

A Characteristic Study of Parameterized Unit Tests in .NET Open Source Projects

Wing Lam

University of Illinois at Urbana-Champaign, USA
winglam2@illinois.edu

Siwakorn Srisakaokul

University of Illinois at Urbana-Champaign, USA
srisaka2@illinois.edu

Blake Bassett

University of Illinois at Urbana-Champaign, USA
rbasset2@illinois.edu

Peyman Mahdian

University of Illinois at Urbana-Champaign, USA
mahdian2@illinois.edu

Tao Xie

University of Illinois at Urbana-Champaign, USA
taoxie@illinois.edu

Pratap Lakshman

Microsoft, India
pratapl@microsoft.com

Jonathan de Halleux

Microsoft Research, USA
jhalleux@microsoft.com

Abstract

In the past decade, parameterized unit testing has emerged as a promising method 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. The use of PUTs can enable powerful test generation tools such as Pex to have strong test oracles to check against, beyond just uncaught runtime exceptions. In addition, PUTs have been popularly supported by various unit testing frameworks for .NET and the JUnit framework for Java. 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, we first present categorization results of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We then use the categorization results to guide the design of the first characteristic study of PUTs in .NET open source projects. We study hundreds of PUTs 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.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Parameterized unit testing, automated test generation, unit testing



© Wing Lam, Siwakorn Srisakaokul, Blake Bassett, Peyman Mahdian, Tao Xie, Pratap Lakshman, and Jonathan de Halleux;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 5; pp. 5:1–5:28



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2018.5

45 **Acknowledgements** This work was supported in part by National Science Foundation under
46 grants no. CCF-1409423, CNS-1513939, and CNS1564274.

47 **1 Introduction**

48 With advances in test generation research such as dynamic symbolic execution [23, 35], pow-
49 erful test generation tools are now at the fingertips of software developers. For example,
50 Pex [37, 39], a state-of-the-art tool based on dynamic symbolic execution, has been shipped
51 as *IntelliTest* [32, 26] in Microsoft Visual Studio 2015 and 2017, benefiting numerous de-
52 velopers in software industry. Such test generation tools allow developers to automatically
53 generate input values for the code under test, comprehensively covering various program
54 behaviors and consequently achieving high code coverage. These tools help alleviate the
55 burden of extensive manual software testing, especially on test generation.

56 Although such tools provide powerful support for automatic test generation, when they
57 are applied directly to the code under test, only a predefined limited set of properties can be
58 checked. These predefined properties serve as test oracles for these automatically generated
59 input values, and violating these predefined properties leads to various runtime exceptions,
60 such as null dereferencing or division by zero. Despite being valuable, these predefined
61 properties are *weak test oracles*, which do not aim for checking functional correctness but
62 focus on robustness of the code under test.

63 To supply strong test oracles for automatically generated input values, developers can
64 write formal specifications such as code contracts [25, 30, 16] in the form of preconditions,
65 postconditions, and object invariants in the code under test. However, just like writing
66 other types of formal specifications, writing code contracts, especially postconditions, can
67 be challenging. According to a study on code contracts [34], 68% of code contracts are
68 preconditions while only 26% of them are postconditions (the remaining 6% are object
69 invariants). Section 2 shows an example of a method under test whose postconditions are
70 difficult to write.

71 In the past decade, parameterized unit testing [40, 38] has emerged as a practical alter-
72 native to specify program behaviors under test in the form of unit tests. Developers can
73 write parameterized unit tests (PUTs), unit-test methods with parameters, in contrast to
74 conventional unit tests (CUTs), without parameters. Then developers can apply an auto-
75 matic test generation tool such as Pex to automatically generate input values for a PUT's
76 parameters. Note that algebraic specifications [24] can be naturally written in the form of
77 PUTs but PUTs are not limited to being used to specify algebraic specifications.

78 Since parameterized unit testing was first proposed in 2005 [40], PUTs have been popu-
79 larly supported by various unit testing frameworks for .NET along with recent versions of
80 the JUnit framework (as parameterized tests [9] and theories [33, 13]). However, there exists
81 no study to offer insights on how PUTs are written by developers in development practices of
82 either proprietary or open source software, posing barriers for various stakeholders to bring
83 PUTs to widely adopted practices in software industry. Example stakeholders are current or
84 prospective PUT writers (e.g., developers), PUT framework designers, test-generation tool
85 vendors, testing researchers, and testing educators.

86 To address the lack of studies on PUTs, we first conduct a categorization of 93 Microsoft
87 MSDN Pex Forum posts [31] (contributed primarily by industrial practitioners) related
88 to parameterized unit tests. We then use the categorization results to guide the design

89 of the first characteristic study of PUTs in .NET open source projects (with a focus on
 90 PUTs written using the Pex framework, given that Pex is one of the most widely used
 91 test generation tools in industry [39]). Our findings from the categorization results of the
 92 forum posts show the following top three PUT-related categories that developers are most
 93 concerned with:

- 94 1. “Assumption/Assertion/Attribute usage” problems, which involve the discussion of us-
 95 ing certain PUT assumptions, assertions, and attributes to address the issues faced by
 96 developers, are the most popular category of posts (occupying 23 of the 93 posts).
- 97 2. “Non-primitive parameters/object creation” problems, which involve the discussion of
 98 generating objects for PUTs with parameters of a non-primitive type, are the second
 99 most popular category of posts (occupying 17 of the 93 posts).
- 100 3. “PUT concept/guideline” problems, which involve the discussion of the PUT concept
 101 and general guidelines for writing good PUTs, are the third most popular category of
 102 posts (occupying 11 of the 93 posts).

103 Upon further investigation into these top PUT-related categories, we find that developers
 104 in general are concerned with when and what assumptions, assertions, and attributes they
 105 should use when they are writing PUTs. We find that a significant number of forum posts are
 106 directly related to how developers should replace hard-coded method sequences with non-
 107 primitive parameters of PUTs. We also find that developers often question what patterns
 108 their PUTs should be written in. Using our categorization and investigation results, we
 109 formulate three research questions and answer these questions using 11 open-source projects,
 110 which contain 741 PUTs.

111 In particular, we investigate the following three research questions and attain correspond-
 112 ing findings:

- 113 1. **What are the extents and the types of assumptions, assertions, and attributes
 114 being used in PUTs?** We present a wide range of assumption, assertion, and at-
 115 tribute types used by developers as shown in Tables 3a, 3b, and 5, and tool vendors or
 116 researchers can incorporate this data with their tools to better infer assumptions, asser-
 117 tions, and attributes to assist developers. For example, tool vendors or researchers who
 118 care about the most commonly used assumptions should focus on `PexAssumeUnderTest`
 119 or `PexAssumeNotNull`, since these two are the most commonly used assumptions. Lastly,
 120 based on the studied PUTs, we find that increasing the default value of attributes as
 121 suggested by tools such as Pex rarely contributes to increased code coverage. Tool ven-
 122 dors or researchers should aim to improve the quality of the attribute recommendations
 123 provided by their tools, if any are provided at all.
- 124 2. **How often can hard-coded method sequences in PUTs be replaced with non-
 125 primitive parameters and how useful is it to do so?** There are a significant number
 126 of receiver objects in the PUTs (written by developers) that could be promoted to non-
 127 primitive parameters, and a significant number of existing non-primitive parameters that
 128 lack factory methods (i.e., methods manually written to help the tools generate desirable
 129 object states for non-primitive parameters). It is worthwhile for tool researchers or
 130 vendors to provide effective tool support to assist developers to promote these receiver
 131 objects (resulted from hard-coded method sequences), e.g., inferring assumptions for
 132 a non-primitive parameter promoted from hard-coded method sequences. Additionally,
 133 once hard-coded method sequences are promoted to non-primitive parameters, developers
 134 can also use assistance in writing more factory methods for such parameters.
- 135 3. **What are common design patterns and bad code smells of PUTs?** By under-
 136 standing how developers write PUTs, testing educators can teach developers appropriate

ways to improve PUTs. For example, developers should consider splitting PUTs with multiple conditional statements into separate PUTs each covering a case of the conditional statements. Doing so makes the PUTs easier to understand and eases the effort to diagnose the reason for test failures. Tool vendors and researchers can also incorporate this data with their tools to check the style of PUTs for suggesting how the PUTs can be improved. For example, checking whether a PUT contains conditionals, contains hard-coded test data, and contains duplicate test code, etc. often accurately identifies a PUT that can be improved.

In summary, this paper makes the following major contributions:

- The categorization of the Microsoft MSDN Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs.
- The first characteristic study of PUTs in open source projects, with a 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 PUTs and a suite of tools for analyzing PUTs (both are used for our study and are released on our project website [10]). These PUTs and analysis tools can be used by the community to conduct future empirical studies or to evaluate enhancements to automated test generation tools.

The work in this paper is part of the efforts of our industry-academia team (including university/industrial testing researchers and tool vendors) for bringing parameterized unit testing to broad industrial practices of software development. To understand how automatic test generation tools interact with PUTs, we specifically study PUTs written with the Pex framework. Besides the Pex framework, other .NET frameworks such as NUnit also support PUTs. In recent years, PUTs are also increasingly adopted among Java developers, partly due to the inclusion of parameterized test [9] and theories [33, 13] in the JUnit framework. However, unlike the Pex framework, these other frameworks lack powerful test generation tools such as Pex to support automatic generation of tests with high code coverage, and part of our study with PUTs, specifically the part described in Section 5, does investigate the code coverage of the input values automatically generated from PUTs.

The remainder of this paper is organized as follows. Section 2 presents an example of parameterized unit testing. Section 3 discusses the categorization of Pex forum posts that motivates our study. Section 4 discusses the setup of our study. Section 5 presents our study findings and discusses the implications to stakeholders. Section 6 discusses threats to validity of our study. Section 7 presents our related work, and Section 8 concludes the paper.

2 Background

Consider the method under test from the open source project of NUnit Console [6] in Figure 1. One way to supply strong test oracles for automatically generated input values is to write preconditions and postconditions for this method under test. It is relatively easy to specify preconditions for the method as `(sn != null) && (sv != null)` but it is actually quite challenging to specify comprehensive postconditions to capture this method's intended behaviors. The reason is that this method's intended behaviors depend on the behaviors of all the method calls inside the `SaveSetting` method. In order to write postconditions for `SaveSetting`, we would need to know the postconditions of the other method calls in

```

1  public class SettingsGroup {
2      private Hashtable storage = new Hashtable();
3      public event SettingsEventHandler Changed;
4      public void SaveSetting(string sn, object sv) {
5          object ov = GetSetting(settingName);
6          //Avoid change if there is no real change
7          if(ov != null) {
8              if((ov is string && sv is string && (string)ov == (string)sv) ||
9                 (os is int && sv is int && (int)ov == (int)sv) ||
10                 (os is bool && sv is bool && (bool)ov == (bool)sv) ||
11                 (os is Enum && sv is Enum && ov.Equals(sv)))
12                  return;
13          }
14          storage[settingName] = settingValue;
15          if(Changed != null)
16              Changed(this, new SettingsEventArgs(sn));
17      }
18  }

```

■ **Figure 1** SaveSetting method under test from the SettingsGroup class of NUnit Console [6].

182 SaveSetting (e.g., GetSetting) as well. In addition, the postconditions can be very long
 183 since there are many conditional statements with complex conditions (e.g., Lines 8-11). If a
 184 method contains loops, its postcondition may be even more difficult to write, since we would
 185 need to know the loop invariants and the postconditions may need to contain quantifiers.
 186 Thus, there is a need for a practical method to specify program behaviors under test in
 187 the form of unit tests. Specifying program behaviors in the form of unit tests can be easier
 188 since we do not need to specify all the intended behaviors of the method under test as a
 189 single logical formula. Instead, we can write test code to specify the intended behaviors of
 190 the method under test for a specific scenario (e.g., interacting with other specific methods).
 191 For example, a real-world conventional unit test (CUT) written by the NUnit developers
 192 is shown in Figure 2. The CUT in this figure checks that after we save a setting by call-
 193 ing the SaveSetting method, we should be able to retrieve the same setting by calling the
 194 GetSetting method. Despite seemingly comprehensive, the CUT in Figure 2 is insufficient,
 195 since it is unable to cover Lines 8-12 of the method in Figure 1. Figure 3 shows an additional
 196 CUT that developers can write to cover Lines 8-12; this additional CUT checks that saving
 197 the same setting twice does not invoke the Changed event handler twice. These two CUTs'
 198 corresponding, and more powerful, PUT is shown in Figure 4.

199 The beginning of the PUT (Lines 3-5) include PexAssume statements that serve as as-
 200 sumptions for the three PUT parameters. During test generation, Pex filters out all the
 201 generated input values (for the PUT parameters) that violate the specified assumptions.
 202 These assumptions are needed to specify the state of SettingsGroup that one may want to
 203 test. For example, according to Lines 2-3 in Figure 2, sg initially does not have "X" and
 204 "NAME" set. Thus, we need to add PexAssume.IsNull(st.Getting(sn)) (Line 5) to force Pex
 205 to generate only an object of SettingsGroup that satisfies the same condition as Lines 2-3
 206 in Figure 2. Otherwise, without such assumptions, the input values generated by Pex may
 207 largely be of no interest to the developers. The PexAssert statements in Lines 7 and 10
 208 are used as the assertions to be verified when running the generated input values. More
 209 specifically, the assumption on Line 5 and the assertion on Line 7 in the PUT correspond
 210 to Lines 2-3 and Lines 6-7, respectively, in the CUT from Figure 2. Lines 8-9 in the PUT
 211 then cover the case of calling the SaveSetting method twice with the same parameters as
 212 accomplished in the CUT shown in Figure 3. Note that writing the PUT allows the test to
 213 be more general as variable sn can be any arbitrary string, better than hard-coding it to be
 214 only "X" or "NAME" (as done in the CUTs).

215 A PUT is annotated with the [PexMethod] attribute, and is sometimes attached with

```

1 public void SaveAndLoadSettings() {
2     Assert.IsNull(sg.GetSetting("X"));
3     Assert.IsNull(sg.GetSetting("NAME"));
4     sg.SaveSetting("X", 5);
5     sg.SaveSetting("NAME", "Charlie");
6     Assert.AreEqual(5, sg.GetSetting("X"));
7     Assert.AreEqual("Charlie", sg.GetSetting("NAME"));
8 }

```

■ **Figure 2** A real-world CUT for the method in Figure 1.

```

1 public void SaveSettingsWhenSettingIsAlreadyInitialized() {
2     Assert.IsNull(sg.GetSetting("X"));
3     sg.SaveSetting("X", 5);
4     sg.SaveSetting("X", 5);
5     // Below assert that Changed only got invoked once in SaveSetting
6     ...
7 }

```

■ **Figure 3** An additional CUT for the method in Figure 1 to cover the lines that the CUT in Figure 2 does not cover.

```

1 [PexMethod(MaxRuns = 200)]
2 public void TestSave1(SettingsGroup sg, string sn, object sv) {
3     PexAssume.IsTrue(sg != null && sg.Changed != null);
4     PexAssume.IsTrue(sn != null && sv != null);
5     PexAssume.IsNull(sg.GetSetting(sn));
6     sg.SaveSetting(sn, sv);
7     PexAssert.AreEqual(sv, sg.GetSetting(sn));
8     sg.SaveSetting(sn, sv);
9     // Below assert that Changed only got invoked once in SaveSetting
10    ...
11 }

```

■ **Figure 4** The PUT corresponding to the CUTs in Figures 2 and 3.

216 optional attributes to provide configuration options for automatic test generation tools.
 217 An example attribute is `[PexMethod(MaxRuns = 200)]` as shown in Figure 4. The `MaxRuns`
 218 attribute along with the attribute value of 200 indicates that Pex can take a maximum
 219 of 200 runs/iterations during Pex’s path exploration phase for test generation. Since the
 220 default value of `MaxRuns` is 1000, setting the value of `MaxRuns` to be just 200 decreases the
 221 time that Pex may take to generate input values. Note that doing so may also cause Pex to
 222 generate fewer input values.

223 3 Categorization of Forum Posts

224 This section presents our categorization results of the Microsoft MSDN Pex Forum posts [31]
 225 related to parameterized unit tests. As of January 10th, 2018, the forum includes 1,436 posts
 226 asked by Pex users around the world. These users are primarily industrial practitioners. To
 227 select the forum posts related to parameterized unit tests, we search the forum with each
 228 of the keywords “parameterized”, “PUT”, and “unit test”. Searching the forum with these
 229 three keywords returns 14, 18, and 243 posts, respectively. We manually inspect each of
 230 these returned posts to select only posts that are actually related to parameterized unit tests.
 231 Finally among the returned posts, we identify 93 posts as those related to parameterized
 232 unit tests. Then we categorize these 93 posts into 8 major categories and one miscellaneous
 233 category, as shown in Table 1. The categorization details of the 93 posts can be found on
 234 our project website [10]. We next describe each of these categories and the number of posts
 235 falling into each category.

236 The posts falling into the top 1 category “assumption/assertion/attribute usage” (25%
 237 of the posts) involve discussion of using certain PUT assumptions, assertions, and attributes

■ **Table 1** Categorization results of the Microsoft MSDN Pex Forum posts related to parameterized unit tests.

Category	#Posts
Assumption/Assertion/Attribute usage	25% (23/93)
Non-primitive parameters/object creation	18% (17/93)
PUT concept/guideline	12% (11/93)
Test generation	11% (10/93)
PUT/CUT relationship	9% (8/93)
Testing interface/generic class/abstract class	6% (6/93)
Code contracts	5% (5/93)
Mocking	5% (5/93)
Miscellaneous	9% (8/93)
Total	100% (93/93)

238 to address the issues faced by PUT users. The posts falling into the second most popular
 239 category “non-primitive parameters/object creation” (18% of the posts) involve discussion
 240 of generating objects for PUTs with non-primitive-type parameters, one of the two major
 241 issues [42] for Pex to generate input values for PUTs. The posts falling into category “PUT
 242 concept/guideline” (12% of the posts) involve discussion of the PUT concept and general
 243 guideline for writing good PUTs. The posts falling into category “test generation” (11%
 244 of the posts) involve discussion of Pex’s test generation for PUTs. The posts falling into
 245 category “PUT/CUT relationship” (9% of the posts) involve discussion of co-existence of
 246 both CUTs and PUTs for the code under test. The posts falling into category “testing
 247 interface/generic class/abstract class” (6% of the posts) involve discussion of writing PUTs
 248 for interfaces, generic classes, or abstract classes. The posts falling into category “code
 249 contracts” (5% of the posts) involve discussion of writing PUTs for code under test equipped
 250 with code contracts [25, 30, 16]. The posts falling into category “mocking” (5% of the posts)
 251 involve discussion of writing mock models together with PUTs. The miscellaneous category
 252 (9% of the posts) includes those other posts that cannot be classified into one of the 8 major
 253 categories.

254 We use the posts from the top 3 major categories to guide our study design described in
 255 the rest of the paper, specifically with research questions RQ1-RQ3 listed in Section 5. In
 256 particular, our study focuses on quantitative aspects of assumption, assertion, and attribute
 257 usage (top 1 category) in RQ1, non-primitive parameters/object creation (top 2 category)
 258 in RQ2, and PUT concept/guideline (top 3 category) in RQ3.

259 4 Study Setup

260 This section describes our process for collecting subjects (e.g., open source projects contain-
 261 ing PUTs) and the tools that we develop to collect and process data from the subjects. The
 262 details of these subjects and our tools can be found on our project website [10].

263 4.1 Subject-collection Procedure

264 The subject-collection procedure (including subject sanitization) is a multi-stage process. At
 265 a coarse granularity, this process involves (1) comprehensive and extensive subject collection
 266 from searchable online source code repositories, (2) deduplication of subjects obtained multi-
 267 ple times from different repositories, and (3) verification of developer-written parameterized

268 unit tests (e.g., filtering out subjects containing only automatically-generated parameterized
 269 test stubs).

270 For comprehensive collection of subjects, we query a set of widely known code search
 271 services. The used query is “`PexMethod Assert`”, requiring both “`PexMethod`” and “`Assert`”
 272 to appear in the source file of the search results. The two code search services that return
 273 non-empty results based on our search criteria are GitHub [4] and SearchCode [12]. For
 274 each code search service, we first search with our query, and then we extract the source
 275 code repositories containing the files in the search results. When a particular repository is
 276 available from multiple search services, we extract the version of the repository from the
 277 search service that has the most recent commit. Lastly, we manually verify that each of our
 278 source code repositories has at least one PUT with one or more parameters and one or more
 279 assertions.

280 4.2 Analysis Tools

281 We develop a set of tools to collect metrics from the subjects. We use Roslyn [5], the
 282 .NET Compiler Platform, to build our tools. These tools parse C# source files to produce
 283 an abstract syntax tree, which is traversed to collect information and statistics of interest.
 284 More specifically, the analysis tools statically analyze the C# source code in the .cs files of
 285 each subject. The outputs of the tools include but are not limited to the following: PUTs,
 286 PUTs with `if` statements, results in Tables 3 and 6, the number of assumption and assertion
 287 clauses, and attributes of the subjects’ PUTs. In general, the results that we present in the
 288 remainder of the paper are collected either directly with the analysis tools released on our
 289 website [10], manual investigation conducted by the authors, or a combination of the two
 290 (e.g., using the PUTs with `if` statements to manually categorize the number of PUTs that
 291 have unnecessary `if` statements).

292 4.3 Collected Subjects

293 In total, we study 77 subjects and retain only the subjects that contain at least 10 PUTs and
 294 are not used for university courses or academic research (e.g., creating PUTs to experiment
 295 with Pex’s capability of achieving high code coverage). This comprehensive list of subjects
 296 that we study can be found on our project website [10].

297 Table 2 shows the information on the subjects that contain at least 10 PUTs. We count
 298 a test method as a PUT if the test method is annotated with attribute “`PexMethod`” and has
 299 at least one parameter. Our detailed study for research questions focuses on subjects with
 300 at least 10 PUTs because a subject with fewer PUTs often includes occasional tryouts of
 301 PUTs instead of serious use of them for testing the functionalities of the open source project.
 302 Column 1 shows the name of each subject, and Columns 2-3 shows the number of PUTs
 303 and CUTs in each subject. Columns 4-6 show the number of the lines of production source
 304 code, PUTs and CUTs, respectively, in each subject. Columns 7-8 shows the percentage of
 305 statements covered in the project under test by the PUTs on which Pex is applied and by the
 306 CUTs of the subject. Column 9 shows the version of Pex a subject’s PUTs were written with.
 307 If a subject contains PUTs written with multiple versions of Pex, the most recent version of
 308 Pex used to write the subject’s PUTs is shown. Altogether, we identify 11 subjects with at
 309 least 10 PUTs, and these subjects contain a total of 741 PUTs. When we examine the profiles
 310 of the contributors to the subjects, we find that all but one of the subjects have contributors
 311 who work in industry. The remaining one subject, `PurelyFunctionalDataStructures`, referred
 312 to as PFDS in our tables, is developed by a graduate student imitating the algorithms in a

■ **Table 2** Subjects collected for our study.

Subject Name	#Methods		#LOC			Code Cov.		Pex Version
	PUT	CUT	Source	PUT	CUT	PUT	CUT	
Atom	240	297	127916	3570	3983	N/A	N/A	0.20.41218.2
BBCode	17	22	1576	188	219	84%	69%	0.94.0.0
ConcurrentList	23	57	315	243	645	51%	75%	0.94.0.0
Functional-dotnet	41	87	14002	355	1666	N/A	N/A	0.15.40714.1
Henoch	63	149	4793	142	2816	N/A	N/A	0.94.0.0
OpenMheg	45	6	21809	382	100	N/A	N/A	0.6.30728.0
PFDS	10	2	1818	120	34	50%	12%	0.93.0.0
QuickGraph	205	123	38530	1478	2186	5%	50%	0.94.0.0
SerialProtocol	34	0	7603	269	0	49%	0%	0.94.0.0
Shweet	12	42	2481	295	703	N/A	N/A	0.91.50418.0
Utilities-net	51	0	3224	475	0	26%	0%	0.94.0.0
Total	741	785	223158	7496	12352	-	-	-
Average	67	71	22174	681	1123	52%	41%	-

313 data structure textbook. The table shows the percentage of statements covered for only 5
 314 out of 11 subjects because we have difficulties compiling the other subjects (e.g., a subject
 315 misses some dependencies). Part of our future work is to debug the remaining subjects
 316 so that we can compile them. More details about the subjects (e.g., the contributors of
 317 the subjects, the number of public methods in the subjects) can be found on our project
 318 website [10].

319 5 Study Results

320 Our study is based on forum posts asked by Pex users around the world as detailed in Sec-
 321 tions 5.1 to 5.3. Our study findings aim to benefit various stakeholders such as current
 322 or prospective PUT writers (e.g., developers), PUT framework designers, test-generation
 323 tool vendors, testing researchers, and testing educators. In particular, our study intends to
 324 address the following three main research questions:

- 325 ■ **RQ1:** What are the extents and the types of assumptions, assertions, and attributes
 326 being used in PUTs?
 - 327 ■ We address RQ1 because addressing it can help understand developers' current prac-
 328 tice of writing assumptions, assertions, and attributes in PUTs, and better inform
 329 stakeholders future directions on providing effective tool support or training on writ-
 330 ing assumptions, assertions, and attributes in PUTs.
- 331 ■ **RQ2:** How often can hard-coded method sequences in PUTs be replaced with non-
 332 primitive parameters and how useful is it to do so?
 - 333 ■ We address RQ2 because addressing it can help understand the extent of writing
 334 sufficiently general PUTs (e.g., promoting an object produced by a method sequence
 335 hard-coded in a PUT to a non-primitive parameter of the PUT) to fully leverage
 336 automatic test generation tools.
- 337 ■ **RQ3:** What are common design patterns and bad code smells of PUTs?
 - 338 ■ We address RQ3 because addressing it can help understand how developers are cur-
 339 rently writing PUTs and identify better ways to write good PUTs.

340 5.1 RQ1. Assumptions, Assertions, and Attributes

■ Table 3

(a) Different types of assumptions in subjects.

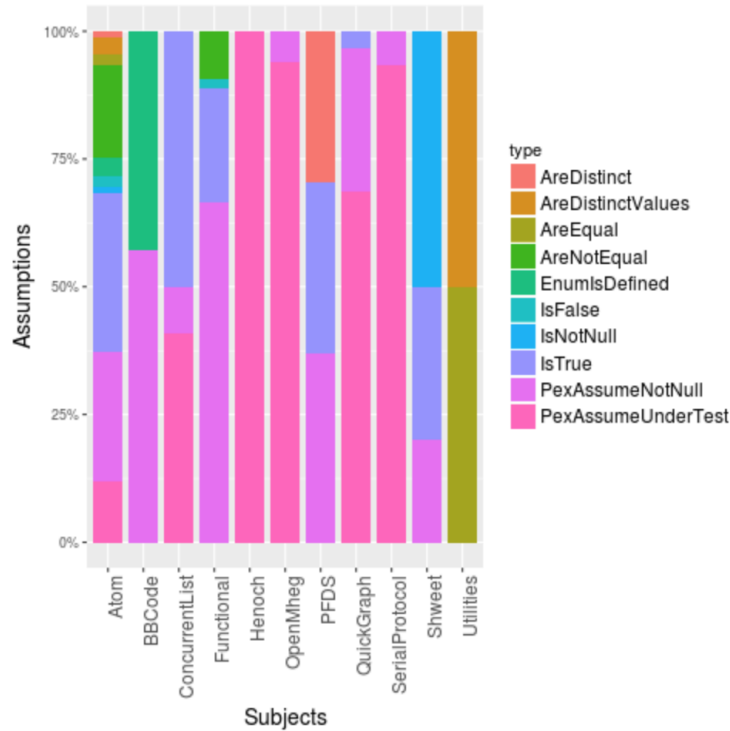
(b) Different types of assertions in subjects.

PexAssume Type	#	#NC	PexAssert Type	#	#NC
PexAssumeUnderTest	273	273	AreEqual	355	0
PexAssumeNotNull	211	211	IsTrue	199	2
IsTrue	158	2	IsFalse	75	3
AreNotEqual	73	0	Inconclusive	43	0
EnumIsDefined	22	0	IsNotNull	26	26
AreDistinct	13	0	Equal	21	1
AreDistinctValues	13	0	TrueForAll	19	0
IsNotNull	10	10	That	17	0
IsFalse	9	0	AreElementsEqual	16	0
AreEqual	9	0	IsNull	9	9
TrueForAll	7	2	AreNotEqual	5	0
IsNotNullOrEmpty	4	4	Fail	5	0
Fail	4	0	Throws	5	0
InRange	3	0	AreBehaviorsEqual	4	0
AreElementsNotNull	1	1	ImpliesIsTrue	3	0
Total	810	503	FALSE	3	0
Null Check Percentage	62% (503/810)		TRUE	3	0
			Empty	2	0
			Implies	2	0
			Contains	1	0
			DoesNotContain	1	0
			ReachEventually	1	0
			Total	815	41
			Null Check Percentage	5% (41/815)	

341 To understand developers' practices of writing assumptions, assertions, and attributes in
342 PUTs, we study our subjects' common types of assumptions, assertions, and attributes. Our
343 study helps provide relevant insights to the posts from the Assumption/Assertion/Attribute
344 usage category described in Section 3. For example, the original poster of the forum post
345 titled "New to Unit Testing" questions what type of assertions she/he should use. Another
346 forum post titled "Do I use NUnit Assert or PexAssert inside my PUTs?" reveals that the
347 original poster does not understand when and what assumptions to use.

348 5.1.1 Assumption Usage

349 As shown in Table 3a, `PexAssumeUnderTest` is the most common type of assumption, used
350 273 times in our subjects. `PexAssumeUnderTest` marks parameters as non-null and to be
351 that precise type. The second most common type of assumption, `PexAssumeNotNull`, is used
352 211 times. Similar to `PexAssumeUnderTest`, `PexAssumeNotNull` marks parameters as non-null
353 except that it does not require their types to be precise. Both `PexAssumeUnderTest` and
354 `PexAssumeNotNull` are specified as attributes of parameters, but they are essentially a conve-



■ **Figure 5** Assumption-type distribution for each of our subjects.

355 nient alternative to specifying assumptions (e.g., the use of attribute `PexAssumeNotNull` on
 356 a parameter `X` is the same as `PexAssume.IsNotNull(X)`). Since PUTs are commonly written
 357 to test the behavior of non-null objects as the class under test or use non-null objects as
 358 arguments to a method under test, it is reasonable that the common assumption types used
 359 by developers are ones that mark parameters as non-null. Figure 5 shows that the combina-
 360 tion of `PexAssumeUnderTest`, `PexAssumeNotNull`, and `IsNotNull`, which are for nullness
 361 checking, appears the most in all of our subjects. Note that Figure 5 contains only the top
 362 10 commonly used assumption types in our subjects. Furthermore, according to the last row
 363 of Tables 3a and 3b, developers perform null checks much more frequently for assumptions
 364 than assertions. Our findings about the frequency of assumption types and assertion types
 365 that check whether objects are null are similar to the findings of a previous study [34] on
 366 how frequently preconditions and postconditions in code contracts are used to check whether
 367 objects are null. Similar to code contracts, we find that 62% of assumptions perform null
 368 checks while the study on code contracts finds that 77% (1079/1356) of preconditions per-
 369 form null checks. Our study also finds that 5% of assertions perform null checks while the
 370 study on code contracts finds that 43% (165/380) of postconditions perform null checks.
 371 Since assertions are validated at the end of a PUT and it is less often that code before the
 372 assertions manipulates or produces a null object, it is reasonable that assumptions check for
 373 null much more frequently than assertions do. For assumption and assertion types such as
 374 `TrueForAll`, developers’ low number of uses may be due to the unawareness of such types’
 375 existence. `TrueForAll` checks whether a predicate holds over a collection of elements. In our
 376 subjects, we find cases such as the one in Figure 6 where a collection is iterated over to check
 377 whether a predicate is true for all of its elements; instead, developers could have used the

```

1  [PexMethod]
2  public void GetEnumerator_WhenMatrixConvertedToEnumerable_IteratesOverAllElements<T>(
3      [PexAssumeNotNull]ObjectMatrix<T> matrix ) {
4      System.Collections.IEnumerable enumerable = matrix;
5      foreach(var item in enumerable.Cast<T>())
6      {
7          Assert.IsTrue( matrix.Contains( item ) );
8      }
9  }

```

■ **Figure 6** PUT (in Atom [1]) that could benefit from Pex’s `TrueForAll` assertion.

378 `TrueForAll` assumption or assertion. More specifically, the developers of the method in Fig-
 379 ure 6 could have replaced Lines 5-8 with `PexAssert.TrueForAll(enumerable.Cast<T>(), item`
 380 `=> matrix.Contains(item))`. It is important to note that in versions of Pex after 0.94.0.0,
 381 certain assumption and assertion types were removed (e.g., `TrueForAll`). However, as shown
 382 in Table 2, none of our subjects used versions of Pex after 0.94.0.0.

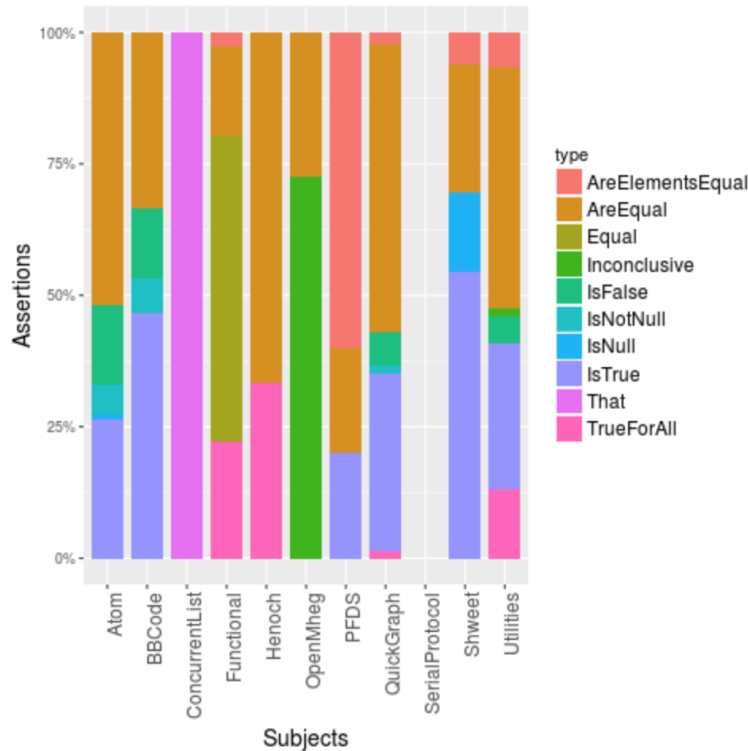
383 5.1.2 Assertion Usage

384 According to Figure 7, in all of the subjects except OpenMhég, the PUTs usually contain
 385 assertions for nullness or equality checking. Instead, OpenMhég’s assertions are mainly
 386 `Assert.Inconclusive`. `Assert.Inconclusive` is used to indicate that a test is still incomplete.
 387 From our inspection of the PUTs with `Assert.Inconclusive` in OpenMhég, we find that de-
 388 velopers write `Assert.Inconclusive("this test has to be reviewed")` in the PUTs. When
 389 we investigate the contents of these PUTs, we find that the developers indeed use these as-
 390 sertions to keep track of which tests are still incomplete. One example of OpenMhég’s PUT
 391 that contains `Assert.Inconclusive` is shown in Figure 8. The example is one of many PUTs
 392 in OpenMhég that create a new object but then do nothing with the object and contain
 393 no other assertions but `Assert.Inconclusive`. When we ignore all PUTs of OpenMhég that
 394 contain only `Assert.Inconclusive`, we find that the remaining assertions are similar to our
 395 other subjects in that most of them are for nullness or equality checking.

396 As shown in Table 4, the PFDS subject has the highest number of assume clauses per
 397 PUT method. Upon closer investigation of PFDS’s assume clauses, we find that these clauses
 398 are necessary because PUTs in PFDS test various data structures and the developers of
 399 PFDS have to specify assumptions for all of its PUTs to guide Pex to generate data-structure
 400 inputs that are not null and contain some elements. When we examine the assume clauses
 401 in Atom, the subject with the second highest number of assume clauses per PUT method,
 402 we also find similar cases. On the other hand, the Shweet subject has the highest number of
 403 assert clauses per PUT method. Shweet’s high number of assert clauses per PUT method
 404 can be attributed to the fact that the subject has multiple PUTs each of which contains
 405 around 8 assertions. The reason why some of Shweet’s PUTs each have around 8 assertions
 406 is that the subject’s PUTs test a web service, and the service returns 8 values every time
 407 it is triggered. Therefore, multiple of Shweet’s PUTs assert for whether these 8 values are
 408 correctly returned or not.

409 5.1.3 Attribute Usage

410 To investigate developers’ practices of configuring Pex via PUT attributes, we study the
 411 number and settings of attributes, as configuration options for running Pex, written by
 412 developers in PUTs. Our findings from the forum posts related to attributes suggest that
 413 developers are often confused on what attributes to use or how they should configure at-
 414 tributes. More specifically, 5 out of 23 of the Assumption/Assertion/Attribute usage forum



■ **Figure 7** Assertion-type distribution for each of our subjects.

```

1 [PexMethod]
2 public Content Constructor03(GenericContentRef genericContentRef) {
3     Content target = new Content(genericContentRef);
4     Assert.Inconclusive("this test has to be reviewed");
5     return target;
6 }

```

■ **Figure 8** PUT (in OpenMheg [7]) that contains `Assert.Inconclusive`.

posts involve an answer recommending the use of a particular attribute or configuring an attribute in a specific way. For example, a post titled “the test state was: path bounds exceeded - infinite loop” discusses how developers should set the `MaxBranches` attribute of Pex. The setting of `MaxBranches` controls the maximum number of branches taken by Pex along a single execution path.

The fourth column of Table 4 shows the average number of attributes added per PUT. The results show that developers add only 1 attribute for every 3-4 PUTs. Table 5 shows the number of attributes added for our subjects. Common attributes that developers add are `MaxRuns`, `MaxConstraintSolverTime`, and `MaxBranches`. The setting of `MaxRuns` controls the maximum number of runs before Pex terminates. Developers commonly set this attribute to be 100 runs when the default value is 1,000. Upon our inspection, most of the PUTs that use this attribute test methods related to inserting objects into a data structure. By setting the value of this attribute, developers make Pex terminate faster. In fact, 14 out of 18 attributes used in QuickGraph are `MaxRuns`.

`MaxConstraintSolverTime` is another type of attribute that some projects contain. The attribute controls the constraint solver’s timeout value during Pex’s exploration. By default,

■ **Table 4** Number of PexAssume clauses, PexAssert clauses, and Pex Attributes per PUT.

Subject Name	# of Assume Cl. / PUT	# of Assert Cl. / PUT	# of Attrs / PUT
Atom	1.72 (412/240)	1.71 (411/240)	0.07 (16/240)
BBCode	1.71 (29/ 17)	1.47 (25/ 17)	2.18 (37/ 17)
ConcurrentList	0.96 (22/ 23)	0.74 (17/ 23)	0.26 (6/ 23)
Functional-dotnet	1.39 (57/ 41)	1.24 (51/ 41)	0.17 (7/ 41)
Henoch	0.78 (49/ 63)	0.05 (3/ 63)	0.38 (24/ 63)
OpenMheg	0.76 (34/ 45)	1.29 (58/ 45)	0.00 (0/ 45)
PFDS	2.70 (27/ 10)	1.10 (11/ 10)	0.00 (0/ 10)
QuickGraph	0.91 (186/205)	0.85 (175/205)	0.10 (21/205)
SerialProtocol	0.44 (15/ 34)	0.00 (0/ 34)	0.00 (0/ 34)
Shweet	1.00 (12/ 12)	3.42 (41/ 12)	0.33 (4/ 12)
Utilities-net	0.18 (9/ 51)	1.37 (70/ 51)	0.00 (0/ 51)
Average	1.14	1.20	0.32

■ **Table 5** Different types of Pex attributes in our subjects' PUTs.

Pex Attribute Type	#
MaxBranches	36
MaxRuns	18
MaxConstraintSolverTime	12
MaxConditions	8
MaxRunsWithoutNewTests	6
MaxStack	5
Timeout	4
MaxExecutionTreeNodes	4
MaxWorkingSet	4
MaxConstraintSolverMemory	4
Total	101

431 `MaxConstraintSolverTime` is set to 10 seconds. Similar to `MaxRuns`, we find that developers
 432 often set the value to be lower than the default value so that Pex would finish sooner. For
 433 example, `BBCode` contains PUTs with `MaxConstraintSolverTime` set to 5 seconds, and `Atom`
 434 contains PUTs with `MaxConstraintSolverTime` set to 2 seconds.

435 In contrast to `MaxRuns`, we find that developers commonly set the value of `MaxBranches`
 436 to be higher than the default value. A common value set by developers is 20,000 when the
 437 default value is 10,000. When we study these PUTs, we find that the code tested by these
 438 PUTs all has loops, and the developers' intention when using this attribute is to increase
 439 the number of loop iterations allowed by Pex. For example, `ConcurrentList` contains several
 440 PUTs with `MaxBranches = 20000` set. When we run Pex without this attribute, Pex suggests
 441 to set `MaxBranches` to 20000. However, when we compare the code coverage with and without
 442 the attribute being set, we find that the code coverage does not increase with the attribute
 443 set. In fact, we find that when we manually unset all attributes of `ConcurrentList`, the code
 444 coverage does not change at all. The number of input values (generated by Pex) that exhibit
 445 a failed test result also does not change. Our findings indicate that increasing the default
 446 values of attributes often does not help increase the code coverage. In fact, for some of BB-

447 Code’s PUTs, its developers set 9 different attributes all to the value of 1,000,000,000. Based
448 on our estimation of running Pex on these PUTs, it would take approximately 2000 days for
449 Pex to terminate. When we run Pex with a time limit of three hours on BBCode’s PUTs
450 with the developer-specified attributes, we notice that the coverage increases marginally by
451 less than 1% compared to running Pex with the same time limit on BBCode’s PUTs without
452 any attributes.

453 5.1.4 Implications

454 With the wide range of assumption and assertion types used by developers as shown in
455 Tables 3a and 3b, tool vendors or researchers can incorporate this data with their tools
456 to better infer assumptions and assertions to assist developers. For example, tool vendors
457 or researchers who care about the most commonly used assumption types should focus
458 on `PexAssumeUnderTest` or `PexAssumeNotNull`, since these two are the most commonly used
459 assumption types. Lastly, based on our subjects’ PUTs, we find that increasing the default
460 value of attributes as suggested by tools such as Pex rarely contributes to increased code
461 coverage. Tool vendors or researchers should aim to improve the quality of the attribute
462 recommendations provided by their tools, if any are provided at all.

463 5.2 RQ2. Non-primitive Parameters

464 Typically developers are expected to avoid hard-coding a method sequence in a PUT to
465 produce an object used for testing the method under test. Instead, developers are expected
466 to promote such objects to a non-primitive parameter of the PUT. In this way, the PUT
467 can be made more general, to capture the intended behavior and enable an automatic
468 test generation tool such as Pex to generate objects of various states for the non-primitive
469 parameter. We find that 4 out of 17 answers from our non-primitive parameters/object
470 creation category of forum posts described in Section 3 are directly related to how developers
471 should replace hard-coded method sequences with non-primitive parameters. For example,
472 in a forum post titled “Can Pex Generate a `List<T>` for my PUT”, one of the answers to
473 the question is that the developer should write a PUT that takes `List` as a non-primitive
474 parameter instead of hard-coding a specific method sequence for producing a `List` object.
475 Doing so enables Pex to generate non-empty, non-null objects of that list. Since many of our
476 forum posts are related to how developers should replace hard-coded method sequences with
477 non-primitive parameters, we decide to study how frequently developers write PUTs with
478 non-primitive parameters and how often hard-coded method sequences in these PUTs could
479 be replaced with non-primitive parameters. More details about the forum posts specifically
480 related to this research question can be found on our project website [10].

481 5.2.1 Non-primitive Parameter Usage

482 As shown in Table 6, our result indicates that developers on average write non-primitive
483 parameters 59.0% of the time for the PUTs in our subjects. In other words, for every
484 10 parameters used by developers, 5-6 of those parameters are non-primitive. However,
485 developers write factory methods for only 17.9% of the non-primitive parameters used in
486 our subjects’ PUTs. The lack of non-primitive parameters and factory methods for such
487 parameters inhibits test generation tools such as Pex from generating high-quality input
488 values. For example, Figure 9 depicts 1 out of 16 PUTs that tests the `BinaryHeap` data
489 structure in the `QuickGraph` subject. Promoting the object that it is testing (`BinaryHeap`)

■ **Table 6** Statistics for factory methods and non-primitive parameters of our subjects. Average is calculated by dividing the sum of the two relevant columns (e.g., 59.0% is from the sum of Column 3 / the sum of Column 2).

Subject Name	Total Params	Non-prim Params	Non-prim / Params	Non-prim Params w/ Factory	w/ Factory / Non-prim Params
Atom	456	290	63.6%	66	22.8%
BBCode	33	9	27.3%	0	0.0%
ConcurrentList	16	0	0.0%	0	-
Functional-dotnet	50	5	10.0%	2	40.0%
Henoch	54	48	88.9%	0	0.0%
OpenMheg	75	55	73.3%	0	0.0%
PFDS	10	10	100.0%	0	0.0%
QuickGraph	125	111	88.8%	21	18.9%
SerialProtocol	51	21	41.2%	12	57.1%
Shweet	21	1	4.8%	0	0.0%
Utilities-net	66	15	22.7%	0	0.0%
Average			59.0%		17.9%

490 to a non-primitive parameter enables Pex to use factory methods such as the one depicted in
491 Figure 10 to generate high-quality input values. Without promoting the `BinaryHeap` object
492 to a parameter and using a factory method such as the one in Figure 10, the input values
493 generated by Pex with the 16 PUTs can cover only 13% of the code blocks in the `BinaryHeap`
494 class as opposed to 80% when the `BinaryHeap` object is promoted and a factory method is
495 provided for it. When developers do not promote non-primitive objects to a non-primitive
496 parameter or provide factory methods for it, the effectiveness of their tests really depends
497 on the values that the developers use to initialize the objects in their tests. For example, if
498 developers do not promote the `BinaryHeap` object to a parameter or provide factory methods
499 for it, then depending on the values that the developers would use to initialize the `BinaryHeap`
500 object, the code blocks covered by the 16 PUTs could actually range from 13% to 80% (the
501 same as that achieved by promoting the `BinaryHeap` object to a parameter and providing
502 a factory method for it). Promoting the `BinaryHeap` object to a parameter and providing
503 factory methods for it not only enable tools such as Pex to generate objects of `BinaryHeap`
504 that the developers may not have thought of themselves, but also alleviate the burden of
505 developers to choose the right values for their tests to properly exercise the code under
506 test. It is important to note that if we just promote the `BinaryHeap` object in the 16 PUTs
507 but do not provide a factory method for it, the percentage of code blocks covered by the
508 PUTs is 52%. Our findings here suggest that to enable tools such as Pex to generate input
509 values that cover the most code, it is desirable to promote non-primitive objects in PUTs to
510 non-primitive parameters and provide factory methods for such parameters. However, even
511 if no factory methods are provided, simply promoting non-primitive objects in PUTs may
512 already increase the code coverage achieved by the input values generated by tools such as
513 Pex.

```

1 [PexMethod(MaxRuns = 100)]
2 [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
3 public void InsertAndRemoveMinimum<TPriority, TValue>(
4     [PexAssumeUnderTest] BinaryHeap<TPriority, TValue> target,
5     [PexAssumeNotNull] KeyValuePair<TPriority, TValue>[] kvs)
6 {
7     var count = target.Count;
8     foreach (var kv in kvs)
9         target.Add(kv.Key, kv.Value);
10    TPriority minimum = default(TPriority);
11    for (int i = 0; i < kvs.Length; ++i)
12    {
13        if (i == 0)
14            minimum = target.RemoveMinimum().Key;
15        else
16        {
17            var m = target.RemoveMinimum().Key;
18            Assert.IsTrue(target.PriorityComparison(minimum, m) <= 0);
19            minimum = m;
20        }
21        AssertInvariant(target);
22    }
23    Assert.AreEqual(0, target.Count);
24 }

```

■ **Figure 9** InsertAndRemoveMinimum PUT from the BinaryHeapTest class of QuickGraph [11].

```

1 [PexFactoryMethod(typeof(BinaryHeap<int, int>))]
2 public static BinaryHeap<int, int> Create(int capacity)
3 {
4     var heap = new BinaryHeap<int, int>(capacity, (i, j) => i.CompareTo(j));
5     return heap;
6 }

```

■ **Figure 10** Factory method for the BinaryHeapTest class of QuickGraph [11].

5.2.2 Promoting Receiver Object

To determine how often developers could have replaced a hard-coded method sequence with a non-primitive parameter, we manually inspect each PUT to determine the number of them that could have had their receiver objects be replaced with a non-primitive parameter. We consider an object of a PUT to be a receiver object if the object directly or indirectly affects the PUT's assertions. The detailed results of our manual inspection effort can be found on our project website [10] under "PUT Patterns". As shown in Table 7, 95.7% (709/741) of the PUTs in our subjects have at least one receiver object. However, we find that 49.2% (349/709) of these PUTs with receiver objects do not have a parameter for the receiver objects, and 89.4% (312/349) of them can actually be modified so that all receiver objects in the PUT are promoted to PUT parameters. As shown in Table 8, we categorize the 349 PUTs whose receiver objects could be promoted into the following four different categories.

- (1) In 47.9% (167/349) of the PUTs, we can easily promote their receiver objects into a non-primitive parameter (e.g., removing the object creation line and adding a parameter).
- (2) In 41.5% (145/349) of the PUTs, their receiver objects are static (which cannot be instantiated).
- (3) In 9.7% (34/349) of the PUTs, they are testing their receiver objects' constructors.
- (4) In 1.6% (3/349) of the PUTs, they are testing multiple receiver objects with shared variables (e.g., testing the equals method of an object).

Of the PUTs belonging to the first category shown in Table 8, 33.0% (55/167) of them test specific object states. Figure 11 shows an example of a PUT that tests a specific object state. The developers of this PUT could have promoted `_list` and `element` to parameters and updated `index` accordingly before the assertion in Line 9. Figure 12 depicts a more general version of the PUT in Figure 11. Notice how the initial contents of the list and the element being added to the list are hard-coded in Figure 11 but not in Figure 12.

■ **Table 7** Statistics of PUTs with receiver objects (ROs).

Subject Name	# of PUTs w/ ROs	# of PUTs w/o promoted ROs	# of PUTs whose ROs should be promoted
Atom	90.4% (217/240)	59.4% (129/217)	98.4% (127/129)
BBCode	88.2% (15/ 17)	100.0% (15/ 15)	100.0% (15/ 15)
ConcurrentList	100.0% (23/ 23)	56.5% (13/ 23)	100.0% (13/ 13)
Functional-dotnet	85.4% (35/ 41)	91.4% (32/ 35)	100.0% (32/ 32)
Henoch	100.0% (63/ 63)	25.4% (16/ 63)	43.8% (7/ 16)
OpenMheg	100.0% (45/ 45)	25.0% (11/ 45)	18.2% (2/ 11)
PFDS	100.0% (10/ 10)	100.0% (10/ 10)	100.0% (10/ 10)
QuickGraph	99.5% (204/205)	20.1% (41/204)	73.2% (30/ 41)
SerialProtocol	100.0% (34/ 34)	55.9% (19/ 34)	68.4% (13/ 19)
Shweet	100.0% (12/ 12)	100.0% (12/ 12)	100.0% (12/ 12)
Utilities-net	100.0% (51/ 51)	100.0% (51/ 51)	100.0% (51/ 51)
Total	95.7% (709/741)	49.2% (349/709)	89.4% (312/349)

```

1 [PexMethod]
2 public void GetItem(int index) {
3     IList<int> _list = new ConcurrentList<int>();
4     PexAssume.IsTrue(index >= 0);
5     const int element = 5;
6     for (int i = 0; i < index; i++)
7         _list.Add(0);
8     _list.Add(element);
9     Assert.That(_list[index], Is.EqualTo(element));
10 }

```

■ **Figure 11** PUT testing a specific object state in ConcurrentList [2].

```

1 [PexMethod]
2 public void GetItem_Promoted(int index, IList<int> _list, int element) {
3     int size = _list.Count;
4     PexAssume.IsTrue(index >= 0);
5     for(int i = 0; i < index; i++)
6         _list.Add(0);
7     _list.Add(element);
8     index += size;
9     Assert.That(_list[index], Is.EqualTo(element));
10 }

```

■ **Figure 12** Version of the PUT in Figure 11 with receiver object promoted.

538 Upon further investigation, we find that the 145 PUTs in the second category shown in
539 Table 8 can and should actually be promoted by making the class under test not be static.
540 On the other hand, the PUTs that test their receiver objects' constructors have no need to
541 be improved by promotion. Lastly, the PUTs that test multiple receiver objects are best
542 left not promoted. In the end we find that the 167 PUTs in the first category (their receiver
543 objects can be easily promoted) and the 145 PUTs in the second category (their receiver
544 objects are static) are PUTs whose receiver objects could be promoted and they should
545 actually be promoted. These two categories of PUTs form the total of 89.4% (312/394) of
546 the PUTs that could be promoted and should be promoted. Promoting these objects enables
547 test generation tools such as Pex to use factory methods to generate different states of the
548 receiver objects (beyond specific hard-coded ones) for the PUTs.

549 Based on our promotion experiences, often the time, after we promote receiver objects
550 (resulted from hard-coded method sequences) to non-primitive parameters of PUTs, we need
551 to add assumptions to constrain the non-primitive parameters so that test generation tools

■ **Table 8** Categorization results of the PUTs whose receiver objects could be promoted.

Category	#PUTs
(1) Their receiver objects can be easily promoted	167 (47.9%)
(2) Their receiver objects are static	145 (41.5%)
(3) Testing their receiver objects' constructors	34 (9.7%)
(4) Testing multiple receiver objects with shared variables	3 (0.9%)
Total	349

```

1  [TestMethod]
2  public void GetItem_CUT()
3  {
4      GetItem_Promoted(0, null, 5);
5  }

```

■ **Figure 13** Example of a CUT generated from the PUT in Figure 12.

552 will not generate input values that are of no interest to developers. For example, for the
553 `GetItem_Promoted` PUT in Figure 12, one of the input values generated by Pex with this
554 PUT can be found in Figure 13. Although the value of `index` (0) from the `GetItem_CUT` in
555 Figure 13 is reasonable for both the `GetItem` and `GetItem_Promoted` PUTs and the value of
556 `element` (5) is reasonable for the `GetItem_Promoted` PUT, the additional value of `_list` (null)
557 is unreasonable. The value is unreasonable because the `GetItem` PUT is expected to test
558 adding various elements to `_list` but it is not expected to test the case when `_list` is null.
559 However, due to our promotion of `_list`'s hard-coded method sequence to a non-primitive
560 parameter, input values generated from `GetItem_Promoted` would actually test such a case.
561 In order for developers to prevent such nonsensical input values from being generated, the
562 developers would have to add the assumption of `PexAssume.IsNotNull(_list)` before Line 3 of
563 `GetItem_Promoted`. Such assumption writing can be time-consuming: essentially promoting
564 hard-coded method sequences to be non-primitive parameters and adding assumptions to
565 these parameters are going from specifying “how” (to generate specific object states) to
566 specifying “what” (specific object states need to be generated).

567 5.2.3 Implications

568 There are a significant number of receiver objects (in the PUTs written by developers)
569 that could be promoted to non-primitive parameters, and a significant number of existing
570 non-primitive parameters that lack factory methods. It is worthwhile for tool researchers
571 or vendors to provide effective tool support to assist developers to promote these receiver
572 objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a non-
573 primitive parameter promoted from hard-coded method sequences. Additionally, once hard-
574 coded method sequences are promoted to non-primitive parameters, developers can also use
575 assistance in writing effective factory methods for such parameters.

576 5.3 RQ3. PUT Design Patterns and Bad Smells

577 Our categorization of forum posts as described in Section 3 shows that 5 out of 11 of the
578 PUT concept/guideline posts discuss patterns in which PUTs should be written in. For
579 example, two of the posts titled “Assertions in PUT” and “PUT with PEX” involve answers
580 informing the original poster that assertions are typically necessary for PUTs. One such
581 forum post contains the following response: “You should write Asserts, in order to ensure

```

1 [PexMethod]
2 public void Clear<T>([PexAssumeUnderTest]ConcurrentList<T> target) {
3     target.Clear();
4 }

```

■ **Figure 14** PUT (in `ConcurrentList` [2]) that should be improved with assertions.

■ **Table 9** Categorization results of bad smells in PUTs

Category	#PUTs
(1) Code duplication	55
(2) Unnecessary conditional statement	39
(3) Hard-coded test data	37
Total	131

582 that the Function (`TestInvoice` in this case) really does what it is intended to do”. To better
 583 understand how developers write PUTs, we manually inspect all of the PUTs in our subjects
 584 to see what the common design patterns and bad smells are. The detailed results of our
 585 manual inspection effort can be found on our project website [10] under “PUT Patterns”.

586 5.3.1 PUT Design Patterns

587 We find that the majority of the PUTs are written in the following patterns: “AAA” (Triple-
 588 A) and Parameterized Stub. Triple-A is a well-known design pattern for writing unit tests [8].
 589 These tests are organized into three sections: setting up the code under test (Arrange),
 590 exercising the code under test (Act), and verifying the behavior of the code under test
 591 (Assert). On the other hand, a Parameterized Stub test is used to test the code under test
 592 that already contains many assertions (e.g., code equipped with code contracts [25, 30, 16]).
 593 In general, Parameterized Stub tests are easy to write and understand, since the test body
 594 is short and contains only a few method calls to the code under test. In our subjects,
 595 we find that 34.6% (270/741) and 32.1% (251/741) of the PUTs to exhibit the Triple-
 596 A and Parameterized Stub test pattern, respectively. Of the 251 PUTs that exhibit the
 597 Parameterized Stub pattern, we find that 74.5% (187/251) of them are PUTs that should
 598 be improved with assertions, given that the code under test itself does not contain any
 599 code-contract assertions or any other type of assertions. For example, the PUT in Figure 14
 600 contains only a single statement to test the robustness of the `Clear` method, which by itself
 601 does not contain any assertions. Developers of this PUT should at least add an assertion
 602 such as `Assert.That(target.Count, Is.EqualTo(0))`; to the end of the PUT to ensure that
 603 once `Clear` is invoked, then the number of elements in a `ConcurrentList` object will be 0.

604 Similar to the bad smells typically found in conventional unit tests [29], we consider the
 605 following three categories of bad smells in our PUTs: (1) code duplication, (2) unnecessary
 606 conditional statement, and (3) hard-coded test data. These three categories of bad smells
 607 can cause tests to be difficult to understand and maintain. Table 9 shows the number of
 608 PUTs containing each category of bad smells. Our analysis tools as described in Section 4.2
 609 assist our manual inspections of the PUTs by listing the PUTs that contain conditional
 610 statements or hard-coded test data (as arbitrary strings). Using these lists of PUTs, we
 611 then manually inspect each of these PUTs to determine whether it really has bad code
 612 smells. To determine code duplication, we manually compare every PUT with every other
 613 PUT of the same class. Next, we discuss each of the categories in detail.

```

1  [PexMethod]
2  public void GetItem(int index)
3  {
4      PexAssume.IsTrue(index >= 0);
5      const int element = 5;
6      for (int i = 0; i < index; i++)
7      {
8          _list.Add(0);
9      }
10     _list.Add(element);
11     Assert.That(_list[index], Is.EqualTo(element));
12 }

```

■ **Figure 15** PUT (from the `ConcurrentListHandWrittenTests` class of `ConcurrentList` [2]) that contains many lines of test-code duplication with another PUT named `SetItem` from the same class.

■ **Table 10** Categorization results of why conditional statements exist in PUTs.

Category	#PUTs
(1) Testing particular cases	16
(2) Forcing Pex to explore particular cases	9
(3) Testing different cases according to boolean conditions	9
(4) Unnecessary if statements	5
Total	39

614 5.3.2 Code Duplication in PUTs

615 Similar to conventional unit tests, PUTs also contain the bad smell of test-code duplication.
616 Test-code duplication is a poor practice because it increases the cost of maintaining tests.
617 Duplication often arises when developers clone tests and do not put enough thought into
618 how to reuse test logic intelligently. As the number of tests increases, it is important that
619 the developers either factor out commonly used sequences of statements into helper methods
620 that can be reused by various tests, or in the case of PUTs, consider merging the PUTs and
621 using assumptions/attributes to ensure that the specific cases being tested previously are
622 still tested. In our subjects' PUTs, we find that 7.4% (55/741) of them contain test-code du-
623 plication. In other words, for 55 of our subjects' PUTs, there exist another PUT (in the same
624 subject) that contains a significant amount of duplicate test code. One example of such PUT
625 is shown in Figure 15. The PUT in this example is from the `ConcurrentListHandWrittenTests`
626 class of `ConcurrentList` [2] and is almost identical to another PUT named `SetItem` in the
627 same class. More specifically, the only lines that differ between the two PUTs are Lines 6 and
628 10. For Line 6 the loop terminating condition is set to `i <= index` as opposed to `i < index`.
629 For Line 10, instead of adding an element with the `Add` method, the line is `_list[index] =`
630 `element;`. In .NET, the use of brackets and an index value to add elements to a collection is
631 enabled by Indexers [14]. Since the intention of the two PUTs is to test whether setting and
632 getting an element from a list of arbitrary size correctly set and get the correct element, the
633 two differences in Lines 6 and 10 between the two PUTs actually do not matter. Instead
634 of duplicating so many lines of test code, the developers of these two PUTs should just
635 delete one of them. Doing so will not only help decrease the cost for developers to maintain
636 the tests, but also to speed up the testing time, since there will be fewer tests that cover
637 the same parts of the code under test. Developers can also make use of existing tools for
638 detecting code clones [18, 19] to automatically help detect code duplication in PUTs.

```

1 IList<int> _list = new ConcurrentList<int>();
2 [PexMethod(MaxBranches = 20000)]
3 public void Clear(int count)
4 {
5     var numClears = 100;
6     var results = new List<int>(numClears * 2);
7     var numCpus = Environment.ProcessorCount;
8     var sw = Stopwatch.StartNew();
9     using (SaneParallel.For(0, numCpus, x =>
10    {
11        for (var i = 0; i < count; i++)
12            _list.Add(i);
13    }))
14    {
15        for (var i = 0; i < numClears; i++)
16        {
17            Thread.Sleep(100);
18            results.Add(_list.Count);
19            _list.Clear();
20            results.Add(_list.Count);
21        }
22    }
23    sw.Stop();
24    for (var i = 0; i < numClears; i++)
25        Console.WriteLine("Before/After Clear #{0}: {1}/{2}", i, results[i << 1], results[(i << 1) + 1]);
26    Console.WriteLine("ClearParallelSane took {0}ms", sw.ElapsedMilliseconds);
27    _list.Clear();
28    Assert.That(_list.Count, Is.EqualTo(0));
29 }

```

■ **Figure 16** PUT with hard-coded test data in the `SaneParallelTests` class of `ConcurrentList` [2].

639 5.3.3 Unnecessary Conditional Statements in PUTs

640 Typically developers are expected not to write any conditional statements in their tests,
641 because tests should be simple, linear sequences of statements. When a test has multiple
642 execution paths, one cannot be sure exactly how the test will execute in a specific case. In our
643 subjects, 7.0% (52/741) of the PUTs contain at least one conditional branch. To understand
644 why developers write PUTs with conditionals, we study whether the conditionals in these
645 PUTs are necessary and if they are not, why the developers write such conditionals in their
646 PUTs. We find that 25% (13/52) of the PUTs contain conditional statements that could not
647 be removed. These PUTs are typically testing the interactions of two or more operations
648 of the code under test (e.g., adding and removing from a data structure). The remaining
649 75.0% (39/52) of the PUTs with conditionals can have their conditionals removed or each
650 of these PUTs should be split into two or more PUTs. Table 10 shows the reasons for why
651 the conditionals of such PUTs should be removed and the number of PUTs for each of the
652 reasons. The PUTs in the first and second categories should replace their conditionals with
653 `PexAssume()` statements to force Pex to explore and test particular cases. The PUTs in
654 the third category should be each split into multiple PUTs each of which tests a different
655 case of the conditional. For the PUTs created from the third category, developers can use
656 `PexAssume()` statements in the new PUTs to filter out inputs that do not satisfy the boolean
657 conditions of the case that the new PUTs are responsible for. The PUTs in the last category
658 contain conditionals that can be removed with a slight modification to the test (e.g., some
659 conditionals in a loop can be removed by amending the loop and/or adding code before the
660 loop). The automatic detection and fixing of unnecessary conditional statements in PUTs
661 would be a valuable and challenging line of future work due to the following. There are
662 various reasons for why a PUT may have conditionals as shown in Table 10, and depending
663 on the reason why a PUT may have conditionals, the fix for removing the conditionals, if
664 removal is possible, can be quite different.

665 5.3.4 Hard-coded Test Data in PUTs

666 Another bad smell that we identify in our subjects' PUTs is hard-coded test data. This
667 smell can be problematic for three main reasons. (1) Tests are more difficult to understand.
668 A developer debugging the tests would need to look at the hard-coded data and deduce how
669 each value is related to another and how these values affect the code under test. (2) Tests are
670 more likely to be flaky [28, 22, 15]. A common reason for tests to be flaky is the reliance on
671 external dependencies such as databases, file system, and global variables. Hard-coded data
672 in these tests often lead to multiple tests modifying the same external dependency and these
673 modifications could cause these tests to fail unexpectedly. (3) Hard-coded test data prevent
674 automatic test generation tools such as Pex from generating high-quality input values. In
675 our subjects' PUTs, we find that 5.0% (37/741) of them use hard-coded test data. One
676 example of such PUT is shown in Figure 16. In this example, the developers are testing the
677 `Clear` method of the `ConcurrentList` object (`_list`). The PUT adds an arbitrary number of
678 elements to the `_list` object, clears the list, and records the number of elements in the list.
679 The process of adding and clearing the list repeats 100 times as decided by `numClears` on
680 Line 5. As far as we can tell, the developers arbitrarily choose the value of 100 for `numClears`
681 on Line 5. When we parameterize the `numClears` variable and add an assumption that the
682 variable should be between 1 and 1073741823 (to prevent `ArgumentOutOfRangeException`), we
683 find that the input values generated by Pex for the `numClears` variable to be 1 and 2. These
684 two values exercise the same lines of the `Clear` method just as the value of 100 would. The
685 important point here is that contrary to the developers' arbitrarily chosen value of 100, Pex
686 is able to systematically find that using just the values of 1 and 2 would already sufficiently
687 test the `Clear` method. That is, as we manually confirm, even if the developers devote more
688 computation time to testing the `Clear` method by setting `numClears` to 100, they would not
689 cover any additional code or find any additional test failures. Therefore, the developers of
690 this PUT should not hard code the test data, and instead they should parameterize the
691 `numClears` variable. Doing so would enable automatic test generation tools such as Pex to
692 generate high-quality input values that sufficiently test the code under test. Developers can
693 also make use of existing program analysis tools [41] to automatically detect whether certain
694 hard-coded test data may exist between multiple PUTs.

695 5.3.5 Implications

696 By understanding how developers write PUTs, testing educators can suggest ways to improve
697 PUTs. For example, developers should consider splitting PUTs with multiple conditional
698 statements into separate PUTs each covering a case of the conditional statements. Doing so
699 makes the developer's PUTs easier to understand and eases the effort to diagnose the reason
700 for test failures. Tool vendors and researchers can incorporate this data with their tools
701 to check the style of PUTs for better suggestions on how the PUTs can be improved. For
702 example, checking whether a PUT is a Parameterized Stub, contains conditionals, contains
703 hard-coded test data, and contains duplicate test code often correctly identifies a PUT that
704 can be improved.

705 6 Threats to Validity

706 There are various threats to validity in our study. We broadly divide the main threats into
707 internal and external validity.

708 6.1 Internal Validity

709 Threats to internal validity are concerned with the validity of our study procedure. Due
710 to the complexity of software, faults in our analysis tools could have affected our results.
711 However, our analysis tools are tested with a suite of unit tests, and samples of the results
712 are manually verified. Results from our manual analyses are confirmed by at least two of
713 the authors. Furthermore, we rely on various other tools for our study, such as dotCover [3]
714 to measure the code coverage of the input values generated by Pex. These tools could have
715 faults as well and consequently such faults could have affected our results.

716 6.2 External Validity

717 There are two main threats to external validity in our study.

- 718 1. We use the categorization of the Microsoft MSDN Pex Forum posts [31] to determine
719 the issues surrounding parameterized unit testing. These forum posts enable us and
720 the research community to access the issues of developers objectively and quantitatively,
721 but the issues identified from the posts may not be representative of all the issues that
722 developers encounter.
- 723 2. Our findings may not apply to subjects other than those that we study, especially since we
724 are able to find only 11 subjects matching the criteria defined in Section 4. Furthermore,
725 we primarily focus on projects using PUTs in the context of automated test generation,
726 so our findings from such subjects may not generalize to situations outside of this setting
727 (e.g., general usage of Theories [33] in Java). In addition, our analyses focus specifically
728 on subjects that contain PUTs written using the Pex framework, and the API differences
729 or idiosyncrasies of other frameworks may impact the applicability of our findings. All of
730 our subjects are written in C#, but vary widely in their application domains and project
731 sizes. Finally, all of our subjects are open source software, and therefore our findings
732 may not generalize to proprietary software.

733 7 Related Work

734 To the best of our knowledge, our characteristic study is the first on parameterized unit
735 testing in open source projects. In contrast, previous work focuses on proposing new tech-
736 niques for parameterized unit testing and does not provide any insight on the practices of
737 parameterized unit testing. For example, Xie et al. [43] propose a technique for assessing
738 the quality of PUTs using mutation testing. Thummalapenta et al. [36] propose manual
739 retrofitting of CUTs to PUTs, and show that new faults are detected and coverage is in-
740 creased after such manual retrofitting is conducted. Fraser et al. [21] propose a technique
741 for generating PUTs starting from concrete test inputs and results.

742 Our work is related to previous work on studying developer-written formal specifications
743 such as code contracts [16]. Schiller et al. [34] conduct case studies on the use of code
744 contracts in open source projects in C#. They analyze 90 projects using code contracts and
745 categorize their use of various types of specifications, such as null checks, bound checks, and
746 emptiness checks. They find that checks for nullity and emptiness are the most common
747 types of specifications. Similarly we find that the most common types of PUT assumptions
748 are also used for nullness specification. However, the most common types of PUT assertions
749 are used for equality checking instead of null and emptiness.

750 Estler et al. [20] study code contract usage in 21 open source projects using JML [27]
751 in Java, Design By Contract in Eiffel [30], and code contracts [16] in C#. Their study

752 also includes an analysis of the change in code contracts over time, relative to the change
753 in the specified source code. Their findings agree with Schiller's on the majority use of
754 nullness code contracts. Furthermore, Chalin [17] studies code contract usage in over 80
755 Eiffel projects. They show that programmers using Eiffel tend to write more assertions than
756 programmers using any other languages do.

757 **8 Conclusion**

758 To fill the gap of lacking studies of PUTs in development practices of either proprietary
759 or open source software, we have presented categorization results of the Microsoft MSDN
760 Pex Forum posts (contributed primarily by industrial practitioners) related to PUTs. We
761 then use the categorization results to guide the design of the first characteristic study of
762 parameterized unit testing in open source projects. Our study involves hundreds of PUTs
763 that open source developers write for various open source projects.

764 Our study findings provide the following valuable insights for various stakeholders such
765 as current or prospective PUT writers (e.g., developers), PUT framework designers, test-
766 generation tool vendors, testing researchers, and testing educators.

- 767 1. We have studied the extents and types of assumptions, assertions, and attributes being
768 used in PUTs. Our study has identified assumption and assertion types that tool ven-
769 dors or researchers can incorporate with their tools to better infer assumptions and
770 assertions to assist developers. For example, tool vendors or researchers who care
771 about the most commonly used assumption types should focus on `PexAssumeUnderTest` or
772 `PexAssumeNotNull`, since these two are the most commonly used assumption types. We
773 have also found that increasing the default value of attributes as suggested by tools such
774 as Pex rarely contributes to increased code coverage. Tool vendors or researchers should
775 aim to improve the quality of the attribute recommendations provided by their tools, if
776 any are provided at all.
- 777 2. We have studied how often hard-coded method sequences in PUTs can be replaced with
778 non-primitive parameters and how useful it is for developers to do so. Our study has
779 found that there are a significant number of receiver objects in the PUTs written by de-
780 velopers that could be promoted to non-primitive parameters, and a significant number
781 of existing non-primitive parameters that lack factory methods. Tool researchers or ven-
782 dors should provide effective tool support to assist developers to promote these receiver
783 objects (resulted from hard-coded method sequences), e.g., inferring assumptions for a
784 non-primitive parameter promoted from hard-coded method sequences. Additionally,
785 once hard-coded method sequences are promoted to non-primitive parameters, develop-
786 ers can also use assistance in writing effective factory methods for such parameters.
- 787 3. We have studied the common design patterns and bad smells in PUTs, and have found
788 that there are a number of patterns that often correctly identify a PUT that can be
789 improved. More specifically, checking whether a PUT is a Parameterized Stub, con-
790 tains conditionals, contains hard-coded test data, and contains duplicate test code often
791 correctly identifies a PUT that can be improved. Tool vendors and researchers can in-
792 corporate this data with their tools to check the style of PUTs for better suggestions on
793 how these PUTs can be improved.

794 The study is part of our ongoing industry-academia team efforts for bringing parameterized
795 unit testing to broad industrial practices of software development.

796 ——— **References** ———

- 797 **1** Atom. URL: <https://github.com/tivtag/Atom>.
- 798 **2** ConcurrentList. URL: <https://github.com/damageboy/ConcurrentList>.
- 799 **3** dotCover. URL: <https://www.jetbrains.com/dotcover>.
- 800 **4** GitHub code search. URL: <https://github.com/search>.
- 801 **5** The .NET compiler platform Roslyn. URL: <https://github.com/dotnet/roslyn>.
- 802 **6** NUnit Console. URL: <https://github.com/nunit/nunit-console>.
- 803 **7** OpenMheg. URL: <https://github.com/orryverducci/openmheg>.
- 804 **8** Parameterized Test Patterns for Microsoft Pex). URL: <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.216.282>.
- 805 **9** Parameterized tests in JUnit. URL: <https://github.com/junit-team/junit/wiki/Parameterized-tests>.
- 806 **10** PUT study project web. URL: <https://sites.google.com/site/putstudy>.
- 807 **11** QuickGraph. URL: <https://github.com/tathanhdinh/QuickGraph>.
- 808 **12** SearchCode code search. URL: <https://searchcode.com>.
- 809 **13** Theories in JUnit. URL: <https://github.com/junit-team/junit/wiki/Theories>.
- 810 **14** Using Indexers (C# Programming Guide). URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/using-indexers>.
- 811 **15** Stephan Arlt, Tobias Morciniec, Andreas Podelski, and Silke Wagner. If A fails, can B still
812 succeed? Inferring dependencies between test results in automotive system testing. In *ICST*
813 *2015: Proceedings of the 8th International Conference on Software Testing, Verification and*
814 *Validation*, pages 1–10, Graz, Austria, April 2015.
- 815 **16** Michael Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Till-
816 mann. Exploiting the synergy between automated-test-generation and programming-by-
817 contract. In *ICSE 2009: Proceedings of the 31st International Conference on Software*
818 *Engineering*, pages 401–402, Vancouver, BC, Canada, May 2009.
- 819 **17** Patrice Chalin. Are practitioners writing contracts? In *Rigorous Development of Complex*
820 *Fault-Tolerant Systems*, pages 100–113. 2006.
- 821 **18** Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie.
822 XIAO: Tuning code clones at hands of engineers in practice. In *ACSAC 2012: Proceedings*
823 *of 28th Annual Computer Security Applications Conference*, pages 369–378, Orlando, FL,
824 USA, December 2012.
- 825 **19** Yingnong Dang, Dongmei Zhang, Song Ge, Ray Huang, Chengyun Chu, and Tao Xie.
826 Transferring code-clone detection and analysis to practice. In *ICSE 2017: Proceedings*
827 *of the 39th International Conference on Software Engineering, Software Engineering in*
828 *Practice (SEIP)*, pages 53–62, Buenos Aires, Argentina, May 2017.
- 829 **20** H-Christian Estler, Carlo A Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer.
830 Contracts in practice. In *FM 2014: Proceedings of the 19th International Symposium on*
831 *Formal Methods*, pages 230–246. Singapore, May 2014.
- 832 **21** Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *ISSTA 2011:*
833 *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages
834 364–374, Toronto, ON, Canada, July 2011.
- 835 **22** Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making
836 system user interactive tests repeatable: When and what should we control? In *ICSE*
837 *2015: Proceedings of the 37th International Conference on Software Engineering*, pages
838 55–65, Florence, Italy, May 2015.
- 839 **23** Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random
840 testing. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Program-*
841 *ming Language Design and Implementation*, Chicago, IL, USA, June 2005.

- 845 **24** John V. Guttag and James J. Horning. The algebraic specification of abstract data types.
846 *Acta Informatica*, pages 27–52, 1978.
- 847 **25** C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the*
848 *ACM*, pages 576–580, October 1969.
- 849 **26** Pratap Lakshman. Visual Studio 2015 – Build better software with Smart Unit Tests.
850 *MSDN Magazine*.
- 851 **27** Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behav-
852 iorial interface specification language for Java. Technical Report TR 98-06i, Department of
853 Computer Science, Iowa State University, June 1998.
- 854 **28** Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis
855 of flaky tests. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the*
856 *Foundations of Software Engineering*, pages 643–653, Hong Kong, November 2014.
- 857 **29** Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper
858 Saddle River, NJ, USA, 2006.
- 859 **30** Bertrand Meyer. Applying "Design by Contract". *Computer*, pages 40–51, October 1992.
- 860 **31** Microsoft. Pex MSDN discussion forum, April 2011. URL: [http://social.msdn.](http://social.msdn.microsoft.com/Forums/en-US/pex)
861 [microsoft.com/Forums/en-US/pex](http://social.msdn.microsoft.com/Forums/en-US/pex).
- 862 **32** Microsoft. Generate unit tests for your code with IntelliTest, 2015. URL: [https://msdn.](https://msdn.microsoft.com/library/dn823749)
863 [microsoft.com/library/dn823749](https://msdn.microsoft.com/library/dn823749).
- 864 **33** David Saff. Theory-infected: Or how I learned to stop worrying and love universal quantifi-
865 cation. In *OOPSLA Companion: Proceedings of the Object-Oriented Programming Systems,*
866 *Languages, and Applications*, pages 846–847, Montreal, QC, Canada, October 2007.
- 867 **34** Todd W Schiller, Kellen Donohue, Forrest Coward, and Michael D Ernst. Case studies
868 and tools for contract specifications. In *ICSE 2014: Proceedings of the 36th International*
869 *Conference on Software Engineering*, pages 596–607, Hyderabad, India, June 2014.
- 870 **35** Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C.
871 In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference*
872 *and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*,
873 pages 263–272, Lisbon, Portugal, September 2005.
- 874 **36** Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan
875 de Halleux. Retrofitting unit tests for parameterized unit testing. In *FASE 2011: Proceed-*
876 *ings of the Fundamental Approaches to Software Engineering*, pages 294–309. Saarbrücken,
877 Germany, March 2011.
- 878 **37** Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .NET. In
879 *TAP 2008: Proceedings of the 2nd International Conference on Tests And Proofs (TAP)*,
880 pages 134–153, Prato, Italy, April 2008.
- 881 **38** Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Parameterized unit testing: Theory
882 and practice. In *ICSE 2010: Proceedings of the 32nd International Conference on Software*
883 *Engineering*, pages 483–484, Cape Town, South Africa, May 2010.
- 884 **39** Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test
885 generation tool to practice: From Pex to Fakes and Code Digger. In *ASE 2014: Proceedings*
886 *of the 29th Annual International Conference on Automated Software Engineering*, pages
887 385–396, Västerås, Sweden, September 2014.
- 888 **40** Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ESEC/FSE 2005:*
889 *Proceedings of the 10th European Software Engineering Conference and the 13th ACM*
890 *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–262, Lisbon,
891 Portugal, September 2005.
- 892 **41** Matias Waterloo, Suzette Person, and Sebastian Elbaum. Test analysis: Searching for
893 faults in tests. In *ASE 2015: Proceedings of the 30th Annual International Conference on*
894 *Automated Software Engineering*, pages 149–154, Lincoln, NE, USA, November 2015.

- 895 **42** Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identifica-
896 tion of problems for structural test generation. In *ICSE 2011: Proceedings of the 33rd*
897 *International Conference on Software Engineering*, pages 611–620, Waikiki, HI, USA, May
898 2011.
- 899 **43** Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mutation analysis
900 of parameterized unit tests. In *ICSTW 2009: Proceedings of the International Conference*
901 *on Software Testing, Verification and Validation Workshops*, pages 177–181, Denver, CO,
902 USA, April 2009.