A Survey on Unit Testing Practices and Problems

Ermira Daka and Gordon Fraser
University of Sheffield
Sheffield, United Kingdom
{ermira.daka,gordon.fraser}@sheffield.ac.uk

***Abstract*—Unit testing is a common practice where developers write test cases together with regular code. Automation frameworks such as JUnit for Java have popularised this approach, allowing frequent and automatic execution of unit test suites. Despite the appraisals of unit testing in practice, software engineering researchers see potential for improvement and investigate advanced techniques such as automated unit test generation. To align such research with the needs of practitioners, we conducted a survey amongst 225 software developers, covering different programming languages and 29 countries, using a global online marketing research platform. The survey responses confirm that unit testing is an important factor in software development, and suggest that there is indeed potential and need for research on automation of unit testing. The results help us to identify areas of importance on which further research will be necessary (e.g., maintenance of unit tests), and also provide insights into the suitability of online marketing research platforms for software engineering surveys.**

I. Introduction

Today, almost every programming language has its own unit testing framework (e.g., JUnit for Java, NUnit for C#), which enables the use of small, automatically executable unit tests. Unit testing has become an accepted practice, often even mandated by development processes (e.g., test-driven development). Nevertheless, software quality remains an issue. Software engineering researchers therefore argue that there is a need to push automation in testing further — to even automatically generate unit tests.

Research has produced many different variations of automated unit test generation, and even some commercial tools have started appearing: Fully automated tools can exercise units to find crashes and unexpected exceptions (e.g., Randoop [22]). Given a formal specification (e.g., JML), automated unit testing can generate tests that check the implemented unit against its specified behaviour fully automatically (e.g., [3]). Parameterized unit tests can be used as a type of specification, where automated tools like Pex [26] can instantiate the parameterized unit tests with parameters that explore the behaviour of the unit. A common assumption in software testing research is also that simply providing the developer with a small set of efficient test cases will make the task of testing easier (e.g., [8], [15], [20]). However, it is unclear whether the need perceived by *researchers* meets the actual demands of *practitioners*.

To gain insight into common practice and needs in unit testing, we conducted an online survey using the global online marketing research platform *AYTM*[1]. As we are not aware of a previous use of this source for surveys in software engineering research, we tried to establish the quality of responses, and how to best use this source of data for research. We queried responses from AYTM's global pool of 20 million respondents in several iterations, and in this paper present the results of the final refinement of the survey and the qualified responses to it. Based on this data, we investigated the following research questions on unit testing:

RQ1: What motivates developers to write unit tests?
RQ2: What are the dominating activities in unit testing?
RQ3: How do developers write unit tests?
RQ4: How do developers use automated unit test generation?
RQ5: How could unit testing be improved?

The survey responses suggest that developers are driven by their own conviction and use systematic approaches like code coverage. They seem to need more guidance on what makes a test good or how to write tests, and this could be achieved by providing new tools, which they are keen on using. To produce these tools and underlying techniques, an important insight for researchers is that the actual writing of unit tests is only part of the difficulty of unit testing — maintenance activities add further constraints, such as the need for unit tests to represent realistic scenarios. Besides these immediate implications for unit testing research, our experiences show the limits of what is possible with online marketing research platforms: In principle, everything is possible, but costs can quickly explode, e.g., if one requires a specialised respondent pool.

II. Survey Design

Before conducting the final survey, the underlying questionnaire and sampling approach went through several iterations. In this section we summarise the approach in detail.

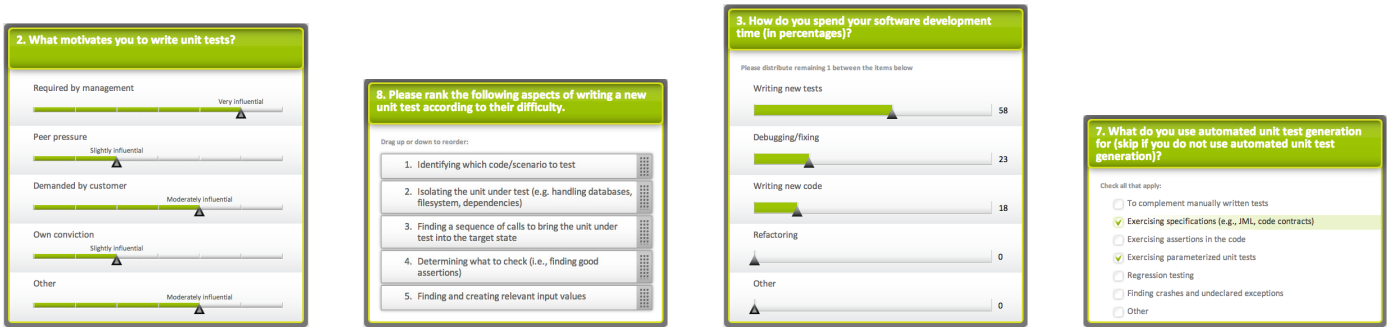
A. The Questionnaire

We designed the questionnaire in four iterations: Initially, we set up an online survey system (LimeSurvey[2]) on our web site and created a survey related to RQ2–RQ5. We advertised this survey in social media (Google+, Facebook, Twitter, LinkedIn), Usenet news groups, developer forums, and mailing lists of industrial contacts, and promised participants a chance to win an Amazon voucher of GBP 20. Within a time frame of two months we received no more than 30 completed responses. From this response rate it became quite clear that with the channels we were exploring we would not achieve a satisfying response rate for any revised follow-up surveys.

Nevertheless, the 30 initial responses allowed us to redesign and improve the survey questions. To have a hope of a satisfying response rate, we considered different commercial marketing research platforms, and decided on AYTM as their overall set of features and prices was most appealing. On November 30,

[1] http://www.aytm.com

[2] http://www.limesurvey.org/



(a) Rating question (b) Ranking question (c) Distribution question (d) Selection question

Fig. 1. Example questions on AYTM.

2013 we launched a revised survey on AYTM, and this time had 100 responses within 48 hours.

Though satisfied with the response rate, some of the data was rather difficult to interpret or surprising. For example, when asked about which techniques developers applied, almost 50% of participants claimed to be using automated test generation. This is more than we expected, therefore we decided to drill down further into this aspect, revised the survey questions again and added an open text question asking respondents to list the automated test generation tools they used. We ran a revised survey including this question in two batches: The first 50 responses were obtained in the time from December 19 to December 20. After analysing the data in detail, we launched a second batch on January 9, 2014, where participants from the first batch were excluded. The second batch completed 150 responses on January 13, 2014. Out of the 200 responses we received 153 answers to the open text question, and many of the tools listed were not actually test generation tools. There are 33 tools that we could not identify as testing tools, 42 tools that are related to testing (e.g., coverage analysis tools) but not to test generation. 23 tools are test generation tools; however, only 8 out of these are actually unit test generation tools. Interestingly, these are all tools from academia (Randoop, TestGen4J, GenUTest, Pathcrawler, PEX, Crest, JCrasher, Klover).

The quality of these responses put a question mark over all data retrieved via AYTM, and we thus designed a final iteration of the survey. In this final iteration, besides revising the questions, we added a qualification question (Q13) asking respondents to select all true statements out of a list of three questions about software testing, such that we can determine the quality of responses (two answers are expected to be true, the other one false). We intentionally asked this question last in the questionnaire, as we would expect some participants to be put off by an exam-like question at the beginning of the survey. The data from the revised version is the one on which this paper is based.

We had four different types of questions: For rating questions we used a 7-point Likert scale with answer choices adapted to the questions[3], for which AYTM provides an interface based on sliders (see Figure 1(a)). Ranking questions are provided using a drag and drop interface in AYTM (see Figure 1(b)). In the final revision of the survey we restricted the number of choices in ranking questions to five to increase chances of getting truthful responses. Distribution questions asked participants to distribute 100% on several available options. AYTM offers a very intuitive interface for this type of question which makes it easy and quick to answer (see Figure 1(c)). There is one choice question where respondents had to tick boxes of which options apply (see Figure 1(d)).

Table I lists the survey questions used (in the order in which they were asked, and together with the raw response data). To answer RQ1 (*What motivates developers to write unit tests?*) we asked this question to participants directly with a choice of options and a Likert scale rating. We investigated RQ2 (*What are the dominating activities in unit testing?*) by asking developers using two questions (Q2, Q3) how they perceive to be spending their time, where participants had to distribute the proportions between different coding and testing activities. To answer RQ3 (*How do developers write unit tests?*) we asked which aspects they find most important when writing tests (Q4) and which techniques they apply when doing so (Q5). To answer RQ4 (*How do developers use automated unit test generation?*) we asked participants to rate the frequency of different application scenarios of automated unit testing (Q6). Finally, to answer RQ5 (*How could unit testing be improved?*) we asked participants to rank the difficulty of different aspects of writing tests (Q7) and of fixing failing tests (Q8), and asked them about agreement to general statements about the difficulty of unit testing (Q9).

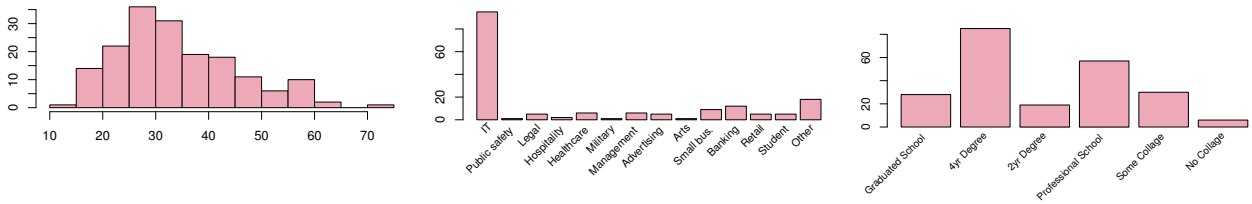
AYTM obtains its responses from a large pool of participants (>20 million), and demographics for this pool are already known such that we did not have to ask standard demographic questions. However, we added questions about the years of professional programming experience (Q10), size of the typical software development team (Q11), and programming languages typically used (Q12). Thus, in total respondents had to answer 13 questions, which should take less than an estimated 15 minutes to complete.

We configured AYTM to ask the survey questions in the same order to all respondents, but the order of answer choices was randomised for each respondent (except for options such as "Other").

B. Selection of Participants

The AYTM platform claims to have a pool of over 20 million respondents globally. Once a survey is launched, it is made available to this pool and members can choose to

[3]We started with 5-point scale questions, but increased the number of points to 7 in later revisions of the survey to increase discriminating power [4].



(a) Age (b) Background (c) Education

Fig. 2. Demographics of the survey respondents.

participate until the target response rate has been achieved. Respondents are paid a fee that depends on the number and type of questions. The fee to pay to AYTM also depends on the restrictions on the sample of the population, i.e., the more difficult it is to obtain the target sample the more expensive the survey is. For example, global responses are cheaper than US-only responses, and the more restrictions one puts on the sample, the further the price per response increases.

As AYTM has demographic information about its pool members, the target respondents can be filtered according to the available demographic information (e.g., gender, age, income, children, race, education, employment status, career, relationship status), and respondents can also be chosen from different pools geographically (World Wide, UK, US, etc.). We decided not to filter respondents by any of these criteria, as unit testing should be of interest to any software developer, regardless of these aspects.

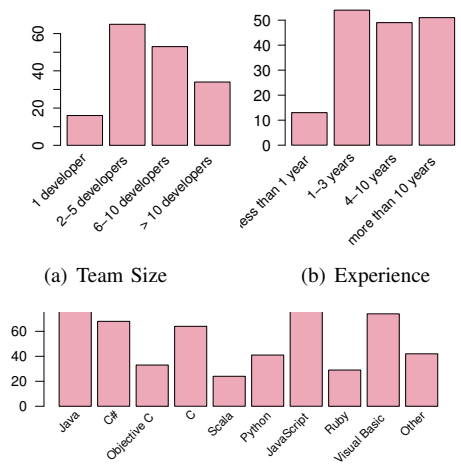
To target a questionnaire to specific groups (e.g., software developers), AYTM offers the possibility to ask up to three pre-qualification questions, such that respondents who do not answer the pre-qualification questions as desired will be ejected from the survey. We initially used the default choice of "software developers" offered by AYTM, which results in the pre-qualification question, "Are you a software developer?". Only respondents who answered "yes" were allowed to complete the survey. One can add up to three further custom questions, and we considered doing this in the final iteration of the survey; however, every additional pre-qualification question leads to a significant increase of the cost per question (in our case this doubled the price per response). Thus, in our case it was cheaper to include the pre-qualification question as part of the survey, and to filter the data *after* the survey has completed; this is what we did in our final iteration.

As our learning process on AYTM required several iterations of the survey, we hit the limits of the standard AYTM interface when trying to run repeated surveys. The standard interface offers only the possibility to exclude participants of *one* past survey. Thus, as soon as we wanted to post our third iteration of the survey, we were unable to select the two previous surveys for participant exclusion. The AYTM support staff is very helpful and responsive in this respect, but asks for a 25% fee to exclude further surveys.

A more severe problem we hit with AYTM was that after running several iterations of our survey, we no longer were able to achieve the response rates we desired with further iterations, while excluding all past participants. Interaction with AYTM staff revealed that the number of software developers contained in the pool of 20 million users varies frequently, and is apparently small, only in the order of a few hundred at most. Once we had finalised our survey, we were thus unable to get the sample sizes we would have liked to, and therefore had to make do with 100 global responses. In addition, we were able to retrieve 125 responses from the US pool (with the AYTM staff excluding overlap). For the final survey, AYTM charged a fee of US$3.85 per response for completing the survey from the global pool, and US$9.25 for per response for the US pool (plus exclusion fee). These two batches were launched on March 9, 2014, and both completed within 48 hours. According to a chi-square test with p=0.05 the two sets only significantly differ on Q6 (item "Finding crashes and undeclared expression"), Q7 (items "Find/create relevant input values" and "Isolating unit under test"). For the analyses in this paper, we thus combine the responses of these two pools, leaving analysis of geographic groups for future work.

C. Threats to Validity

In designing survey questions as unbiased as possible, conducting the survey, and analysing the results we tried to follow general guidelines for survey design [18]. Nevertheless, there are, of course, several threats to the validity of our findings. First, our sample has to be seen as a convenience sample of respondents available in the global AYTM pool. We used a simple pre-qualification question and a more thorough qualification question to filter the data. However, it may be that a targeted survey in a particular industry or domain would lead to different responses. The survey may suffer from the self-selection principle, as participants in AYTM's pool chose to participate in the survey on their own. This may bias the results towards people that are more likely to answer such surveys, for example people with more extra spare time or trying to earn extra money through AYTM. The use of payment for survey completion may in principle lead to participants trying to achieve their remuneration as quickly as possible, by clicking through the questions without seriously answering them. However, Q6 was the only question where no response is necessary to proceed to the next question: Rating questions require all items to be rated, distribution questions require 100% to be distributed, and ranking questions require at least one reordering action before proceeding to the next question is possible. Even though AYTM has a very convenient interface and accepts only surveys that can be swiftly answered, we still had 13 questions in total each with several answer choices. This may potentially reduce the reliability of responses. We analysed the inter-rater reliability for each question (see Section III-B), which is good for most questions, suggesting that the data is reliable. It is possible that our sample size of 225 in total (171 for the analysis) is too small; for example, some of the differences between individual sub-groups might turn out to be statistically significant if we had a larger sample. However, the



(a) Team Size (b) Experience

(c) Programming language

Fig. 3. Software development background of the survey respondents.

number of software engineers in the global pool of AYTM is limited, and we queried as many participants as possible given the available pool and our budget.

III. Survey Results

A. Demographics

Figure 2 and 3 show the demographics of our sample. The age ranges from 14 (the minimum age for AYTM respondents) to 72, with 37% of all respondents between 25 to 35. The majority of respondents work in IT (55.5%), although there is a mix of respondents from other domains as well. The largest share of respondents (59.06%) comes from the USA (necessitated by our two-part sampling), followed by India. However, in general the sample achieves a global spread with responses from Europa (32), North (143) and South America (13), Africa (1), Australia (1), and Asia (35). The majority of respondents have either a 4 year degree or a professional degree.

The programming languages most used are JavaScript and Java (Figure 3(c)), followed by C#, Objective C, Scala; thus, the majority of responses is based on programmers using object-oriented languages.

B. Suitability of online marketing research platforms for software engineering surveys

When conducting a survey, obtaining a good sample of responses from a representative population can be a challenging task. During our pilot studies, we did not receive many responses when publishing surveys in online discussion boards and social networking platforms, and so online marketing research platforms such as AYTM offer an attractive alternative. The large pool of respondents comes with demographic details one can use to specify the target population, and one has the ability to use qualification questions to filter the respondents. The standard pre-qualification question when selecting software developers in AYTM is a simple question of "Are you a software developer?", and the question remains how good the quality of the resulting sample is.

We designed an additional qualification question that requires a basic level of understanding of what unit testing is (Q13). The aim of this qualification question is not to filter out only experts on unit testing — quite the opposite, as our survey should target all software developers, good and bad. However, we would like to disregard potential respondents who try to cheat the system to collect payment from AYTM. For this, our assumption is that any software developer answering the questions truthfully should be able to classify at least two out of the three statements correctly.

Less than a quarter of the respondents (22%) classified all three statements correctly; 76% had at least two questions right, and 88% had at least one question correct. Of course some of those giving wrong answers may simply be less competent; for example, 32% of the respondents with more than 10 years of programming experience had all answers correct, whereas for less experienced respondents (up to three years of experience) that number is only 14%. However, assuming that a real software developer who reads the question and answers the survey questions properly rather than randomly should be able to get at least two answers correct, the 24% of responses with no or only one correct answers are "suspicious". Although it seems that the pre-qualification question ("Are you a software developer?") did a relatively good job at filtering out unreliable responses, our conclusion is therefore that it is not sufficient, and an additional qualification question is necessary. We thus only use the data of the 76% (i.e., the data of 171 out of 225 respondents).

There are two options provided by AYTM to improve the data sample: First, our qualification question could be asked as a pre-qualification question, such that respondents who do not answer it correctly are ejected from the survey. Second, as we learned from interactions with AYTM after conducting the survey, when using open text questions one has the possibility to reject (a certain percentage of the) answers that are obviously of low quality.

To analyse the inter-rater reliability [14], we calculated the intraclass correlation (ICC, using a two-way model with average-measures unit, and consistency type ICC measurement) for ordinal and Fleiss Kappa for nominal type questions in the survey (only for the 171 qualified responses). The details for each question are listed in Table I. There is only slight agreement ($k = 0.04$) in Q6, the only nominal type question in our survey. For five questions (Q1, Q2, Q3, Q5, Q9) ICC is in the excellent range $(0.86 < ICC < 0.98)$, suggesting that the variance in the data is due to true score variance between ratings rather than error variance. ICC is good for Q4 and fair for Q7 and Q8, but the confidence intervals are also much larger. However, considering the average inter-rater reliability, it seems plausible that for these questions there truly is less agreement, and this is not just the result of "random" answers.

C. RQ1: What motivates developers to write unit tests?

To better understand unit testing practice we would first like to understand *why* developers do unit testing. The first survey question (Q1) lists five options, each as a 7-point Likert scale rating question ranging from "very influential" to "not at all influential". Figure 4 summarises the responses to this question.

The main reason for unit testing is own conviction — developers test because they believe that testing is helpful. However, the requirement by management is listed as almost as influential as the own conviction. This is interesting as it is

TABLE I. SURVEY QUESTIONS AND RESPONSE DATA.

Question	Answer							IRR	95 %-CI	p-value
Q1: What motivates you to write unit tests?								0.965	[0.903,0.996]	< 0.01
	Extremely influential	Very influential	Moderately influential	Somewhat influential	Slightly influential	Hardly influential	Not at all influential			
Required by management	41	36	45	28	12	2	7			
Demanded by customer	32	19	58	29	17	9	7			
Own conviction	37	39	47	28	14	3	3			
Peer pressure	15	21	40	29	27	24	15			
Other	17	21	36	57	21	8	11			
Q2:How do you spend your software development time (in percentages)?	***(Listing averages)***							0.972	[0.921, 0.997]	< 0.01
Writing new code	33.04%									
Writing new tests	15.8%									
Debugging/fixing	25.3%									
Refactoring	17.4%									
Other	8.4%									
Q3: When a test case fails, what do you typically do? (Percentage of test failures)	***(Listing averages)***							0.985	[0.952, 0.999]	< 0.01
Fix the code until the test passes	47.2%									
Repair the test to reflect changed behaviour	25.9%									
Delete or comment out the failing test or assertion	15.6%									
Ignore the failure	11.2%									
Q4:How important are the following aspects for you when you write new unit tests?								0.621	[0.082, 0.922]	0.0153
	Extremly important	Very important	Moderately important	Neutral	Slightly important	Low importance	Not at all			
Code coverage	30	41	51	34	12	2	1			
Execution speed	28	32	60	31	11	7	2			
Robustness against code changes (i.e., test does not break easily)	32	36	57	31	10	4	1			
How realistic the test scenario is	37	57	35	29	8	3	2			
How easily faults can be localised/debugged if the test fails	37	33	58	28	11	3	1			
How easily the test can be updated when the underlying code changes	31	30	60	33	12	4	1			
Sensitivity against code changes (i.e., test should detect even small code changes)	30	35	57	30	12	3	4			
Q5:Please select which techniques you apply when writing new tests (select "Never" if you don't know a technique)								0.921	[0.81, 0.984]	< 0.01
	Always	Very frequently	Frequently	Occasionally	Rarely	Very rarely	Never			
Automated test generation (e.g., test inputs are created automatically)	22	28	43	40	15	5	18			
Code coverage analysis	27	31	47	44	8	5	9			
Mutation analysis	22	18	44	34	11	10	0			
Test-driven development (i.e., tests are written before code)	24	26	46	44	13	12	6			
Systematic testing approaches (e.g., boundary value analysis)	29	29	53	40	7	2	11			
Mocking/stubbing	25	20	40	41	18	10	17			
Other	21	11	32	59	11	14	23			
Q6:What do you use automated unit test generation for (skip if you do not use automated unit test generation)?								0.04	-	0
Exercising specifications(e.g., JML, code contracts)	51	Finding crashes and undeclared exceptions			63					
Exercising assertions in the code	58	Regression testing			58					
Exercising parameterised unit tests	58	To complement manually written tests			12					
Other	12									
Q7: Please rank the following aspects of writing a new unit test according to their difficulty.								0.477	[-0.496, 0.935]	0.113
	Position 1	Position 2	Position 3	Position 4	Position 5					
Determining what to check (i.e., finding good assertions)	35	32	32	35	37					
Finding and creating relevant input values	40	30	39	34	28					
Identifying which code/scenario to test	28	26	37	39	41					
Finding a sequence of calls to bring the unit under test into the target state	41	42	26	34	28					
Isolating the unit under test (e.g., handling databases, filesystem, dependencies)	27	41	37	29	37					
Q8: What makes it difficult to fix a failing test? Please rank by importance.								0.538	[-0.295, 0.944]	0.0713
	Position 1	Position 2	Position 3	Position 4	Position 5					
The test reflects outdated behaviour	40	45	29	33	24					
The test is difficult to understand	35	40	29	38	29					
The code under test is difficult to understand	38	27	32	32	42					
The test reflects unrealistic behaviour (e.g., unrealistic mock objects)	31	37	39	20	44					
The test is flaky (i.e., it fails nondeterministically)	27	22	42	48	0					
Q9: Please indicate your level of agreement with the following statements.								0.865	[0.653, 0.978]	< 0.01
	Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly Disagree			
Writing unit tests is difficult	19	21	47	47	20	12	5			
I enjoy writing unit tests	30	22	35	41	19	18	6			
I would like to have more tool support when writing unit tests	34	27	65	32	7	4	2			
I would like to have more unit tests	27	26	42	43	21	7	5			
Maintaining unit tests is difficult	26	14	56	40	21	9	5			
I usually have sufficiently many unit tests	28	28	53	40	10	9	3			
Q10: How many years of professional programming experience (i.e., in your job) do you have?										
	less than 1 year	1-3 years	4-10 years	more than 10 years	No prof. exp.					
	13	54	49	51	4					
Q11: How large is the typical software development team you work in?										
	1	2-5	6-10	More than 10						
	16	65	53	34						
Q12: Which programming languages do you commonly use?										
	Java	90	Python	41						
	C#	68	JavaScript	95						
	Objective C	33	Ruby	29						
	C	64	Visual Basic	74						
	Scala	24	Other	42						
Q13: Please select all statements that are true										
A unit test is a small piece of code testing a unit under test	126	(Correct = yes)								
Unit tests are written (or generated) by the developer	132	(Correct = yes)								
If all unit tests pass, the software is correct	133	(Correct = no)								

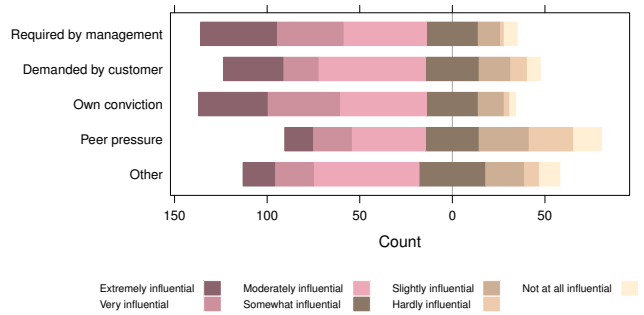


Fig. 4. What motivates developers to write unit tests?

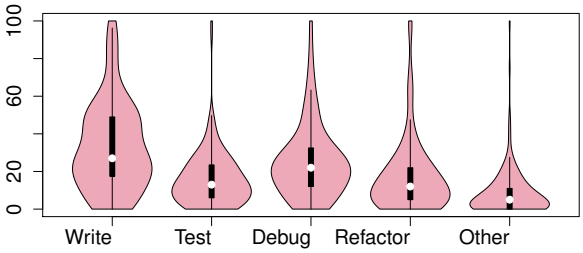


Fig. 5. How do developers spend their development time? The white dot in the violin plot is the median; the black box represents the interquartile range, and the pink region represents the probability density, ranging from min to max.

commonly said that management will happily cut down testing time in order for a product to be delivered on time. However, given this response it is reasonable to assume that unit testing is a standard practice. Customer demand is listed as quite important, too, though clearly lower than the main two reasons. Likely in many cases customers will not know enough about software development to make any demands on unit testing. Peer pressure is rated comparatively low; maybe being "test infected" (i.e., being hooked to the concept of writing tests) is not infectious after all. In fact, the "other" response got higher agreement than peer pressure, suggesting that there are reasons we did not think of.

From the point of view of unit testing research, this means the two main criteria that will be influential on whether a new technique will be adopted is a) whether measurable benefit can be provided to convince management, or b) whether developers see an immediate benefit. For example, a great technique without a robust tool is unlikely to be adopted by practitioners.

RQ1: The driving forces behind unit testing are developers' conviction, and management requirements.

D. RQ2: What are the dominating activities in unit testing?

How much time do developers believe to spend on unit testing, and what are the dominant activities when working with unit tests? While we cannot measure hard data on how much time is spent on what with a survey, we can query developers' perception, which will help us to understand how receptive they might be to different unit testing advances produced by software engineering research.

1) Time spent on writing unit tests: Figure 5 shows the time that programmers estimate to spend on different coding activities (Q2). The violin plots show median, minimum, maximum and quartiles like a box plot, but in contrast to regular box plots additionally shows the probability density of the data.

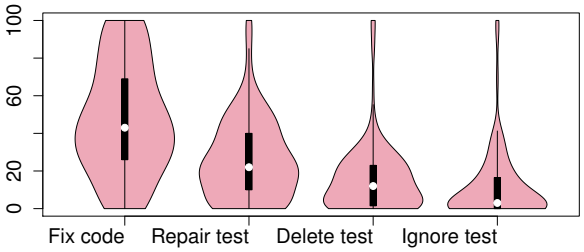


Fig. 6. What do developers usually do when a test fails?

As expected, the largest chunk of time (33.04% on average) is spent on writing new code. The second largest chunk of time is debugging (25.32%), followed by refactoring (17.4% on average), and writing tests (15.8%). In total, developers estimate that they spend 66.96% of their time doing things other than coding, which seems realistic.

Although out of the named activities, the least time is perceived to be spent on writing tests, the time claimed to be spent on writing new tests is still quite substantial compared to the time spent on writing new code: almost half as much time as for writing new code is spent on writing new tests. This reminds of the often cited estimate that 50% of the software development time goes to testing. In terms of writing code and tests, this may not be true, and one may speculate that spending more time on writing tests would be necessary. On the other hand, the broader term "testing" will include the treatment of failing tests, i.e., debugging and fixing, and in that sense developers perceive to be spending close to half (42.72%) of their time on activities related to testing. In that sense, expecting more time being spent on writing tests is maybe not realistic, but there clearly is potential for unit testing research to help developers produce *better* tests that make debugging and fixing easier.

Notably there are some extreme cases for time spent on writing new tests; for example, there are respondents who spend much more time on testing. These might include developers who are "test infected". Interestingly, there are 21 respondents who declare to do *no* testing at all — 12% of all (qualified) respondents!

We investigated whether there are any trends when comparing different groups of developers, and checked statistical differences between groups by calculating the median value and the 95% confidence interval. Confidence intervals were calculated using bootstrapping with 1,000 replications [6]. If the confidence intervals of two groups do not overlap, then the difference between these groups is statistically significant. In terms of programming experience there seems to be a slight trend that with increasing experience less time is spent on writing tests, but there are no significant differences. However, it is reasonable to assume that with increasing experience developers become more efficient at writing their tests — or more sloppy. There are no differences in terms of the programming languages used; however, respondents were allowed to select more than one language. Considering the team size, there seems to be a slight trend that more time is spent on writing tests in larger teams.

2) Handling of failing unit tests: When using unit testing, the time spent on debugging and fixing (25.32%+17.40% of the development time on average) is heavily influenced by the tests

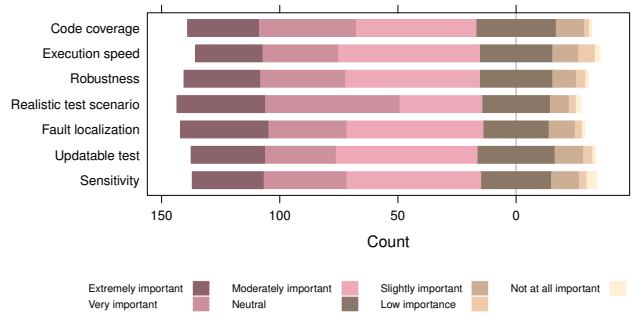


Fig. 7. Which aspects do developers aim to optimise when writing tests?

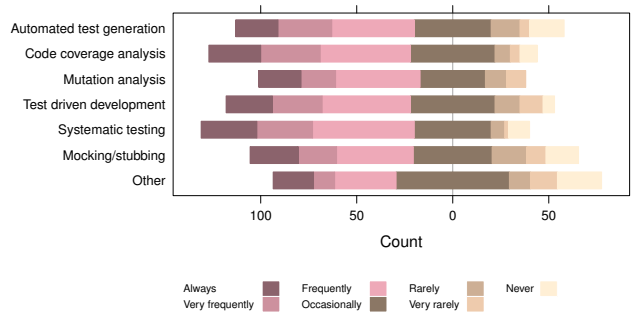


Fig. 8. Which techniques do developers apply when writing new tests?

— every failing test requires inspection and corrective actions. To investigate this, we asked respondents to estimate how often they react in different ways to a failing test (Q3), again in terms of percentages. Figure 6 summarises the responses: In almost half of the cases (47.2% on average), developers think that unit tests fulfilled their purpose and detected an error that requires code fixing. However, in 52.8% of the cases they believe that is not the case, and either have to fix the tests, or do not treat the failure at all (either by deleting or by ignoring the test). This is plausible — during development, functionality can change, and tests need updating. This is, however, an important point for automated unit test generation: Automated test generation research is typically driven and evaluated in terms of how much and how difficult code can be covered. However, depending on the usage scenario, unit testing tools are not used as fire-and-forget tools — they produce unit tests that need to be maintained. For this, different aspects such as readability and robustness against code changes are very important.

Again, the results are quite homogeneous across groups. Developers with more than 10 years experience claim to fix code more often than the others, and claim to ignore or delete failing tests less often than other groups. On the other hand, inexperienced developers claim to spend more time on repairing tests than other experience groups. Notably, less experienced developers (1-3 years of experience) delete tests significantly more often than developers with more than 10 years of experience.

RQ2: Writing new tests is perceived less dominant than writing, refactoring, and fixing code.
More often than not, a failing test is treated with a fix of the test (rather than the code) or deletion of the test.

E. RQ3: How do developers write unit tests?

The respondents claimed to be spending about 16% of their time on writing unit tests; the next research question now looks at *how* developers write these tests, by asking two questions.

1) What do developers optimise their tests for?: The first question (Q4) asked respondents to rate which aspects they try to optimise when writing new unit tests, listing several options. Figure 7 displays the results; it turns out that this is a tricky question: The answer choices are all generally regarded as properties of good unit tests, so most respondents tend to agree with all of the choices to a good degree. The highest level of agreement can be seen for "realistic test scenario", which is of particular interest for unit test generation tools. For example, a recent study [13] found that using a random unit test generation tool all of 181 reported failures were due to unrealistic scenarios in the context of the system. However, this is clearly not a solved problem and requires further attention from testing researchers, and it is a prime concern for developers.

Execution speed and sensitivity against code changes are aspects that developers seem to regard less important than the other properties, but overall there is rather strong agreement to all options, and it is difficult to discern real trends – developers seem to be claiming that they are optimising their tests to everything that makes a good test, at least to some degree.

While the conclusions we can draw about what specifically developers do when writing a new test are limited, this is still useful information for research on automated unit test generation: Random unit test generation tools, which arguably are very popular among testing researchers, typically optimise for *none* of these aspects. Coverage oriented tools tend to optimise for *one*: coverage, which is not a top rated concern. The other dimensions are areas where there are opportunities for automated testing research, in particular making the tests robust against changes, or easy to update.

2) Which techniques do developers apply when writing tests?: The second question in this group (Q5) asked respondents to select techniques that they apply when writing their unit tests. We listed five different techniques, again asking for a rating on a 7-point Likert scale. Figure 8 shows the results of this question: Most developers claim to apply some systematic approach when writing unit tests. (In the question, we listed boundary value analysis as an example of a systematic technique.) Code coverage analysis is listed as the second most frequently applied technique. These two techniques could be seen as related: If one analyses code coverage, then it is likely that one also systematically aims to cover code. Test-driven development also seems to be quite common practice, suggesting that many of the tests are written before the code.

Somewhat surprising, almost 54% of developers answered to use automated test generation at least "frequently". This result needs to be taken with a grain of salt; as we saw from our pilot studies the concept of automated test generation is not something that developers are particularly familiar with, and so, unlike for more common techniques like code coverage, the interpretation of what is automated test generation is not coherent across developers. Clearly, this suggests that practitioners need to be made better aware of progress in automated unit testing research. However, in discussions with developers one sometimes observes an indisposition against the idea of automated test generation, with arguments about

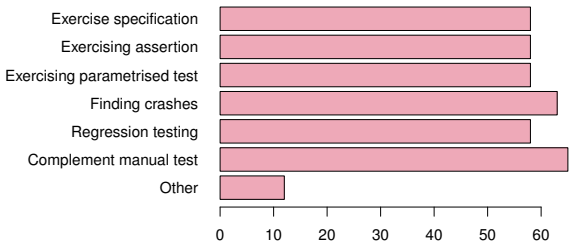


Fig. 9. Common usage scenarios of automatic test generation.

how the act of writing the tests improves the software. The response to this question seems to suggest that this is not the prevailing opinion, and developers may be more open minded about automated test generation than often believed.

Mutation analysis is applied less often than other techniques, but we would classify mutation analysis as an advanced technique – it is liked and endorsed by researchers, but would not be expected to be common in practice. Thus, the result that 69% of the respondents claim to use mutation analysis at least occasionally is surprising: For example, recent workshops of the mutation testing community featured many discussions on why mutation analysis is not being picked up by practitioners. Although the survey results confirm that mutation analysis is not as common as code coverage, it does seem that mutation analysis is more common in practice than believed. Indeed we have subjectively perceived a recent increase of availability of new mutation analysis tools for various languages, which would confirm this trend.

At the lower end of the scale, stubbing and mocking are less common than the other techniques. This is of relevance to automated test generation, as high coverage could easily be achieved by automatically generating stubs of all dependencies and tailoring their behaviour to what is needed to cover code. However, adding stubs may cause maintenance problems and unrealistic behaviour – which are aspects developers would like to optimise in their tests.

RQ3: Developers claim to write unit tests systematically and to measure code coverage, but do not have a clear priority of what makes an individual test good.

F. RQ4: How do developers use automated unit test generation?

The previous question suggested that automated unit test generation is applied quite frequently in practice. The question now is, how is it used?

Figure 9 summarises which options respondents selected in Q6: The main application area for automated test generation tools in unit testing is to complement manual tests. From a test generation research point of view, this is relevant as it suggests that such automatically generated tests are elevated to the same importance as manually written tests, and will therefore suffer from the same problems: Test oracles need to be devised, failing tests need to be debugged, and test code needs to be maintained, posing challenges on the readability and other aspects.

The second most frequently given answer is that automatically generated tests are used to find crashes. Indeed, this is

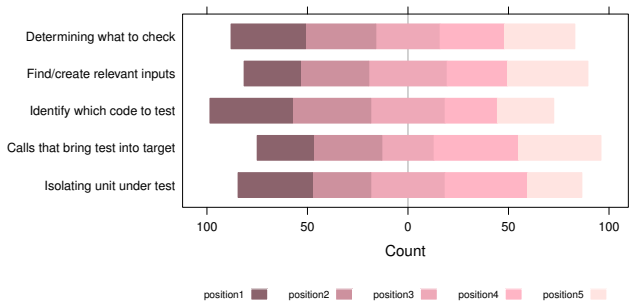


Fig. 10. What is most difficult about writing unit tests?

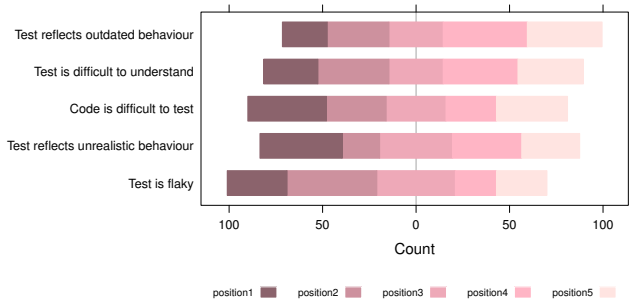


Fig. 11. What is most difficult about fixing unit tests?

what automated test generation tools are particularly good at, as this can be done fully automatically without any specification or test oracles. Here, there is a difference between the US and the global response pool: Significantly more respondents from the global pool (49%) claimed to apply automated test generation for this than US respondents (31%).

Exercising specifications, assertions, and parameterised unit tests are less frequent. This is unfortunate, as the existence of specifications and related artefacts would allow the application of test generation tools in the same fully automated way as they seem to be more popular to find crashes and undeclared exceptions. However, it is conceivable that this would change with availability of more advanced (and usable) automated test generation tools: Automated test generation could serve as better incentive to write assertions or code contracts. Regression testing is also ranked less often, and again that seems like a missed opportunity, as in regression testing the previous program version can typically take the place of the specification, thus allowing full automation in principle.

RQ4: The main uses of automated test generation are those that do not require any types of specifications.

G. RQ5: How could unit testing be improved?

The final part of our questionnaire aims at understanding where developers perceive difficulties and potential for improvement. What do developers believe to be the most difficult aspects of unit testing? To answer this, our questionnaire posed two ranking questions (Q7, Q8) about what is most difficult about writing and fixing tests, respectively, and one question of agreement to general statements about unit testing (Q9). For the ranking questions, respondents were given five answer choices and were requested to rank them by their difficulty. In earlier iterations of the survey we offered more choices, and reduced them to five in order to increase chances of obtaining useable data. We analysed the data for these questions using

TABLE II. WHAT MAKES IT DIFFICULT TO WRITE A NEW UNIT TEST?

Rank	Aspect	Borda count
1	Identifying which code to test	552
2	Isolating unit under test	521
3	Determining what to check	520
4	Finding relevant input values	493
5	Finding a sequence of calls	479

TABLE III. WHAT MAKES IT DIFFICULT TO FIX A FAILING TEST?

Rank	Aspect	Borda count
1	The test is flaky	549
2	The code under test is difficult to understand	526
3	The test reflects unrealistic behaviour	522
4	The test is difficult to understand	499
5	The test reflects outdated behaviour	469

Borda count: For each answer choice a score is calculated such that every time the answer is ranked last 0 points are added to the score, up to 4 points for each first rank.

1) Writing new unit tests: Figure 10 shows the results on what makes it difficult to write a new unit test (Q7), and Table II summarises the Borda ranking. In contrast to the previous questions, the inter-rater agreement has much larger confidence intervals, suggesting that there is less agreement.

According to the Borda ranking (Table II), the most difficult aspect is to identify which code to test. In automated unit test generation, this aspect is generally addressed by using coverage criteria (e.g., search-based testing), ignored completely (random testing), or irrelevant because all paths are explored (dynamic symbolic execution). Considering that 149 out of 171 respondents answered to use code coverage tools at least occasionally, one would expect that this should be an easy problem—a code coverage tool would point out exactly which code to test. However, as also indicated in the lower agreement on the priority of code coverage when writing new tests (Section III-E), the problem of deciding what to test seems to be more difficult than that, and developers proceed fundamentally differently to automated test generation tools. Indeed, the responses to Q4 confirmed that developers are striving to find realistic scenarios. Under this light, it is questionable whether the use of code coverage to drive automated test generation will make developers happy in the long run.

Ranked second is the problem of what to check in a test. Even if automated test generation tools would solve the problem of generating realistic tests, there remains the oracle problem that developers ranked highly. The checking is often assumed to be relayed to some sort of specification (e.g. assertions) or left as a problem for the tester (the well-known *oracle problem*). Consequently, this ranking can be seen as positive reinforcement for research in this area, for example to support developers in adding test oracles (e.g., [10], [25]).

The third aspect, ranked very similarly to the oracle problem, is to isolate the unit under test. Indeed this is also a major challenge for automated unit test generation [7], but there is little tool support one could get in this task (e.g., [1], [5]). The test generation literature often ignores this aspect, and it seems there is opportunity to improve software testing in this respect. This also raises the question to what extent this problem is an effect of low testability (i.e., bad design) in the code written by developers, and whether improving automated tools to cope with bad code is the right approach is debatable. Interestingly, respondents from the US pool ranked this issue lower (4th rank; difference in ranking is significant) than respondents from the global pool (2nd rank). However, for the US respondents the first four items of the ranking are closer together in general.

The ranking shows a group of aspects ranked clearly lower, consisting of finding the right input values and the right sequences of calls. Interestingly, this is precisely what automated test generation tools are being optimised for and evaluated against.

We looked at the ranking for different sub-groups, but largely found agreement with the global ranking. Notably, inexperienced programmers rank the problem of what to check higher, whereas there is little agreement on how difficult it is to determine relevant input values. With increasing experience the problem of deciding which code to test also seems to become more important.

2) Treating failing tests: Figure 11 shows the results on what makes it difficult to fix a failing unit test (Q8), and Table III summarises the Borda ranking.

The inter-respondent agreement is slightly higher than for Q7, although the confidence intervals are still large. The top ranked response of what makes a test difficult to fix is if the test is flaky, i.e., it fails sometimes, but not always. For example, tests that depend on non-deterministic code, environmental properties, or multi-threaded code are inherently difficult to debug and thus to fix. To some degree, the problem with a flaky test is in the code, not the test, and thus the second ranked choice is related: Tests are difficult to fix if the code under test is difficult to understand. This is not unexpected, but from a test generation point of view the code usually has to be taken as a given, and improvements can only be achieved in the generated tests. However, ensuring that generated tests are not flaky is an important question, and we are not aware of any work in that area.

The last three options reflect properties of the tests, rather than the code. Out of the three, the top ranked option is the problem of unrealistic tests, and this mirrors the concern of developers to derive realistic scenarios when writing tests (Q4). This is cause for concern for automated test generation tools — how should an automatic test generation tool distinguish between realistic and unrealistic behaviour? Specifications or other artefacts may provide hints; a recent idea is also to use the system context of a unit to determine realistic behaviour [13], but this may not always be an option in unit test generation, leaving an open research problem.

This is followed by the test being difficult to understand, and considering that understandability of tests is ranked less difficult than understandability of code, there may be an opportunity for automated test generation: If good tests can be generated, then these may help in understanding the code. Finally, the last ranked option is if the test represents outdated behaviour — which one would actually expect to be a common case when software evolves.

3) General impressions: The final question (Q9, followed by the demographic questions in the questionnaire) of the survey asked respondents to specify their level of agreement with different statements about unit testing. Figure 12 shows these statements and the responses. A clear response is given to the question of whether more tool support is needed when

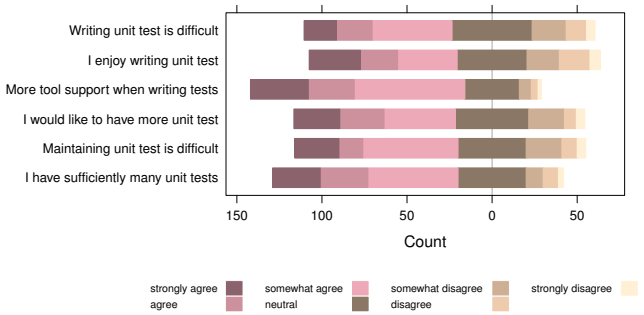


Fig. 12. General questions on perception of unit testing.

writing tests — developers seem to be very open to this idea, and eager to see more tools.

Interestingly, there is quite strong agreement that developers already have sufficiently many tests, although still quite many developers state they would like to have more tests. On the other hand, the lowest agreement can be seen for the question of whether developers actually enjoy writing these unit tests. With only about half of the developers attributing a positive feeling to writing unit tests, there seems to be a clear need to act; providing advanced tools that help developers may be one way to make testing more enjoyable and productive; for example, by providing the tools that generate the tests automatically. On the other hand, in line with Q3, there is more agreement by developers that maintaining unit tests is difficult compared to writing unit tests. This poses a challenge to automated test generation tools, where research is still very much focused on solving the latter problem, with few exceptions (e.g., [23]).

RQ5: Developers do not seem to enjoy writing tests, and want more tool support—in particular to identify what to test, and how to produce robust tests.

IV. Related Work

Although there is a body of empirical results on unit testing [17], surveys are used less frequently in software engineering compared to other disciplines such as marketing and psychology [16]. Most closely related to our survey, Runeson conducted a survey on unit testing practices [24]. This survey asked 50 questions related to the questions of 1) what is unit testing? and 2) strengths and weaknesses of unit testing. Our interpretation of what unit testing is matches the one that emerged from this survey; some of the weaknesses identified in Runeson's survey also emerged in our survey. For example, identifying units to test is difficult, as is maintenance of tests. In general, our survey differs from Runeson's survey by focusing more on identifying potential where automated unit test generation can improve testing.

Torkar and Mankefors [27] conducted a survey on software reuse and testing. This survey shares some of the insights of our survey; for example, boundary value analysis is commonly applied by developers, and their survey also revealed a strong desire of developers for better tools when writing unit tests.

Geras et al. [12] conducted a survey on testing practices in Alberta, and found that unit testing was the most popular testing approach, yet was only applied by 30% of testers. A larger survey in all of Canada [11] confirmed the relative popularity of unit testing, and interestingly also identified growing interest in mutation analysis, similar to our results. Ng et al [21], in their survey of software testing practice in Australia, also found a high willingness to use automated tools, as our results also suggest. The need for tools to support automation was also observed in Lee et al.'s survey [19]. Causevic et al. [2] found that unit testing was more popular for web applications than for other domains such as embedded systems. Of relevance to our findings is also that they found that open source tools are mostly used for unit testing, whereas commercial testing tools are used more for higher levels of testing. Zhao and Elbaum surveyed open source projects [28] and found that, although testing consumes significant resources, only about half of the projects had a baseline test suite, and only few projects used coverage analysis tools to support their testing.

V. Conclusions

The need to improve software quality is generally accepted, but the question remains how to achieve this. Improving unit testing is one possibility, and software engineering research is providing practitioners with new techniques that can improve the quality and productivity of their testing activities. Our aim was to better understand common practice in unit testing, in order to identify possibilities for improvement, in particular automated unit test generation.

Our online survey of 171 participants from 29 countries confirmed that unit testing plays an important role during software development: Developers spend a good amount of their time on writing tests. There clearly is room for more testing: 12% of the developers do no testing at all, and 73% stated a strong desire to have more tests. To strengthen our findings we are exploring the use of controlled experiments (e.g., [9]), and plan to use survey replications as well as observational methods (e.g., industrial case studies) as future work.

In general, the responses point out areas of unit testing where automated unit test generation could support developers, and where further research is necessary:

- Developers need help to decide *what to test*, rather than which specific input value to select. Most research on automated unit testing side-steps this problem (e.g., by making the choice random or driven by code coverage).
- Unit tests need to be *realistic*: A prime concern of developers when writing new tests, but also when treating failing tests, is whether they are realistic. An unrealistic scenario will make it more difficult to fix the test, and developers may not be convinced to fix their code based on unrealistic tests.
- Unit tests need to be *maintainable*: Even when automatically generated, a unit test may be integrated into the regular code base, where it needs to be manually maintained like any other code. If the test no longer reflects updated behaviour, it needs to be easy to detect this, and it needs to be easy to fix it.
- Unit test generation is most efficient at determining input values, but the unsolved challenge is to determine *what to check*. Ultimately, this means the long standing test oracle problem bugs unit testing like any other test generation approach, and advances on this end will make test generation tools more valuable to practitioners.

Acknowledgments. Thanks to Jose Miguel Rojas and Jose Carlos Campos for comments on earlier versions of this paper.

REFERENCES

[1] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *International Conference on Automated Software Engineering (ASE)*. ACM, 2014. To appear.

[2] A. Causevic, D. Sundmark, and S. Punnekkat. An industrial survey on contemporary aspects of software testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 393–401. IEEE, 2010.

[3] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 231–255. Springer-Verlag, 2002.

[4] R. A. Cummins and E. Gullone. Why we should not use 5-point likert scales: The case for subjective quality of life measurement. In *Proceedings of the Second International Conference on Quality of Life in Cities*, pages 74–93, 2000.

[5] J. de Halleux and N. Tillmann. Moles: Tool-assisted environment isolation with closures. In *International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 253–270. Springer-Verlag, 2010.

[6] B. Efron and R. Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[7] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[8] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[9] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings International Symposium on Software Testing and Analysis*, pages 291–301. ACM, 2013.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 28(2):278–292, 2012.

[11] V. Garousi and J. Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.

[12] A. M. Geras, M. Smith, and J. Miller. A survey of software testing practices in alberta. *Electrical and Computer Engineering, Canadian Journal of*, 29(3):183–191, 2004.

[13] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2012.

[14] K. A. Hallgren. Computing inter-rater reliability for observational data: An overview and tutorial. *Tutorials in quantitative methods for psychology*, 8(1):23, 2012.

[15] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 182–191. IEEE, 2010.

[16] A. Höfer and W. F. Tichy. Status of empirical research in software engineering. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, pages 10–19. Springer, 2007.

[17] N. Juristo, A. M. Moreno, S. Vegas, and M. Solari. In search of what we experimentally know about unit testing. *Software, IEEE*, 23(6):72–80, 2006.

[18] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[19] J. Lee, S. Kang, and D. Lee. Survey on software testing practices. *IET software*, 6(3):275–282, 2012.

[20] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, pages 1–4. ACM, 2010.

[21] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen. A preliminary survey on software testing practices in Australia. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 116–125. IEEE, 2004.

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84. IEEE Computer Society, 2007.

[23] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *International Conference on Automated Software Engineering (ASE)*, pages 23–32. IEEE Computer Society, 2011.

[24] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

[25] M. Staats, G. Gay, and M. P. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *International Conference on Software Engineering*, pages 870–880. IEEE Press, 2012.

[26] N. Tillmann and J. N. de Halleux. Pex — white box test generation for .NET. In *TAP'08: International Conference on Tests And Proofs*, volume 4966 of *LNCS*, pages 134 – 253. Springer, 2008.

[27] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Software: Science, Technology and Engineering, 2003. SwSTE'03. Proceedings. IEEE International Conference on*, pages 164–173. IEEE, 2003.

[28] L. Zhao and S. Elbaum. Quality assurance under the open source development model. *Journal of Systems and Software*, 66(1):65–75, 2003.