WAP: Cognitive Aspects in Unit Testing

The Hunting Game and the Hunter's Perspective

Marllos P. Prado*†‡, Eric Verbeek*, Margaret-Anne Storey *, Auri M. R. Vincenzi§
*University of Victoria
Victoria B.C., Canada
Email: {mprado, everbeek, mstorey}@uvic.ca
†Instituto Federal de Goiás
‡Universidade Federal de Goiás
Goiania GO, Brazil
§Universidade Federal de São Carlos
Sao Carlos SP, Brazil
Email: auri@ufscar.br

***Abstract*—Humans are hunters and love the chase—they hunt for food, they hunt for bugs in software. In the last decade, testing research has gone deeper and broader to help with the challenging task of catching bugs. Much of the literature approaches the problem from a theoretical-technical perspective and is often oriented to automated solutions. Yet, there is a gap between industry testing problems and research testing solutions. We take a different perspective and consider the human component as a major part of the solution for practical testing problems. Many of these human-related issues are reported in academic surveys of practitioners. We highlight the importance of human factors in testing by introducing a hunting metaphor. We also bring attention to evidence on cognitive support demands in unit test practices. An initial framework is proposed as an effort to bring understanding of cognitive support demands, and provides direction for further research on unit testing tools which support tester skill improvement.**

I. Introduction

Imagine a world where automated solutions perform software testing. Solutions that search for errors, discover promising inputs to exercise code where the humans have made mistakes. Better than that, imagine they can decide if the generated output is correct or not, according to the specification written by humans. Sounds perfect? This would be an ideal world—this is the holy grail of testing researchers.

Now think about the real world where one of the most remarkable testing tools, JUnit [1], is a simple framework made by practitioners to make developers' work lives easier. It has strongly impacted the way tests are built, in both industry and research. Instead of hands-on tests that are difficult to replicate and perform "on the fly", it allows modular and precise tests to be run and re-run, refactored and reused. It also incorporates the "red/green bar" feedback mechanism, a practical solution for simplifying one's perception of the test suite's state. Rather than getting the job done by itself, this tool facilitates the tester's skills in the task. We must also recognize that software companies deal daily with numerous quality issues in their products. Companies invest money and time to leverage employee expertise, and they cannot wait for a promissory technology to do all, or almost all of the work needed.

We refer to "testing" as the activity of detecting failures. This differs from "debugging", the activity of finding and eliminating faults responsible for failures. It is also worth mentioning that while automation is a relevant matter for testing, it cannot be an exclusive or self-contained solution.

We may explain this more easily with a metaphor; the human aspect of the software testing problem can be seen as hunting for game. Testers are the hunters, bugs are the game animals, testing tools are the weapons and snares, and strategies and tactics underlay both as techniques. In order to succeed, hunters must catch as many game animals as they can, especially the most valuable ones. A hunter relying only on their natural strength and human abilities will not be successful—specialized weapons and traps are needed for the hunt. But if these tools are hard to use or not designed to leverage hunter's skills and abilities, they can be cumbersome, take too long to master, and in the worst case, provide false expectations. Replacing people with automated weapons is less adaptable, and has weaknesses in finding some skittish and valuable game animals.

The purpose of this paper is to address how research in software testing has dealt with the tester's needs, how researchers adopting the tester perspective could help improve testing "weapons", and how this would ultimately serve practical needs in "bug-hunting" activities. There is a clear intersection here between two different fields: human-computer interaction (HCI) and software testing. Thus, this paper considers issues rooted in the tester's perspective (testers' claims, behavior, knowledge, etc.), and proposes a conceptual framework to help organize and understand the demands of testers as users, from a human-cognitive point of view. The evidence we consider focuses on the unit testing level. The proposed framework is composed of three cognitive dimensions: Tester Artifact Comprehension; Tester Orientation and Guidance; and Tool Interaction Usability. Our pilot framework is neither final nor low-level, but is proposed to promote evolution in research in this yet immature human aspect of software testing.

In Section II, the motivation for research on this topic is presented. Section III gives a brief background about testing activity, highlighting the importance of test cases and its different levels. Section IV presents the evidence of cognitive

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

support demands in software testing. Section V organizes the demands identified in Section IV into a framework proposal. Finally, Section VI concludes and discusses further research.

II. Hunters left out of the game

Amman and Offut noted that "[...] faults in software are design mistakes. They do not appear spontaneously, but rather exist as a result of some (unfortunate) decision by a human" [2]. In this statement, they tried to characterize the extent to which any process can hope to control software faults. But consider that software does not only mean the software as shipped to the end user—test suites, frameworks, and other tools are software, too. The operation and development of such test software can influence human decisions, and their usefulness can, in turn, be influenced by human decisions.

So why is automation so important? Software testing is a dynamic analysis of software, and research efforts have aimed at developing tools to support techniques, criteria and heuristics. The goal has been to transform the exhaustive testing problem into a feasible and automatic (or semi-automatic) solution for revealing failures. This approach seems to have reinforced the belief that the important testing solutions will originate from machines, to be executed by machines, while understating or forgetting the impact that the fundamental human-component in the middle of this loop may have. On the other hand, studies like that of Kasurine and Taipale [3] demonstrate that research results in automated solutions have not been adopted in practice, showing a disconnect between industry and research.

In a very recent paper, Orso and Rothermel [4] analyze the state of the art in software testing, spanning the last decade and a half. They summarized a number of research topics and presented challenges to be investigated. Their data is from a pair of open-ended questions collected from fifty testing researchers. Notably, although there is a wide variety of ideas in the "Challenges and Opportunities" section of their paper, the authors explicitly say that they would not consider "human factors" and "technology transfer" as research challenges, even though they declared that these topics were among those mentioned in their peers' responses.

However, in the subsection devoted to "Empirical Studies and Support for Them", Orso and Rothermel [4] mention the need for "users studies", the predominance of "automation of algorithms and heuristics" in testing research, and the concerns about threats to validity that research results are subject to when they fall into the "hands of engineers". They also note the influence that industry contributions have had on research (a reverse transfer of technology). They mention JUnit [1], a "straightforward [testing] framework", as an example of this influence. Altogether, this demonstrates the division between research and practice, and gives us pause for concern. Omitting human factors and technology transfer as "Challenges" seems inconsistent with the discussion in their paper. In the following, we consider what other researchers have to say about the role of the tester and transfer of technology in testing.

Daka and Fraser [5] recently conducted a broad survey of testers to investigate the current practice of unit testing. They found that the main motivation for conducting testing is the tester's "Own Conviction", which exceeds that of the "Management Requirements" placed upon them. On the other hand, only half of the respondents "attributes a positive feeling to writing unit tests". These findings are aligned with the findings of Ng et al. [6] and the survey carried out by Lee et al. [7]. In the former, the "difficulty of using [tools]" was identified as the major barrier in adopting a tool, and consequently the testing methods associated. In the latter, more than half of the respondents answered that software testing is performed based on personal knowledge and without guidance. As indicated by these results, despite being unpleasant work, resilience in the hunt for bugs can be motivated by genuine trust in its purpose.

The work of Jia et al. [8] draws attention to human-related problems, but gives little consideration to human participation in testing solutions. They reviewed mutation testing literature and discussed prospective research directions. The unresolved problem of "barrier[s] to wide application of Mutation Testing" is the "Equivalent Mutant Problem". The "Human Oracle Problem"—the checking of the original program output for each test case generated—is also cited as another barrier. Finally, the authors point out that "more tooling is required to ensure widespread industrial uptake" and that "automated practical tool[s]" for test case generation could be a promising way of creating test solutions. Throughout the paper, the authors discussed past and future connections between automation and mutation testing, but little mention was made of a human-in-the-loop as part of the solution. Considering that the Equivalent Mutant Problem is recognized as undecidable and that the test case generation requires an as-yet non-mechanized "human oracle", it is difficult to suppose that automation will be a panacea in the large spectrum of existing software development domains.

These are just a few examples of how the software testing research community has dealt with personal and practical human aspects of testing. The common threads among them demonstrate that the omission of both human factors and the transfer of technology that we see in Orso and Rothermel's work [4] is by no means an isolated occurrence. Thus, it is the research community in software testing as a whole that must claim *mea culpa*. Are we attacking the problems considering the primary stakeholders, those who deal with testing the most? We are researching powerful weapons, but are they satisfying the hunt? In order to leverage testers' skills and make their critical role in testing easier, how should we tailor their tools in an effective and ergonomic fashion?

III. Test cases and the quality of the shots

The quality of a hunter's shot determines if the target is hit or not. In this sense, a good shot can be classified as intentional, planned and capable of reaching the target. In order to be a good hunter, one must improve their skills at producing a good shot. A correspondence can be established with test cases in software testing: a test case embodies and enacts the intention of the tester. It is planned following a strategy and catches errors depending on whether assertions are passed or not. This way, focusing on the quality of the test case means enabling the tester to hone in on the failure.

A. *Techniques and Exploration in the Field*

The classical paradigm to apply software testing (sometimes called the test-case based approach) has its process organized in four steps: test case planning; test case design; test case execution; and test evaluation [9], [10], [11]. In this paradigm, test cases are generated and evaluated using testing techniques. A testing technique is classified in terms of the type of source used to derive the test cases. Testing techniques can be classified into three main categories: Functional or Black-box (based on specification information); Structural or White-box (based on the structure of the software under test); and Fault-based (based on common mistakes committed by developers). Such techniques are composed of testing criteria. A criterion defines which are the properties of the artifact under test (test requirements) that should be exercised, in order to evaluate the quality of test cases [12].

Exploratory testing (E.T.) is another paradigm to perform software testing. It relies on elements such as the knowledge of the tester, scenarios, and experience of the user domain to guide the activity. In this sense, it differs from the classical approach by being less formal. Within it, learning, design, execution, and modification of the tests can be parallel activities [13]. It does not imply that testing needs to be random or *ad-hoc*, since it can be governed by a strategy (such as the tourist metaphor explained in [14]). As mentioned by Kaner et al. [15], E.T. can be particularly useful in situations where the planned tests cases quickly become outdated, making automation impractical. This way, the idea of E.T. does not exclude the classical approach, but is actually complementary. To focus testing activities on the tester, it seems reasonable to start by understanding how testers think. The kind of study performed by Itkonen et al. [13] looks like a promising way to understand how practitioners think and act in E.T. in the real world. However, the same kinds of questions should also be investigated with the classical testing approach; this is the fundamental form of software testing used in both the literature and in tester training. Once we have scientific confidence about tester knowledge in these two main testing paradigms, the task of creating and adapting testing solutions to their practical needs will be possible.

B. *Adjusting Shots for the Scale in the Sights*

Considering the granularity of the system to be addressed when looking for failures, three main categories are commonly considered in the literature: unit testing, integration testing and system testing. The unit level is employed to find faults in the smallest cohesive working modules of the artifact being considered (usually the method, class, procedure etc.), depending on the paradigm and author interpretation; the integration level looks for faults particular to the interaction between these modules (method or procedural calls for example); the system level looks for errors resulting from the behavior emanating from all the units working together and the integration of the software with its environment (*e.g.* OS, databases, services). This way, different levels of testing demand different kinds of analysis from the tester.

In practice, integration and system level tests differ from unit level testing in who is assigned responsibility for generating tests. Since unit testing is a fine-grained test and requires fine grained knowledge, it can become inefficient and expensive for a test team to perform. Besides the great number of units to test, the test cycle could be long: (i) the developer implements a unit; (ii) the tester receives and understands the unit, implements test cases and finds the error; and (iii) the tester delivers it to the developer for resolution. Instead, developers are often assigned the responsibility for developing and testing units—or to test and then develop, in the test driven development (TDD) paradigm [16]. This way, the test team is free to concentrate on integration or system level problems, while developers leverage their fine grained knowledge in performing unit tests. On one hand, the transfer of unit testing to the developer is advantageous to testers and management, but on the other hand, it risks diverting the developer's mental resources away from development, their primary responsibility.

IV. Ready, Aim, Fire and Enjoy - Under the Storm

Cognition is an interdisciplinary topic and its study is generally found within "cognitive science". Many definitions of cognition can be found in the literature, as in Neisser [17], Norman [18] and APA [19]. A common idea among them is that a mental process is applied to build and use knowledge. In this section, we summarize the literature we gathered, which contains evidence of cognitive problems in testing. We group the identified demands into three proposed *dimensions*: Tester Artifact Comprehension Problems, Tester Orientation and Guidance Problems, and Tool Interaction Usability Problems.

A. *Tester Artifact Comprehension Problems*

Daka and Fraser [5] reported the results of a survey on unit testing conducted with software developers from different countries. The purpose of the study was to align the current research with existing common practices and identify research opportunities coming from the industry. The results of their second research question (RQ2) demonstrated that developers tend to treat failing tests as defects in the test cases themselves more often than defects in the code. They also reported (RQ4) that the top two developer uses of automated test generation are: (i) automation as a complement to manual testing, and (ii) discovery of failures of the "crash" type. These results indicate that both manual and automated testing are current and non-exclusive practices. Also, manual testing is probably dominant in areas with less obvious errors (i.e. "non-crash" types). As mentioned by Leitner [20], although automated test generation is effortless, it cannot substitute for manual testing because developers are experts on defining complex input data, and thus, defining effective test cases.

Nowadays, it is widely accepted that xUnit test frameworks are dominant in unit test activity. A notable example is JUnit. Besides wide adoption of JUnit in industry, it is also an example of reverse transfer of technology, as subsequent adoption in academia influenced and aided software testing research [4]. However, as observed by Atkison et al. [21], tools like JUnit still require tests to be written as programs, which is beneficial for their natural integration with the code base. Unfortunately, it also creates dependencies between details of the program code and language, and the strategy to test logic, test data, and later test results. The work of Lappalainen et al. [22] is concerned with similar issues in the educational

field. Based on identified difficulties of novice developers who are concurrently learning unit testing and TDD, their work proposes a tool to make test cases more readable and easier to write. We see similar observations in the study of Daka and Fraser [5]. They asked "how could unit testing be improved" (RQ5), in particular "what makes it difficult to fix a failing unit test". The results of interest are that: "the code under test is difficult to understand", "the test [itself] is difficult to understand", and "the test reflects unrealistic behavior".

B. Tester Orientation and Guidance Problems

Runeson [23] reported a survey to characterize unit testing as it occurs in practice. The survey included one round of focus group discussions with subjects and another with their respective companies. One of the results revealed that there was no consensus in what constitutes a unit under test. Despite the confidence exhibited by the companies' questionnaire responses in this matter, the focus group round revealed that it wasn't uncommon to consider a unit as a group of cohesive linked units instead of an individual one. Almost a decade after, Daka and Fraser [5] revealed that two of the most difficult aspects related to writing new tests correspond to (i) the identification of which code to test and (ii) the isolation of the unit under test. Similarly, the results of Runeson [23] and Daka and Fraser [5] revealed difficulties in the evaluation of unit testing. The former study indicated a need for clear and measurable test quality criteria, and the second revealed "what to check in a test" as a common difficulty in writing new tests.

Such concerns in the surveys of Runeson [23] and Daka and Fraser [5] are related to the uncertainty about what and how to perform the activity. They can be influenced by many causes, such as the lack of adequate training, the lack of strict standardization in the activity, different rigor demanded between projects and team backgrounds. The fact is that, independent of the reason, unit testing may still be underestimated by testers due its apparent simplicity compared to other levels of test. In this sense, some *guidance* could contribute to mitigate such problems.

Even when testers know how to create a test and what to test, they often lack *clues of reference* to rely on. Lee et al. [7] surveyed current testing practices and found that 50% of respondents "perform their software testing based on individual's know-how or personal knowledge without standardized guidance". This closely matches what Daka and Fraser [5] found in their third research question, which addressed how unit tests are written by developers. According to them, "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". Runeson [23] reported that due to the lack of strategies at companies, test cases were defined using the personal judgment of developers, "leading to varying practices".

Lee et al. [7] found that the usage of testing methods and tools in unit testing is especially weak, compared to other levels of testing. In fact, unit testing frameworks do not impose the application of any particular strategy or criterion. In this sense, they are like a blank slate where developers may define test cases in their own way. Moreover, Garousi and Zhi [24] and Ng et al. [6] agree that functional testing is a more popular approach than structural techniques in software testing as a whole. Considering that functional approaches rely on specifications to derive and evaluate the testing requirements, functional testing is more prone to interpretation and subjectivity than other techniques. Furthermore, functional end-to-end testing via a Graphical User Interface (GUI) requires a lot of script maintenance, demanding subsequent improvements of the underlying unit tests [25]. Thus, it is reasonable to question if the *lack of restrictiveness* of the aleatory combination between functional testing and unit testing could be leading to unproductive practices, such as the generation of a high number of low efficacy test cases.

C. Tool Interaction Usability Problems

Two last opportunities of investigation regarding cognitive support for testing are the apparent lack of enjoyability and ease-of-use of unit testing tools. Runeson [23] acknowledged the difficulty motivating developers as one of the weaknesses of unit testing. Daka and Fraser [5] corroborate this idea, revealing that only half of the developers they surveyed, attribute a positive feeling to writing unit testing. They also indicate that tool improvement is a path to address such problems. Kasurinen [26] revealed in his study—dedicated to understanding the problems of software testing in practice—that the usability and applicability of testing tools was identified by the companies surveyed as one of the factors that affect the efficiency of testing activity. In particular, large companies investigated in the study reported that false positives generated by the use of complicated or defective tools add additional overhead. Wiklund et al. [27] pointed to difficulties related to the configuration and use of the Integrated Development Environments (IDE)— Eclipse in particular—as a barrier commonly faced by testers when creating their test code in automated testing environments. In addition, Lee et al [7] claim that the relationship between software testing tools and their corresponding activities should be made more clear.

Delahaye and Bousquet [28] defined guidelines for comparing and selecting software engineering tools. They chose mutation testing as their case of study. Their *usability* criteria were segregated into four categories: compatibility, interface, documentation and surviving mutant management. In the "interface" category, they highlight the difficulty faced when operating these tools. Among the eight mutation tools they investigated, only two present a proper Graphical User Interface (GUI). Additionally, most of them do not integrate with popular IDEs. In "documentation" they looked for clear descriptions of the operations and processes executed by the tools. Only three out of the eight tools analyzed are ranked as having "very good" documentation, whereas five are ranked at the lowest level for the inferior quality of the "surviving mutant report" generated.

V. Synthesizing the Hunter Needs into a "Research Compass"

Cognitive support for testing could ameliorate the primary difficulties found in unit testing. Cognitive support can be understood as the aid a tool provides to the user in her mental effort of thinking, reasoning, and creating [29]. The coloring of a covered path in a control-flow graph of a structural testing tool is a simple example of existing cognitive support. It helps

the tester to keep focus on what remains to be covered in the code, and helps them to remember what part was already covered. The "green/red bar" of JUnit is another example of cognitive support that helps one avoid tracking if the test cases have changed their assertive state, between different testing executions. The unit testing problems discussed in Section IV indicate that non-trivial mental effort is required of the tester, despite any assistance from existing automated testing solutions. Based on this evidence, we propose a framework (Figure 1) combining the three cognitive dimensions discussed. The framework indicates initial directions of investigation, for the evolution of unit testing tools.

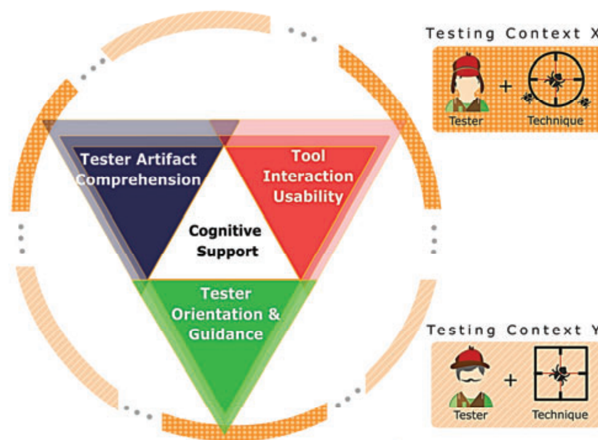

Fig. 1. Cognitive framework containing three dimensions of cognitive demands to be met by unit testing tools

- *Tester Artifact Comprehension:* Testers must reason about test cases, units under test, and test reports. They spend effort in comprehending these artifacts both individually and as collections. Different ways of reformulating artifact information could offload this cost. Since these problems center around comprehension of either testing code or program code, one way to address them is by using the contributions provided by *programming psychology* literature. It includes analysis of models of how human reads code and theories about how knowledge is built and used [30].
- *Tester Orientation and Guidance:* Testers can experience confusion or disorientation, at the cost of time and morale, which can decrease efficiency and commitment to testing. Tools should offer assistance to the testers by: providing direction for the testers in their personal cognitive work-flow (*guidance*); helping them make decisions under uncertainty or with ambiguous problems (*reference*); and adjusting the scope of tasks to bring focus to the task goals (*restrictiveness*). Examples include but are not restricted to: electing, ranking, identifying and summarizing testing tasks and artifacts. This involves first understanding what strategies humans apply in such tasks, making *decision-making* theories [31] a natural choice.
- *Tool Interaction Usability:* Testers may face interface and other usability challenges in setup and operation. This contributes to the unpleasantness of testing [5], [23]. Some examples include: availability of a tool's self installation and configuration; visibility of tool operations; clear visual feedback; alternative interface operations; and detailed and up-to-date documentation. Several guidelines and principles are available in the HCI literature to approach such problems, as presented by Stone et al. [32] and Norman [33].

A tool can assist in more than one dimension simultaneously and according to the testing context (Tester and Testing Technique). Distinct testing techniques may require different precedence of cognitive support, within each cognitive dimension. For example, the support for comprehension of test cases differences within a test suite may be priority in mutation testing but secondary in functional testing. Also, the amount of cognitive support needed may be dependent on the tester expertise. One example is the use of external memory support, more frequent among experts than novices [34].

Norman [35] classified cognition in two different types: *experiential* and *reflective*. Experiential cognition is the type of mental effort spent during automatic tasks such as talking, riding a bicycle, cooking the every-day meal etc. Reflective cognition requires much more concentration and is related to decision making, creativity, and reasoning. The framework's cognitive dimensions can be experiential or reflective. We consider *Tester Orientation and Guidance* and *Tester Artifact Comprehension* dimensions to be reflective cognition. In this sense, they are dimensions related to the intermediate steps and results produced during testing tasks; that is, they support conscious and creative activities of the tester. The *Tool Interaction Usability* dimension, on the other hand, is closer to experiential cognition. Thus, it involves auxiliary or aesthetic aspects, giving more importance to how the features contribute to a more practical and comfortable experience.

VI. Conclusions and Future Work

This paper calls attention to how software testing researchers have not sufficiently emphasized the tester's perspective when formulating research problems. By ignoring or minimizing the tester's influence, we may be missing valid and helpful solutions that meet the demands of practitioners. With our background in both HCI and software testing, we are especially confident that the intersection of both research fields is valuable to forming solutions for the presented problems. The former deals with the humanistic aspects, and the latter looks to the theory and current practice of testing activities.

The unit tester's needs, summarized from the literature and described in Section IV, consist of cognitive demands that need to be supported by tools. We condensed these cognitive demands into a framework composed of three cognitive dimensions: Tester Artifact Comprehension, Tester Orientation and Guidance, and Tool Interaction Usability. In order to establish a foundation for the framework, a connection is made between the cognitive dimensions of the framework and the types of cognition as characterized by Norman [35].

This paper makes two main contributions. First, as the starting point for investigating the improvement of unit testing

tools, it moves the primary focus away from automation and looks at the tester's cognitive needs, a change from the current literature. The second contribution is the proposal of a conceptual framework for unit testing, which characterizes newly identified cognitive problems. In this sense, the framework indicates new research directions to be investigated in each cognitive dimension.

This is a work in progress that characterizes newly defined problems in practice. The validation of the proposed framework should involve multiple testing contexts, and should be conducted over multiple research efforts. We are particularly interested in exploring the functional and fault-based techniques as our next steps of investigation, including validation through both prototypes and experiments with users. With further refinement, the framework will indicate with greater precision how unit testing tools should support each cognitive dimension in different testing contexts. As a "wild and provocative" idea, we conclude this paper with a statement that supports our introduction: *testers are hunters because they are human*. The implication is that new opportunities of research are open, and their investigation should help make the bug-hunting game less arduous for the hunters in their practice.

ACKNOWLEDGMENT

This work was partially supported by FAPESP under grant number 201415514-2. We thank CNPq for the financial support and Cassandra Petrachenko for the feedback on our paper.

REFERENCES

[1] K. Beck, *JUnit Pocket Guide*. Beijing ; Sebastopol, Calif: O'Reilly Media, 1 edition ed., Oct. 2004.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York: Cambridge University Press, 1 edition ed., Jan. 2008.

[3] J. Kasurinen, O. Taipale, and K. Smolander, "Software Test Automation in Practice: Empirical Observations," *Adv. Soft. Eng.*, vol. 2010, pp. 4:1–4:13, Jan. 2010.

[4] A. Orso and G. Rothermel, "Software Testing: A Research Travelogue," in *Proceedings of the on Future of Software Engineering*, FOSE 2014, (New York, NY, USA), pp. 117–132, ACM, 2014.

[5] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 201–211, Nov. 2014.

[6] 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. Australian*, pp. 116–125, 2004.

[7] J. Lee, S. Kang, and D. Lee, "Survey on software testing practices," *IET Software*, vol. 6, pp. 275–282, June 2012.

[8] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, Sept. 2011.

[9] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. John Wiley & Sons, Sept. 2011.

[10] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[11] R. S. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach*. New York, NY: McGraw-Hill Science/Engineering/Math, 8th revised edition edition ed., Jan. 2014.

[12] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software Unit Test Coverage and Adequacy," *ACM Comput. Surv.*, vol. 29, pp. 366–427, Dec. 1997.

[13] J. Itkonen, M. Mantyla, and C. Lassenius, "The Role of the Tester's Knowledge in Exploratory Software Testing," *IEEE Transactions on Software Engineering*, vol. 39, pp. 707–724, May 2013.

[14] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Upper Saddle River, NJ u.a.: Addison-Wesley Professional, 1 edition ed., Aug. 2009.

[15] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software, 2nd Edition*. New York: Wiley, 2 edition ed., Apr. 1999.

[16] K. Beck, *Test Driven Development: By Example*. Boston: Addison-Wesley Professional, 1 edition ed., Nov. 2002.

[17] U. Neisser, *Cognitive Psychology*. New York, NY: Appleton-Century 1967, first edition ed., 1967.

[18] D. Norman, "Emotion & Design: Attractive Things Work Better," *interactions*, vol. 9, pp. 36–42, July 2002.

[19] APA - American Psychological Association, "Glossary of Psychological Terms." [Online]. Available: http://www.apa.org/research/action/glossary.aspx. [Accessed: Aug. 14, 2015].

[20] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling Manual and Automated Testing: The AutoTest Experience," in *40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007*, pp. 261a–261a, Jan. 2007.

[21] C. Atkinson, F. Barth, and D. Brenner, "Software Testing Using Test Sheets," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 454–459, Apr. 2010.

[22] V. Lappalainen, J. Itkonen, V. Isomttnen, and S. Kollanus, "ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming," in *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, (New York, NY, USA), pp. 63–67, ACM, 2010.

[23] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, pp. 22–29, July 2006.

[24] V. Garousi and J. Zhi, "A survey of software testing practices in Canada," *Journal of Systems and Software*, vol. 86, pp. 1354–1376, May 2013.

[25] M. Fowler (2012, May 01), "Test pyramid." [Online]. Available: http://martinfowler.com. [Accessed: Jun. 09, 2015].

[26] J. Kasurinen, O. Taipale, and K. Smolander, "Analysis of Problems in Testing Practices," in *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, pp. 309–315, Dec. 2009.

[27] K. Wiklund, D. Sundmark, S. Eldh, and K. Lundvist, "Impediments for Automated Testing An Empirical Analysis of a User Support Discussion Board," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pp. 113–122, Mar. 2014.

[28] M. Delahaye and L. du Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, Jan. 2015.

[29] A. Walenstein, *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002.

[30] F. Detienne, *Software Design - Cognitive Aspect*. London ; New York: Springer, softcover reprint of the original 1st ed. 2002 edition ed., Nov. 2001.

[31] H. A. Simon, G. B. Dantzig, R. Hogarth, C. R. Piott, H. Raiffa, and T. C. Schelling, *Research Briefings 1986*. National Academies Press, Jan. 1986.

[32] D. Stone, C. Jarrett, M. Woodroffe, and S. Minocha, *User Interface Design and Evaluation*. Morgan Kaufmann, Apr. 2005.

[33] D. Norman, *The Design of Everyday Things: Revised and Expanded Edition*. New York, New York: Basic Books, revised ed., Nov. 2013.

[34] S. Davies, "Display-based problem solving strategies in computer programming," *Empirical Studies of Programmers: Sixth Workshop*, pp. 59–76, 1996.

[35] D. A. Norman, *Things That Make Us Smart: Defending Human Attributes In The Age Of The Machine*. Reading, Mass: Basic Books, reprint edition ed., Apr. 1994.