition, our analysis focused specifically on those subject programs equipped with PUTs written using the Pex framework, and the API differences or idiosyncrasies of other frameworks may impact their usage relative to Pex. All of our subjects are written in C#, but vary widely in their application domains and project sizes. Finally, all of our subjects are open source software, and therefore our conclusions may not generalize to proprietary software systems.

5 Related Work

To the best of our knowledge, our empirical study is the first on parameterized unit testing in the open source wild. In contrast, without studying practices of parameterized unit testing, previous work propose new techniques for parameterized unit testing. For example, Xie et al. [29] propose a technique for assessing the quality of PUTs using mutation testing. Thummalapenta et al. [24] propose the manual retrofitting of CUTs to PUTs, and show that new faults are detected and coverage is increased after such manual retrofitting was conducted. Fraser et al. [14] propose a technique for generating PUTs starting from concrete test inputs and results.

Our work is related to previous work on studying developer-written formal specifications such as code contracts. Schiller et al. [22] conduct case studies on the use of code contracts in open source projects in C#. They analyzed 90 projects using Code Contracts [10] and categorized their use of various types of specifications, such as null-checks, bounds checks, and emptiness checks. They find that checking for nullity and emptiness are the most common types of specifications. Estler et al. [13] study code contract usage in 21 open source projects using JML [18] in Java, Design By Contract in Eiffel [19], and Code Contracts [10] in C#. Their study also includes an analysis of the change in code contracts over time, relative to the change in the specified source. Their findings agree with Schiller's on the majority use of nullness code contracts. Furthermore, Chalin [12] study code contract usage in over 80 Eiffel projects.

Casalnuovo et al. [11] study the use of assertions in open-source projects hosted on Github. They find that 69 of the top 100 most popular C and C++ projects on Github have significant assertion use, and that methods containing assertions are less likely, on average, to contain faults.

6 Conclusion

To fully leverage the power of automatic test generation tools available to developers in software industry, developers can write parameterized unit tests (PUTs), which are unit-test methods with parameters, in contrast to conventional unit tests, without parameters. Then developers can apply an automatic test generation tool such as Pex to generate input values for the PUT parameters. In this way, separation of two main testing duties is well accomplished: developers focus on specifying comprehensive program behaviors under test in PUTs while automatic test generation tools focus on generating high-quality input values for the PUT parameters.

To fill the gap of lacking studies of PUTs in either proprietary or open source development practices, we have conducted the first empirical study of parameterized unit testing in open source development practices. We have studied hundreds of parameterized unit tests that open source developers wrote for various open source projects. Our study findings provide valuable insights for various stakeholders such as current or prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool vendors, testing researchers, and testing educators.

References

1. Automatones. `http://automatones.codeplex.com/`.

2. Black Duck Open Hub code search. `http://code.openhub.net`.
3. concolic-walk. `https://github.com/osl/concolic-walk`.
4. Github code search. `https://github.com/search`.
5. Parameterized tests in JUnit. `https://github.com/junit-team/junit/wiki/Parameterized-tests`.
6. Pexfaultlocalization. `https://github.com/lukesandberg/PexFaultLocalization`.
7. Put study project web. `https://sites.google.com/site/putstudy/`.
8. SearchCode code search. `https://searchcode.com`.
9. Theories in JUnit. `hhttps://github.com/junit-team/junit/wiki/Theories`.
10. M. Barnett, M. Fähndrich, P. de Halleux, F. Logozzo, and N. Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *Proc. 31st International Conference on Software Engineering*, pages 401–402, 2009.
11. C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering*. ACM, 2015.
12. P. Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 100–113. Springer, 2006.
13. H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *FM 2014: Formal Methods*, pages 230–246. Springer, 2014.
14. G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proc. 2011 International Symposium on Software Testing and Analysis*, pages 364–374, 2011.
15. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.
16. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.
18. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
19. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.
20. Microsoft. Generate smart unit tests for your code. Online. `https://msdn.microsoft.com/library/dn823749`, 2015.
21. D. Saff. Theory-infected: or how i learned to stop worrying and love universal quantification. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 846–847. ACM, 2007.
22. T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 596–607. ACM, 2014.
23. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.
24. S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In *Fundamental Approaches to Software Engineering*, pages 294–309. Springer, 2011.

25. N. Tillmann and J. De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.
26. N. Tillmann, J. de Halleux, and T. Xie. Parameterized unit testing: Theory and practice. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 483–484, New York, NY, USA, 2010. ACM.
27. N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 385–396, 2014.
28. N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.
29. T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mutation analysis of parameterized unit tests. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 177–181. IEEE, 2009.