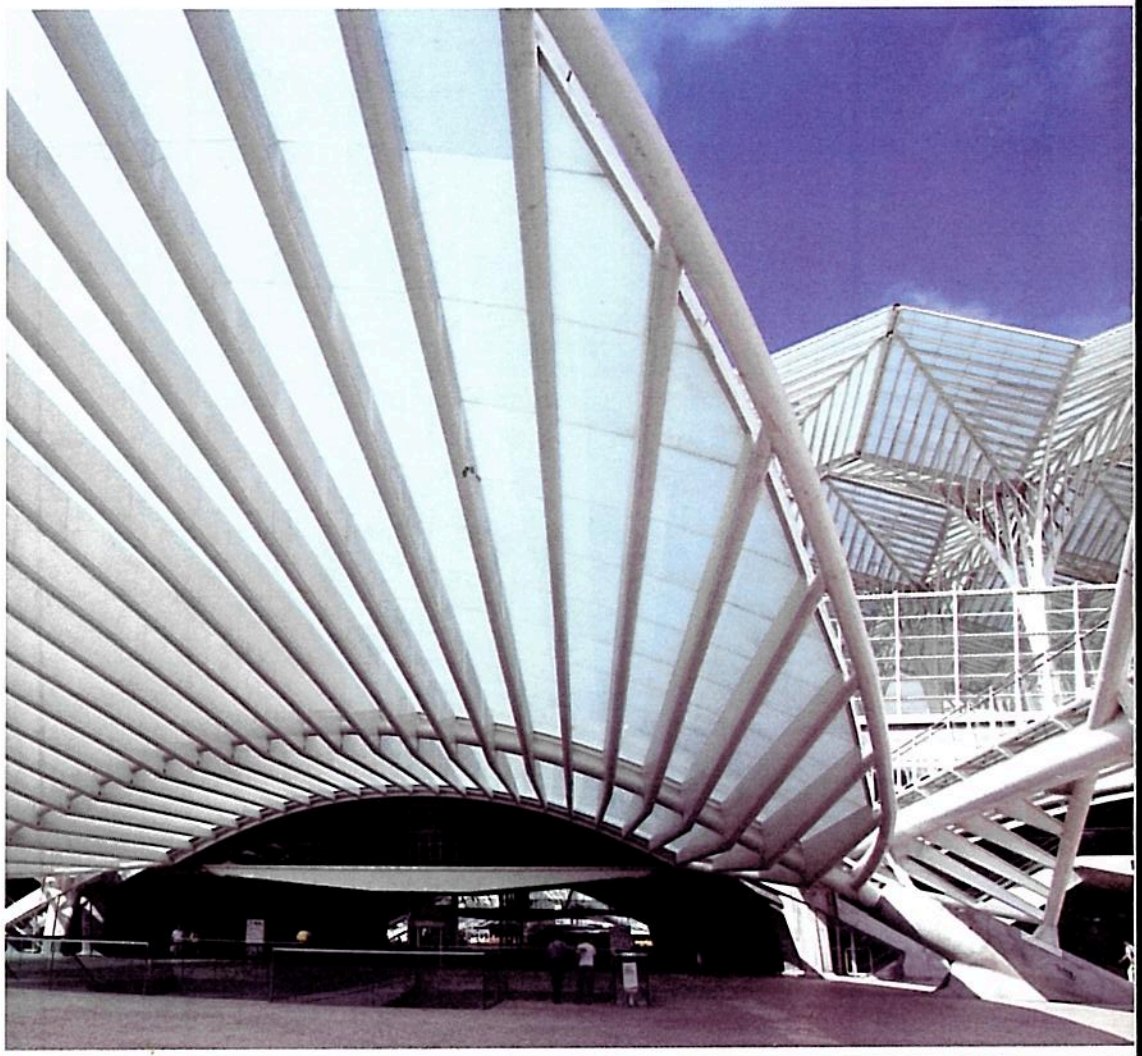
Optimize Quality for Business Outcomes

A Practical Approach to Software Testing

Third Edition



HP SOFTWARE

We had no idea what it takes to put a book together and we've learned a lot from the Oak Hill team, so big thanks to Beverly Hanly, Linda Gallagher, Mary Rosberg, and Amy Guzules for all their help and guidance — we would have been lost without you.

Last and perhaps most important of all - behind every author stands a very supportive family. Sincere thanks go to Petra Harren, Jennifer Sarbiewski, and Claire, Agathe, and Margot Zahm for their love, support, and understanding throughout this project.

Andreas Golze

Mark Sarbiewski

Alain Zahm

Introduction

The word *testing* has many meanings. Teachers test their students to see how much they've learned, small children test the taste and feel of almost anything by putting things in their mouths, and scientists test the strength of materials to see how far they can be stressed before they break. (Some people say teenagers do the same thing to their parents!)

But those kinds of testing are not what we discuss in this book. Software testing is a process where we check a behavior we observe against a specified behavior the business expects.

The trial-and-error behavior that helps us learn and understand the world as children is testing without understanding the expected result.

Both methods of testing have elements in common. In both methods, we want to find out what happens if we're using something in a specific way. The primary difference is that in software testing, the tester should know what behavior to expect as defined by the business requirements. We agree on this defined, expected behavior and any user can observe this behavior.

Today, we see folks in the IT industry still confusing these two concepts of testing. Often, two groups of people are doing software testing:

- The developers or engineers who build the software application want to make sure the final product works as they intended. The testing is done from an IT or technical perspective without relating back to the business.
- The end users who will be using the final product in their jobs, in many cases, do ad hoc testing based on their day-to-day work experience. They make assumptions of how the application should work that come from their individual needs and ideas. Their approach is more trial-and-error than systemized testing — it's often ineffective and creates a lot of rework.

If each group is testing based on their needs and ideas, who's testing for the business outcomes? Who validates that the application will meet business requirements? Who mitigates the risk of failure of a software application in production?

Software testing, like software development, is a profession. Testers need to master proven techniques. Organizations like the British Computer Society and the International Software Testing Qualifications Board (ISTQB) have begun building standardized training and granting testing certification levels.

Like developers, software testers are a special breed with unique talents that can bridge the gap between the business and IT. They must have technical skills coupled with a business mentality. To build a professional test team and do testing with rigor, not by trial and error, takes time and money. For most companies using IT as the backbone for their business, time and money are the two things that are not readily available.

We, along with our colleagues, have consulted with thousands of customers on quality and testing and have seen the same issues time after time. In this book, we present a pragmatic solution to help address some of these challenges.

Our goal is not to write a comprehensive book on testing or advanced testing theory. We're showing you practical solutions, based on implementation experience. If we can help organizations solve some of their most important quality challenges quickly, we have succeeded in what we've set out to achieve here.

In this book, we address fundamental questions around testing from a business perspective.

- In Chapter 1, we examine why testing is so important to a successful product launch.
- In Chapter 2, we discuss how we know what to test and how to determine that testing has met its objectives.
- In Chapters 3 and 4, we present efficient, effective methods of software test design, which is based on equivalence partitioning.
- In Chapter 5, we define practical implementation strategies for optimizing software testing, and we show how to prioritize testing tasks and which parts of testing should be automated.
- In Chapter 6, we describe the non-functional test activities that complement the functional tests and are often underestimated when planning and preparing the test activities.
- In Chapter 7, we focus on security testing, a non-functional test stage, but one that is critical enough to devote a chapter to it.
- In Chapter 8, we focus on the organizational and process requirements for effective outsourcing. With the correct approach, you'll be able to effectively move parts of the described testing activities to off-shore or near-shore facilities.
- In Chapter 9, we give you a SUCCESSFUL approach to measuring and using key performance indicators.

- The appendices contain technical details, tips, tricks, and guidelines for those who want to go further with these concepts and begin to implement them.

Throughout this book, we use the phrase *business function(s)* to define a set of discrete end-user steps that constitute a basic business activity such as "log in," "search for a flight," or "register user." In some application environments the terms, *process steps* or *work steps* may be used instead.

We hope you gain a new business perspective on software testing and find some practical guidance for implementing quality improvements within your organization.

We might say, "Enjoy!", but we know that what you'll really enjoy is the confidence that you know how to move through your software development cycle in the most efficient, cost-effective way — and that you know your customers will be satisfied at the end of it. We will share this win-win process with you.

Chapter 1

What Is the Big Deal About Testing?

"Time waste differs from material waste in that there can be no salvage. The easiest of all wastes and the hardest to correct is the waste of time, because wasted time does not litter the floor like wasted material."
-Henry Ford

This book is about saving time. The work we've done in the IT industry for several years has taught us that professional testing can help a business save a significant amount of money on product development, reduce time to market, and gain satisfied customers.

Finding defects early in the development process makes for a product development cycle that satisfies the business stakeholders and the customers — we can vouch for that.

In manufacturing, for instance, a perfect example of the approach we favor is Toyota's. In a huge departure from the hallowed big three U.S. automakers, Toyota aligned its business requirements with its users.

- They test a cup holder like a user would, not like a designer.

- Any person on the manufacturing floor can stop the entire manufacturing process if they notice something is not right.
- Everybody on the assembly line is a quality checker.
- When a problem is spotted early, it's a lot cheaper to fix at that point, instead of cranking out a bunch of flawed cars and later having to rework the design, or possibly even do a recall.

The HP Quality Model was born out of the question, "How can we save IT costs for testing and quality management?"

To find the answer, we followed the approach made popular by Taiichi Ohno, inventor of the Toyota Quality System, and others. The 5-Why method asks five questions that help lead us down the right path.

First, we asked: **Why is IT so expensive, when hardware costs are dropping and development technologies have improved developer productivity?**

We found that despite the advancements in IT process and technology, IT projects are becoming more complex and they almost always run out of time. When time is short, people tend to sacrifice the quality of the product.

The Standish Report over the past 15 years shows the rate of successful projects (that is, under budget and on time) is below 30 percent (Figure 1). The advancements haven't lowered the cost. On the contrary, with more complexity, the need for proper planning and project management is greater.

When budget and time constraints dictate the project parameters, functionality and quality get the short end of the stick.

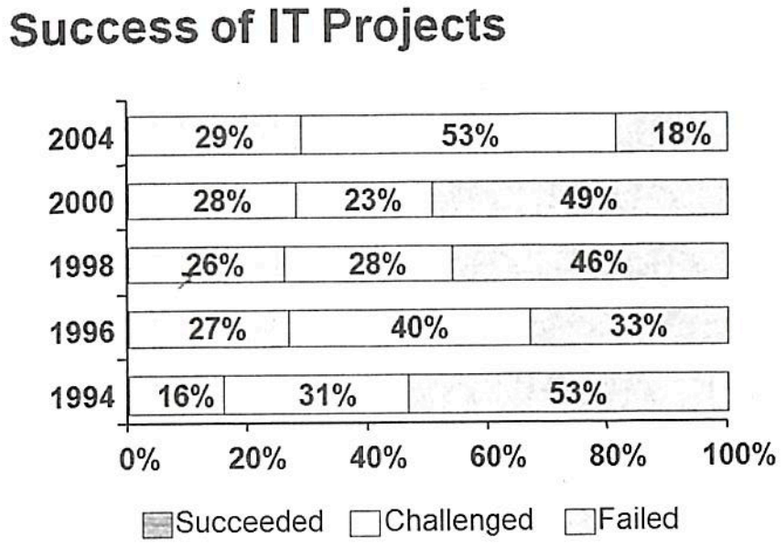


Figure 1: IT project success rate has changed over the past 10 years

To learn why the success rate is so low, we wondered: **Why were the successful projects successful?**

The Standish group also collected data on success factors, asking respondents what contributed to their project's success. Executive support and user involvement were high on the list. Here are the other factors that figured in project success.

- Executive support (18%)
- User involvement (16%)
- Experienced project manager (14%)
- Clear business objectives (12%)
- Minimized scope (10%)
- Standard software infrastructure (8%)
- Firm basic requirements (6%)

- Reliable estimates (5%)
- Other criteria (5%)

While we expected to see executive support and experienced project management, we were interested in other noteworthy elements like user involvement, clear business objectives, and firm basic requirements on the list.

All three of those factors have to do with the end user or the business, which leads us to the next question: **Why is user involvement so important?**

The answer lies in the following statistics (Figure 2), based on a 2002 study conducted by the National Institute of Science and Technology (NIST).

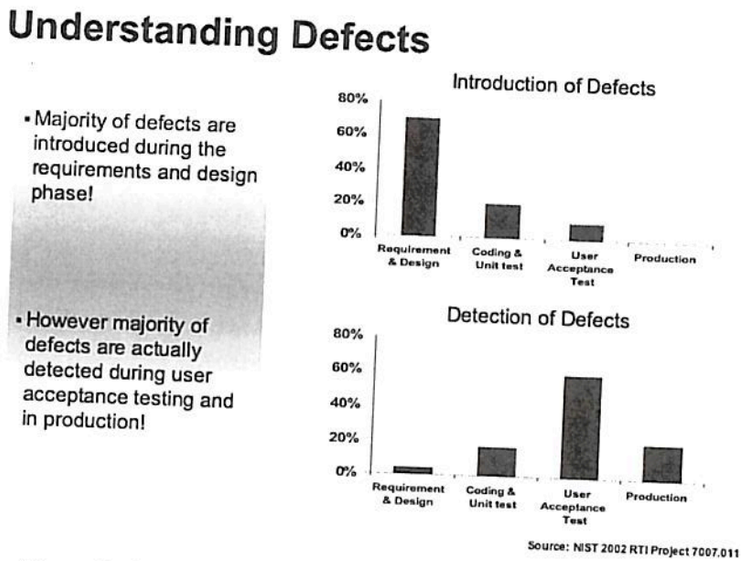


Figure 2: Comparison of where defects are introduced against where they are discovered

Figure 2 shows that most defects are introduced in the requirements phase of the traditional software development lifecycle (SDLC).

Defects in this phase happen mostly because the requirements themselves are incomplete, ambiguous, or contradictory.

When unclear requirements are translated into developers' technical specifications, misalignment of the original ambiguous business requirement and misinterpreted technical specification results in significant rework. (Interestingly, the same people who created the requirements and introduced problems in the requirements phase couldn't see their mistakes until the user acceptance phase, when they could spot them.)

So, we recommend as a best practice for those people creating requirements — double-check initial requirements for ambiguity, contradictions, and incompleteness.

This effect is not limited to software. The following example shows what can happen in situations where people omit necessary details — they have a clear idea of what they want and have made preconceived assumptions, but neglect to communicate the specifics of what they require.

The whole process begins when somebody needs something — in this case a means of transport (the product) — and expresses an expectation (the requirement).

The contractor asks no questions because he has a picture in his mind of what the customer wants. Now, the contractor has made an assumption without checking with the customer. We're clearly missing some communication — a requirement proof point here.

The building process starts and soon, the contractor hands the product over to the customer for acceptance.

BUT I DON'T WANT TO GET WET!
AND HOW AM I GOING TO CARRY MY BRIEFCASE?

During the acceptance phase, the customer rejects the product because important features are missing, and more requirements are added. Instead of starting from scratch, the developer tries to incorporate the new features into the existing construction.

At this point, people rarely go back to their initial assumption (in this case, that the customer needs a bicycle) to find out whether the additional information (the customer doesn't want to get wet and needs to carry a briefcase) means the whole design should be reconsidered.

The reason seems to be that we make decisions based on our own perspective of the situation, rather than use a mechanism that would help obtain a more complete picture. As a result, we attempt to solve the problem without solving the problem.

Despite all efforts to the contrary, we've got an unhappy customer along with a wasted investment into something that doesn't solve the problem.

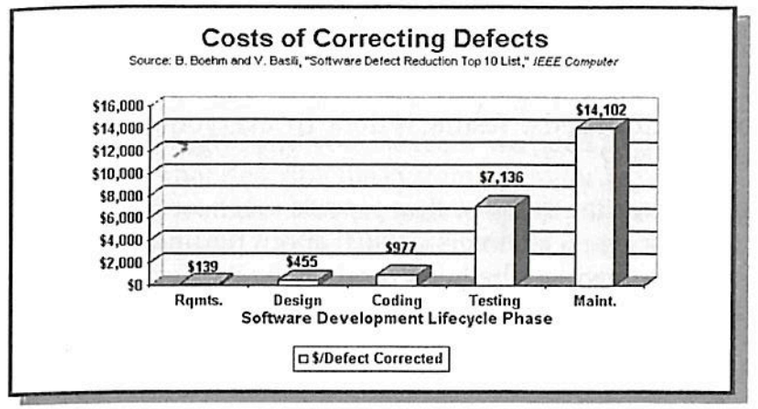
After more iterations, the solution that the contractor comes up with will be more what the customer originally wanted and the problem is solved. But taking the time to sort that out in today's dynamic marketplace means we have compromised:

- Time to market
- Cost efficiency
- Efficiency of the solution

Applying this common example to the concepts of software development, we come to the next question: **Why is the late discovery of defects a major concern, as long as defects are detected and fixed?**

The answer lies in the cost of fixing defects in various stages of the SDLC (Figure 3).

Assessing Cost of Defects



This industry average is used as a baseline for arriving at cost savings

Industry References: 3 B. Boehm and V. Basili, "Software Defect Reduction Top 10 List," IEEE Computer, IEEE Computer Society, Vol. 34, No. 1, January 2001, pp. 135-137.

Figure 3: The cost of correcting defects increases dramatically as we move toward the end of the SDLC

If we are complacent, neglecting to identify defects introduced in the requirements phase until we get to user acceptance testing, we lose approximately $7,000 per defect. This quickly adds up to significant numbers. It also results in the failure of many IT projects when most of the budget for the project is already spent and a huge additional monetary outlay is needed toward the end of the project.

In our bicycle/car example, all the extra work and investment was for something essentially useless to the business. In a business situation, if we consider other indirect costs that crop up, from added time-to-market and increased project risk, the cost can really break a business.

This leads to the final question: **Why can't we make use of this knowledge to improve the testing process?**

To answer that question let's first look at some testing definitions:

> Testing is about finding defects, bugs, or flaws.

This is true for software, as well as for all other products built to suit specific needs. Testing is done by everyone all the time, so why bother?

If we start looking at the testing process more carefully, we find that in most cases, testing is not just about finding defects, but also about knowing whether a product meets business expectations.

> Testing is to determine if a product meets business expectations.

Although this sounds pretty much the same, there is a fundamental difference. A defect is any flaw or imperfection in a software product or process. This implies that a definition of the perfect working product has been made and is clear to all stakeholders. It's not a defect just because the expectations of an individual tester have not been met.

Unfortunately, intuitive testing based on undocumented expectations is a common approach in the industry today. This testing approach is limited because it only works with the completed product and does not allow testing to be done early enough in the SDLC or in parallel with development.

The answer to the last question is that we can't improve the testing process using this knowledge because in today's market, IT often dominates discussions around product expectations and defines the desired functionality around technological aspects rather than business needs. Therefore, the testing process is often misaligned with the needs of the business or ideal business outcomes.

> The HP Quality Model is a pragmatic approach to software testing that aligns IT with business outcomes.

Quality Goals and Test Phases

To build meaningful and concise tests, we need to define requirements that describe the system behavior we anticipate. But that alone would not be enough to specify the intensity or the coverage needed for the tests to meet the end users' quality goals.

Quality goals are non-functional requirements. Typically in IT projects, we use these quality goals:

- **Adaptability** — How easily software can be modified to meet new requirements.
- **Maintainability** — How easily a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.
- **Modularity** — How much of a system or computer program is composed of discrete components and a change to one component has minimal impact on other components.
- **Generality** — The breadth of applicability of the component.
- **Portability** — How easily a system or component can be transferred from one hardware or software environment to another.
- **Reliability** — The ability of a component to perform its required functions under stated conditions for a specified period of time.
- **Correctness** — How free a component is from faults in its specification, design, and implementation. It is also

how well a component meets specified requirements or user needs and expectations. And, it's the ability of a component to produce specified outputs when given specified inputs, and the extent to which they match or satisfy the requirements.

- **Completeness** — How well the component implements all required capabilities.
- **Efficiency** — How well a component performs its designated functions using minimal resources.
- **Understandability** — How clear the meaning of a software component is to the user.
- **Performance** — How well the article under test (AUT) meets the performance needs of the user. Performance may be a measure of throughput, response time, transaction or data volume, or any other related measure.
- **Security** — How well the system as a whole is protected from unauthorized access, disclosure, use disruption, modification, and destruction.
- **Availability** — The degree to which a system or component is operational when required for use.
- **Scalability** — How easily a system or component can handle growing amounts of work in a graceful manner; or the criteria to readily enlarge the capacity of a system vertically or horizontally.

Obviously it is not possible to test against all the different goals in one step. So testing is usually conducted using test phases in a phased approach. Each test phase concentrates on phase-specific quality goals and requires a different level of readiness of the product.

This phased approach is a common practice in engineering and is done to assure the quality of a single unit, before it is verified as an integrated unit to other units. The positive effects are obvious:

- When you discover an error before getting too far down the production line, it pays off in tremendous cost savings.
- It's easy to troubleshoot a product or process, or do root-cause analysis, when each element has been tested. (For instance, if you're building a brick wall, each brick has been tested before being placed in the wall so you know the problem is in the wall structure — you don't have to test the bricks again.)
- You enter every integration step with a known quality of the components (the bricks).
- Testing is easier because the functionality to be tested is limited to the component to be tested.

The test phases build on each other and are defined in such a way that testing can be done as early as possible in the SDLC. Because cost efficiency is crucial, it typically does not make sense to enter a test phase before the previous phase has met its exit criteria. For example, it may not be effective to do Performance testing before Unit testing is complete.

The prerequisite for each test phase is that the preceding test phase was executed successfully to a certain extent. The entry and exit criteria used for each test phase are called quality gates. They are introduced at the beginning and end of each phase and determine the success of the test phases.

Quality goals are requirements that describe the quality characteristics of a software product. Because different people may be involved, and the timing and the nature of the tests to be conducted are different, we divide up the test phases, as you can see in Figure 4.

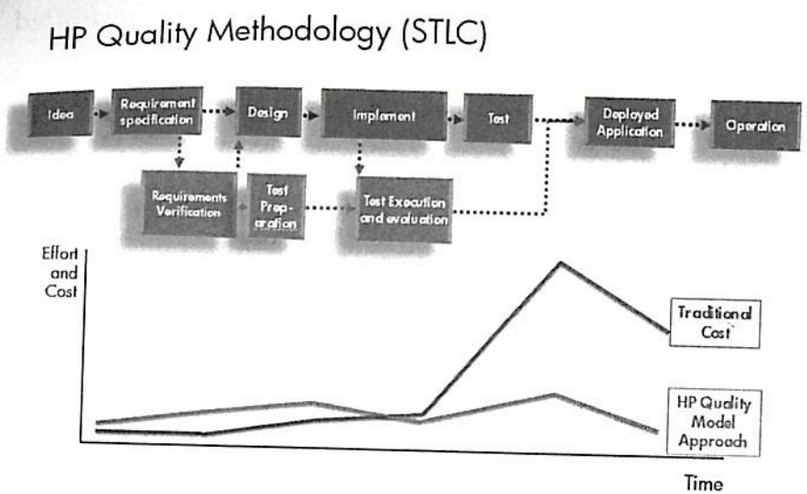


Figure 4: HP Quality Model

In the HP Quality Model, we recommend the following test phases:

Requirements Verification

In the Requirements Verification phase, the end user adds acceptance criteria to each requirement in order to answer the question, "What would make me sign off on this requirement?" The acceptance criteria needs to:

- Validate the completeness and correctness of the requirements.
- Associate the requirements with the business functions.
- Eliminate ambiguities.
- Validate that the result of each requirement is testable.
- Identify the business functions that are changed during this project (maintenance projects only).

Unit Test

The Unit Test is a white-box test conducted by the developers to find out whether the application is stable and completely implemented from a developer's perspective. (We look into white-box and black-box testing later in this chapter.)

Unit testing validates that a particular module of source code is working as the developer intended. A developer conducts the testing, not an end user. A unit has to be executable and the developer uses a debugger or self-developed tools to find out whether the code meets his expectations and the technical specification.

Integration Test

The Integration Test phase has two stages:

- **Business Function Test** — Business Function testing is the first black-box test stage where the functionality of each business function is tested against the requirements.
- **Business Process Test** — Business Process testing checks the business process end-to-end inside the application and focuses on the correct implementation of the interfaces between the business functions.

System Test

The System Test is conducted on a complete, integrated system to evaluate the entire system's compliance with its specified business requirements. System testing falls within the scope of black-box testing, and as such, should require no knowledge of the inner design of the code or logic.

Performance Test

The Performance Test determines the performance and scalability of some aspect of a system — how it performs under a real-world workload. This test may measure throughput, volume limits, transaction time, or any other similar metrics.

User Acceptance Test

The User Acceptance Test (UAT) is done by users or their representatives on a new or changed information system. If the system behaves according to their expectations, the user acceptance testers approve deploying the system. Cosmetic and other small changes may still be required as a result of the test, but the system is considered stable and behaving according to requirements.

Operational Readiness Test

The Operational Readiness Test is the final phase of testing performed on the prospective production system. This test is scheduled and conducted as a part of the production go-live plan. It is to be executed after all other cutover activities are complete, and prior to any go/no-go considerations. The Operational Readiness Test Summary Report is the final business acceptance signoff.

Each test phase requires a particular completeness of the components to be tested. These components have to be executed in a certain sequence, which is synchronized with the progress of the development team. The test phases are repeated for every release of the software and they define the software test lifecycle — a parallel process with the SDLC, but the two are also integrated (Figure 5).

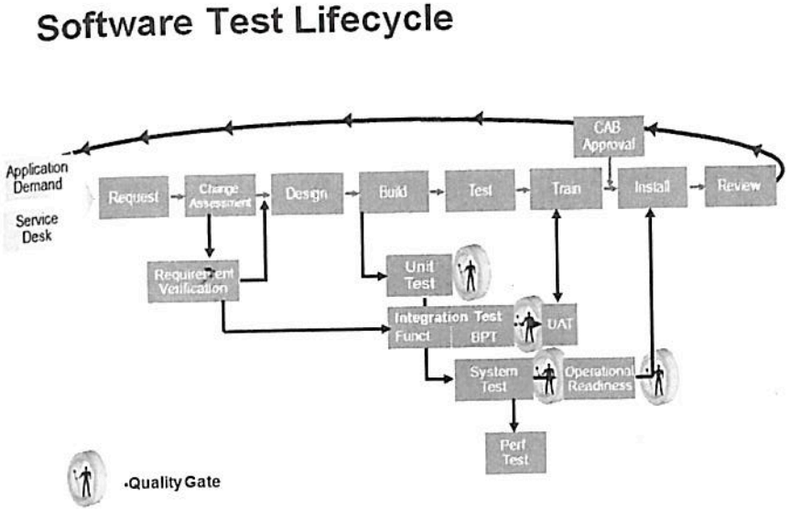


Figure 5: The software test lifecycle is shown running parallel to and integrated with the SDLC

Case Study: Flight Application

We'll illustrate how the phased approach is applied by examining the operation of a web-based portal that allows customers to book airline flights.

The portal is an easy-to-use, straightforward solution that allows customers to search, select, and book flights in the flight database. The system interfaces with the centralized booking system based on the Systems, Applications, and Products in the Data Processing (enterprise resource planning [ERP]) financial module.

Customers have to register before they can use the system. The customer-related information is stored in the centralized customer relationship management (CRM) system, along with users' passwords.

The following functions are needed:

- Log in
- Log out
- Register
- Search flight
- Book flight

Requirements Verification

The prerequisite for the Requirements Verification phase is broken down into logical business functions (Figure 6).

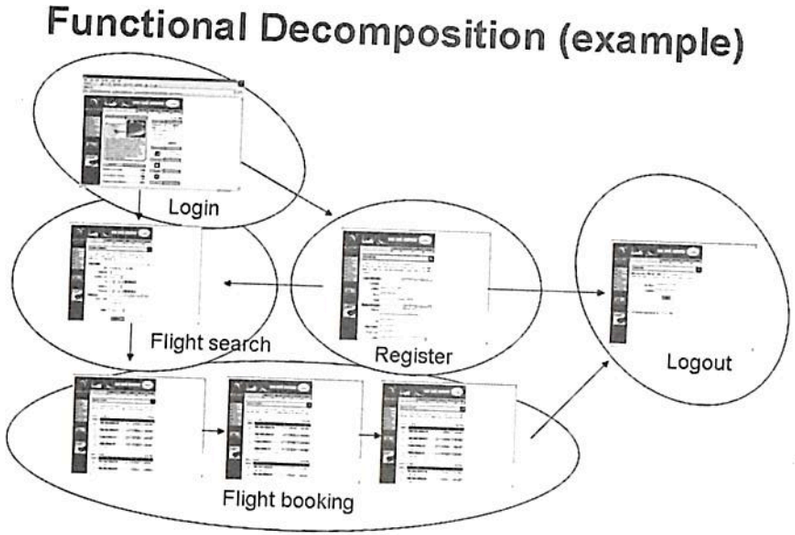


Figure 6: The business process is broken down into business functions

The Requirements Verification phase documents the acceptance criteria for each requirement of each separate business function.

Suppose one requirement for the business function LOGIN is: "The user has to enter a username and a password. By pressing a <LOG IN> button or ENTER on the keyboard, the validation of username and password should start."

The acceptance criteria for that requirement could be:

- If a username without a password is entered, an error message is displayed.
- If the username and the password are a correct combination, the system shows the flight-search screen.
- If the username and the password are not a valid combination, an error message is displayed.

Ideally, the business analyst or business user writing the acceptance criteria identifies missing information and ambiguities in the requirements, documents those as defects, and hands them over to the development team.

Possible quality gates for the Requirements Verification phase are:

- One hundred percent of the requirements have at least one documented acceptance criterion (such as the online flight booking).
- We have divided the application into business functions and prioritized each business function according to the quality needed to meet the business goal of allowing the user to book a flight online.

Unit Test

Unit testing is part of the development lifecycle, so we don't make specific recommendations for this phase in our airline booking example.

However, we need to agree on the quality gates for the Unit Test, because it defines the quality level we can expect when the software is handed over. In other words, if the software does not meet the defined criteria, the test team will not accept it for the Integration Test.

Possible quality gates for the Unit Test are:

- **Statement Coverage** — Has each line of the source code been executed and tested?
- **Condition Coverage** — Has each evaluation point (such as a true/false decision) been executed and tested?
- **Path Coverage** — Has every possible route through a given part of the code been executed and tested?

Integration Test

During the Integration Test we first look at the implementation of each business function. After we validate that the business function is functioning correctly, we can move to the next phase, the various combinations of business functions.

Business Function Test

We're testing these business functions in our example applications:

- Log in
- Log out
- Register
- Flight search
- Flight booking

We test each function independently as soon as it has passed the Unit Test, using various approaches to complete the function test. We suggest using equivalence partitioning for the test case development (we'll explain more in Chapters 4 and 5).

Regardless of the applied methodology, the quality gates need to be defined and met.

Possible quality gates for the Business Function Test are:

- All planned test cases have been executed.
- All incidents have been logged as defects.

- All identified defects have been resolved. ("Resolved" means they're either closed or a decision has been made to delay the closing.)

Business Process Test

When it comes to the Business Process Test, we want to make sure to test the various possible business function sequences. In our flight-booking example, the possible process is shown in Figure 7.

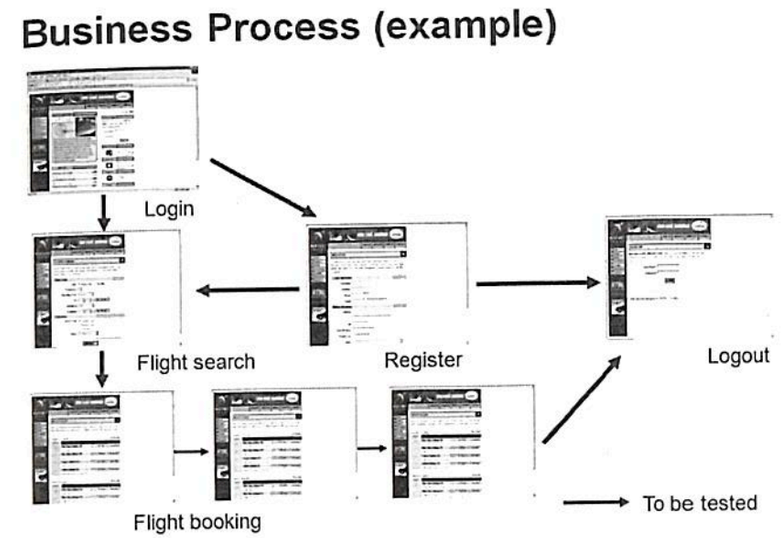


Figure 7: Sample business process for a flights application

It's important to test each possible connection between the business functions once. The focus is the transition from one business function to the next and not the functional correctness of the single business function (addressed during the Business Function Test).

Possible quality gates for the Business Process Test might be:

- All end-to-end processes can be executed.
- A workaround exists for all defects found.

System Test (End to End testing)

During the System Test, each individual application should have passed the quality gates for the Integration Test. The System Test often involves large and complex environments with multiple independent applications working together to support a business process (Figure 8).

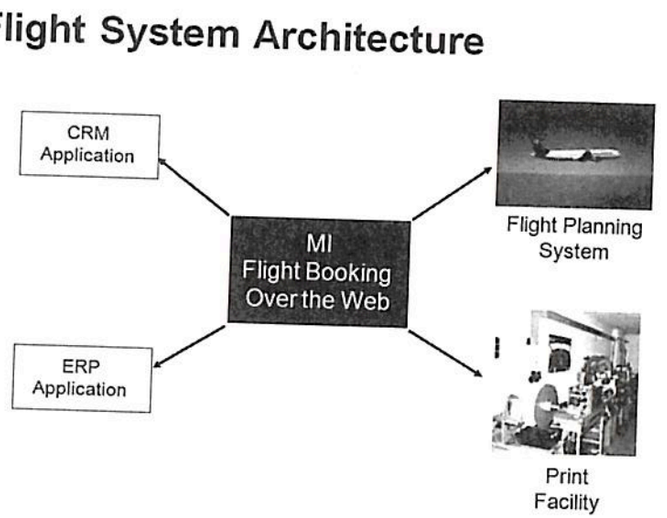


Figure 8: A flight-booking system is supported by multiple independent applications

The System Test makes sure various applications work together as the business process requires — flight planning, ticketing, customer relations. The focus of the System Test is:

- **Functional correctness** — All interfacing applications are in place and the application functions correctly in the defined environment.
- **Reliability** — The system can perform and maintain its functions in routine circumstances, as well as in hostile or unexpected circumstances.

- **Accessibility** — A system is usable by as many people as possible with modification.

Possible quality gates for the System Test are:

- All end-to-end processes can be executed.
- No severe defects exist.

Performance Test

The Performance Test helps us:

- Demonstrate that the system meets performance criteria.
- Compare two systems to find which performs better.
- Measure which parts of the system or workload cause the system to perform badly.

When troubleshooting, software engineers use tools such as profilers to measure which parts of a device or software contribute most to the poor performance or to establish throughput levels (and thresholds) to maintain acceptable response time.

> The later a performance defect is detected, the higher the cost of remediation.

When Performance testing begins at the inception of the development project and extends through to deployment, you have a better chance of catching defects early and thereby saving money and time that would otherwise have to be budgeted to fix problems down the line.

In Chapter 6, we focus on the different aspects of the Performance Test and describe how it can be performed throughout the product lifecycle.

User Acceptance Test

When users are satisfied that the business functional specifications and business requirements have been met, system sign-off and release can take place. The User Acceptance Test (UAT) phase involves turning loose a number of real end users to try the system, using it as they would in real life to accomplish whatever task the system is designed for.

The goal is to make sure that the end users are comfortable with the system they will be working with and that all their requirements are met.

Often a team will leverage training of the end users as a means to perform User Acceptance testing. In our flight example, we might have reservation agents go through the system and make flight reservations and help identify any issues in the flight system.

The quality gate for the UAT is the end-user sign-off.

Operational Readiness Test

The project requirements and design specification documents clearly define the expectations for systems operational readiness. We've been aware of operational readiness levels while we've conducted previous testing phases (Unit, System, Integration, Performance, and UAT).

Operational Readiness testing ensures that the system we are about to deliver has the appropriate production data loaded, is ready to communicate with required external systems, and is ready to run the necessary regular batch processes, while meeting any specific security and compliance requirements.

For the flight application, we might check to make sure that the flight data is being populated correctly, that billing processes that use an external system are working, and that the ticket purchase process is done through secure channels so users are not worried about sharing their private information.

This test phase is the last check of the production system for operational readiness before the system go-live point. This phase should be a part of the Project Implementation Plan.

Key elements of Operational Readiness testing are:

- End-user setup
- Role definitions and requirements
- Security allocations and passwords
- End-to-end connectivity verification
- Production data verification
- Core business functionality verification
- Removable or read-only system interaction
- Application monitoring tool verification
- Call/Service center support setup
- Support documentation

The quality gates for the Operational Readiness Test are normally defined by the operations team or, if the operations task is outsourced, these quality gates can be defined by a combination of the service providers that host the application and the internal operations team.

Test Concepts and Techniques

Many test techniques have been developed over the years for the different test phases, but there are two basic types of test: The white-box test and the black-box test.

White-Box Testing

White-box testing is also known as clear-box testing, glass-box testing, or structural testing. The term *white box* (or glass box) indicates that the tester knows the code used to execute certain functionality.

In software testing, it's usually a programmer that performs white-box tests. Often, multiple programmers will write tests based on certain code to gain various perspectives on possible outcomes.

In the fields of computer programming, software engineering, and software testing, white-box testing is used to check that the outputs of a program, given certain inputs, conform to the structural specifications of the program. In electrical hardware testing, every node in a circuit may be probed and measured (in-circuit test, or ICT).

The most common techniques and approaches used in white-box testing are syntax testing, statement testing, branch/decision testing, data flow testing, branch condition testing, branch condition combination testing, and modified condition decision testing. (For detailed descriptions of these test techniques, see Appendix A.)

Black-Box Testing

In white-box testing we saw that the tester — usually the developer — had insight into how the application was developed. However, when a third-party or impartial tester does not know how the programs were developed, they can conduct black-box testing.

In computer programming, software engineering, and software testing, we use black-box testing (also known as concrete box or functional testing) to check that the outputs of a program, given certain inputs, conform to the functional specification of the program. In electrical hardware testing, we black-box test the specifications of the interface between the device and application circuit.

The term *black box* indicates that the tester does not examine the internal implementation of the program being executed. For this reason, black-box testing is not normally carried out by the programmer. In most engineering firms, one group does design work and a separate group does the testing.

The most common techniques and approaches used in black-box testing are the smoke test, equivalence partitioning, boundary value analysis, and user input validation. (For detailed descriptions of these tests techniques, see Appendix A.)

Wrapping Up

By now, you can see how testing can help your company's bottom line in a new product release, saving you considerable money by catching flaws early in the process and reducing rework for your team.

We've showed you how our approach works in the software field, but you can apply this method in most any industry and it's an excellent way to troubleshoot internal business processes.

> **By setting out clear requirements that align with your business goals, you'll save yourself some headaches and preserve good customer relations.**

Our experiences with other test methodologies have led us to adopt an approach to test design that departs somewhat from traditional testing. It's aligned with business outcomes and is focused on the end-user experience — testing to make sure the customer will have a satisfactory experience, not just that developers see the results they expect.

But before we start testing, we have to better understand what to test.