Random Visual GUI Testing: Proof of Concept

Emil Alégroth
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg, Sweden
Emil.Alegroth@Chalmers.se

***Abstract*—Market demands for higher quality software and shorter time-to-market delivery have resulted in a need for new automated software testing techniques. Most automated testing techniques are designed for regression testing that limit their fault finding ability to faults explicitly tested in scenarios/scripts. To overcome this limitation, companies define test processes with several test techniques, e.g. unit testing and random testing (RT). RT is a technique that can be performed manually or automatically with tools such as Fuzz, DART and Quickcheck. However, these tools operate on lower levels of system abstraction, leaving a gap for a Graphical User Interface (GUI), bitmap level, automated RT technique.**

In this paper we present proof of concept for Random Visual GUI testing (RVGT), a novel automated test technique that combines GUI based testing, Visual GUI Testing, with random testing. Proof of concept for RVGT is evaluated in a three phase study with results that show that RVGT is applicable for both functional and non-functional/quality requirement conformance testing. Furthermore, results from a survey performed in industry indicate that there is industrial need for the technique. These pivotal results show that further research into RVGT is warranted.

***Keywords*-Visual GUI testing; Random testing; Proof of Concept**

I. Introduction

The demands for higher software quality and faster time to market delivery are continuously increasing in software industry. These demands force software development companies to focus a larger percentage of time on development, leaving less time for software quality assurance. Quality assurance that can constitute more than 40 percent of the development cost of a system [1]. To aid companies to reach higher quality and lower development time a plethora of automated testing techniques have been developed, e.g. unit testing [2], record and replay [3], and Visual GUI Testing [4], [5]. These automated testing techniques are used for regression testing, i.e. to ensure that the system under test (SUT) still conforms to the system requirements after change or maintenance of the system [6].

Most test tools focus on functional requirement conformance testing but there are also tools that can test a system's non-functional/quality requirement (NFR) conformance. However, most of these tools focus on performance, availability and other quantifiable types of NFRs, whilst NFRs like usability and user experience are left without support. This is one reason why automated testing is not considered a replacement to manual testing. Another reason is because most automated testing techniques are only able to find faults, functional or NFR related, which have been specified in the scripted test scenarios. Therefore, a common industrial practice is to complement the automated scenario-based testing with manual test techniques such as random or exploratory testing. Both these techniques are generally performed through interaction with the SUT's GUI but with the important distinction that with exploratory testing the practitioner aims at identifying the cause of a fault, not just that there is a fault, which is the case with random testing [7], [8]. However, in contrast to exploratory testing, random testing can be performed automatically with tools such as Direct Automated Random Testing (DART) [9], Fuzz [10] and Quickcheck [11]. Automated random testing is however generally performed on lower levels of system abstraction. A gap therefore exists for a high-level technique that can perform user emulated random testing through the SUT's GUI.

In our previous work we have evaluated an automated test technique, referred to as Visual GUI Testing (VGT) [4], [5]. VGT uses image recognition to interact with a SUT through its GUI bitmap layer, i.e. what is shown to the human user on the computer monitor. The image recognition is what differentiates VGT from previous GUI based test techniques, e.g. coordinate- or component/widget-based record and replay [3]. These techniques require access to the SUT's components, underlying APIs or source code, which limits their applicability dependent on SUT implementation. In contrast, VGT is more flexible since the image recognition makes it non-intrusive and independent of SUT implementation, operating system or even platform, e.g. desktop, mobile or cloud. Furthermore, the image recognition allows the technique to emulate human user behavior since all interactions with the SUT are performed through, and with, the same interfaces a human uses, i.e. GUI bitmaps and the operating system's mouse and keyboard operations. Therefore, VGT also has the potential to perform automated GUI bitmap based random testing, which in the continuation of this paper will be referred to as Random Visual GUI Testing (RVGT). However, to the author's best knowledge, no research has been conducted on RVGT, either for conformance testing of functional or non-function/quality requirements.

In this paper we aim to bridge this gap in research with results from a three phase study with the goal of providing proof of concept that VGT is applicable for automated random functional requirement and NFR conformance testing. Furthermore the study will present a survey performed at the

Swedish safety-critical software development company, Saab AB, which shows that there is a need for further research into RVGT and that it perceivably can have positive impact on industrial testing. Thus, the research questions that this study aims to answer are,

1) RQ1: Can random testing be combined with Visual GUI Testing to perform automated, GUI bitmap based, random testing?
2) RQ2: Can random Visual GUI Testing be used to verify system conformance to non-functional/quality requirements?
3) RQ3: Is there a need for/interest in random Visual GUI Testing in industrial practice?

The continuation of this paper is structured as follows. Section II will present related work regarding random testing, automated random testing and VGT, followed by the research methodology used in this study in Section III. In Section IV the results of the study will be presented. Finally the paper vill be concluded in Section V.

II. Related Work

Random testing (RT) is a technique that is commonly used to complement automated testing in industry. The technique is performed through random generation of, and/or random execution, of test cases with the overall aim to cover the input space of a system. Thus, providing test coverage of both common and uncommon cases that appear during system usage and which may be faulty. Furthermore, the technique can be performed manually, e.g. through random interaction with the system under test (SUT) to force it into a faulty state, or automatically by using tools, e.g. Direct Automated Random Testing (DART) [9], Fuzz [10] and Quickcheck [11]. In addition, even though RT is based on random execution of, often mutually exclusive, SUT interactions, studies have shown that RT has equal or even higher fault finding ability than structured test techniques because of higher input coverage [11], [12].

Furthermore, RT allows the user to quantify the significance that a test will not fail and formulate statements like "it is certain that program P will not fail more than once in 10.000 calculations". The conventional theory behind such a statement comes from the equation,

$$\frac{1}{\phi} = \frac{1}{1-(1-e)^{1/N}}$$

where ϕ is the failure rate, $1/\phi$ is the Mean Time To Failure (MTTF), e is the probability that one failure will be observed and N the number of test runs [8]. Hence, the number of tests required to acquire a confidence of 1-e for a given MTTF is,

$$\frac{log(1-e)}{log(1-\phi)}$$

This value can for instance be used as a quality metric of the tested software. However, as with all techniques, RT has drawbacks including, but not limited to, that it can be difficult to define the input space, observe/evaluate the RT output, develop proper oracles to support RT, etc.

Another technique similar to RT that is often used in industry is exploratory testing (ET), defined as *simultaneous learning, test design, and test execution* [13]. ET is based on random input to identify faults but instead of using mutually exclusive interactions this technique relies on cognitive decision making and the user to use previous test results to narrow in on the cause of a fault. Because of this cognitive element, ET is primarily a manual practice that human testers perform without knowing about it, i.e. if a human finds a fault, he/she naturally tries to find its cause [7]. In summary, ET focuses on depth, i.e. finding the cause of a fault, whilst RT focuses on breath, i.e. finding many faults but necessarily not what causes them.

RT and ET are primarily performed manually on a GUI level because there is a gap in terms of tool-support. However, this gap could potentially be filled by Visual GUI Testing (VGT) [4], a novel test technique that is currently emerging in industry. VGT is a tool supported technique that uses scenario-based scripts and image recognition to interact with the system under test (SUT) through its GUI on a bitmap level, i.e. interaction against what is actually shown to the user on the computer monitor. The image recognition allows VGT to emulate human user behavior and could perceivably therefore be used for automated RT and ET. This hypothesis is supported by previous work into VGT that has shown the technique's industrial applicability for scenario-based system and acceptance test automation, resulting in modular scripts that could be randomized on a test suite level of abstraction [4], [5]. However, to the author's best knowledge, there are no studies that use image recognition to perform automated GUI based RT.

As stated, VGT is a novel automation technique, and as with any new automated test technique it requires verification. One approach that is common to verify test techniques, oracles, etc., is mutation testing. Mutantion testing is conducted through fault injection into the SUT, e.g. by randomly modifying input to, or operations in, the SUT to produce faulty output that the tested technique should be able to capture. Even though these faults are artificially created, studies have shown that these faults are equally difficult, or even more difficult, to identify than industrial grade faults [14]. In this study, mutation testing is used to verify RVGT's ability to find faults related to a system's functional requirements.

III. Methodology

The study presented in this paper consists of three phases. In the first phase, a proof of concept study was performed to evaluate VGT's applicability for random testing of functional requirement conformance. The evaluation was performed on two calculator applications, tested with a RVGT script written in the open source VGT tool, Sikuli [15]. In the second phase, another RVGT script was developed to evaluate VGT's applicability for random testing of non-functional/quality requirement (NFR) conformance. This evaluation was performed with a commercial Swedish bus-travel application to show RVGT's applicability on real-world applications. Both phase

one and two were performed on a computer with an 3.07GHz Intel(R) Core(TM) i7 CPU, 6GB of RAM, two GeForce GTX 460 graphic cards with SLi support and the Windows 7 Professional Operating System. In the third phase of the study a survey was performed in industry, at the company Saab AB, to evaluate the industrial need of RVGT. Thus, this study follows the principles of the circular knowledge transfer model described by Gorschek et al. [16]. The model highlights the importance of research knowledge transfer/development in incremental parts. It also states that laboratory experimentation is a good idea before industrial deployment to catch research problems and thereby save unnecessary costs for industry. Hence, this study aims to be an initial building block for future research into RVGT, such as industrial evaluation of the technique, further technical improvement of random testing algorithms, etc. The continuation of this section will describe each phase of the study in more detail.

A. Phase One: Calculator evaluation

In the first phase of the study a RVGT script was written to test the functional requirement conformance of calculator applications. Thus, this phase was performed with simplistic applications that have limited generalizability. However, this choice of applications was motivated by the exploratory nature of this phase of the study since it was unknown if VGT was at all applicable for GUI based random testing. Therefore, a SUT was required that could receive random input but for which it was possible to construct a dynamic oracle. Hence, an oracle that based on random input values could evaluate an expected output for comparison to the actual output from the SUT. The RVGT script was written using Sikuli [15], an open source VGT tool, which uses Python as a scripting language. Python is an object-oriented programming language that supports all aspect of a conventional programming language such as loops, branching, randomization, etc. However, in Sikuli the Python language has been extended with a set of methods that use the tool's image recognition capabilities. These additional methods allow the user to write scripts that can interact with any bitmap displayed on the computer monitor and through scenarios that exactly emulate human user interaction with the SUT.

The RVGT script was developed using a modularized architecture, consisting of four main parts. The first part of the script contained a set of variables to configure the script's speed, the tool's image recognition sensitivity, etc. Furthermore, this part of the script contained the GUI bitmap components to allow Sikuli's image recognition algorithm to interact with the calculators. The bitmaps were stored in lists which made them simple to change in order to migrate the RVGT script from one calculator to another. The second part of the script contained the script oracle that based on the randomized input numbers, generated in the range of -100.000 to 100.000 and a randomized calculator operation, calculated the expected output. The third part of the script defined the GUI interaction with the SUT, i.e. translated the randomized input numbers into click interactions that Sikuli could perform using the mouse-cursor. Input to the calculators was given as

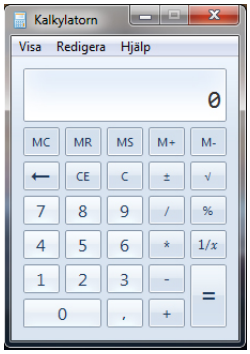

Fig. 1. Windows calculator interface.

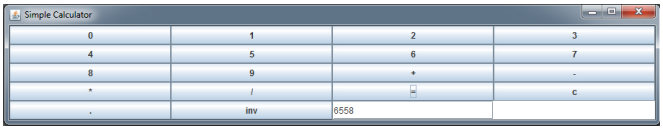

Fig. 2. Simple Java calculator interface

mathematical operations following the pattern,

$$Clearcalculator,$$
$$Input1[0, 100.000], (Optional)Negation,$$
$$Operation[addition, subtraction, multiplication, division],$$
$$Input2[0, 100.000], (Optional)Negation,$$
$$Equals.$$

Hence, if two positive single digit numbers were randomly generated the script would perform five interactions with the SUT whilst two negative six digit random numbers would require 17 interactions. Each mathematical operation was followed by an assertion to verify that the output from the calculator was equal to the expected output. The fourth and final part of the script produced output. Output that consisted of what the randomized numbers were, what mathematical operation that was used, what expected value the oracle had calculated, what the actual value from the calculator was and finally a verdict of the performed assertion.

In order to evaluate the RVGT script it was applied on two calculators, the standard Windows calculator and a custom Java calculator, GUI's shown in Figure 1 and Figure 2. The RVGT script was developed for the Windows calculator and then migrated to the custom Java calculator. To test the RVGT script's capabilities for long-term testing they were executed 10.000 times for each calculator during which the execution time and number of faults were measured. Due to the simplicity of the SUTs, the hypothesis was that there would be no faults in either the Windows or the custom Java calculator. Hence, any fault found by the RVGT script would most likely be a false positive caused either by the image recognition failure or faulty implementation of the RVGT script itself.

Consequently, since the RVGT script, as expected, did not find any faults in the calculators, other than false positives, the

custom Java calculator was modified by introducing mutants into it, for mutantion testing [14]. These mutants caused the calculator to randomly calculate a faulty value in 10 percent of all calculations, also when calculations were performed by a human. In addition, the Java calculator was modified with an additional output module to track which of the calculations were faulty. The RVGT script was then executed against the mutated Java calculator during which 1000 tests were generated. After the execution the result logs from the RVGT script and the calculator were analyzed through visual inspection to determine how many of the randomly infused mutants the RVGT script had been able to kill. The visual inspection was performed by comparing the output logs from the calculator and the RVGT script.

It must however be stressed that a calculator is a quite simplistic system in comparison with most industrial grade systems. However, it is not completely irrelevant to showcase proof of concept since many industrial systems build on basic GUIs with only a few buttons and minimalistic output to the user even though the functionality of the SUT's backend might be quite extensive and/or complex. An example of such a system is the airport management system presented in [4], which had a shallow simplistic GUI, but which controlled safety-critical functionality.

In summary, phase one aimed to provide the proof of concept support for RVGT's applicability to test functional requirement conformance. This was achieved through development of a RVGT script to test calculator applications.

B. Phase Two: Commercial web-application

In phase two, the evaluation of RVGT was expanded also to verification of non-functional/quality requirement (NFR) conformance. Furthermore, to gain a broader view of RVGT's applicability, the evaluation in this phase was performed with a commercial web-service, which in the continuation of this paper will be referred to as the travel planner. The travel planner, screenshot of its GUI shown in Figure 3, is a web-application that allows the user to schedule bus-travel in parts of Sweden. Bus-travel is calculated based on user input that should consist of a start location, an end location and timing information. These input can be given by the user either as text strings or by clicking on a map. Output from the application is presented in a list of different bus travel alternatives, including departure times, if the bus fare will be late, etc. One feature of the travel planner, which is relevant for the evaluation, is that it automatically suggests a list of alternative locations based on the first three letters that the user inputs to the application.

The choice to perform the evaluation on this application was motivated by its representativeness for commercial web-applications and thus of importance/interest for the knowledge transfer to industry, as expressed by Gorschek et al. [16]. However, since access to the actual non-functional requirements of the application could not be acquired during the study, they had to be reverse-engineered based on industrial best practice. Ten NFRs were created for the evaluation, presented in Table I, based on their representativeness for actual NFRs used in industry. Representativeness was evaluated through comparison to industrial NFRs, i.e. best practice, which were available during the study. Furthermore, the NFRs were chosen to cover different types of NFRs, including both qualitative and quantitative requirement types, i.e. Performance, Reliability, Usability, Availability and Portability.

Fig. 3. The traveler planner text input GUI.

Nr	NFR Description	Type
1	The "travel planner" shall provide the user with alternative bus stops within 1 second after the first three letters of a bus stop, or location, has been given as input.	Performance
2	No trips before the current, or user selected, time shall be presented as available.	Reliability
3	After a user clicks the "search-trip" button it shall take maximum 5 seconds before a result is presented.	Performance
4	Late fares shall be presented in a clear way that they are late.	Usability
5	When a user clicks the update button, for a previous search, he/she shall receive new data from the server regarding the trip.	Availability
6	The service shall be available 90 percent of the time.	Availability
7	It shall be possible for a user to provide traveling input in different ways, as textual input or by clicking on a map.	Usability
8	The "travel planner" application shall work in several different browsers.	Portability
9	It shall be possible for a blind user to tab through the entire interface without getting stuck in a "sink-hole".	Usability
10	The "travel planner" shall not accept negative time input.	Usability

TABLE I
THE NFRS THAT WERE TESTED BY THE NFR RVGT SCRIPT.

Once the NFRs had been defined, a RVGT script was developed to test them. Similar to the script in phase one, the travel planner RVGT script was based on a modular architecture with one part for setting up properties of the script, one part that contained the actual tests, one part for output, etc. The largest difference between the scripts developed in phase one and phase two was that randomization in phase two was performed on two different levels of abstraction, i.e. on both an input level but also in what order the tests were executed. Hence, during script execution against the

travel planner, which was performed with 1000 tests, each of the test cases were chosen randomly with a 1/10 probability. Furthermore, all textual and numeric input to the application was randomized. For the textual input the randomization was achieved by randomly inputting three letters to the application and then choosing the top alternative that was suggested by the application. A similar scheme was used for numeric input, e.g. time.

In summary, phase two aimed to identify support for RVGT's applicability for automated random testing of NFR conformance, evaluated on a commercial web-application using constructed, yet realistic, NFRs. However, it should once again be stressed that due to the small number of NFR tests the evaluation only has limited representativeness for industrial NFR testing.

C. Phase Three: Industrial survey

In the third phase of the study, a questionnaire survey was performed at the company Saab AB to evaluate if there is a need for RVGT in practice. The questionnaire was anonymous, voluntary and was handed out to 13 industrial practitioners that were chosen through convenient sampling. Ten people answered the survey, leaving a response rate of 77 percent. The industrial need for RVGT was evaluated with the question, "What effect would the introduction of RVGT perceivably have at Saab?", which was presented as a forced choice question with the three alternative answers,

1) Make current testing worse
2) Not change anything
3) Improve current testing

Included in the questionnaire was a one page summary of RVGT and the proof of concept evaluations performed in phase one and phase two.

IV. Results

In phase one of the evaluation, a RVGT script was developed to randomly test the functionality of two calculators. Results of the first and second part of phase one found no faults in either the Windows calculator or the custom Java calculator that was developed for the evaluation. However, the script still reported that 0.59 percent (less than one percent) of the tests failed with false positive test results. The false positives occurred either because of a mismatch between how the calculator and the oracle rounded fractions, i.e. the precision of the fractions were different which caused the result comparison to fail, or due to image recognition failure. The image recognition failure occurred when an input number with two subsequent nines was generated, e.g. 199 or 991, since the mouse curser changed the GUI state such that the image recognition algorithm could not find the second nine. This problem only appeared for nines and could easily have been solved by replacing the image for the button in the script. However, both these issues show the difficulties of constructing a perfectly functioning oracle, even for simplistic applications.

In the third part of phase one, mutants were introduced into the Java calculator, causing it to produce faulty output 10 percent of the time. Analysis of the output from the calculator showed that 112 mutants were introduced, for the 1000 generated tests, of which 100 percent could be identified and killed by the RVGT script. However, the script reported 114 failures, i.e. 2 false positives. Analysis of these two failures showed that they were caused by the same oracle problem as in the first two parts of phase one. In summary, phase one provided proof of concept of RVGT's applicability to test functional requirement conformance but that oracle creation can be difficult, which caused the RVGT script to have a trustworthiness of 99 percent.

In phase two, 1000 tests were randomly performed with random input to test 10 quality attributes, summarized in Table I, for the travel planner application. Out of the 1000 generated tests, two threw exceptions of which one was a fatal exception that caused that particular test to fail. Hence, 999 tests were completed and one failed due to an exception. Figure 4 visualizes the test results and also shows that the test cases were chosen with an even distribution. Further analysis of the results shows that the tests had overall high success-rate but with test number 4 being an outlier. Figure 4 shows that the travel planner primarily struggled with performance and availability tests. Worth noting is that the 10 test cases, due to the eighth NFR conformance test (See Table I), were performed in two different web-browsers, i.e. every time this test case was randomly chosen the script switched from one web-browser to the other. Analysis of the test case success-rate showed no significant difference between the two browsers since the 114 browser switches were uniformly distributed over the 1000 generated tests. This test also showcases VGT's flexibility and ability to work seamlessly with several GUI based application once.

Analysis of the outlier, test 4, showed that its low success-rate was caused by faulty oracle implementation. If the resulting list of available bus fares from a search did not include any late fare the test failed. Thus, once again showing the difficulties of developing a perfect oracle. Further analysis of the results showed that most failures were caused by high network latency that caused the scripts' assertions to time out when the travel planner application did not respond within the specified time frame. The result that was the most puzzling was that the travel planner in some cases accepted negative time input in test 10. No explanation was found for this behavior since it could not be replicated manually.

In phase three, a survey was performed with 10 industrial practitioners to evaluate if there is an industrial need for RVGT. Results of the survey provide support for a need of the technique since most of the industrial practitioners stated that RVGT would perceivably improve the company's current testing (Median value 3, see question in Section III-C).

Consequently, the results from the evaluations performed in phase one, two and three provide proof of concept that RVGT is applicable for conformance testing of both functional and non-functional/quality requirements. However, this research is only pivotal, showing that the technique is at all applicable and that there is an industrial need for the technique. Future

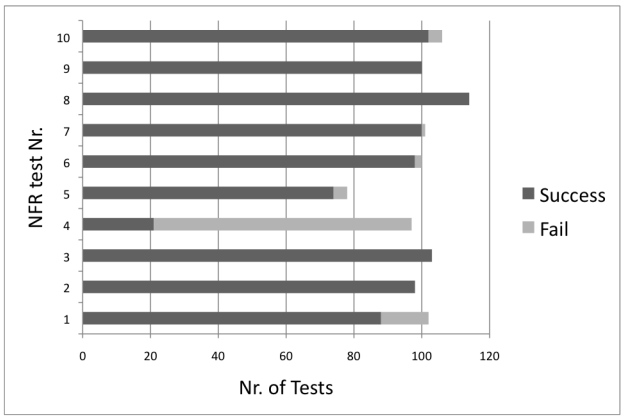

Fig. 4. Summary of the results from experiment 4. Each bar shows the number of successful and failed NFR test cases.

RQ	RQ answer	Summary
1	Yes	The random test script, developed in phase one to test calculators, provides proof of concept for random Visual GUI Testing's (RVGT) applicability to test functional requirements.
2	Yes	The random test script, developed in phase two for the commercial "travel planner" application provides proof of concept for random Visual GUI testing's applicability to test non-functional/quality requirements of different types, i.e. performance, availability, reliability and usability requirements.
3	Yes	The results from the questionnaire survey provides support that there is an industrial need for random Visual GUI Testing.

TABLE II
SUMMARY OF THE COLLECTED RESULTS FROM EACH EXPERIMENT AND THEIR CONNECTION TO THE RESEARCH QUESTIONS.

work therefore includes evaluation of RVGT on more types of software systems, web, desktop, industrial software, etc., as well as other requirement types, e.g. robustness, safety and security.

V. CONCLUSION

Random Visual GUI Testing (RVGT) takes the strengths from Visual GUI Testing (VGT), e.g. flexibility and user emulation, and combines it with the practices of random testing to create an automated GUI based random testing technique. To the author's best knowledge there is no previous work that focuses on the combination of random testing and image recognition-based system under test (SUT) interaction. However, there are other tools available for automated random testing but these tools interact with the SUT on a lower level of system abstraction, which limit their capabilities for testing, for instance, non-functional/quality requirement (NFR) conformance, e.g. SUT usability. These tests are instead performed manually in industry.

In this paper we have presented a three phase evaluation study with the goal of providing initial proof of concept for RVGT, i.e. that the technique is applicable for both functional and NFR conformance testing. Phase one of the study was performed with two calculator applications for which a test script was written in the open-source tool Sikuli [15] that generated and inputted random numbers through GUI interaction and then compared the visual output automatically. To test the fault-finding ability of RVGT, mutation testing was used to introduce faults in a calculator's operations that caused it to produce faulty output 10 percent of the time in a long-term test with 1000 generated RVGT test cases. 112 mutants were generated that could all be identified by the RVGT script. In phase two, a script was written for a commercial web-application to test its NFR conformance. Results showed that RVGT works for test cases executed in random order, with random textual and numeric input, asserted through automated visual inspection. Furthermore, in phase three, a questionnaire survey was performed with 10 industrial practitioners at the company Saab AB that showed a need for the technique.

In summary, this study provides initial proof of concept that RVGT can be applied for GUI based random testing of functional as well as non-functional/quality requirement conformance testing. Furthermore, the study shows that there is a need for the technique in industry which also shows that further research into RVGT is warranted.

REFERENCES

[1] I. Jovanović, "Software testing methods and techniques," *The IPSI BgD Transactions on Internet Research*, p. 30, 2009.
[2] M. Olan, "Unit testing: test early, test often," *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319–328, 2003.
[3] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated gui performance testing," *Software Quality Journal*, pp. 1–39, 2011.
[4] E. Börjesson and R. Feldt, "Automated system testing using visual gui testing tools: A comparative study in industry," *ICST*, 2012.
[5] E. Alegroth, R. Feldt, and H. Olsson, "Transitioning manual system test suites to automated testing: An industrial case study," *ICST*, 2012.
[6] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.
[7] J. Itkonen and K. Rautiainen, "Exploratory testing: a multiple case study," in *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 2005, p. 10.
[8] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 1994.
[9] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.
[10] J. Forrester and B. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium-Volume 4*. USENIX Association, 2000, pp. 6–6.
[11] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Acm sigplan notices*, vol. 35, no. 9. ACM, 2000, pp. 268–279.
[12] W. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 661–674, 1999.
[13] J. Bach, "Exploratory testing explained," *Online: http://www. satisfice. com/articles/et-article. pdf*, 2003.
[14] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?[software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 2005, pp. 402–411.
[15] T. Yeh, T. Chang, and R. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. ACM, 2009, pp. 183–192.
[16] T. Gorschek, C. Wohlin, P. Carre, and S. Larsson, "A model for technology transfer in practice," *Software, IEEE*, vol. 23, no. 6, pp. 88–95, 2006.