3 WHAT IS TESTING?

TESTING IS TO ask questions of a test object and compare the answers with some kind of expected result in order to decide whether a particular aspect of the test object works as intended. Some have formulated it a little differently:

> *Testing is the process of using a system or component under given circumstances, of observing or noting the results, and carrying out an evaluation of the system or component from a given perspective.*[7]
>
> *Testing is the process of executing a program with the purpose of detecting defects.*[8]
>
> *Testing is questioning a product in order to evaluate it.*[9]

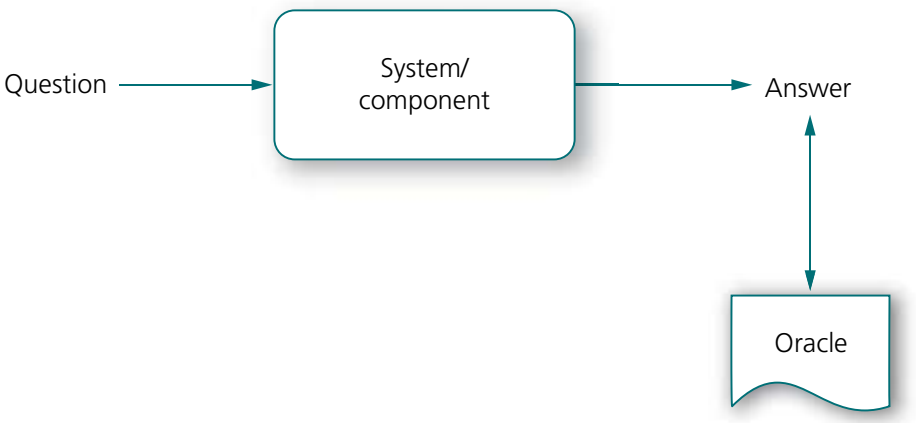

Figure 3.1: Testing is to ask questions and evaluate the answers against an oracle of some kind. All tests have four main problems: how do we ask the questions; what questions do we ask; how do we look at the answers and; what is our Oracle, i.e. where do we find our expected result?

The above picture speaks volumes about the challenges we face when working as testers. We have to know both how to *ask* questions and also to *know which* questions to ask. We must be able to interpret the answers and have something to compare them with. In more detailed tests, and in component testing, we even have to study what happens within the system. Testing can even be regarded as a way of measuring how far we have come in terms of development. Testability is the sum of all factors above.

Aside from the above definitions, there is Static Testing. This involves us researching and checking whether what is in place is correct, without executing any code. The techniques for this are entirely separate from the above descriptions and, although static techniques are counted as testing, according to most testing standards, all the listed definitions exclude this fact, which can be considered a little remarkable.

3.1 Controllability – How Do We Ask the Questions?

In certain cases, we can feed in data ourselves, as users, via a graphical or text-based interface. Here, we mostly use a keyboard and mouse, although there may be other ways of influencing the system, such as remote controls (TV, stereo), breath-input devices (alcohol lock) or push-buttons (traffic signals, lifts).

It is also common to use some type of file interface, whether predefined or generated just for testing. There is often even a programmable interface (API) which makes it possible to take advantage of functions within the program or in input modules, for certain available functions in another program or group of functions.

Practical problems we often come up against are that it is difficult to ask questions before the whole system is complete. Moreover, the interface which the end-user will end up with may turn out to be impractical or impossible to use for us to run the whole range of test scenarios we want to carry out.

If possible, we can require the system to be built with extra functionality for the purpose of carrying out tests – i.e. built-in testability. This means that controllability and observability of the system is increased which enables s to test faster and better.

3.2 Test Design – What Questions Do We Ask?

From the infinite number of questions it is a matter for us to make the right choice. We must choose the questions that are good enough and which, together, provide us with acceptable coverage. The hard part is determining what and how much is good enough! The more questions we manage to ask, the more we know about how well our system functions. The whole of this book is about precisely this problem, which we call test design.

3.3 Observability – How Do We See the Answers?

In the same way as we ask the questions, we can also get the answers via a graphical or text-based interface. It is also common for us to obtain a result file or a paper report and, sometimes, an audible signal or mechanical reaction, for example, from a mobile telephone, lift or microwave oven.

3.4 Oracle – How Do We Know if The Answers are Correct?

One of our problems lies in obtaining an expected result to compare with, and this is often called the Oracle problem. Depending on the type of test we carry out, there are different oracles to suit. Some of the most common are:

- An expected result described as a part of the requirements
- Manually calculated
- Calculations in some type of tool: e.g. MS Excel™
- Earlier versions of the same product: in those cases we perform an update of existing code
- Other similar products that we estimate are correct: e.g. internal or external systems with the same data or functions
- Completed suites of tests which somebody else has built and tested. This is often used for testing compilers, web browsers and SQL-tools[10]

3.4.1 Heuristics

James Bach provides what he calls heuristics for oracles, which are more general answers to the question of how a system should behave.[11] Below are some of the things to consider.

- Consistent with history and our image
- Consistent with similar products
- Consistent within product
- Consistent with user´s expectations

3.4.2 High Volume Automated Testing

When automated tests are carried out there are completely different oracles that apply. Harry Robinson, who is one of the leading experts on automated robustness testing, talks about running the system for a long period with masses of input data, and checking the following:[12]

- Unauthorised defect messages should not appear – e.g. search for the text *Defect*
- The system should not crash
- Features which lock up, where you stick in one place, should not be present

3.5 Example: Loan Application

We are going to test a system which handles applications for loans. From the user's point of view, the system consists of a GUI-dialogue, where particular details about the customer are fed in, for example, salary, while information about the customer's assets and current loans and any remarks are already in the dialogue. So, from a system-testing perspective, we both ask questions, and

get answers, via a dialogue. However, we have another problem, namely that, because time is limited, we must also test the component handling the set of rules, before the dialogue is complete.

- Problem 1: Testing the set of rules before the dialogue is complete, since time is limited
- Problem 2: Partial results – is the test data being used correctly? (It is difficult to test the set of rules in detail without seeing the partial results)

Testing the set of rules component in advance naturally requires an interface for both questions and answers.

1. The test cases are written in a text file which has the same format as the *input area* that the set of rules component has. Thus, we pretend that the information comes from the dialogue.
2. Since we not only want the answer, but also to check that the input variables are being handled correctly, we ask for the partial results to be compiled through what we call a *display* being written to a file during the period that the set of rules is working- these are the temporary results.
3. We generate the test cases in an Excel spreadsheet where the set of rules is simulated. This template contains both the partial and final results.
4. Hand all test cases and the expected results over to the developer, who personally conducts an initial test.
5. In the analysis of our test results, we detect only a few defects. This I contribute to the fact that we have written the test cases early on, and given them to the developer, we have thus contributed to an improved component test.
6. The final step is testing the dialogue connected with the set of rules. Now, we know that the rules are being implemented correctly, and can focus on ensuring that the dialogue is sending the right data to the set of rules. As we had expected, we detect defects in the connection between the dialogue and the set of rules, but no new defects in the set of rules component itself.

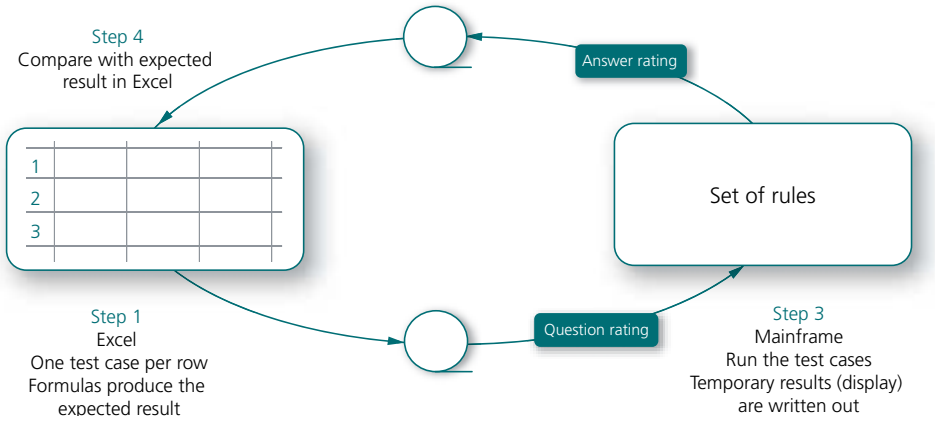

Figure 3.2: Strategy for testing the set of rules component separately. For this, we have to compile an interface for both questions and answers. The answer is compiled in an Excel template

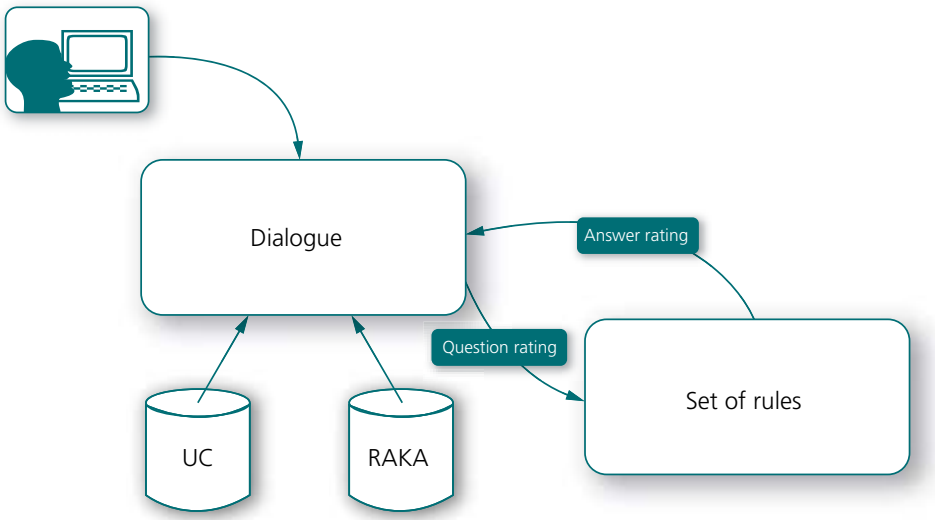

Figure 3.3: Testing the dialogue, connected to the set of rules. We use the dialogue's graphic interface for both questions and answers. The oracle is the same Excel sheet as before. Our focus is now on integration between the parts.

FOOTNOTES

[7] *IEEE Standard* 610.12-1990

[8] Myers, Glenford [1979] *The Art of Software Testing*, p.5

[9] Bach, James [2005]: *Rapid Testing course material.*

[10] Copeland, Lee [2003]: *A Practitioner´s Guide to Software Test Design* pp. 6–7

[11] Bach, James [2005]: *Rapid Testing course material.*

[12] Robinson[2005]: *EuroSTAR Tutorial Model-Based Testing*

5 APPROACHES

5.1 Test Levels

Here is a description of the distribution of tests across levels used in the standard developed by ISTQB.[23] There is no requirement for you to use all levels of testing in all projects you are working on, all the time, but, for practical reasons, it is good to have at least a couple of different levels.

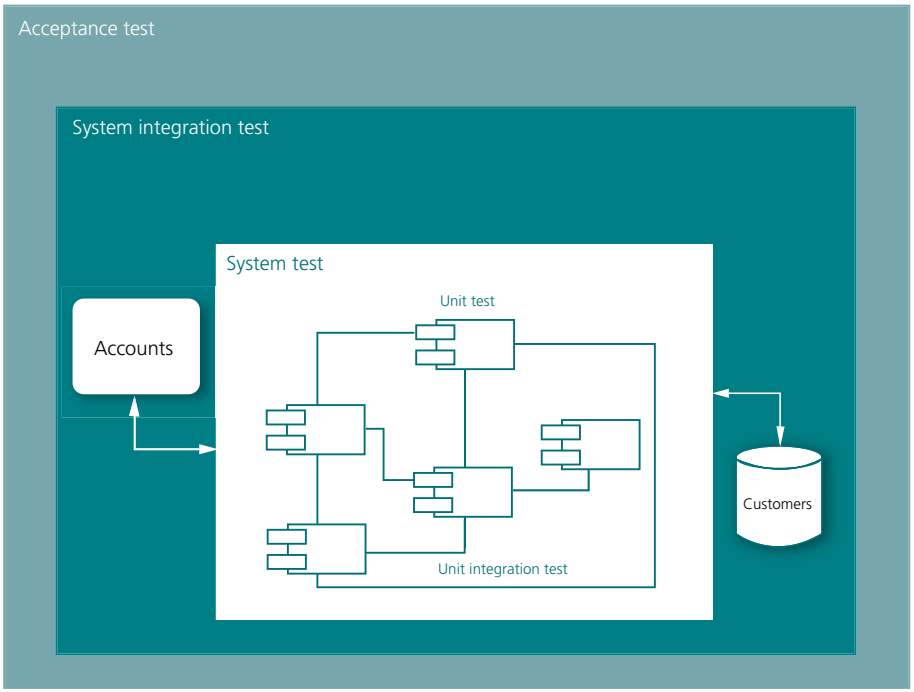

Figure 5.1: Distribution of test levels. Component testing is carried out by the developers, system testing by the development project's testers, and acceptance testing by the person who ordered the system. Every level has a different focus.

5.1.1 Unit Testing

Unit testing, also called module or component testing, is intended for checking the smallest testable parts. For this, aids are used in the form of stubs, drivers[24] or some form of tool.[25] The tests are of a technical nature and require knowledge

about the programming language, so they are commonly carried out as part of the developers' coding work.

As the basis for the testers, program and design specifications are used. Test techniques suitable for this are control flow testing and data flow testing

5.1.2 Unit Integration Testing

Unit integration testing verifies interfaces between components in a system. This level is also of a technical nature, and is about piecing together components in preparation for the system tests. This level is counted amongst the structural tests and its focus is on whether the internal logic works.

5.1.3 System testing

Now, the system is tested as a whole from the users' perspective. Both functional and non-functional aspects are included. Now, we perform *behaviour-based* tests, which involve focusing on external functionality, rather than the software's inner structure.

5.1.4 System Integration Testing

If our system communicates with other systems, it can be a good idea to divide the system tests into two, by adding an extra level. This is, firstly, to check whether the system works in its own right, and then to check all the external connections in place.

5.1.5 Acceptance Testing

Acceptance testing is a means for customers to approve what has been delivered. In simple terms, it can be said that if the system does not solve the problems it was built to solve, we have not been successful. This level is often part of closing a deal between the customer and the supplier and, as such, is very important. Acceptance testing can also be a way for a production organisation to get approval for production.

To emphasise the importance of acceptance testing, here are two examples.
Example 1: Company A orders a system for ticket booking and administration. The assignment is carried out by a consultancy at a fixed price. The price means that the supplier does the least possible amount of work for the sum they have been paid. As soon as the customer has approved what has been delivered, they stop developing the system any further. All changes after approval go on a separate account. In this case, the acceptance tests were non-existent and Company A does not, therefore, discover that the system has severe quality

problems at the point of delivery. These had to be corrected afterwards, and the final bill was double the original estimate. Had Company A carried out a proper acceptance tests, the consultancy firm would have been obliged to carry out all the required corrections in order that the system fulfilled the original requirement specification as part of the original fixed price contract.

Example 2: Company B orders a new administrative system for handling insurance policies, again at a fixed price. The consultancy taking on the assignment quotes a price lower than its competitors and therefore wins the competition stage. Since the estimate is so much lower than the others, Company B actually decides to check that they are getting what they ordered. They deploy a practised test manager, who sets up a thorough acceptance test with support from a group of operations experts. Over the first three deliveries, the acceptance tests show that the system does not succeed in meeting the requirements in the contract, and is therefore returned to the supplier. Only after lengthy delays does the supplier get approval on its system. The work put in on the acceptance test by Company B saved several millions on the final bill.

5.1.6 Test Levels in Production

The last verification before you go to production with a system is to check whether the system works:

- with the volumes of data which will be present
- in the environment in which it is intended to be used
- with any connections to other elements
- together with other systems being run in the same environment

Use a test environment that is as close to its production setting as possible, and preferably the final environment. If possible, you should use the production data under controlled conditions, otherwise try to generate fake data of the same size as the system has to be able to deal with. If you are going to sell the application on the open market, you should include details about what performance requirements it satisfies, and in which environments it works. All the documentation about installation and ownership should be included.

If you use production data in any phase, remember that this type of test places heavy requirements on security and integrity, and must be carried out under controlled conditions, together with the production staff and, perhaps even specialists in load testing. It is generally illegal to use genuine data in test situations, but several companies end up unwittingly in the spotlight each year because they have not only used data in an unauthorised manner, but have also

managed to send the test results out to real customers. For legal reasons, I will refrain from giving any actual examples of this.

Sometimes, you run something called a pilot for a limited number of end users. This is a way of being able to work with real production environments and real data, and of having actual end users carry out the work.

Areas where you often detect defects are:

- response times to end users are too long
- batch processes done daily, monthly or yearly take too much time
- memory or database capacity is insufficient
- parameters for connections outside the system differ between testing and production
- the data communication network cannot cope with the load
- the system is attacked by, or itself attacks, other systems

For those parts relating to load, a tool is often required. In such cases, you often bring in specialists in the field who carry out these tests according to specifications that the test group have compiled. Remember that the person who is to carry out the test has to know which type of user groups there are, how many in each group may work simultaneously and what functions they will be using. This is called compiling user profiles, also known as operational profiles. Not least do they have to know what they are to measure, and preferably, what the response time requirements are! Otherwise, it becomes more a case of stating that under these conditions we have these response times, and will that do?

There are a number of other activities which are called tests by some people. One of them is where an application is run on a *pilot basis*. This means that we work, as in real time, but with a limited number of users who know that they are the first to use the system. The object is to verify actual use in the production environment.

Another variation is what is called beta-testing. If we sell a product commercially, we can choose to distribute an early version in order to obtain end users' views and to have time to deal with potential defects and technical glitches, which only show up during actual use by real users on their site.

5.2 Which Tests Do We Run?

One suggestion for an approach is to start from what a test is worth in relation to its cost, where this value is directly related to the risk. This means that a test case of relatively little benefit to us is only worth running if it is cheap enough.[26]

The same principle is used when we have to decide which requirements are to be included in a release/iteration of a project. This applies regardless of the approach we have chosen to develop the system. This way of thinking is an alternative to a wholly risk-based approach.

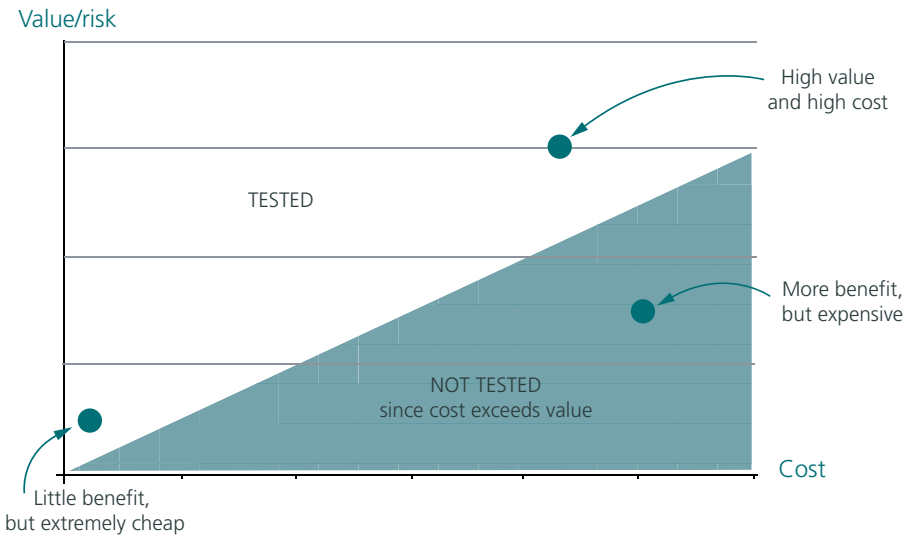

Figure 5.2: Cost is compared to value when we are choosing which tests to carry out. Tests of limited value can be carried out if their cost is very low.

5.3 Scripted Testing

The traditional approach, that stems both from the waterfall model and the V-model, means that a project is linear, and that one activity is completed in full before the next one starts. This means that all requirements are clearly stated before coding begins, and that the testers only begin when the coding is complete. This also means that all test cases must be fully specified, with the expected results, before test execution begins. This traditional method is often called *scripted testing* to distinguish it from *exploratory testing*. The obvious problems with scripted testing, when applied in reality, have led to great debate in the testing world, between those who consider that all test cases, and their expected results, can and must be documented in full before the tests are run, and those who say that we can never know everything we need to know in advance. Out of this was born the concept of exploratory testing, which

involves test design work carrying on in parallel with the test execution. This is described in full in the next chapter.

A number of reasons why you write down all the information in advance are:

1. Legal or regulatory requirements, for example, in the aerospace industry, medical or other safety-critical work
2. Complex relationships which mean that you must prepare test data in detail
3. The person who has to carry out the test case must have the information in detail
4. You make the judgement that it is most effective to write down a great deal in advance
5. You know enough about how the finished system will look to be able to design effective test cases up front
6. You need the test cases for the archive or for regression testing

All operations which handle «life and death» services and products have safety as their most important governing requirement for product and service development. This applies to sectors such as aircraft, the space industry and pharmaceuticals. In the pharmaceutical field, the threat of an inspection by the US Federal Drug Administration is a great one. The risk of having your sales licence withdrawn in the largest market there is (the USA) means that all quality management work has to be documented in order to show what you have done. Many people I have spoken to say that the enormous amount of administrative and documentation work means that an extremely large part of their resources are focused in this area, and so the tests, in reality, are less effective than under a less formal approach. Obviously, well documented tests can also be supplemented with exploratory tests in order to minimise the risks further, which also happens.

5.4 The Context-Driven School

An interesting viewpoint on testing is summarised by something called *The Context-Driven School*.[27] The basic idea is that the testing depends on the context, and there are great similarities with *Agile* development. There are seven fundamental rules:

1. The value of each practice depends on its context.
2. There is good practice in its own context, but there is no best practice which applies at all times.
3. People who work together are the most important part of how every project fits together.

4. The project is developed over time in a way which often cannot be predicted.
5. The product is a solution: if the problem is not solved, the product does not work.
6. Good software testing is a challenging intellectual process.
7. Simply through judgement and skill, put into practice together throughout the whole project, we can do the right thing at the right time in order to test our products effectively.

Consequences of the above:
It pays to produce different results within a test to the extent that they satisfy the relevant requirements of the interested parties. That means that neither documentation nor anything else is compiled for its own sake, if it does not add value.

Different types of test detect different types of defect. If a program appears to be stable under one set of test techniques, it is time to change techniques.

Rather than seeing automation as a way of cutting costs, by means of a machine doing imperfectly what a human does, see it as a way of carrying out tasks which would be difficult to achieve manually.

Supporters of this school make regular use of exploratory testing.

5.5 Striking the Golden Balance

There may be laws and directives requiring formal documentation, as in the medical industry. There may be practical requirements for detailed planning, such as an extremely large number of interfaces and mutually-connected data dispersed over several systems. In such cases, we plan and document carefully in advance.

Other situations mean that we are not delivered what we expected, and changed or new functionality may have crept in. Situations can arise that we really have not prepared ourselves for, so we use exploratory testing. Since defects rarely come alone, there are often several others close to where you found the first one. The work you do in isolating the defect, and finding out whether there are defects close by, is not written into the test case: you must improvise. This can be seen as an example of exploratory testing.

I have had discussions with many experienced testers of critical systems in medical science, who carry out a whole range of carefully documented tests because of legal requirements, They supplement this work with exploratory tests with good results. However well they design the prepared test cases, there are still defects which are first detected in the more free-form part of the testing.

Earlier, we concluded that variation is good, both for test design, and for the question of which people participate. This seems to also apply to the test approach, where we successfully combine thoroughly prepared test cases with a more exploratory approach. So, in reality, we do carry out both scripted and exploratory testing in every project. We find ourselves somewhere on the scale in the diagram below.[28]

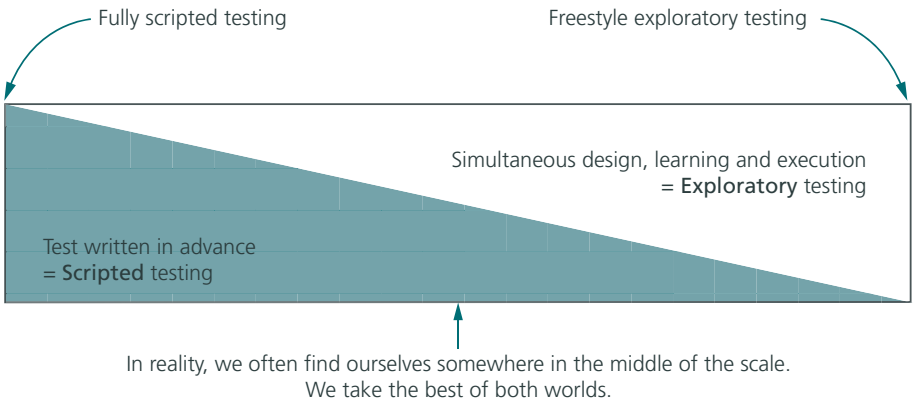

Figure 5.3: Most often, the tests consist of a mixture of scripted and exploratory testing. Some parts are scripted and others are supplemented afterwards.

FOOTNOTES

[23] *ISTQB course plan version 0.2. www.sstb.se*

[24] *Code used to execute other code*

[25] *For example, the XUnit tools, J-Unit for Java, N-Unit for C# and .Net*

[26] Bach, James [2005]: *Rapid Testing course material.*

[27] «*http://www.context-driven-testing.com/*»
«*http://www.context-driven-testing.com/*»www.context-driven-testing.com

[28] Bach, James [2005]: *Rapid Testing course material.*

7 TEST DESIGN TECHNIQUES: AN OVERVIEW

7.1 Static and Dynamic Techniques

We usually differentiate between static and dynamic test design techniques. A static technique involves examining documentation, design or code, while a dynamic one involves studying the program while it is being run. Here is a diagram of a common subdivision.[29]

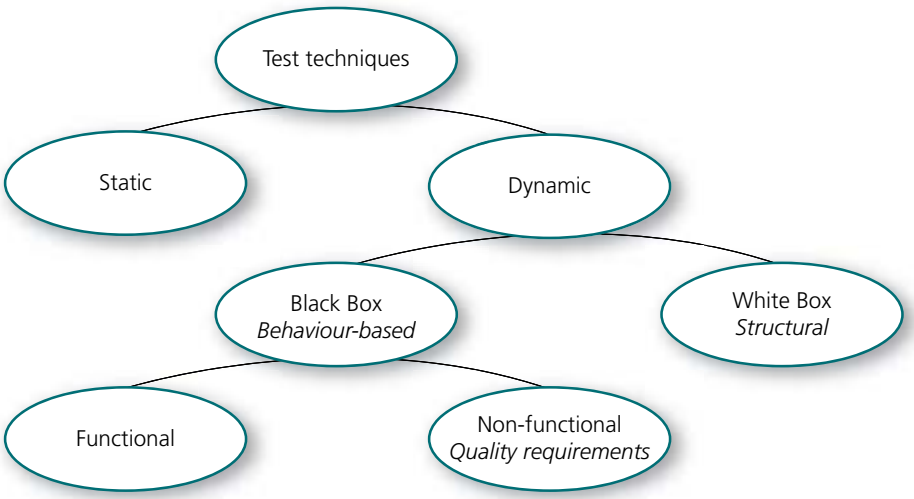

Figure 7.1: Overview of test design techniques. Dynamic techniques may focus on functions or on quality factors. Static techniques, which involve scrutinising code or documentation, are usually also counted as testing, even though no code is executed.

7.2 Static Techniques

A static test design technique involves no program code being executed. Instead different types of documentation in the form of text, models or code are analysed, often by hand. In a number of cases, e.g. in the compilation of program code, tools are used. The defects detected are often related to requirements and design: either there are parts missing or the documents are not consistent within themselves or do not conform with the standard in force. The advantages of reviews are that we intervene early and we supplement the dynamic testing by detecting other types of defects. The formality of the review is usually governed by the criticality of the material being examined. A requirements specification

which is fundamental to all other work is examined more carefully than a program specification for a small part of the system. It is a question of choosing the right type of review in order to derive the greatest possible benefit. Reviews are regarded by most organisations today as valuable and are commonly used in many different sectors. I have worked with different types of reviews within telecommunications, banking, insurance and the public sector. In the majority of cases these reviews are informal, and nowhere outside the academic world have I been involved in an inspection of the kind described below.

7.2.1 Inspection

The most formal review technique is called an *inspection* and is strictly governed. The method was developed by Michael E. Fagan at IBM.[30] The participants prepare themselves carefully by examining selected areas according to role descriptions and check-lists. At the review meeting all points of view are recorded and how they are to be addressed can be discussed. On the other hand, the problems are solved afterwards. Since the way of working is formal, only a limited amount of material is examined on each occasion, mostly no more than a dozen pages, depending on the test basis. For this type of review to be effective requires the participants to be trained, check-lists to be compiled, and the review meeting itself to be chaired by an experienced moderator. For those who wish to learn more, there is a detailed description in the book *Software Inspection*[31].

7.2.2 Walkthrough

One of the simpler and less formal techniques is called walkthrough. It involves the author presenting his or her material to a selected group of participants in order to get them involved in the test basis more quickly as a result. Questions and explanations are welcomed during the meeting. The purpose is more one of creating a common picture, than of identifying defects, but the technique is still counted as static testing.

7.2.3 Technical Review

As the name suggests, a technical review focuses on the technical parts of the project like architecture and program design. Here, primarily the technical experts and the architects take part together with other developers. The purpose is both to evaluate choices of solution, and compliance with standards and other documentation. The result is often documented in a log.

7.2.4 Informal Review

Informal review involves two or more people looking through a document or

code that one or the other of them has written. The purpose is still to detect defects, but there are usually no check-lists used and the result does not need to be documented.[32]

7.2.5 Modelling – an Alternative to Reviews

The method that I prefer for static testing is not listed with the techniques above. It is more a matter of analysing the test basis I have and, from it, creating models of how I believe it should work. By compiling the models and discussing them with people involved, I often detect defects in the form of information that is missing, inconsistent, or plainly wrong. We will cover modelling in detail later in the book.

7.3 Dynamic Test Design Techniques

Dynamic testing involves usmeans testing code by executing it. We usually split up the dynamic techniques into *behaviour-based*, often termed black-box, and *structural*, usually called white- or glass-box. Regardless of whether we work with behaviour-based or structural methods, there are a great many different techniques which can be used in order to create good test cases

7.3.1 Data

For handling test data, we split up the mass of values into *equivalence partitions*. For numerical intervals, most of all, there is also *boundary value* analysis for the groups we have identified. Handling test data in these ways reappears as an important part of most other techniques, and are therefore called *the elementary techniques* in this book.

7.3.2 Flows

There are flows at many different levels, from overarching business processes, to program code, which have both control flows and data flows. The principles of flow testing are similar to each other, whatever the level. Firstly, we draw up the flow graph, then we cover the variations of the flows by using branch or path analysis. This is the same principle as white box testing.

How good the test coverage needs to be, depends on risk-level and how complicated the flow is. The perhaps somewhat unsatisfactory answer is that it just depends. The lowest level of coverage that I make use of is where all branches have to be covered, and this mostly works well when it comes to more overarching flows, such as business processes. More detailed flows, such as use case flows, in the form of activity diagrams, often require you to have more combinations of different flows. There is no general rule for which level you should or must cover in order to have a good set of tests.

7.3.3 Logic: Sets of Rules, Formulae

In order to handle sets of rules and mathematical formulae, *decision tables* are powerful aids for creating test cases. In order to check whether a decision table is correct – complete and not overlapping – for some types of sets of rules, you can draw up a *decision tree*. The graph is also an excellent aid even when you run your tests and evaluate potential defects.

7.3.4 Combinatorial Analysis

A commonly occurring problem is where you have a large number of variables which can be combined with each other in an almost infinite number of variations. Testing all combinations for a range of mutually dependent variables is often not possible. We quickly get what we call a combinatorial explosion, which is common in testing.

For handling this type of problem, there are a number of useful means of simplification, without us missing too much. Elementary comparisons[33] involve all variables included in a condition being capable of determining the result of the condition at least once, and all results of the condition being obtained. Another way of combining variables is to test them with each other in pairs so that *all pairs* are covered.

7.3.5 Experience based testing

There are a great many different techniques which all build on earlier experience of what usually goes wrong. To this group belong the variants called *risk lists*, *defect guessing*, *taxonomies*, *defect classifications and attack patterns*.

Error guessing is described in some works as a technique in its own right.[34] The purpose is to point out areas that we think have defects in them from experience, technical knowledge, old defect reports, etc. An alternative viewpoint is to see *error guessing* as a natural element of all techniques.[35] In this book, it is counted as a risk-based technique.

7.3.6 Advanced Testing

Once the individual functions in a program have been tested, it is a question of checking whether the whole works. We have already dealt with the *testing of business processes* earlier in this chapter under *Flows* ther useful techniques are the testing of *time cycles*, *data cycles*, *soap-opera testing*[36], *life cycles* and *syntax testing*. We can also work with user profiles in order to find out how the actual users will be using our application, and from this, test the relevant combinations. A technique that has worked out exceptionally well for some of my colleagues is exploratory testing using test charters which are basically a test

case description without detailed steps that states the task but gives the tester great freedom to perform. The chapter on exploratory testing explains this concept further and there is an example of a simple test charter in the chapter on testing business processes.

7.4 Black, White and Grey

7.4.1 Black Box Techniques – Behaviour-Based Testing

We often call behaviour-based testing input or output data-driven testing. This is aimed at verifying whether a system behaves correctly from the outside, without us being concerned about what is happening inside – hence the commonly used expression black box testing. As a starting point, you often have a specification in the form of a use case, a work process or other requirement documentation. We ask the system a question and get an answer, and if the result corresponds to what we are expecting, the test case is completed with a satisfactory result. The fact that we are not concerned about what is happening inside the system does not necessarily mean that we do not want to know how it works. That definition, I think, is wrong. Often, from the design of the system, we can obtain valuable information which helps us generate better test cases. One example is where a developer has performed an update on a program, and we want to know which parts of the system may be affected by the change, and which regression tests we should therefore conduct. There is more about this in the section on Grey Box Testing.

7.4.2 White Box Techniques – Structural Testing

White box tests are also called structural- or glass-box tests. The purpose of the tests is to verify the internal logic in the test object. The technique is used at the component testing and program integration testing stages and requires detailed knowledge of how the system is built. The structural tests encompass *control flow* (program flow) and *data flow*. Control flow tests check that the internal program logic is working. Data flow tests verify whether the way a variable is defined, used and destroyed is handled correctly. [37]

The test basis used are program and design specifications where the details about the program's internal structure are clearly described. It is especially important to test all boundary values and error conditions carefully. You can carry out structural tests without any tools, but if you are to measure the degree of coverage this is very difficult without automated aids. Structural tests depend on programming language, and this is the help the test tools offer.

7.4.3 Combined: Grey Box Testing

Often, we make use of a combination of behaviour-based and structural

techniques in order to obtain better test cases. This involves us looking into the box to determine the internal structure, in order to create better behaviour-based tests. Knowledge of programming and databases is therefore a strength when it comes to detecting more defects.[38]

7.5 Non-functional Tests

Besides pure functional requirements, there are also quality requirements. The quality factors requirements category contains so many aspects that it is appropriate to create an additional categorisation in order to make the requirements profile more visible. A good foundation for such a classification is given in the standard ISO 9126 Software Product Evaluation which describes how to structure the main non-functional requirements.

ISO 9126 [1991] states that a product's quality must be determined according to the following six headings. An appendix in the standard contains a model for how these can be interpreted, providing a number of subcategories for each heading:

1 **Functionality:** Are the desired functions present? *suitability, correctness, compatibility, compliance with standards, security*
2 **Reliability:** Is the system robust, and does it work in different situations? *maturity, defect tolerance, restart (after defect), accessibility*
3 **Usability:** Is the system intuitive, comprehensible and simple to use? *intelligibility, learning requirements, ownership*
4 **Efficiency:** Does the system use resources well? time aspects *(e.g. performance characteristics), resource requirements (e.g. scalability)*
5 **Maintainability:** Can the workforce, developers and users upgrade the system when needed? *analysability, modifiability, configurability, testability*
6 **Portability:** Can the system work on different platforms, with different databases, etc.? *adaptability, installation requirements, compliance with standards, replaceability*

It is important for us not to forget the above characteristics. The question is, whose responsibility is it to ensure that they work? Functionality is the part that we testers work with most, while reliability and efficiency are more technical aspects which often require some form of tool and specialist knowledge. Usability is brought into focus when the user interface is designed. This should be done by experts in usability,[39] [40] but this is too often neglected. It is not a good idea to leave this type of testing to testers who lack specialist knowledge of these issues!

7.6 How Many Design Techniques are There?

Actually, the question of how many design techniques there are is, in itself, not very interesting. What is interesting, is which techniques you need to know in order to do a good job, and how you adapt the techniques for each situation, so that they suit precisely that situation. Just because you vary a technique to suit the situation does not mean that it is a whole new technique. It is not recommended that you have a favourite technique that you always use, but rather have a range of basic techniques that you master well and can apply in different situations.

For many years, I have been learning the martial art Aikido, which is based on the self-defence techniques of the Samurai. My Sensei talks about twelve basic techniques which can be varied according to which situation you find yourself in. It is very important to master the basic techniques well, the next step being to be able to adapt your defence to your opponent and the situation. It is the precise feeling and timing which marks out a master.

Below is a selection of different organisations' views on which test techniques are important. As you will see, the way to subdivide them and what is considered important differs depending on whom you talk to.

7.7 The Selection in This Book

The techniques described in this book are the selection that I personally consider every tester should have knowledge of, and be able to apply. All these techniques are conventional and well-known in testing internationally. If you want to read more about each technique, there are plenty of references in the bibliography. In no way does this selection claim to be exhaustive, or the only one you should know about, but it is a good base to start from.

- Data: *equivalence partitions, boundary value analysis, domain testing*
- Flows: *work processes, use cases for test cases*
- Event based: *state graphs*
- Logic: *decision trees, decision tables*
- Combinational analysis: *all pairs, elementary comparisons*
- Risk-based testing: *risk, defect guessing, taxonomies, heuristics, attack patterns*
- Advanced testing: *scenario-based, soap opera, time cycles, data cycles*
- For the developer: *control flow, data flow*

7.8 The ISTQB Standard

There is an international alliance called the International Software Testing Qualifications Board, ISTQB[41]. In their syllabus for accreditation on their

foundation course, they deal with the test design techniques they think are the most important for functional tests. From the start, this selection builds on the British standard BS 7925:2 for component testing. In the latest official version, the following techniques are listed: [42]

1. Equivalence partitioning
2. Boundary value analysis
3. State diagrams
4. Decision tables
5. Use case testing
6. Classification tree method

They deal additionally with the following approaches:

7. Defect guessing
8. Exploratory testing

For structural testing, they deal with code coverage in the form of a number of different variants where you cover all of the following:

1. Statements
2. Branches
3. Loop
4. A combination of branches and conditions (multiple combination decision coverage)

7.9 The Academic World

I asked the question of what were the most important test techniques to the well-known authority on testing, Cem Kaner, who is also a Professor at the Florida Institute of Technology, and the author of several books, and got the following answer:[43] [44]

He teaches 10 elementary techniques in his black-box testing course:

1. Function testing
2. Specification-based testing
3. Domain testing
4. Risk-based testing
5. Scenario testing
6. Regression testing
7. Stress testing
8. User testing
9. State machines
10. Volume testing

This is what Cem says:

> *«I have together with James Bach chosen these ten since they exemplify the most important areas within black-box testing:*
>
> *There is a separate course for component testing. However, the purpose of reading out these techniques is not to make people good at just ten techniques. I want the pupils to understand that there are a great many different variants of methodology, that they have fundamentally different strengths and that the skilful tester chooses the technique, or makes up a technique, which suits the needs of the day.*
>
> *An example of variation: some of our techniques focus on input and output data, others on control flows, some on interaction with other programs or units (this can be modelled as input/output data, but because the interactions often contain sequences of messages, it can often be better modelled as a state graph with accompanying data analysis). Certain techniques focus on compliance with legal requirements, specifications, announcements etc. Others focus on actual use of the product etc. All of these drive the tester towards new research, on a quest for different sources of information as an aid in design and evaluation of test cases. «*

7.10 Companies Which are Far Ahead in the Field of Testing

Peter Zimmerer from Siemens in Germany presented what he calls the Test Design Poster at EuroSTAR in December 2005[45]. The fact that I have chosen to reproduce his whole list here is down to the fact that it is one of the most complete catalogues I have seen. He says this:

> *«The Test Design Poster contains a categorised overview of test design methods, paradigms, techniques, styles and ideas for how to generate test cases. We give it out to everyone who works in development and testing in order to give them inspiration. Then, we go out into the organisation, and show them how to use the different methods.*
>
> *Satisfactory testing requires several different methods to be used in combination. The choice of methods depends on many factors, including:*

1. Requirements on the system and desired quality
2. Requirements on the tests in terms of strength and depth
3. Test strategy: which tests are performed in which parts of the chain of development
4. Existing basic test data
5. The system to be tested
6. Technique for soft- and hardware
7. Compatible tool support

The workload/difficulty of the methods, or the actual intensity of them, is divided into five layers, from 1, which is very small/simple, to 5 which is very large/difficult.

Category	Method, paradigm, technique, style and ideas for test cases	Degree of difficulty/ intensity
Black Box Interfaces, data, models	Standards, norms, formal specifications, requirements	3
	Criteria, functions, interfaces	1
	Requirement - based with traceability matrix	3
	Use case based (activity diagram, sequence diagram)	3
	CRUD, Create, Read, Update, Delete (database operations)	3
	Scenario tests, soap opera tests	4
	User profiles: frequency and priority/critical(reliability)	4
	Statistical testing (Markov chains)	4
	Random (ape testing)	4
	Design with contract (built - in self test)	3
	Equivalence Groups	2
	Classification trees	3
	Domain tests, category partition	4
	Boundary value analysis	2
	Special values	1
	Test catalogue for input data values, input data fields	5
	State Graphs	3
	Cause - effect graphs	5
	Decision tables	5
	Syntax tests (grammar)	4
	Combinatory testing (pair tests)	3
	Evolution tests	5
Grey - box	Dependences/relationships between classes, objects, methods, functions	2
	Dependences/relationships between components, services, applications, systems	3
	Communication behaviour (dependence analysis)	3
	Tracking (passive testing)	3
	Protocol based (sequence diagram)	4

Figure 71a Test Design Poster part 1

White - box	Control flows	Coverage	Statements	2
Internal structure, paths		(code - based, model - based)	Branches	3
			Conditions	4
			Interfaces	4
		Static measurement	Cyclomatic complexity (McCabe)	4
			Measurement (e.g, Halstead)	4
	Data flows		Read/write	3
			Define/use	5
Positive, valid cases	Normal, expected construction			1
Negative, invalid cases	Unauthorised construction			3
	Defect management			3
	Exception management			5
Defect - based	Risk - based			2
	Systematic defect analysis (FMEA: Failure Mode and Effect Analysis)			4
	Defect catalogues, bug taxonomies (Biezer, Kaner)			4
	Attack patterns (Whittaker)			3
	Defect models which depend on the technology and nature of the system			2
	Defect patterns: standard patterns or from root cause analysis			3
	Defect reports (previous)			2
	Error guessing			2
	Test patterns (Binder), Question patterns (Q - patterns: Vipul Kochar)			3
	Ad Hoc, intuitive, based on experience			1
	Exploratory testing, heuristics			2
	Mutation tests			5
Regression testing	Parts which have changed			1
	Parts affected by the changes			2
	Risky, highly prioritised, critical parts			3
	Parts which are often changed			3
	Everything			5

Figure 71b Test Design Poster part 2

7.11 Other Sources

There are many further sources where you can find lists of techniques. Some of the best known are the recent new version TMAP Next [46], which is widespread in countries like the Netherlands and Belgium. Boris Beizer has written a couple of classic works on testing techniques: Software Testing Techniques[47] and Black Box Testing Techniques[48]. Finally, there is Glenford Myers' book The Art of Software Testing[49] from 1979, which was one of the first to be written in the field. Many of his ideas stand up well even today and there is hardly a single new book on testing on the market which does not contain a reference to this book.

FOOTNOTES

[29] ISTQB [2005]: *Course plan*, chapter 4

[30] Fagan, Michael[1976]: *Design and Code inspections to Reduce Errors in Program Development*. IBM Systems Journal, 15(3), pp. 182–211

[31] Gilb, Tom, Graham, Dorothy [1993]: *Software Inspection*

[32] Weinberg, Gerald[1971]: *The Psychology of Computer Programming*

[33] Pol, Martin, Teunissen, Ruud, van Veenendaal, Erik [2002] *Software Testing – A Guide to the TMAP Approach* pp. 219–223

[34] Myers, Glenford [1979] *The Art of Software Testing*, p.37

[35] Beizer, Boris [1995], *Black Box Testing* p. xiv

[36] *Soap-opera testing, in short, is about thinking through all the strange and extreme cases that can arise.*

[37] Copeland, Lee [2003]: *A Practitioner´s Guide to Software Test Design* pp. 139–142

[38] Copeland, Lee [2003]: *A Practitioner´s Guide to Software Test Design* p. 8, and Whittaker, James [2003]: *How to Break Software* pp. 57–58

[39] *www.funkanu.se*

[40] Cooper, Alan[2004]: *The Inmates are Running the Asylum*

[41] *www.istqb.org*

[42] *ISTQB course plan version 0.2.*

[43] Kaner, Cem [2005] *groups.yahoo.com/group/software-testing/message/2880*

[44] *www.testingeducation.org/BBST/BBST--IntroductiontoTestDesign.html*

[45] Zimmerer, Peter [2005]: *Eurostar presentation*

[46] Koomen et al [2006] *TMAP Next*

[47] Beizer, Boris, [1993] *Software Testing Techniques*

[48] Beizer, Boris [1995], *Black Box Testing*

[49] Myers, Glenford [1979] *The Art of Software Testing*