

Preface

Software testing is an important phase in the software-development cycle. Well-tested software that is free of bugs is the goal of the software developer and the expectation of the user. Today's world has "gone digital" and offers services to people for their ease and comfort. Many essential services are also being delivered online over the Internet. Thus, our life style is becoming increasingly dependent on gadgets and even more on the software that controls the gadgets. The malfunctioning of services offered online by way of these gadgets is what users least expect. For the purpose, well-tested services are the first requirement before software products appear on the market for consumer use.

Software testing broadly aims to certify not only the accuracy of the logic embedded in code but also adherence to functional requirements. Traditional testing strives for verification of these two aspects. Academia as well as industry have been working on software testing in their own ways and have contributed to the body of research on testing algorithms and practices. As a result, tools exist to automate software-testing efforts so that testing can be accomplished in a faster and less-expensive manner. However, as new computing paradigms and platforms emerge, so does the need to re-examine software testing. Now both academia and industry are searching for solutions to the problems currently faced by software development. Some of these current challenges are addressed in this book.

The challenges software testing faces now require an integrative solution; this means that new solutions must run alongside all traditional solutions. However, well-developed and well-practiced concepts should not be abandoned in search of the new. New issues are arising as changes in technology and varied applications are identified and dealt with zeal by both academia and industry professionals. For that reason, this book includes the views and findings of both academia and industry. The challenges they address include test debt, agile testing, security testing, uncertainty in testing, separation, software-system evolution, and testing as a service. In order to motivate readers, we will brief them on these issues and invite interested readers to explore chapters of their interest.

Software developers usually race against time to deliver software. This mainly happens in response to users who require fast automation of tasks they have been

managing manually. The sooner these tasks are automated, the faster software developers reap financial gains. This has initiated unbelievable growth in the software industry. Software business houses compete with each other to reach out to users with the promise of deliverables as quickly as possible. In the process, developers take short cuts to meet deadlines. Short cuts can be made in each phase of the software-development cycle. Among all of the phases, the testing phase is most prone to short cuts being taken. Because quality software requires exhaustive testing, the process is time consuming. Although taking short cuts ensures that users' delivery expectations are met, they open the door for problems to creep in that appear at later time. These problems later may make things messy and even expensive for a company to fix. This phenomenon is termed "test debt." Test debt occurs due to inadequate test coverage and improper test design. The chapter, *Understanding Test Debt*, by SG Ganesh, Mahesh Muralidharan, and Raghu Kalyan Anna, takes up this issue and elaborates on defining technical debt as a whole and test debt specifically. As the authors say, the causes for technical debt could be many including work-schedule pressure, lack of well-trained personnel, lack of standards and procedures, and many other similar issues that bring in adhocism into software development. Test debt can occur during the requirement-engineering, design, coding, and testing phases. This chapter considers test-debt management from the authors' experience. The concepts presented here include quantifying test debt, paying off debt periodically, and preventing test debt. The best method to avoid test debt is to take appropriate measures, such as adopting best practices in coding and testing, so that test debt is prevented. Finally, the authors illustrate their ideas by quoting two case studies and ending with some suggestions such as the development of tools that can detect smells when debt occurs in software development, particularly during the testing phase.

Early detection of test smell is better because it restricts test debt from increasing. Thus, the idea of testing should start at an early stage of development; however, testing traditionally occurs at the end of coding. The recent idea is to develop in small steps and then test; the cycle is repeated to complete development of the software. Thus, test smell detection happens early in development. This process is known as "agile testing." One chapter, *Agile Testing* by Janakirama Raju Penmetsa, discusses methodology along with presenting case studies to familiarize beginners with this new concept. The hallmark of agile testing is collaboration, transparency, flexibility, and retrospection. In traditional testing methodology, the developer and the tester "chase" each other with negative attitudes: Developers find "devils" in testers, and testers try to pin down developers. This attitude provokes an unwanted working atmosphere. Agile method works on the principle of co-operation and transparency and thus results in a congenial working atmosphere.

The chapter on agile testing presents a brief discussion on the agile process by quoting extreme programming and scrum. A reader new to this concept gets a basic idea of the agile process and then explores the use of this concept in the realm of testing. The author, with the help of an illustration, draws a picture of the agile-testing process, which is based on an iterative and collaborative approach in software development. An application is divided into smaller atomic units with respect to user requirements. Each unit is developed, tested, and analysed by all

concerned including a designer, a code developer, and a tester. In case there is any problem, issues are sorted out then and there instead of carrying test debt forward. Advantages and disadvantages of agile testing are also identified. The success of agile testing greatly depends on a new class of testing tools that automate the process of testing on the basis of the agile-testing process.

Traditionally functional testing has been a prime concern in software testing because this ensures users that requirements will be met. Among nonfunctional issues, responsiveness testing takes the first priority because users usually want a system to perform as quickly as possible. Recently the security of software systems has become the main priority because it is a prime need, particularly when systems become complex as distributed and loosely coupled systems. The integration of third-party software has become prevalent even more so in the case of large systems. This provides a great degree of vulnerability to systems it could many open doors for hackers to sneak through and create havoc in terms of system-output accuracy and performance. This underscores the urgency of security testing to ensure that such doors are closed. Security testing minimizes the risk of security breach and ensures the confidentiality, integrity, and availability of customer transactions. Another chapter, *Security Testing* by Faisal Anwer, Mohd. Nazir, and Khurram Mustafa, takes up the issue of security testing. On discussing the challenges involved, the chapter proceeds to detail the significance of security testing. The authors explore the life cycle of software development and argue that security concerns must be addressed at every single phase because security is not a one-time activity. Security in requirement, design, coding, and testing is to be ensured in order to minimise system vulnerability to security threats. Static security testing involves an automatic check of system vulnerability without the execution of programs, whereas dynamic security testing involves testing of programs while the gadget is running. Static-testing techniques include code review, model checking, and symbolic checking. Techniques such as fuzz testing, concolic testing, and search-based testing techniques come under the category of dynamic testing. The authors present a case study to illustrate the ideas of security testing. They argue that security testing must be embedded in the software-development process. Software security should be checked in a two-layer approach including checking at each phase of the software-development life cycle and again in integrated manner throughout the development life cycle. A discussion of industry practices followed in security testing is contributed, and the authors go forward to explore industry requirements and future trends. The chapter concludes with a positive remark expecting a bright future in software-quality assurance, particularly in the context of security testing, by adopting phase-based security testing instead of demonstration-based conventional testing.

Testing itself is a process consisting of test selection, test classification, test execution, and quality estimation. Each step of testing is dependant on many factors, some of which are probalistic, e.g., the selection of a test case from a large set of candidate test cases. In that case, there could be a compromise in test quality. Thus, uncertainty in software testing is an important issue to be studied. At each phase of software development, it is required to estimate uncertainty and take appropriate measures to prevent uncertainty. Because the greater the uncertainty, the farther

away the actual product could be from the desired one. Because this issue is serious and not has been sufficiently addressed in software engineering, this book includes a chapter—*Uncertainty in Software Testing* by Salman Abdul Moiz—on the subject.

The chapter starts with a formal specification of a test case and introduces some sources of uncertainty. It discusses at length about what contributes to uncertainty. Uncertainty starts from uncertainty in requirement engineering and prioritising user requirements. Uncertainty, if not eliminated early on, affects other activities, such as prioritising test cases and their selections, at a later stage of software development. In the case of systems with inherent asynchrony, the results differ from one test run to another. Uncertainty plays a significant role in assuring the quality testing of such systems. Currently many advanced systems adopt heuristics for problem solving, and some use different machine-learning techniques. These also contribute to uncertainty regarding software output because these techniques depend on variables that could be stochastic in nature.

In order to manage uncertainty, there must be a way to measure it. The chapter reviews some of the important techniques used to model uncertainty. Based on the type of system, an appropriate model to measure uncertainty should be used. The models include Bayesian approach, fuzzy logic, HMM, and Rough sets. Machine-learning techniques are also proposed to determine uncertainty from previous test runs of similar systems. The chapter also provides rules of thumb for making design decisions in the presence of uncertainty.

Measuring uncertainty, ensuring security, adopting agile testing, and many such software-engineering tasks mostly follow a bottom-up approach. That is the reason why modularisation in software engineering is an important task. Modularisation aims at isolation with minimal dependence among modules. Nevertheless, the sharing of memory with the use of pointers creates inconsistency when one module changes value to a location without the knowledge of others who share the location. This book has a chapter, *Separation Logic to Meliorate Software Testing and Validation* by Abhishek Kr. Singh and Raja Natarajan, on this issue.

This chapter explores the difficulties in sharing of memory locations through presenting some examples. It introduces the notion of separation logic. This logic is an extension to Hoare logic. The new logic uses a programming language that has four commands for pointer manipulation. The commands perform the usual heap operations such as lookup, update, allocation, and deallocation. Formal syntax for each command is introduced. Sets of assertions and axioms are defined to facilitate reasoning for changes due to each operation. For this purpose, basic structural rules are defined; then for specific cases, a set of derived rules are also defined. This helps in fast reasoning because these rules are used directly instead of deriving the same from basic rules at the time of reasoning. Before and after each operation, the respective rules are annotated in the program to verify anomalies, if any, created by the operation. The idea is illustrated by a case study of list traversal. The chapter, while introducing separation logic, also proposes a formal language that naturally defines separation in memory usages. The technique proposed here presents a basic method to assure separation among modules while using common pointers. This helps in building systems by modules and reasoning over its correctness.

Top-down system building has been a practice well-accepted by software developers. Agile methodology inspires developers to build a system in modules by taking user concerns as a priority. The system building is iterative, and the system architecture evolves through iterations. At each stage, the system architecture must be rationalized to build confidence, which is required for the subsequent design, testing, and maintenance of complex systems. This is also required for the re-engineering of a system because re-engineering may result in changes to the system architecture. Considering the importance of stability study in evolving systems, this book includes a chapter on the topic, titled *mDSM: A Transformative Approach to Enterprise Software Systems Evolution*, by David Threm, Liguu Yu, SD Sudarsan, and Srini Ramaswamy. mDSM is an extension of the DSM (Design Structure Matrix) approach to software system design and testing. It was developed by the authors to address the design-driven rationalization of such complex software system architectures.

When modelling a complex system, three structural aspects are considered: instantiation, decomposition, and recombination. The first aspect aims to ensure that all design entities belong to the same-type domain. The second aspect considers that a system design can be refined into smaller subdomains. The third aspect assures a system can be reassembled from the constituting components. The authors propose that mDSM-enabling analysts view the software system in its entirety without losing sight of how the modules, units, subsystems, and components interact. Evolutionary-stability metrics is proposed to evaluate a software system's evolution at the lowest level and ensure that the software system can still be rationalized in its totality.

In the case of agile testing, the software is tested at each stage, and changes are made incrementally. At the end of each iteration, stability is computed to rationalize the evolved architecture. For this purpose, the authors introduce evolutionary-stability metrics for software systems and mDSM methodology. Essentially, the concept of normalized-compression distance (NCD) is introduced to study the difference between two versions of a software artifact. Metrics such as version stability, intercomponent version stability, branch stability, structure stability, and aggregate stability are the few ideas introduced in the chapter.

mDSM prescribes five steps to develop component-based architecture: (1) decomposition of a system into components; (2) documentation and understanding of interaction among the components; (3) calculating the evolutionary stability of the components; (4) laying out a mDSM matrix with the components labeled in rows and stability values presented in columns with the corresponding matrix cells; and (5) performing potential reintegration analysis. The concept of mDSM and its impact, particularly in testing, are illustrated by a number of case studies. The chapter ends with listing the advantages of mDSM as well as a concluding remarks on the future direction of research.

Software testing is becoming sophisticated as system complexity increases. This not only requires a large pool of experts but also requires sophisticated technology to perform qualitative testing. Further rapid changes in technology call for the augmentation of testing resources. At this point, many software houses find it difficult to maintain testing resources to meet ever-increasing challenges. At the

same time, testing as a service is emerging. Corporate houses specializing on testing augment their resources to meet emerging testing requirements. This becomes a win-win business proposition among software developers as well as test-service providers. Again, with the advent of new technology, such as cloud computing, this service can be provided with optimal cost because test resources can be shared on the cloud by several service providers. Thus, testing as a service has not only become a feasible solution but also ready to create good returns on investments. Corporates are engaging in making testing happen on a large scale. Considering the impact of this fledgling concept, the book has included a chapter on the topic, *Testing as a Service*, by Pankhuri Mishra and Neeraj Tripathi.

The chapter lists difficulties in traditional testing: It is expensive, inflexible in resource sharing, and demands high-level testing experts. This makes a case for testing as a service by a service provider with expertise in software testing. The authors propose a generic model for such a service and define the roles of test-service providers and consumers. Service providers aim for quality testing, efficiency, and good returns. At the same time, they must exercise caution for testing services. Among the main concerns, security is crucial because service consumers are required to provide input patterns to testers, whereas from a business-interest point of view, these input patterns may contain many business secrets. High-level abstraction of testing-service architecture is proposed. A workflow depicting the chain of actions required to create a service is proposed. A higher-level implementation of this architecture familiarizes with a tool that automates services. The working of the architecture is explained with an example. On making a comment on the pricing of testing service, the authors hint at the usability of cloud technology for providing test services. They base this idea on the suitability of the technology because it helps in the optimal sharing of resources. The chapter ends with a concluding remark emphasizing standardization of this new upcoming service.

This book intends to give a broad picture of trends in software testing. For this purpose, we have included seven chapters addressing different issues that are currently being addressed by testing professionals. We do not claim totality in presenting the trends in software testing, but the chosen topics cover some important issues. Of seven chapters, four are contributed by practitioners from different software houses engaged in testing. The other three chapters from academia add rigour to the exploration of the issues. We hope that this book will be useful to students, researchers, and practitioners with an interest in software testing.

Hrushikesh Mohanty
J.R. Mohanty
Arunkumar Balakrishnan

Trends in Software Testing

Mohanty, H.; Mohanty, J.R.; Balakrishnan, A. (Eds.)

2017, XVII, 176 p. 65 illus., 60 illus. in color., Hardcover

ISBN: 978-981-10-1414-7