Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

5-2019

Practitioners' views on good software testing practices

Pavneet S. KOCHHAR

Xin XIA

David LO
Singapore Management University, davidlo@smu.edu.sg

Practitioners' Views on Good Software Testing Practices

Pavneet Singh Kochhar[*], Xin Xia[†*], David Lo[‡]

[*]Microsoft, Canada
[†]Faculty of Information Technology, Monash University, Australia
[‡]School of Information Systems, Singapore Management University, Singapore
pavneetk@microsoft.com, xin.xia@monash.edu, davidlo@smu.edu.sg

Abstract—**Software testing is an integral part of software development process. Unfortunately, for many projects, bugs are prevalent despite testing effort, and testing continues to cost significant amount of time and resources. This brings forward the issue of test case quality and prompts us to investigate what make good test cases. To answer this important question, we interview 21 and survey 261 practitioners, who come from many small to large companies and open source projects distributed in 27 countries, to create and validate 29 hypotheses that describe characteristics of good test cases and testing practices. These characteristics span multiple dimensions including test case contents, size and complexity, coverage, maintainability, and bug detection. We present highly rated characteristics and rationales why practitioners agree or disagree with them, which in turn highlight best practices and trade-offs that need to be considered in the creation of test cases. Our findings also highlight open problems and opportunities for software engineering researchers to improve practitioner activities and address their pain points.**

I. INTRODUCTION

Testing is a crucial element in the software development lifecycle. The purpose of testing is to ensure that a software product meets its functional as well as non-functional requirements. The increase in size and complexity of software has further necessitated the need of software testing as bugs can have serious implications on the economy [1]. Test cases are a central piece in testing and practitioners put in a significant amount of time writing and maintaining them. However, despite testing effort, it is often seen that bugs appear in programs. Moreover, for many projects, testing effort continues to be high as systems evolve. These bring forward the issue of test case quality and prompt us to investigate the question of what make good test cases.

Several empirical studies have looked into various aspects of testing such as coverage, mutants, bug detection, and automated test generation tools [2]–[5]. While many of these studies consider automatically generated test cases, most test cases today are created manually – c.f., [6]. Furthermore, studies described above only analyze artifacts that practitioners make (e.g., code and bug reports) rather than surveying or interviewing practitioners. The latter is often needed to get deeper insights into rationales behind practitioner actions.

The closest work to our study is by [7], which surveys developers to understand motivation for writing unit tests, main activities during unit testing, how developers write unit tests, and usage of automated unit test tools. They also ask respondents about importance of different aspects such as coverage, execution speed, robustness, etc. when writing new test cases. The study shows that the developers perform unit testing out of conviction and management requirements and 73% mention strong desire to have more test cases. Our study extends the above-mentioned study by going beyond unit testing, and investigating *many more factors* practitioners consider while writing test cases (e.g., test cases' contents, size & complexity, maintainability, etc.) Furthermore, different from the above study, we conduct interviews before doing a survey with large number of practitioners and ask *rationales* for selecting ratings.

While the above study looks particularly into unit testing practices, our study investigates *many more factors* such as contents, size & complexity, maintainability etc. practitioners consider while writing test cases.

In this study, we complement the existing empirical studies that investigate test case quality by conducting interviews with industrial and open-source practitioners to understand the characteristics of good test cases and testing practices. We validate the hypotheses that we formulate from the interviews by doing a survey on 261 practitioners from Facebook, Microsoft, Google, LinkedIn, Salesforce, other small to large companies, and top 650 projects (ranked based on their popularity, i.e., number of stars + number of forks) on GitHub. Our study produces 29 validated hypotheses on characteristics of good test cases in several dimensions: test case contents, size and complexity, coverage, maintainability, bug detection, and others.

The following is our list of contributions:

1) We interview and survey hundreds of industrial and open-source practitioners from 27 countries to investigate characteristics of good test cases.
2) We provide a list of 29 characteristics of what make good test cases in six dimensions: test case contents, size and complexity, coverage, maintainability, bug detection, and others. These characteristics can give practitioners insight on factors that they need to consider for the creation of test cases and follow good testing practices.
3) We describe rationales why practitioners agree or disagree with each of the characteristics and in the process

---

[*]Corresponding author

highlight trade-offs and special circumstances that practitioners need to consider in the creation of test cases.

II. RESEARCH METHODOLOGY

Our study consists of two parts: open-ended practitioner interviews and a validation survey. The goal of the first part (Section II-A) is to get insights into practitioner views to help us formulate a set of hypotheses. These hypotheses are then checked by the validation survey (Section II-B) which is sent to a large number of practitioners (i.e., hundreds of them).

A. Open Ended Interviews

1) Participants: We contact the top 42 practitioners who contributed the most to Apache projects hosted on GitHub (ranked based on their number of commits), and practitioners from our industry partner in China to find practitioners who are willing to spend a block of their time to get interviewed. Many Apache practitioners are highly experienced and many Apache projects are well-known. This motivates us to pick Apache practitioners as our candidate interviewees. Our industry partner in China, i.e., Insigma Hengtian [8], is a large software outsourcing provider in China. Its service include delivering test cases (i.e., test outsourcing) and solutions for its clients which include Fortune 500 companies. We pick Hengtian due to its long experience as a test outsourcing provider and our prior experience conducting research with them – c.f., [9]–[11]. Many practitioners in the company have created a large number of test cases for many external systems belonging to many clients coming from different industries and parts of the globe.

Following [12], to get more participants, we use several ways to conduct interviews: face-to-face, via Skype, via email, and via an online form. At the end, we get 5 participants from Apache who are willing to be interviewed – either via Skype, email, or online form. These participants include members of Apache Hadoop, Hive, SystemML, CommonsMath, Sling, etc. with an average professional experience of 20 years. We also get 16 participants from Hengtian who are willing to be interviewed face-to-face or via online form. The average experience of these practitioners is 4 years. In total, we interview 13 practitioners face-to-face or via Skype, and 8 practitioners via email or online form. We also translate our questions to Chinese to make it easier for respondents from China.

2) Protocol: **Asynchronous: via email or online form.** We first ask participants some demographic questions (e.g., their number of years of professional experience, etc.). Next, we ask a set of open-ended questions including:
a) How would you define a good and a bad test case?
b) What criteria do you use to characterize a test case quality?
c) What factors do you consider while writing test cases?
d) What kinds of issues do you face in the creation and management of test cases?
The participants respond to these questions in writing via an online form or through email.

Synchronous: face-to-face or via Skype. We start the interview by describing our study and asking for permission to record the interview. Then, initial questions which are related to the participant demographics are asked. Next, we start our discussion which is *loosely* guided by a set of open-ended questions which we prepare in advance. These questions are the same questions that we ask participants who prefer to provide their responses via email or online form. We encourage the practitioners to talk in detail about any relevant topic which our questions do not cover. We ask follow up and clarification questions for answers we find interesting. At the end of the interview, we allow the participants to provide suggestions, comments, and opinions about writing better test cases. The interviews typically last between 30 minutes to 1 hour.

3) Data Analysis: At the end of the interviews, we create interview transcripts manually by replaying the recordings. These transcripts are then analyzed to create a set of hypotheses. We also consider various blog posts, online articles and past experience of authors as software engineers to supplement the information from interviews to create hypotheses. We group similar hypotheses into a small set of dimensions. All the authors were involved in creating and categorizing the hypotheses. Table I lists the hypotheses that we have created divided into seven dimensions. We choose to create hypotheses that can hold true for testing at various levels of granularity (unit, integration, or system). These hypotheses are the input of the second part of our study (i.e., validation survey).

B. Validation Survey

1) Respondents: In the validation survey, we try to get as many practitioners as possible to support or refute our hypotheses. We follow a multi-pronged approach to get survey respondents:

- First, we contact professionals in our network who are working for various organizations such as Facebook, Microsoft, Google, Box.com, LinkedIn, Salesforce, Infosys, Tata Consultancy Services (TCS) and many other small to large companies in various countries. We ask them to fill in our survey and distribute it to their friends and colleagues. Doing this helps us in getting diverse set of responses from industrial practitioners around the world.
- Second, we invite people working on the top 650 most popular open source projects in GitHub (based on the sum of their number of stars and number of forks). Many projects in GitHub are "toy" projects, and thus, similar to prior studies, e.g., [13], we only consider highly popular ones. We analyze the commit history of practitioners and rank them based on the number of commits in which a test file was added or edited. Following [14], we heuristically identify test files by looking for the occurrence of the word "Test" in the file name. We send invitations to the top 1,000 practitioners who have committed at least 10 commits in which at least a test file was changed in each commit. Doing this helps us in getting diverse set of responses from open source practitioners around

	Contents	
H1	A good test case is specific or atomic, i.e., one test case should be testing one aspect of a requirement.	3.93
H2	Test cases in a test suite should be self-contained, i.e., independent of one another.	3.95
H3	Good test cases should check for normal and exceptional flow.	4.47
H4	Test cases must perform boundary value analysis i.e., take as input values at the extreme ends of an input domain.	4.24
H5	Test cases should serve as a good reference documentation.	3.93
	Size and Complexity	
H6	Most test cases should be small in size (in terms of its lines of code).	3.85
H7	Large test cases are often hard to understand and maintain.	3.73
H8	Large test cases may be needed to detect difficult bugs.	3.59
H9	A good suite contains lots of small test cases (with fewer LOC) and few large test cases.	3.97
H10	Increased complexity in a test case can lead to bugs in the test code itself.	4.04
	Coverage	
H11	Code coverage is necessary but not sufficient.	3.97
H12	Code coverage should be used to understand what is missing in the tests and create tests based on that.	3.97
H13	Higher coverage does not mean that a test suite can detect more bugs.	4.02
H14	Each test case should have a small footprint, i.e., the amount of code it executes.	3.92
H15	A test case that is designed to maximize coverage is often long, not understandable and brittle (i.e., breaks easily).	3.50
H16	Designing test cases to cover different requirements is often more important than designing test cases to cover more code.	4.00
	Maintainability	
H17	A good test case should be well-modularized.	4.62
H18	A good test case should be readable and understandable.	4.58
H19	Test cases should be simpler than the code being tested.	4.20
H20	Test code should be designed with maintainability in mind since evolution of code often requires changing of test code.	4.16
H21	Traceability links should be maintained between test code, requirements, and source code.	3.97
	Bug Detection	
H22	A good test case should attempt to break functionality to find potential bugs.	4.11
H23	Test even the simplest things that cannot go wrong.	3.89
H24	During maintenance, when a bug is fixed, it is good to add a test case that covers it.	4.40
H25	Test assertions can help detect subtle errors that might otherwise go undetected.	4.51
H26	Adding common errors and possible causes as comments in test code is helpful to debug failures.	3.98
	Others	
H27	A good test case should be designed such that its results are deterministic.	4.07
H28	Test cases in a test suite should not have side effects so running a test before or after another should not change the results.	4.28
H29	Test cases should use tags or categories, such as slow tests, fast tests etc., so as to be able to run a specific set of tests easily at a time.	3.93

the world. Of the 1,000 invitations, 64 of these are not successfully delivered and we receive one automatic reply notifying the receiver's absence.

In total, we receive 261 responses. The top two countries where the respondents come from are China and United States. The professional experience of these 261 respondents vary from 0.2 years to 30 years, with an average of 6.22 years and median of 5 years. 53% of the respondents have a Bachelor's degree and 33% have an advanced degree, i.e., Master's or Ph.D. Our respondents are spread out in 29 countries.

2) Protocol: Our validation survey consists of two parts: *hypotheses* and *rationales*. We describe them below:

1) In the first part, we present our hypotheses as statements that we ask our respondents to rate. Each respondent can rate each statement as: *strongly agree*, *agree*, *neutral*, *disagree*, *strongly disagree*, and *N/A or I don't understand*. We include the option *N/A or I don't understand* to prevent respondents providing arbitrary ratings to hypotheses that they are not clear about. Respondents can also choose not to provide any rating to any question.

2) Although ratings help us to understand respondent positions on the hypotheses, they are not sufficient for us to understand respondent reasonings. Thus, in the second part, we ask a few additional questions. First, we randomly select two statements that a respondent has rated as *strongly agree* or *agree*. We then ask the respondent the reason why he/she has provided such ratings. Second, we randomly select two statements that a respondent has rated as *strongly disagree* or *disagree* and ask he/she to provide his/her reasons. Answering these questions is optional.

3) Data Analysis: Hypotheses part: We collate the ratings that the practitioners provide to the hypotheses. We convert each rating to a Likert score from 1 to 5. We map strongly disagree, disagree, neutral, agree, and strongly agree to 1, 2, 3, 4, and 5, respectively. We then compute the average Likert score of each statement and plot Likert scale graph. A Likert scale graph (▉▪▪▄▄) is a bar chart which shows number of responses corresponding to strongly agree, agree, neutral, disagree, strongly disagree, and N/A or I don't understand, respectively.

Rationale part: We collect arguments that practitioners have provided to support or refute each hypothesis. We then summarize these arguments.

III. TEST CASE CHARACTERISTICS

In this section, we describe characteristics of good test cases. We first describe how software engineers rated the 29 different hypotheses that we have grouped into 6 dimensions: test case contents, size and complexity, coverage, maintainability, bug detection, and others. And then we present the rationales that the software engineers provided to support or refute their ratings. We consider the following three research questions:

- **RQ1:** How respondents rate the 29 hypotheses describing characteristics of good test cases? (Section III-A)
- **RQ2:** What are the reasons why practitioners agree or disagree with certain test case characteristics? (Section III-B)

A. RQ1: Ratings of the 29 Hypotheses

The third column of Table I presents the average Likert score of the 29 hypotheses. Participants varying degree of agreement to the hypotheses; the average Likert score varies from 3.50 (H15: A test case that is designed to maximize coverage is often long, not understandable and brittle) to 4.58 (H17: A good test case should be well-modularized.). Thirteen out of the 29 hypotheses receive an average Likert score of 4.0 (i.e., agree) and above.

B. RQ2: Reasons

To answer RQ2, we present arguments that support or refute each hypothesis as provided by our interview participants and survey respondents.

1) Contents: Intuitively, the contents of a test case would significantly affect its quality. In this dimension, we investigate practitioners agreement on some hypotheses that describe characteristics of good test cases based on their contents.

Specific (H1). ▊▊■▄▗▗

In general, practitioners advice that a test case should be specific, i.e., it should try to test only one functionality. Out of the responses that we receive, 96 indicate strong agreement and 92 agreement with hypothesis H1. The overall Likert score is 3.93 (i.e., close to "agree"). The following are some of the comments that support (👍) or refute (👎) the hypothesis:

> 👍 *"I prefer atomic things or smaller test cases that test one thing if possible. It's easy to understand, easy to manage."*
> 👍 *"One test case should be testing one aspect of a use case."*
> 👎 *"...If you are in a scenario where test is actually taking little extra time then at least I do not see a problem in verifying multiple different things in the same test or testing multiple scenarios in the same test."*

From the above comments, we note that many practitioners support this hypothesis since specific (or atomic) test cases are easier to understand. However, in cases where tests take longer to run, testing multiple things in one test case may be a more efficient alternative.

Self-Contained (H2). ▊▊■▄▗▗

Most respondents express that test cases should be self-contained with no or minimal dependency on other test cases present in a suite. The average Likert score for this hypothesis is 3.95 (i.e., mostly "agree"). Interestingly, 48, 19, and 6 respondents neither agree/disagree (i.e., they are neutral), disagree, or strongly disagree with this hypothesis, respectively. The following are some comments that support or refute the hypothesis:

> 👍 *"The more isolated the tests, the better. "*
> 👍 *"I try to be maybe have 3 or 4 instance variables within the setup and I am using the nested classes to minimize the scope so you are not sharing, not a lot of globals floating around, fairly localized to where they are used but a bit of reuse is fine I think."*
> 👎 *"Some test cases may share commonalities."*
> 👎 *"There may be inherent relationships or dependencies between test cases."*

From the above comments, again we note that respondents prefer self-contained test cases since they are easier to understand. On the other hand, we note that there is a trade-off between the simplicity achieved by self-contained test cases and reuse potentials. Respondents that disagree with this statement often highly value reuse over simplicity. Some test cases are inherently related or dependent on one another and keeping them self-contained may mean a lot of duplication.

Consider Different Flows (H3). ▊▊▗▗▗▗

Almost all of our respondents strongly agree (140 respondents) or agree (107 respondents) that it is important for test cases to check both normal and exceptional flow. The Likert score for this statement is 4.47 which is substantially higher than the scores for H1 and H2. We do not receive any comment that refutes the hypothesis. The following are comments that support the hypothesis:

> 👍 *"You focus on the happy case to verify the business functionality was needed ... then [write tests] to make sure any edge cases have been properly addressed."*
> 👍 *"A test case will typically have some assertions to check for the happy cases ...It is important to write test cases for failure path... "*

This hypothesis seems to be more or less universally supported, at least among the practitioners whom we interview and survey. Among the five hypotheses in the content dimension, this hypothesis gathers the most support.

Perform Boundary Value Analysis (H4). ▊▊▗▗▗▗

Boundary value analysis refers to testing at the boundaries between partitions of the input space, which include both valid and invalid values. This hypothesis receives the second highest support among the five hypotheses with an average Likert score of 4.24. The comments that we receive include:

> 👍 *"Test cases should be considered as a whole. Some must address nominal input with intermediate values well within the application domain, some must address nominal input with special values (zero, input at boundaries, null size), some must address invalid input in order to check errors are correctly detected."*
> 👍 *"You have always want to test corner cases because that is where things tend to go wrong.*
> 👎 *"Too much effort for too little benefit most of the time."*
> 👎 *"Not every situation requires boundary value analysis."*

There is higher probability of finding bugs at the boundaries of input partitions (i.e., corner cases) as they are often the less tested parts of the code. However, there is a trade-off between time and effort and thus, respondents suggest performing this analysis in specific circumstances.

Serve as a Reference Documentation (H5). ▊▊■▄▗▗

Well commented, named and designed test cases may serve as a good reference documentation. Most of our survey respondents agree that test cases should be designed as such – its average Likert score is 3.93. This hypothesis however receives the lowest support among the hypotheses in this dimension. The following are some comments that we receive:

> 👍 *"I am a big fan of using tests as reference documentation. Writing readable tests so that you don't have, if want, to document the details of an API... If you can stay at the overview level in the documentation and details in the tests, it is very efficient."*
> 👍 *"Test cases are often written before documentation examples and should provide example use cases for functionality."*
> 👎 *"Writing easy to understand tests is hard and is not worth. It's better to have separate reference code, which can be runnable as tests."*

From the comments, practitioners view test cases as a good complement to traditional documentation (e.g., API's textual documentation). High-level overview can be given in the documentation, while details are pushed to test cases. However, some practitioners push back on the idea because writing easy-to-understand test cases is hard, and they view the benefit is not worth the effort. Previous research on over

1400 projects shows that 96.44% of the unit test methods did not have outer comments and 85.98% did not have inner comments [15]. This might explain the lower score obtained for this hypothesis as *"Comments need to be maintained which adds complexity to the task." [15].*

2) Size and Complexity: Size and complexity of test cases are important attributes to consider. The size and complexity of a piece of code have often been associated to its quality [16]. Unfortunately, no or little study has focused on test code. In this dimension, we consider five hypotheses that describe characteristics of good test cases in terms of their size and complexity, and investigate developer support, or lack of, to them.

Small in Size (H6). ▪❚▪␣␣

A large number of respondents agree that test cases should be small whereas some are neutral or even disagree with this hypothesis (average Likert score = 3.85). Some of the comments we receive are:

- 👍 *"A good test should be short, should fit on to 10 lines or less of code, is self-contained, has a clear intent and its scope is obvious..."*
- 👍 *"I am more a fan of many small tests than few big ones."*
- 👍 *"Each test method should test one feature."*
- 👎 *"Some codes need complicated test logic to cover logics of it."*
- 👎 *"I am careful to keep the simplicity, but not care the number of lines."*

Practitioners mention that as test cases should be clear, test only one functionality, and should be small in size as larger test cases are often complex.

Understandability and Maintainability of Large Test Cases (H7). ▪❚▪␣␣

In general, practitioners confirm that large test cases are hard to understand and maintain (average Likert score = 3.73):

- 👍 *"Large cases attempting to do everything at once are difficult to understand and more importantly difficult to maintain. When code changes, the tests must be rewritten, which is bad."*
- 👍 *"For me it is a bad sign if test becomes long and complicated. It can also be a sign of bad design."*
- 👎 *"Large test cases are often necessary, especially in testing cases that require bootstrapping."*
- 👎 *"It is Ok to have some for smoke testing...It is the exception not the rule and it is for a particular purpose."*

From the comments, large test cases are often viewed as harder to understand and maintain and can also be an indication of bad design (either in the test code or in the system under test (SUT)). However, they might be required for specialized testing or for cases that require bootstrapping – these should be exceptions and not the rule though.

Large, Complex Test Cases and Difficult Bugs (H8). ▪❚▪␣

Previously, practitioners express that large test cases are hard to understand and maintain (H7). In this hypothesis, we would like to confirm whether practitioners agree that large test cases can be useful to detect difficult-to-find bugs. We find that 150 respondents strongly agree or agree with this hypothesis. A substantial number of respondents choose to be neutral or disagree (73 neutral respondents, and 35 respondents who disagree or strongly disagree). The average Likert score is 3.59 which is the lowest among hypotheses in this dimension. Some of the comments that we receive are:

- 👍 *"Complex test cases will cover integration environment and they can lead to some very good bugs being discovered."*
- 👍 *"Sometimes the most awkward bugs appear when a series of steps are happening in the code."*
- 👎 *"...will detect less bugs ultimately because they would be harder for us to understand and maintain. It goes together with the readability factor. "*
- 👎 *"...strategy matters, not the size of test case."*

From the comments, many practitioners agree that long and complex test cases can detect some hard-to-find bugs since they can cover long series of steps that cannot be simulated by simple test cases. However, some practitioners disagree by stating that poor understandability will make such test cases less able to find bugs *in the long run*. Others argue that what matters is the strategy practitioners apply for testing – with a good strategy, small and simple test cases can be sufficient to find many hard-to-find bugs.

Large and Small Test Case Mix (H9). ▪❚▪␣␣

Most practitioners are of the opinion that a test suite should contain a good mix of many short and a few large test cases (average Likert score = 3.97). Few of the comments that came out during interview and survey are:

- 👍 *"A combination of lots of small tests and some large tests is ideal but you cannot throw away large test by a lot of small tests."*
- 👍 *"Small tests eg unit tests and large tests like fuzzers, integration tests, etc will find *different* bugs."*
- 👍 *"This would cover most situations of requirements."*
- 👎 *"For me it is better to have lots of [small] tests."*

Complexity and Bugs in Test Cases (H10). ▪❚␣␣␣

In our interviews, several practitioners state that test cases can often become long and hard to manage. This increased complexity of test cases can lead to bugs in the test code. A large number of our survey respondents agree with this hypothesis (average Likert score = 4.04). Some of the comments practitioners made to justify their support or lack of support are:

- 👍 *"If the test is really hard to read and understand and it is complex in its own right there is a good chance that... there is a bug in the test itself."*
- 👍 *'We might put ourselves at risk of not understanding the test when we come back to it later. "*
- 👎 *"Complex test cases make an environment less productive, but do not directly cause bugs."*

In general practitioners find that there is a higher likelihood of bugs appearing in the test code if the complexity of test cases increases. Empirical evidence shows that projects often have bugs in test code and they can cause false alarms, which correspond to test failure when production code is correct [17]. This hypothesis received the maximum agreement in the size and complexity dimension.

3) Coverage: Code coverage, the amount of code covered by test cases, is often used as a measure of test quality. Coverage information can help practitioners in finding parts of the code which are not covered and might contain bugs.

Code Coverage, Necessary but Insufficient (H11). ▪▪▪▫▫▫

A hundred and ninety four of our respondents support (agree or strongly agree with) this hypothesis – resulting in an average Likert score of 3.97. The following are some of their comments that support or refute the hypothesis:

- 👍 *"The more coverage you got, generally speaking, the better."*
- 👍 *"It does not measure the combinatorial explosion of possible interactions."*
- 👎 *"I could certainly write tests that provide good code coverage but do not actually test what users are going to use from the software."*
- 👎 *"Code coverage is nice but not all that useful. Running a line isn't an indicator that you've tested it."*

In general, many practitioners find that code coverage is a good starting point as it gives information whether we have exercised a piece of code. However, some practitioners argue that covering a code may not mean that it has been tested, and a test case that covers a code may not mimic what real users would do in practice. This is in line with the past research which advises developers not to use coverage blindly [3], [4].

Code Coverage and New Test Cases (H12). ▪▪▪▫▫▫

Practitioners in general agree that coverage information can be leveraged to understand shortcomings of current test cases to write new tests (average Likert score = 3.97). Some practitioners provide these rationales:

- 👍 *"Use code coverage to understand what is missing in the tests and then create intelligent test based on that."*
- 👍 *"I look at my code coverage, I am not at 100% then I know I must have not got any tests for place order where there is probably some interesting business functionality"*
- 👎 *"I prefer to focus on features, rather than code coverage."*

Higher Coverage and Detecting More Bugs (H13). ▪▪▪▫▫ ▫

In general, practitioners agree that a higher coverage *does not mean* that a test suite can detect more bugs. This hypothesis receives an average Likert score of 4.02, which is the highest for hypotheses in this dimension. More than 200 practitioners agree or strongly agree with this statement. Here are some of the comments:

- 👍 *"Because high coverage is useless unless you are also making the right assertions."*
- 👍 *"Because code coverage does not consider semantic of the code."*
- 👎 *"More coverage, less chance of bugs."*
- 👎 *"Hitting all code passes increases probability of finding edge test case that was not thought of."*

From the comments, many practitioners complain that code coverage does not consider the semantic of the code and is useless without good assertions. However, eighteen practitioners whom we survey disagree with the statement stating that coverage has its place in detecting bugs.

Small Footprint (H14). ▪▪▪▫▫

This hypothesis is a slightly controversial one; only a slight majority of our survey respondents (52.17%) agree or strongly agree that a single test case should have a small footprint (i.e., the amount of code it executes). Still, the average Likert score is 3.92, and thus the balance tips towards agreement with many practitioners (85 of them) on the fence. The following are the rationales that practitioners give to support or refute the hypothesis:

- 👍 *"Developer test should test a single responsibility but does not necessarily mean a method "*
- 👍 *"Simple. The larger the footprint the more bottlenecks there are in the testing process and the slower the testing process is."*
- 👎 *"...this is overrated. Tests should be maintainable...But worshipping this principle can often be the enemy of maintainable tests."*
- 👎 *"If you can write a simple test that covers a lot of code, that can make writing tests more efficient."*

The proponents of this hypothesis argue that a single test case should have a single responsibility. Also, test cases that cover a lot of code can cause bottlenecks and slow down the testing process. Others disagree that by covering as much code as possible with as few test cases, one can save the cost of writing test cases.

Maximizing Code Coverage, and Long, Not Understandable, and Brittle Test Cases (H15). ▪▪▪▪▫▫

This hypothesis is also a slightly controversial one; only 54.25% of the respondents agree or highly agree that a test case that is designed to maximize coverage is often long, not understandable and brittle (i.e., breaks easily). Although this hypothesis receives the lowest agreement among others in this dimension, the balance again tips towards agreement with an average Likert score of 3.50. The following are some comments given by practitioners:

- 👍 *"If you try to over-focus on code coverage people will try to go through all sorts of loops... it is quite difficult. You have to go through a lot of effort to trigger that to occur and it is not just worth the effort."*
- 👍 *"Because the desire for coverage often makes people lose sight of the true goal of a given test case."*
- 👎 *"Optimizing for coverage doesn't mean complicated tests unless the code being tested is complicated."*
- 👎 *"The more code I can test with a maintainable test the better."*

Most practitioners agree that focussing solely on coverage can create problems since people can often lose sight on the true goal of testing, and start creating peculiar code – one respondent puts it as "all sorts of loops" – which can be harmful. However, 46 respondents disagree and 6 strongly disagree with the hypothesis stating that one can often optimize coverage without causing issues mentioned in the hypothesis.

Code Versus Requirement Coverage (H16). ▪▪▪▫▫

Most practitioners agree that requirement coverage is more important than code coverage resulting in the average Likert score of 4.00. We only receive positive comments supporting this hypothesis which include:

- 👍 *"The core goal for me would not be to maximize code coverage. It will be to maximize testing basic case and corner cases for a feature."*
- 👍 *"I don't believe it is an effective use of time to test the most basic of code (getter/setter, etc...)"*
- 👍 *"Code coverage is a technical measure that isn't directly related to user-facing features. User-facing features are the actual thing that an application should care about."*

From the comments, we find that practitioners prefer requirement coverage, since some code is of little value and is less likely to be buggy (e.g., getter or setter methods). Moreover, test cases that achieve requirement coverage often mimic well how clients would use a piece of SUT.

4) Maintainability: Software system evolves and so should its test cases. Maintainability of code (including test code) is an important aspect as it helps to ensure that a software system continues to serve its intended purpose. In a recent survey conducted by Li et al. [15] on over 200 practitioners, 89.15% developers "agree" or "strongly agree" that maintenance of good test cases is important for the quality of a system.

Well-Modularized (H17). ▮▮▬▬▬

Most respondents agree or strongly agree that test cases should be well modularized. Among the hypotheses in this dimension, this one receives the highest Likert score of 4.62. Only 4 respondents disagree or strongly disagree with this hypothesis. Following are some of the comments that support or refute the hypothesis:

- 👍 *"Test code is code. If the test is simple for people to understand, it should be short and simple in the code."*
- 👍 *"...It might mean that you are trying to test too much stuff at once and maybe you should break that down into smaller modules or units."*
- 👍 *"It is easier to maintain it if it is..."*
- 👎 *"Tests that are too modularized, tend to make debugging of regressions more complex."*

From the comments, most practitioners agree that if a test case is large, it should be broken down into smaller modules or units, since it would then be simpler to read and easier to change as a software system evolves. However, too modularized test may make debugging more complex.

Readable (H18). ▮▮▬▬▬

More than 96% of the respondents agree that test cases should be readable and understandable. Practitioners give a number of supportive comments, including the following:

- 👍 *"Like any code, if you have to maintain it you better be able to understand it."*
- 👍 *"Tests reflect intent. Tests should tell a story of how the code is supposed to work. Tests are one of our best tools for understanding the way code is meant to work. Tests communicate across time to future developers about the code."*

From the comments, we find that practitioners highly value readable and understandable code. A few respondents mention that this is hard to achieve though. One of them mentions: "It is challenging to keep the unit test looking nice." We do not receive any comments that refute this hypothesis. Similar observations were made by previous research works - [7] observe that test cases that are difficult to understand can make it harder to fix failing tests. Li et al. find that over 60% of the developers indicated a "moderate" to "very hard" difficulty with respect to understanding the test cases [15]. These works strengthens the argument that tests should be readable and understandable.

Simpler than Tested Code (H19). ▮▮▮▬▬

57.42% of the respondents agree or strongly agree that test code should be simpler than the tested code, while 16.41% indicate their disagreement or strong disagreement (average Likert score = 4.20). We receive the following rationales:

- 👍 *"If the test is complicated it is harder to understand what is the actual failure. Code could get complicated but tests never should."*

- 👍 *"If the test is more complicated than the code being tested then the API being tested is too complicated."*
- 👎 *"Sometimes a fairly simple algorithm can have a fair number of corner cases that warrant complicated test cases."*
- 👎 *"Because sometimes test cases have a more elaborate setup and teardown requirements than the code under test."*

From the comments, although many practitioners support the hypothesis, some express their reservations. The earlier group of respondents argues that simple tests are essential, for example, for effective debugging, while the latter group argues that some functionalities have many corner cases requiring complicated tests, and others require elaborate setup and tear-down requirements. The findings suggest that this hypothesis can be used as a guiding principle, barring some exceptions.

Designed with Maintainability in Mind (H20). ▮▮▬▬▬

Most practitioners agree or strongly agree that test code should be designed with maintainability in mind (average Likert score = 4.16). Some comments which support or refute this hypothesis are:

- 👍 *"Strongly Agree, it can be less fast, but should be designed with maintainability"*
- 👍 *"...if your tests aren't maintainable the code they test isn't."*
- 👎 *"Personally I will rewrite my tests instead of changing them a lot."*
- 👎 *"Spending too much time making tests clean and maintainable is a waste of time when the requirements change and the test case is no longer applicable."*

Traceability Links (H21). ▮▮▬▬▬

More than 190 practitioners agree or strongly agree that traceability links should be maintained between test cases, code and requirements (average Likert score = 3.97). Only seven respondents disagree or strongly disagree with the hypothesis. The following are some of the rationales that our respondents give to support the hypothesis:

- 👍 *"Can reduce other workload and help improve the efficiency of the team."*
- 👍 *"You can quickly locate the part needs to be updated, to make quick updates and to update documentation."*
- 👎 *"It sounds like a lot of project management overhead, which would lead to slower development velocity. "*

5) Bug Detection: Bug detection is one of the main reasons of writing test cases. When practitioners write a new functionality or add a piece of code, they need to test whether that code is working fine or not.

Attempt to Break Functionality (H22). ▮▮▬▬

A total of 215 respondents agree or strongly agree that a test case should attempt to break a functionality. The hypothesis receives an average Likert score of 4.11. The following are some comments that we receive:

- 👍 *" The more ways we can think of to try to break our code, the less it will break when users actually go do crazy things."*
- 👍 *"Many bugs can be found more easily by testing edge cases that developers didn't think about. "*
- 👍 *"In a distributed system, it is common that some component can't perform the designed function well either due to network issues or machine hang etc."*
- 👎 *"First and foremost, tests should ensure the code works as expected, in the environment its expected to run. Having other negative tests is less important."*

Overall, practitioners agree that test cases should try hard to break functionalities. This can be done by testing corner cases

or performing "crazy" things, which can assure that a system would work well in practice under diverse environments and usage patterns. On the other hand, due to schedule contraints some practitioners mention that testing positive cases may matter more.

Test Even the Simplest Things (H23). ▪▪▪▫▫

The majority of respondents agree that testing even the simplest things is valuable (average Likert score = 3.89). However, a minority of respondents (i.e., 9.76%) disagree or strongly disagree with 18.36% respondents on the fence. The following are their rationales:

> 👍 *"Even the simplest of things tested can give you useful information... like hashcode for example or an equality check, people do not actually break that in the future."*
> 👍 *"Whenever you think something cannot go wrong, it probably will."*
> 👍 *"Even write the stupid test because sometimes it is the one that will find the very stupid bugs."*
> 👎 *"It is wasting time and codes."*
> 👎 *There's a line where test cases become more of a burden to carry than the value they provide. "*

The proponents argue that "the simplest things" (e.g., equals() and hashcode()) may also break sometime in the future, and people make "stupid" mistakes. The opponents on the hand argue that testing simplest things may not add much value and adding them is a waste of time and code.

Add New Test Cases For Fixed Bugs (H24). ▪▪▫▫

We receive a high agreement for this hypothesis (average Likert score = 4.40), which is the highest for this dimension. We only receive positive comments, which include the following:

> 👍 *"The test should be written *before* fixing the bug, to ensure you actually understand the bug. Then, once the bug is fixed, you *have* the test, so keep it."*
> 👍 *"If a bug happened once, it can happen again."*

Practitioners support this hypothesis since writing test cases helps one to understand a bug. Moreover, the generated test case can help to ensure that the bug will not happen again without being detected.

Use Assertions to Detect Subtle Errors (H25). ▪▪▪▫▫

A test assertion contains an expression which describes a property that should be (or should never be) observed for a system under test. Most of our respondents agree that assertions are a crucial part of test code and can be helpful in detecting subtle errors (average Likert score = 4.51). We present some practitioner comments below:

> 👍 *"Yes because something you might be taking for granted to be true could very well be false."*
> 👍 *"You need something to fail, you need to have assertions in a test otherwise you are just exercising the system and not making any statements about what it should be doing."*

Practitioners argue that assertions are essential and one cannot only rely on the appearance of exceptions alone to detect failures. However, expressions used in the assertions need to be designed well so that they can detect bad cases effectively. Our results corroborate past empirical findings from data that test assertions are strongly correlated with test suite effectiveness [18].

Commenting Test Code with Common Errors and Possible Causes (H26). ▪▪▪▫▫

A large number of our respondents (i.e., 200) agree or strongly agree that commenting test code with common errors and possible causes is a good idea (average Likert score = 3.98). A few disagree though. Following are some of the comments:

> 👍 *"...comments is usually a convenient way to document those things..."*
> 👍 *"New people don't know your code/history."*
> 👍 *"The name often will say the scenario I am trying to test out or there will be; this is especially true for complicated test, where if I write a test today and go back a month later, I think it is going to be difficult to understand what I am trying to test."*
> 👎 *"Comments that are outdated can do more harm than good. If the comments are misleading then they can cause people to waste time exploring dead ends."*

6) Others: **Deterministic (H27).** ▪▪▪▫▫

Most practitioners we survey agree that a good test case should be deterministic and produce the same output every time it is run (average Likert score = 4.07). However, again a few disagree. The following are some of their explanations:

> 👍 *"If a test involves some aspect of randomness, it can be very hard if not impossible to reproduce a failure"*
> 👍 *"If tests pass or fail due to random factors then they get ignored and become useless."*
> 👎 *"Sometimes it is good to see transient failures to detect a race condition, for example."*

These test cases, often called as flaky tests, make it difficult to rely on the ouput of test results. Empirical data shows that a large number of test failures are caused by flaky tests and past research gives several causes and fixing strategies for such tests [19].

Side Effect Free (H28). ▪▪▪▫▫

Almost all our respondents agree that test cases should be side effect free. In this dimension, we receive the highest agreement for this hypothesis with 213 respondents agreeing or strongly agreeing with it. We present some of the comments below:

> 👍 *"If tests impact each other, it becomes extremely hard to reproduce and interpret test failures."*
> 👎 *"Test cases cannot guarantee the absence of side effects, but it can be reduced."*

Tag Test Cases as Slow or Fast (H29). ▪▪▪▫▫

Several common testing frameworks like JUnit provide the functionality of adding tags to test cases. Most of our respondents agree or strongly agree that the use of such tags to indicate, for example, fast or slow tests, is helpful (average Likert score = 3.93). There are many who are on the fence though (i.e., 62 respondents). We only receive positive comments and the following are some of them:

> 👍 *"For practitioners' convenience when debugging suite-wise problems or regressions."*
> 👍 *"It is very important to have fast tests and if you have slow tests, maybe define tags or categories. It can be the fast ones and slow ones are activated by a different switch."*
> 👍 *"I usually use BDD develop my project. And it's important to me that it is running fast test when I am developing and more detailed but slower test before I commit my code."*

IV. DISCUSSION

A. Implications

For Researchers: Our research suggests new directions for empirical software engineering research. Developer perception matters [20]–[23] but they may not always be correct [24]. Moreover, some of the hypotheses are slightly controversial with two sizable camps for and against them. For example, hypothesis H14 (each test case should have a small footprint, i.e., the amount of code it covers) only receives an average score of 3.52 and is only supported by 51.85% of our respondents. One way to nicely augment our study is to mine software repositories and analyze history of projects to get a deeper understanding of such slightly controversial hypotheses. For example, one can correlate test case footprint with its effectiveness to find bugs based on historical data. Another way, is to perform controlled experiments or field studies, and investigate the correlation between test case footprint and the time it takes for debugging test case failures and/or maintaining test cases. Clearly, it is not possible to perform all such studies and describe them in one paper. Thus, we encourage others to perform such future studies to provide further empirical evidence to further support or refute our hypotheses.

Our results also highlight opportunities for automated software engineering researchers to build tools that can help practitioners create better test cases:

- One can envision a tool that can detect smells in test code by looking for violations of some of the 29 hypotheses, especially those that receive high average Likert scores.
- From the ratings and comments that we receive for H17 and H18, many practitioners value well-modularized, well-written and well-commented test code which follows a consistent coding style. However, creating such test cases is a challenging task. Automated tools can potentially be built to suggest suitable test code refactoring or renaming to improve the modularity, readability, and understandability of test cases. Past tools work on reducing system tests into many unit tests [25], [26], however, refactoring goes beyond that [27].
- From ratings and comments that we receive for H21, practitioners value traceability links between test cases, source code, and requirements. However, for many projects, these links may not have been made explicit and kept up-to-date. Past studies have looked into recovering traceability links between source code and requirements by employing information retrieval [28] and future tools can extend these existing works by incorporating static analysis to infer and maintain 3-way links between test code, source code, and requirements.

For Practitioners: Novices are often unsure on characteristics of good test cases and what factors they need to consider to write such test cases. Our findings provide a list of characteristics that matter to experienced practitioners. The average Likert score of all the hypotheses are above 3.5 (somewhat/close to "agree") and 12 hypotheses are above 4.0 (between "agree" and "strongly agree"). The top 5 hypotheses agreed by most respondents are: H3, H17, H18, H24 and H28. We encourage novices to consider these important factors when designing test cases. For example, following H3, they should check for both normal and exception flow, and following H28, test cases should not have side effects. For H18, practitioners can follow best practices as used by other practitioners, such as for writing unit test cases, they can follow the "Arrange, Act, Assert (AAA)" paradigm [29] that can aid in understandability.

Our survey respondents consist of experienced practitioners, and they disagree on a number of hypotheses. Our results present different practitioner perspectives which often highlight tradeoffs and special circumstances. For example, based on practitioner ratings and comments for H23, we find that testing "simplest things" may detect future problems or "stupid" mistakes, but these "simplest things" may be large in number and testing them (e.g., hashcode(), equals() methods) may consume much time and resources. For H26, we find that commenting test code with common errors and possible causes may be helpful to aid understanding, but these comments may also be a source of problems if they get outdated. For H1, most respondents agree that a test case that tests one aspect of a requirement is good since the test case would be easier to understand; however, for test cases that require long time to run, putting many things in one test may have its place. Our findings bring up such tradeoffs and special considerations which may not be obvious to even experienced practitioners (and thus the difference in opinions).

B. Threats to Validity

External Validity: We interview 21 practitioners working on various industrial and open source projects through which we get numerous insights, yet it is a small sample. Also, our interviewees either contributes to an Apache project or work for Hengtian, which might introduce some bias. To mitigate this threat, we survey 261 respondents from various small and large organizations spread out in 27 countries around the world. Still, our findings may not generalize to all practitioners.

Internal Validity: To reduce bias during interviews, we keep our questions open-ended and let practitioners talk most of the time. It is possible that practitioners might have missed out some points and given more time to think, practitioners might give more suggestions. Interviewing developers via Skype or email might introduce some bias, however, we have tried to reduce the bias by following up with the interviewers if we needed more information. In most of the cases, developers thoroughly answered our questions. Also, it is possible that some of our survey respondents do not understand our hypothesis well. Although most of our hypotheses hold for various levels of granularity such as unit testing, integration testing, however, some of them might hold true only for a specific testing level. To minimize this threat, we provide the option "I don't understand". We also translate our survey to Chinese to make it easier for respondents from China. Moreover, it is possible that we might draw wrong conclusions. To reduce this

threat, we (all of the three authors separately) read interview transcripts and comments we receive from survey respondents several times. The creation of hypotheses from interviews can introduce bias as they are based on perspectives of an individual. However, each of the three authors vetted through the hypotheses created by the other authors to to minimize the bias. The hypotheses were also sent to our survey participants to validate them.

V. Related Work

Daka et al. conducted a survey on 225 developers to understand unit testing practices such as motivation of developers, their usage of automation tools, and their challenges [7]. Meszaros provided guidance on writing automated tests using xUnit covering several aspects such as improving coverage, using test smells to find issues and refactoring tests for greater simplicity, roustness and execution speed [30]. Rompaey et al. proposed a set of metrics using unit test concepts to detect test smells that are analogous to code smells and suggested the need for reliable test smells detection mechanism [31]. Greiler et al. studied indutrial projects to understand whether test smells are caused by test fixtures, i.e., code that initializes and configures the system under test so that automated tests can be run [32]. Palomba et al. studies the relationship between flaky tests and test smells, in particular, Resource Optimism, Indirect Testing and Test Run Wa, on 18 software systems and find that for 54% of the tests, a test code smell can cause flakiness [33]. Kochhar et al. performed surveys on open-source and industrial practitioners to understand the test automation culture of mobile app developers [6]

Among the aforementioned studies, the closest one is Daka et al.'s; different from their work which broadly looks into testing practices, we focus deeply on characteristics of good test cases and factors practitioners consider while writing them, which we divide into several dimensions. Daka et al.'s survey includes a question that asks respondents to rate the importance of: code coverage, execution speed, robustness against code changes, how realistic the test scenario is, how easily faults can be localised/debugged if the test fails, and how easily the test can be updated when the underlying code changes. In this work, we consider *many more factors*. Moreover, not only we ask practitioners to provide their ratings, but also the *rationales of their ratings*; by so doing we can highlight reasons behind certain actions, differences in perspectives, tradeoffs and special circumstances.

VI. Conclusion and Future Work

Testing is an indispensable part of software development activities. In this study, we investigate practitioner perception of characteristics of test cases and different factors practitioners consider while writing test cases. We interview and survey practitioners from 27 countries to create and validate a list of 29 characteristics of good test cases in 6 dimensions (i.e., test case contents, size and complexity, coverage, maintainability, bug detection and others). In the future, we plan to further investigate some of the slightly controversial hypotheses which

have both proponents and opponents. We also plan to develop tools to identify smells in test code, automatically refactor test code, and recover implicit 3-way traceability links between requirements, source code and test cases.

References

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testingthe economic impacts of inadequate infrastructure for software testing," in *National Institute of Standards and Technology. Planning Report*, 2002.

[2] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *ICSE*, 2014, pp. 72–82.

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *ICSE*, 2014, pp. 435–445.

[4] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*, 2014, pp. 654–665.

[5] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *SANER*, 2015, pp. 560–564.

[6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *ICST*, 2015, pp. 1–10.

[7] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *ISSRE*, 2014, pp. 201–211.

[8] Insigma Hengtian, http://www.hengtiansoft.com/.

[9] X. Xia, D. Lo, P. S. Kochhar, Z. Xing, X. Wang, and S. Li, "Experience report: An industrial experience report on test outsourcing practices," in *ISSRE*, 2015, pp. 370–380.

[10] X. Xia, D. Lo, J. Tang, and S. Li, "Customer satisfaction feedback in an IT outsourcing company: a case study on the Insigma Hengtian company," in *EASE*, 2015, pp. 34:1–34:5.

[11] X. Xia, D. Lo, F. Zhu, X. Wang, and B. Zhou, "Software internationalization and localization: An industrial experience," in *ICECCS*, 2013, pp. 222–231.

[12] R. Opdenakker, "Advantages and disadvantages of four interview techniques in qualitative research," *Forum: Qualitative Social Research, (Last accessed on March 9, 2016)*, vol. 7, no. 4, 2006.

[13] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *FSE*, 2014, pp. 155–165.

[14] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[15] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *ICST*, 2016, pp. 341–352.

[16] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Software Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

[17] A. Vahabzadeh, A. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *ICSME*, 2015, pp. 101–110.

[18] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *ESEC/FSE*, 2015, pp. 214–224.

[19] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014, pp. 643–653.

[20] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *ESEC/FSE*, 2015, pp. 50–60.

[21] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How do professionals perceive legacy systems and software modernization?" in *ICSE*, 2014, pp. 36–47.

[22] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *ICSE*, 2016.

[23] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann, "Quantifying developers' adoption of security tools," in *FSE*, 2015, pp. 260–271.

[24] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *ICSE*, 2016.

[25] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *FSE*, 2006, pp. 253–264.

[26] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *ASE*, 2005, pp. 114–123.

[27] "Catalog of Refactorings," http://refactoring.com/catalog/index.html, (Last Accessed on April 8, 2018). [Online]. Available: http://refactoring.com/catalog/index.html

[28] J. Cleland-Huang and J. Guo, "Towards more intelligent trace retrieval algorithms," in *RAISE*, 2014, pp. 1–6.

[29] "Unit Test Basics," https://msdn.microsoft.com/en-us/library/hh694602.aspx, (Last Accessed on April 8, 2018). [Online]. Available: https://msdn.microsoft.com/en-us/library/hh694602.aspx

[30] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.

[31] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[32] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *ICST*, 2013, pp. 322–331.

[33] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *ICSME*, 2017, pp. 1–12.