
# Modern Software Testing Techniques

A Practical Guide for Developers and Testers

—

István Forgács
Attila Kovács

Apress®

# Modern Software Testing Techniques

## A Practical Guide for Developers and Testers

István Forgács
Attila Kovács

Apress®

*Modern Software Testing Techniques: A Practical Guide for Developers and Testers*

István Forgács
Budapest, Hungary

Attila Kovács
Budapest, Hungary

ISBN-13 (pbk): 978-1-4842-9892-3
ISBN-13 (electronic): 978-1-4842-9893-0
https://doi.org/10.1007/978-1-4842-9893-0



Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Divya Modi
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (https://github.com/Apress). For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Paper in this product is recyclable

# Table of Contents

# About the Authors

**István Forgács**, PhD, was originally a researcher at the Computer and Automation Research Institute of the Hungarian Academy of Sciences. He has had more than 25 scientific articles published in leading international journals and conference proceedings. He is the co-author of the book *Agile Testing Foundations: An ISTQB Foundation Level Agile Tester guide* and the book *Practical Test Design: Selection of Traditional and Automated Test Design Techniques*. His research interests include test design, agile testing, model-based testing, debugging, code comprehension, and static and dynamic analysis. He left his academic life in 1998 to be a founder of Y2KO, the startup company that offered an efficient solution to the Y2K project. He is the founder and Chief Executive Officer of 4Test-Plus and is a former CEO of 4D Soft. He is an author of the Advanced Test Analyst Working Group and former member of the Agile Working Group of ISTQB. István is the creator and key contributor of the only two-phase model-based test automation tool Harmony.

**Attila Kovács** is full professor at Eötvös Loránd University, Budapest, Faculty of Informatics. His research interests are in software engineering, software quality, number theory, and cryptography. He received an MSc in computer science and mathematics and a PhD in informatics. His teaching activity covers a wide range of subjects, including several courses in mathematics and software engineering. He spent several years as a researcher at the University of Paderborn, Germany, and at the Radboud University, Nijmegen, the Netherlands. He received an award for outstanding research from the Hungarian Academy of Sciences (Young Scientist Prize). He is a project leader of several research and development projects and consults for companies on issues including software quality, software testing, safety, and reliability. He is an ISTQB and IREB trainer and a founding member of the Hungarian Testing Board. He is currently the professional head of the "Software and Data-Intensive Services" Competence Center at Eötvös Loránd University.

# About the Technical Reviewer

**Kenneth Fukizi** is a software engineer, architect, and consultant with experience in coding on different platforms internationally. Prior to dedicated software development, he worked as a lecturer and was then head of IT at different organizations. He has domain experience working with technology for companies mainly in the financial sector. When he's not working, he likes reading up on emerging technologies and strives to be an active member of the software community.

Kenneth currently leads a community of African developers through a startup company called AfrikanCoder.

# Acknowledgments

We are grateful to all of those whom we have had the pleasure to work with during this book-writing project: colleagues, friends, the Apress and Springer Nature teams, and especially Kenneth Fukizi.

We are grateful to Bernát Kalló for continuously helping us with good pieces of advice when Harmony's test design features were implemented.

**István**: I would like to thank my wife Margó who helped me a lot and made everything to remain healthy and young.

**Attila**: I want to express my gratitude to my students and colleagues, as well as to my children, Bendegúz and Barnabás, my parents, and my beloved partner.

# Introduction

Oh my gosh, yet another book about software testing…

Indeed, many books have already been written about software testing. Most of them discuss the general framework of testing from different perspectives. Of course, in a test project, the methodological and organizational aspects are essential. However, excellent test organization is only one component of testing effectiveness. You need software that has only a few or no bugs. This can be achieved if the tests can catch all or almost all the faults.

The goal is to create efficient and effective tests. All comprehensive books about software testing that include test design contain the same story: simple examples about use case testing, equivalence partitioning (EP), boundary value analysis (BVA), state transition testing (STT), etc. The present situation of software test design is like classical mechanics: almost nothing has changed since Newton. Current materials about software testing and especially about software test design are either trivial or contradictory or even obsolete. The existing test design techniques are only best practices; the applied techniques are nonvalidated regarding bug-revealing capability and reliability. For example, current EP and BVA techniques give nonreliable results even in the case of a single predicate stating that two or three test cases are enough. The situation with use case testing is not better; that is, several bugs remain undetected. Modern testing needs more efficient test techniques and test models.

In this book, we show a basic classification of the software bugs from the viewpoint of functional specification. We show why traditional test design techniques are ineffective, and we present reliable and efficient solutions for revealing different types of faults. We are convinced that new

techniques in software testing are necessary, and we hope that by applying our methods, software becomes better in the future.

In the last decade, the adoption of new test automation technologies such as codeless test automation has led to more efficient testing that could save time and effort. Virtualization enables optimizing technological processes. Agile testing made testing tasks more customer focused. All the mentioned improvements influence software testing positively. However, those improvements didn't result in significantly better software. Our goal is fundamentally improving software quality. This can be achieved by reliable test design techniques with more efficient fault-revealing capabilities. This is exactly what we are aiming at in our book.

You surely realize that software applications contain numerous bugs. In 1969, we were able to send humans to the Moon and back, but more than 50 years later, we are not able to produce "almost bug-free" software. We know that exhaustive testing is impossible. Is this the real reason for the bugs that remain in the software after testing? Not at all. You are unable to find all the bugs in large software. However, you can find "almost" all of them. You may say: "OK, but the cost will be too high, and I had better leave the tricky bugs in the code." And you may follow the usual path of converting faults into features. Don't do that! Almost all the bugs can be found by applying the test design techniques described in this book.

In this book, we introduce the notion of two-phase model-based testing, action-state testing, and optimized domain testing. These techniques can be used for finding most bugs if the competent tester and the coupling effect hypotheses hold. The first hypothesis states that testers create test cases that are close to being a reliable test set for the correct program. The second hypothesis states that a test set that can detect the presence of single faults in the implementation is also likely to detect the presence of multiple faults. As all the case studies have justified this hypothesis so far, we strongly believe in it.

Our book is a part of our test design knowledge and application methodology that consists of three elements:

1. Learn by reading this book.
2. Exercise our tasks made for you by visiting our website: https://test-design.org/practical-exercises/.
3. Apply our new test design techniques for your tasks by using our algorithms and tools.

In our exercise platform, you are not only able to check your knowledge, but the platform also explains how you might improve it, that is, which tests are missing. We suggest trying to solve some exercises by first applying the traditional and then the new techniques. You will see the difference. By applying the new techniques, the test cases you design will find almost all the bugs.

This book is both for developers and testers. Chapter 1 is a clear and short summary of software testing; our developer and tester readers will find it useful. It contains sections explaining why risk analysis is obligatory, how to classify bugs practically, and how fault-based testing can be used for improving test design. The last part of this book contains a chapter on how developers and testers can help each other and work as an excellent team.

If you are interested in studying traditional software test design more deeply, we suggest reading our previous book *Practical Test Design*. That book contains nontrivial examples, explanations, and techniques, and we think that the previous book is an excellent warm-up to this one. On the other hand, this book can be read independently from the previous one.

We hope we can support your career (as a developer or tester) with this book. Our goal is not just to sell books but to improve the overall quality of software, where satisfied customers happily use the applications offered to them.

# Abbreviations

| | |
|---|---|
| **AST** | action-state testing |
| **BVA** | boundary value analysis |
| **CI/CD** | continuous integration/continuous deployment |
| **CPH** | competent programmer hypothesis |
| **CTH** | competent tester hypothesis |
| **DAG** | directed acyclic graph |
| **DevOps** | development and operations |
| **DP** | defect prevention |
| **EFSM** | extended finite state machine |
| **EP** | equivalence partitioning |
| **F** | false |
| **FSM** | finite state machine |
| **ISO** | International Organization for Standardization |
| **ISTQB** | International Software Testing Qualifications Board |
| **LEA** | learn, exercise, apply |
| **MBT** | model-based testing |
| **ODT** | optimized domain testing |
| **PSP** | personal software process |
| **SDLC** | software development life cycle |

| | |
|---|---|
| **SQL** | Structured Query Language |
| **STT** | state transition testing |
| **T** | true |
| **TSP** | team software process |
| **QML** | Qt Modeling Language |

CHAPTER 1

# Software Testing Basics

This chapter overviews the basics of software testing from the point of view of bugs: lifetime, classifications, pursuing processes, and various pesticides against bugs.

**Estimated Time**

- Beginners: 100 minutes.
- Intermediates: 80 minutes.
- Experts: We suggest reading sections "Pesticided Against Bugs", "Classification of bugs", "Fault-based testing", and "Testing principles"; the rest may be skipped. It takes 30 minutes.

## Bugs and Other Software Quality Destroyers

Bugs and other software quality destroyers refer to issues or factors that can negatively impact the quality, reliability, and performance of software. To mitigate these software quality destroyers, it's crucial to follow best


I. Forgács and A. Kovács, *Modern Software Testing Techniques*,
https://doi.org/10.1007/978-1-4842-9893-0_1

practices in software development, including thorough testing, proper design and architecture, effective documentation, security considerations, performance optimization, user-centric design, and ongoing maintenance and updates. This section overviews the quality from the perspective of the bugs.

## Lifetime of Bugs: From Cradle to Coffin

Software is implemented by developers or generated by models designed by software engineers. Non-considering AI created software, the developer is the one who creates the bugs. Why? There can be many reasons. First, because developers are humans. Successful and scalable software products need professional architecting, designing, and coding. There are many places and reasons where and when bugs can arise. Professional developers need to have programming skills, need to know architecture and design patterns, and need to have some domain knowledge and skills for handling databases, networks, hardware architectures, algorithms, etc. Note that the developers build up the quality of the software. The testers support the developers via quality control.

At present, software is produced mainly manually with the help of some artificial intelligence applications. One of the most important levels for building an application is the programming language level. There are plenty of programming languages. Note that in each situation, there can be many factors determining the "best" programming language. Complex applications require applying more programming languages, frameworks, and libraries together.

Programming language paradigms can be imperative or declarative. In the first case, the developer focuses on how the underlying machine will execute statements. Programs define control flow and usually the way how the program states are changed. Imperative programming

is a programming paradigm that uses statements that change the program's states. "Declarative programming is a style of building the structure and elements of computer programs that expresses the logic of a computation without describing its control flow" (Lloyd 1994). The imperative classification can be broken down into multiple paradigms such as structured, procedural, and object-oriented (however, there are declarative object-oriented languages like Visual Prolog or QML), while declarative programming is an umbrella term of constraint, functional, and logic programming including domain-specific languages. Other paradigms, orthogonal to the imperative or declarative classification, may include concurrent and generative programming. Independently from the chosen language, any programming model describes some kind of logic and data.

*This book does not contain the details of programming* (procedures, objects, classes, type systems, generics, pattern matching, etc.); however, we use some abstract programming elements (pseudocode) and sometimes Python.

An *error* is a mistake, misconception, or misunderstanding during the SDLC. Errors can arise in different places: in the requirement specification, in the architecture, in the code, in the data structure, in the documents, etc. Due to different errors, software bugs are inserted into the code.

"*A software bug is a flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways*" (Wikipedia). Note that we use the terms "bug," "defect," and "fault" interchangeably according to the traditional phrasing in the literature. Bugs affect program functionality and may result in incorrect output, referred to as failure. Later in this chapter, we overview the existing bug classification and describe a new one. Bugs can arise in different software life cycle phases. Bugs can celebrate their dawn as a result of faulty requirements analysis, software design, coding, testing, and even erroneous maintenance. Bugs can slip through the quality gate of unit,

integration, system, or acceptance testing. Bugs can survive the regression testing, and new bugs may arise during any change or correction. Developers and testers share the common goal of reducing the occurrence of bugs and, if any exist, detecting them through the creation of "killer" tests before the software release.

Technically, the very first step to avoiding the rise of bugs is to surmount the chosen programming language. Second, the requirements must be well-defined, unambiguous, complete, and well-understood. Hence, developers (and testers) need to understand the requirements (business, product, process, transition, etc.) and be able to understand the various requirements models. In the following, we overview the possible pesticides against software bugs.

## Pesticides Against Bugs

Unfortunately, the unified theory of testing does not exist and will not exist. A good SDLC includes various activities to minimize the bugs such as defect prevention, detection, and correction.

In the early phases of implementing a system, subsystem, or feature, the business rules and requirements that are incomplete or ambiguous will lead to defects during development. Complex code extended or modified many times without refactoring will also lead to avoidable defects.

Some people think that *defect prevention* is bullshit. "How can we prevent bugs that are already there?" Well, defect prevention is a process that emphasizes the need for early staged quality *feedback for avoiding defects in the later software products*. It stresses the need for quality gates and reviews and encourages learning from the previous defects. In other words, defect prevention is a quality improvement process aiming at identifying common *causes of defects* and *changing the relevant processes to prevent reoccurrence*. The common process of defect prevention is (1) classifying and analyzing the identified defects, (2) determining and

analyzing the root causes, and (3) feedback on the results for process improvement (see Forgács et al. 2019). There are numerous techniques for preventing bugs:

- Apply specification or test models.
- Apply specification by examples.
- Apply test-first methods.
- Manage complexity by divide and conquer.
- Apply the right form of reviews.
- Apply checklists.
- Apply automatic static analyzers.
- Refactor the code.

The first two are related to improving the requirements, the others are related to improving the code, moreover, the third one improves both. The first four are "real" preventions as they happen before coding, the others just before testing. Refactoring is a common prevention technique used during maintenance. Defect prevention is a cheap solution while defect correction is more expensive. That's why defect prevention is valid; moreover, it is obligatory. Clearly, the main target of any quality assurance task is to prevent defects.

*Fault detection* can be made ad hoc and can be semistructured or structured. The most common structured ways are the black-box (specification-based) and white-box (structure-based) methods.

In *black-box testing*, the tester doesn't need to be aware of how the software has been implemented, and in many cases, the software source is not even available. Equivalently, the tester knows only the specification. What matters is whether the functionality follows the specification or not. Black-box testing usually consists of functional tests where the tester enters the input parameters and checks whether the application behaves

correctly and properly handles normal and abnormal events. The most important step for producing test cases in black-box testing is called *test design*.

In contrast, *white-box testing* is performed based on the structure of the test object, or more specifically, the tester knows and understands the code structure of the program. Regarding the code, white-box testing can be done by testers, but it's more often done by the developers on their own. The process of producing white-box testing is called *test creation (test generation)*.

We note that both black-box and white-box testing can be applied at any test level. At the unit level, a set of methods or functions implementing a single functionality should be tested against the specification. At the system or acceptance level, the whole application is tested. Black-box testing can be successfully performed without programming knowledge (however, domain knowledge is an advantage), but white-box testing requires a certain level of technical knowledge and developers' involvement. Writing automated test code for black-box and white-box testing needs programming skills. Nowadays, codeless test automation tools allow any tester to automate tests. There are plenty of tools supporting test automation (both free and proprietary). Test automation is essential for continuous integration and DevOps.

When testing is performed based on the tester's experience, in the ad hoc case, we speak about *error guessing*; in the semistructured case, we speak about *exploratory testing*. A special case of the latter is called *session-based testing* (Bach 2000).

Besides the mentioned software testing types, there are various methods for fault detection, such as graph-based approaches (searching for erroneous control flow dynamic), classifiers (based on machine learning or Bayesian aiming at identifying abnormal events), and data-trace pattern analyzers. But none of these methods have been proven to be efficient in practice yet (however, in some "limited" situations, they can be applied).

In this book, we primarily focus on the first and most important step in the fight against bugs: test design. We consider test design as a defect prevention strategy. Considering the "official" definition, *test design* is the "activity of deriving and specifying test cases from test conditions," where a *test condition* is a "test aspect of a component or system identified as a basis for testing." Going forward, the *test basis* is "the body of knowledge used as the basis for test analysis and design."

Let's make it clearer. Requirements or user stories with acceptance criteria determine what you should test (test objects and test conditions), and from this, you have to figure out the way of testing; that is, design the test cases.

One of the most important questions is the following: what are the requirements and prerequisites of *successful test design*? If you read different blogs, articles, or books, you will find the following:

- The time and budget that are available for testing
- Appropriate knowledge and experience of the people involved
- The target coverage level (measuring the confidence level)
- The way the software development process is organized (for instance, waterfall vs. agile)
- The ratio of the test execution methods (e.g., manual vs. automated), etc.

Do you agree? If you don't have enough time or money, then you will not design the tests. If there is no testing experience, then no design is needed, because "it doesn't matter anyway." Does everyone mean the same thing when they use the terms "coverage" and "confidence level"? If you are agile, you don't need to spend time designing tests anymore. Is it not necessary to design, maintain, and then redesign automated tests?

We rather believe that good test design involves three prerequisites:

1. Complete specification (clear and managed test bases)
2. Risk and complexity analysis
3. Historical data of your previous developments

Some explanation is needed. A complete specification unfortunately doesn't mean error-free specification and during test design, lots of problems can be found and fixed (defect prevention). It only means that we have all the necessary requirements, or in agile development, we have all the epics, themes, and user stories with acceptance criteria.

We have that there is an optimum value to be gained if we consider the testing costs and the defect correcting costs together (see Figure 1-2), and the goal of good test design is to select appropriate testing techniques that will approach this optimum. This can be achieved by complexity and risk analysis and using historical data. Thus, risk analysis is inevitable to define the thoroughness of testing. The more risk the usage of the function/object has, the more thorough the testing that is needed. The same can be said for code complexity. For more risky or complex code, we should first apply more linear test design techniques instead of a single combinatorial one.

Our (we think proper) view on test design is that if you have the appropriate specification (test basis) and reliable risk and complexity analysis, then knowing the historical data, you can optimally perform test design. At the beginning of your project, you have no historical data, and you will probably not reach the optimum. It is no problem, make an initial assessment. For example, if the risk and complexity are low, then use only exploratory testing. If they are a little bit higher, then use exploratory testing and simple specification-based techniques such as equivalence partitioning with boundary value analysis. If the risk is

high, you can use exploratory testing, combinative testing, state-based testing, defect prevention, static analysis, and reviews. We note, however, that regardless of the applied development processes or automation strategies, for given requirements, you should design the same tests. This remains valid even for exploratory testing as you can apply it in arbitrary models.

Have you ever thought about why test design is possible at all? Every tester knows that lots of bugs can be found by applying appropriate test design techniques though the number of test cases is negligible compared to all the possible test cases. The reason is the Coupling Effect hypotheses. This hypothesis states that a test set that can detect the presence of single faults in the implementation is also likely to detect the presence of multiple faults. Thus, we only have to test the application to separate it from the alternative specifications which are very close to the one being implemented (see section "Fault-Based Testing").

## Classification of Bugs

Software faults can be classified into various categories based on their nature and characteristics.

Almost 50 years ago, Howden (1976) published his famous paper "Reliability of the path analysis testing strategy." He showed that "there is no procedure which, given an arbitrary program P and output specification, will produce a nonempty finite test set T, subset of the input domain D, such that if P is correct on T, then P is correct on all of D. The reason behind this result is that the nonexistent procedure is expected to work for all programs, and thus, the familiar noncomputability limitations are encountered." What does it mean? In simpler terms, the sad reality is

that, apart from exhaustive testing, there is no universal method to create a reliable test set that guarantees finding all bugs for all programs. Therefore, it is impossible to definitively state that all bugs have been discovered after testing. However, this does not mean that testing should be neglected as it is still possible to find most of the bugs. Howden introduced a simple software fault classification scheme. According to his classification, three types of faults exist:

- Computation fault: This type of fault relates to errors or faults in calculations or computations performed by the implementation. It encompasses issues such as incorrect arithmetic operations, mathematical errors, or flaws in algorithmic implementations.
- Domain fault: Domain faults involve faults in the control flow or logic of the implementation such as problems with loops, conditionals, or branching, resulting in incorrect control flow, unintended behavior, or faulty decision-making.
- Subcase fault: Subcase faults refer to situations where something is missing or not properly implemented within the software. This can include missing or incomplete functionality, unhandled edge cases, or gaps in the implementation that lead to incorrect or unexpected behavior.

However, this classification is based on the implemented code, but when we design test cases, we do not have any code. Thus, we should start from the functional specification/requirements. The requirements should consist of two main elements:

1. What the system should do.
2. In which conditions the systems should do that.

The system's computation represents what it should do, while the conditions under which the computation occurs fall within the domain of the given computation. Both components are susceptible to implementation errors. Therefore, when considering requirements, we encounter two types of errors:

1. Domain error
2. Computation error

The only distinction concerning Howden's classification is that the subcase error is nonexistent from the specification's perspective since the specification should encompass everything to be implemented. If something is missing, it is not a subcase error but rather a requirement for incompleteness, which can be addressed using defect prevention methods as discussed in the section "Pesticides Against Bugs." A comprehensive specification includes all the conditions the system should fulfill, resulting in test cases for each specific domain. These test cases thoroughly examine potential subcase errors. If a predicate is missing or not implemented, the related test case follows a different path, leading to a faulty computation, which is then manifested as a computation error (excluding coincidental correctness, as discussed in the next paragraph). Therefore, we can consider this situation as a computation error as well.

Therefore, we are left with only these two types of errors, and based on them, we can enhance our test design. **In this book, we introduce one test design technique for computation errors and another for domain errors.** Our technique for detecting domain errors is weak-reliable, meaning that the input value used to identify the error is "one dimension higher" than the one where the bug remains hidden. The reason for this is that even if the code follows an incorrect control flow, the computation may still yield the same result for certain inputs. This phenomenon is known as coincidental correctness. For instance, if the correct path involves the computation y = y * x and the incorrect path has y = y + x,

when both y and x are equal to 2, the result will be 4 for both paths. Our technique for finding the computation errors is not (weak) reliable; however, in practice, it can find most of the bugs.

# Software Testing

## Testing Life Cycle

This subsection is a review. If you are an experienced software tester, you can skip it, except "Test Analysis". If you are a developer, we suggest reading it to get acquainted with the viewpoints and tasks of a tester.

You cannot design your tests if you don't understand the whole test process. We mentioned that the selected test design techniques strongly depend on the results of the risk analysis. Similarly, test creation at the implementation phase is an extension of the test design. Figure 1-1 shows the relevant entities of the traditional testing life cycle including the test design activities.