EuroSTAR
eBook Series
MAY 2018

Test Techniques For The Test Analyst

Drs. Erik van Veenendaal

EuroSTAR Software Testing | Huddle

Skill Level

This eBook is for an audience of all skill levels.

Abstract

Test techniques have been around for many years. They are used explicitly or implicitly by most testers. There are many sources of information with respect to test techniques, their methods and coverage measures. It is not the intention of this e-book to completely repeat or re-iterate such information. This e-book intents to provide a to-the-point overview of the most popular test techniques. This overview will support the test technique selection process when determining a test strategy or test approach. It also looks at a number of reasons why we should consider the use of test techniques as a part of our everyday test activities and who should be looking to use them.

Specification-based test techniques have a more-or-less standard procedure for deriving test cases from requirements documents. For each specification-based technique described, a list of characteristics is provided and the level of (requirements) coverage that can be achieved is discussed.

Experience-based testing, e.g., exploratory testing, is a powerful approach to testing. In many situations, it has shown to be very productive. There isn't a tester yet who didn't, at least unconsciously, perform experience-based testing at one time or another. The three most popular experience-based techniques are described in this e-book.

About The Author

Twitter: @ErikvVeenendaal
www.erikvanveenendaal.nl

Drs. Erik van Veenendaal, CISA is a leading international consultant, trainer and a recognized expert in the area of software testing and requirement engineering.

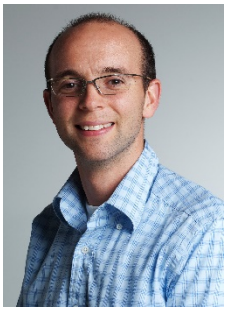
Erik is the (co-)author of numerous papers and a number of books on software quality and testing. He is a regular speaker, e.g., running a tutorial on test design techniques, at both national and international testing conferences and a leading international trainer in the field of software testing.

Since its foundation in 2002, Erik has been strongly involved in the International Software Testing Qualifications Board (ISTQB). From 2005 to 2009, he was the vice president of the ISTQB organization; he currently is the president for the Curaçao Testing Qualifications Board (CTQB).

Erik is one of the core developers of the TMap test methodology and the TMMi test improvement model, and currently the CEO of the TMMi Foundation.
For his major contribution to the field of testing, Erik received the European Testing Excellence Award (2007) and the ISTQB International Testing Excellence Award (2015).

Test Techniques for the Test Analyst

by Erik van Veenendaal



This e-book is dedicated to Paul Quik, a dedicated, passionate, enthusiastic and professional tester, but above all a wonderful person, husband and father. †

Table of Contents

Test Techniques for the Test Analist

Test techniques have been around for many years. They are taught as part of ISTQB classes and are used explicitly or implicitly by most testers. There are many sources of information with respect to test techniques, their methods and coverage measures. It is not the intention of this e-book to completely repeat or re-iterate such information. This e-book intents to provide a to-the-point overview of the most popular test techniques. This overview will support the test technique selection process when determining a test strategy or test approach. It also looks at a number of reasons why we should consider the use of test techniques as a part of our everyday test activities and who should be looking to use them.

Specification-based test techniques have a more-or-less standard procedure for deriving test cases from requirements specification and design documents. For each specification-based technique described, a list of characteristics is provided and the level of (requirements) coverage that can be achieved is discussed. Also, for each technique the specific test case design procedure is briefly described with an example.

Experience-based testing, e.g., exploratory testing, is a powerful approach to testing. In many situations, it has shown to be very productive. There isn't a tester yet who didn't, at least unconsciously, perform experience-based testing at one time or another. The three most popular experience-based techniques are described in this e-book.

What you wil learn

After reading this e-book the reader will be able to:

- Explain the most popular specification-based and experience-based test techniques
- Understand the key characteristics for the various test techniques
- Recall the procedure for deriving test cases using a specification-based testnique
- Compare specification-based techniques to experience-based techniques
- Recommend the most appropriate test techniques in a specific situation.

The e-book is based on a number of chapters from the book **The Testing Practioner** by Erik van Veenendaal.

1. An introduction to Test Techniques

1.1 Different types of Techniques

In this e-book we will look at the different types of test design technique that are commonly used by the test analyst, how they are used and how they differ. Two types or categories of test techniques are distinguished by their primary source: a specification, or a person's experience. Both categories are useful and highly complementary.

There are many different types of test techniques, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and often relatively poor

at finding other types. For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than defects associated with combinations of inputs.

According to the Oxford English Dictionary a technique is a:

- mode of artistic execution in music, painting, etc.
- mechanical skill in art
- means of achieving one's purpose, esp. skilfully.

Software testing has been referred to as an "art", amongst other things, for many years. Perhaps parts two and three of the dictionary definition should be combined to reflect the current state of the testing industry. Test techniques, at least some, make part of the exercise mechanical in that the production of the test cases becomes formalised. However, the overriding need for test techniques, is the need to achieve our purpose. The purpose being; to provide objective and measurable tests, find defects and results to allow users to make a well-founded decision about the likely impact of taking the system into live operation.

What subset of all possible test cases has the highest probability of finding most defects? [Myers]. Well, that's a loaded question if ever there was one! Although Myers' book was published in 1979 many others probably asked that question well before that time and certainly many thousands have asked it since. The testing fraternity are still trying to answer it. However, it may well be that there is no definitive objective answer! The question has an implicit measure within it; in that "subset" implies that testers cannot/will not/should not even try to run every possible test case, even if they have been able to identify them. That being the case, the very simple question requires a very complicated answer, within which there will be caveats, compound decisions and "what if?" statements abounding.

Test techniques come in many shapes and sizes, some formal some not, some dynamic and some static. It is not the intention of this e-book to explain in every detail the functions, coverage or methods of all of these techniques. The overall objective is to discuss the issues surrounding the use, selection, advantages and disadvantages of the various test techniques as a set of tools to support test analysis and design and defect finding.

The test techniques being most relevant to the test analyst and considered in this e-book are divided into the two following categories:

- Specification-based (or black box) test techniques
- Experience-based test techniques.

These categories techniques are complementary and may be used as appropriate for any given test activity, e.g., specification-based techniques may be combined with experience-based techniques to leverage the experience of developers, testers and users to determine what should be tested. Note that both categories of techniques can be used to test both functional or non-functional quality characteristics.

Other categories of test techniques, not considered in this e-book, but that need to be at least briefly mentioned are:

- Static techniques, e.g., reviews, static code analysis
- Structure-based (or white-box) test techniques, e.g., statement and decision testing
- Defect-based techniques, e.g., software attacks, defect taxonomies

As stated before, in this e-book "Test Techniques for the Test Analyst" we will limit ourselves to discussing specification-based test techniques and experience-based test techniques. This is because these are these ones most commonly used by the test analyst (see figure 1), which is one of the clear conclusion from the Software Testing Practices Report [ISTQB]. Within both categories we will be focusing on those techniques that are most popular. This again being based on the result from the Software Testing Practices Report.

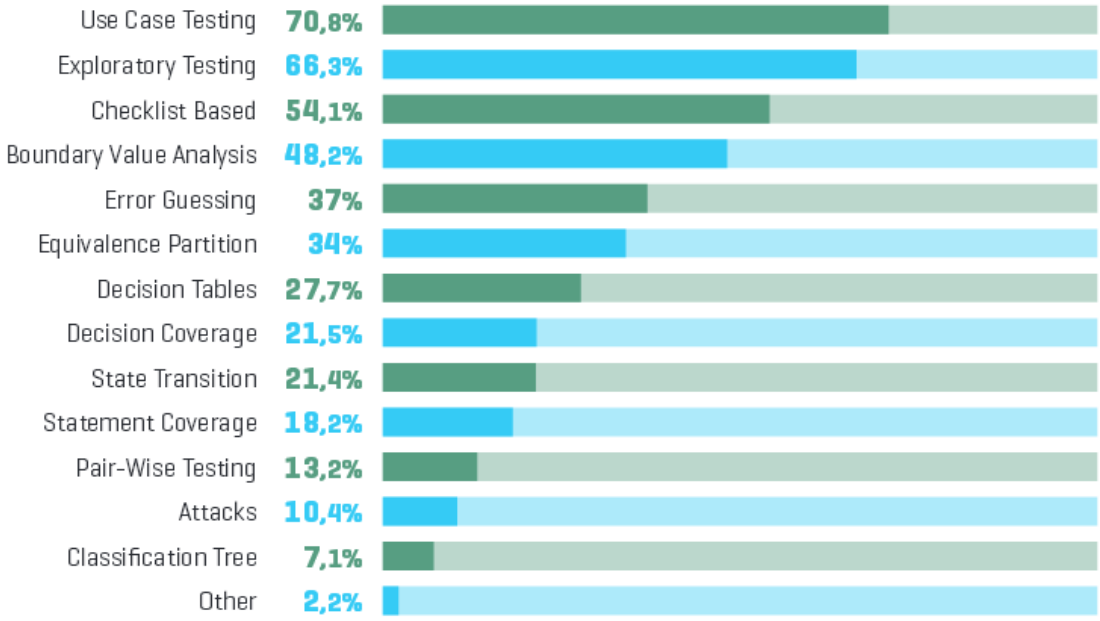


Figure 1: Most adopted test techniques

From the list above of most adopted test techniques, Use Case Testing, Boundary Value Analysis, Equivalence Partitioning, Decision Tables, State Transition, Pairwise testing and Classification Trees, all being specification-based techniques, are discussed in the e-book. Exploratory Testing, Checklist based and Error Guessing, all being experience-based techniques, are also discussed. The structure-based techniques Decision Coverage and Statement Coverage (number 8 and number 10 on the list, and both probably mostly used by technical testers at component testing) and the defect-based technique Attacks (number 12 on the list with a mere 10,4% uptake) are "unfortunately" not discussed.

Specifically for the tester who loves coverage measures, 77% of the most adopted test techniques are covered in this e-book, and if we take into account the uptake percentages, no less than 88% of the situations where test techniques are being applied are covered.

1.2 Agile

In Agile testing, many tests are created by testers concurrently with the developers' programming activities. Just as the developers are programming based on the user stories and acceptance criteria, so are the testers creating tests based on user stories and their acceptance criteria. Some tests, such as exploratory tests and some other experience-based tests, are created later, during test execution, as explained in section 3. Testers can apply "traditional" specification-based techniques such as equivalence partitioning, boundary value analysis, decision tables, and state transition testing to create these tests. For example, boundary value analysis could be used to select test values when a customer is limited in the number of items they may select for purchase. It is important for the test analyst to perform both testing of the individual user story functionality supplied as well as integration of the user story in the overall system. Also non-functional requirements can be documented in the user story format. Specification-based test techniques can also be used to create tests for non-functional quality characteristics. A user story can for example contain performance or reliability requirements. It is important for the test analyst to perform both testing of individual user story fun

User stories

In most Agile projects, requirements come in a user story format. This doesn't change the nature of the testing task. It just means that the test basis is delivered in a different format.

User story: A high-level user or business requirement commonly used in agile software development, typically consisting of one or more sentences in the everyday or business language capturing what functionality a user needs, any non-functional criteria, and also includes acceptance criteria.

User story is one of the primary development artifact for agile project teams. In agile methodologies requirements are prepared in the form of user stories which describe small functional units that can be designed, developed, tested and demonstrated in a single iteration [Cohn]. These user stories include a description of the functionality to be implemented, any non-functional criteria, and also include acceptance criteria that must be met for the user story to be considered complete. A user story contains just enough information so that development team can reasonably give estimate about completing, tester can discuss how it will validated and customer can see its value. One of the common question that, how user stories are different from use cases. User stories are much simpler than use cases. User stories are very easy to create, discuss and develop. They also do not contain any technical details.

Typically good user stories are defined in the following format:
"As a <role>, I want <goal/desire> so that <benefit>".
An user story example: "As a student, I want to be able to buy a parking pass so I can get to school quickly".

It is also common to identify the acceptance criteria, which the stakeholders will use to validate that the user story has been implemented correctly. Some examples of possible acceptance criteria based on the user story example earlier in this section:
"As a student, I want to be able to buy a parking pass so I can get to school quickly".

- A pass should be issued per month
- The student will not receive the parking pass if the payment is insufficient
- One can only buy a parking pass to the school parking lot if the person is a registered student
- The student can only but one parking pass per month.

Test techniques

Experience-based techniques, e.g., exploratory testing, fit almost perfectly with Agile for many reasons. Typically in Agile there is limited time available for test design, the level of detail of the test basis may make it more difficult to apply specification-based techniques, experience-based techniques are low on documentation, center around the experience of testers, etc. When moving to Agile, some testers even stop applying specification-based techniques.

However, in many cases there is still a need for and value to also using specification-based techniques. The application may have a part where the processing depends on a number of combined input parameters and to test this thoroughly a decision table is needed, or a part where boundary values are used, is critical to the functionality. In such cases, and many others, applying the principles of a specification-based techniques will have added value. In Agile context this is sometimes done explicitly, but more often implicitly. We should of course only do things that have added value. The objective should never be to apply a test design techniques, follow a procedure, or to design test cases. The objective is to find defects. Test techniques as such, are a means to achieve an objective. In summary, in order to achieve the best results, experience-based techniques should most often be blended with other categories of techniques, e.g., specification-based techniques.

Some "guidelines" and things to remember for applying specification-based techniques in an Agile context:

- Use them in a pragmatic way, use them as required by the situation or context.
- The objective of the technique, e.g., the specific defects being targeted, are are much important that the method and procedure being used.
- Often a test technique can be applied in different ways and has different levels of thoroughness (see sections hereafter) and thus provides flexibility. These variations must be understood and the derived flexibility should be used as such.
- Tests are much more important than detailed test specifications, test procedures and/or other documentation.
- Different techniques can complement each other; try to combine the principles of different techniques.
- Using techniques implicitly sounds easy, but is often in practice much more difficult than using them explicitly. Get training and study the various test techniques, become a master at using them.

An example of how a specification-based technique can be applied with a set of user stories is provided in section 2.6 “State Transition Testing”.

2. Specification-Based Techniques

2.1 Introduction

The first category of test techniques to be presented are the specification-based test techniques. Specification-based test techniques are also known as “black-box” test techniques because they view the test object as a black-box with inputs and outputs, without having knowledge on how the system or component is structured inside the box. In essence, the tester is concentrating on what the software or system does, not how it does it. All specification-based techniques have the common characteristic that they are based on a model (formal or informal) of some aspect of the specification, which enables test cases to be derived in a systematic way. With specification-based test techniques the test conditions and test cases are derived systematically from these models (the test basis).

Specification-based technique: Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

Note that the specification-based test design techniques discussed in the following sections focus primarily either on determining an optimal number of test cases (e.g., equivalence partitions) or deriving test sequences (e.g., state transition testing). In practice, it is common to combine techniques to create even more “complete” test cases.

The decision to use or not to use specification-based techniques is based on many factors including risk, customer/contractual requirements, type of system, regulatory requirements and time and budget [Foundations]. Initially a basic view of both the advantages and disadvantages of specification-based techniques must be taken.

Advantages	Disadvantages
Higher level of objectivity	Requires training (at least to some degree)
Formal coverage measures possible	Time to implement – it’s usually a culture change
Early defect finding when doing test design	Everyone must be ‘bought in’
Traceability from requirements to test cases (audit trails possible)	Not seen as useful for all applications and all lifecycle models
Coverage less dependent of the tester	Takes more time than informal test design

Advantages	Disadvantages
Way to differentiate test depth based on risks using different techniques	Do not cover all situations (experience-based testing still useful)
High level of re-use (re-usable testware)	Little use of domain and product knowledge of tester
Repeatability of tests and reproducibility of defects	Documentation intense
Good defect finding capability	Difficult to respond to last minute changes – less flexible

Table 1: Advantages / disadvantages specification-based techniques

There are a number of specification-based testing techniques. These techniques target different types of software, systems and scenarios. For each technique a short description is provided in the following sections, describing amongst others how to apply the technique. To make the understanding and selection of the specification-based test techniques easier, and to be able to compare the techniques, a number of characteristics is provided for each technique:

- Test level
- Test basis
- Coverage
- Application area
- Type of defects that are targeted
- Quality characteristics
- Limitations and difficulties that the test analyst may experience.

Test level

Some techniques are more usable at integration level, while others are more suitable for testing at system or acceptance level. An indication is provided regarding at what level the techniques are most appropriate and most often used. A distinction is made in component, integration, system and acceptance test level.

Test basis

Specification-based test techniques are based on a specification. Whether or not a technique can be applied is amongst others determined by the available documentation. Some techniques only need high level documentation to be applicable, where others need very specific information. In some cases there may even be no documented requirements, but only implied requirements such as replacing the functionality of a legacy system.

Coverage

Most specification-based techniques also provide coverage criteria, which can be used for measuring coverage during test design and/or test execution activities. For specification-based test techniques coverage criteria do not relate to the software code, but to the level of thoroughness with which the requirements are tested. Completely fulfilling the coverage criteria

does not mean that the set of test is complete, but rather that the model no longer suggests any additional tests to increase coverage based on that technique.

Application area
Some specification-based techniques are particularly suited to test the interaction between a system and its users (user interfaces, reports), while others are more suited to test the relationship between a business processes and a system or to test batch processing. Yet another group is used to test the integration between components. The applicability of the various techniques is also related to the type of defects that can be found using them, such as incorrect input validation, incorrect processing or integration defects.

Quality characteristics
For each technique a reference is provided to the ISO 25010 quality characteristics answering the question "What quality characteristics can be tested with this specific technique?" Some techniques are for example more suitable for testing usability; others are more suitable for functionality, interoperability or security.

2.2 Equivalence Partitioning

Equivalence partitioning: A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once.

Equivalence Partitioning (EP) is most used specification-based technique and is applicable at all levels of testing. The basic idea is to divide the input domain into equivalence classes (EC's) or partitions which, according to the specification, have the same behavior. The basis of the technique is that any value chosen from an equivalence partition is as valid as any other, since it is expected to be processed in the same way. By selecting one representative value from a partition, coverage for all the items in the same partition is assumed. The technique aims at reducing the number of test cases that is required to test the handling of inputs, outputs, internal values and time-related values to a manageable size, resulting in a small but highly effective test set. An huge advantage of EP is that with a limited set of test cases a good level of coverage can be achieved. The set of test cases that results from applying EP can also be used as a basis for a regression test set. EP can easily be extended to or combined with Boundary Value Analysis (see section 2.3).

Characteristics

Test levels	All test levels, but especially recommend for integration testing and system testing.
Test basis	Requirements and design documents

Coverage	Coverage items are the partitions described by the test basis. Coverage is calculated as follows: Equivalence partition coverage = (Number of covered partitions/Total number of partitions) * 100% Note that using multiple tests for a single partition does not increase the coverage percentage.
Application area	All types of systems
Type of defects	Functional defects in the handling of various data values.
Quality characteristics	Functionality, Interoperability

Table 2: Characteristics Equivalence Partitioning

Applicability

This technique is appropriate when all members of a set of values to be tested are expected to be processed in the same way and where the various sets of values used by the application do not (largely) interact. Selecting test values is applicable to both valid and invalid partitions (i.e., partitions containing values are considered invalid for the test object being tested). This technique is strongest when used in combination with boundary value analysis, which will expand the test values to also include those on the edges of the partitions.

Limitations/Difficulties

If the assumption is incorrect and the values within a partition are not handled in exactly the same way, this technique may miss defects. It is also important to select the partitions carefully. For example, an input field that accepts both positive and negative numbers should be tested as two partitions, one for the positive numbers and one for the negative numbers, because of the likelihood that they are processed differently. If zero is allowed, this may become another partition as well. It is important for the test analyst to have a basic understand the underlying processing in order to determine the best way to partition the values.

Design Procedure

When applying EP four distinct steps have to be taken. The first step is to identify relevant input attributes. The second step is to identify the accompanying equivalence classes. Then, as a third step, the test cases are identified and described. The last step is to also partition the output values.

Identify relevant input attributes

EP starts by studying the requirements specification for input attributes that influence the processing of the test object. This can be straightforward, e.g., the fields on a screen or the parameters of an interface. However, one can also think of attributes such as system parameters, hardware platforms, operating systems etc. Since the number of input attributes that can be managed in one test specification is limited, this step sometimes also involves splitting up the test object in a number of test units that will be tested separately. If for instance a test object has

50 or more input attributes, one may split up the test specification into two test specifications, that partly overlap, to handle the complexity (each with approximately 25 to 30 input parameters).

Consider the following specification:

> ***Benefit***
> Every person receives a benefit of 350. In addition it is determined whether a person has worked and that his/her age is higher than 40. In this case the benefit is raised by 100. Alternatively (else) for persons that are not working and have exactly 4 children the benefit is raised by 50.

For this specification, the following three input attributes can be distinguished:

- Working history
- Age
- Number of Children.

Note that the input attributes are listed atomically (non-compound), e.g., 'A' can be an input attribute, 'B' can be an input attribute, but 'C AND D' has to be split up into two separate input attributes.

Identify equivalence classes

For each attribute that has been identified at step one, the accompanying equivalence classes need to be identified. Each equivalence class (EC) is a representative of (or covers) a large set of possible values. During this step the equivalence classes are identified using the following rules [Kit]:

- For a boundary value, 1 valid EC (within the boundary) and 1 invalid EC (outside the boundary)
- For a boolean, 1 valid EC (true) and 1 invalid EC (untrue)
- For a range, 1 valid EC (within the range) and two invalid EC's (one outside each end of the range)
- If the input is a number (N), 1 valid EC (the exact number) and two invalid EC's (smaller than N and more than N)
- If the input is a set of valid values, 1 valid EC (from within the set) and 1 invalid EC (outside the set)
- If there is reason to believe that elements in a EC are not handled in an identical manner by the program, subdivide the EC into smaller EC's
- For compulsory input also test with an empty input.

For the specification example "benefit" applying these rules would result in the following equivalence classes:

Attribute	Rule	Valid EC's	Invalid EC's
Working history	Boolean	Yes	No
Age	Boundary Value	>40	≤ 40
Number of Children	Number	4	<4, >4

Note that invalid doesn't necessarily mean that the values are not accepted by the system. It can also mean that the inverse of a stated requirement is tested, as is the case in this example.

Identify test cases

The next step is to combine the various EC's to define the content of the test cases. Start by making completely valid test cases until all valid EC's have been covered by test cases. Test cases are written that cover as many of the uncovered EC's as possible. For the "benefit" example only one test case is needed to cover all three valid EC's. Subsequently test cases are designed for the invalid EC's. This is done until all invalid EC's have been covered by the test cases. It's important to note that a test case will cover one, and only one, invalid EC. If multiple invalid EC's are tested in the same test case, some of those tests may never be executed because the first test may mask other tests or even terminate the execution of the test case. At the end it is often interesting to add one test case consisting of only invalid EC's. Of course a necessary part of any test case is a description of the expected result.

For the specification example "benefit" this will result in the following test cases:

Test case	01	02	03	04	05	06
Working history	Yes	No	Yes	Yes	Yes	No
Age	>40	>40	≤40	>40	>40	≤40
Number of Children	4	4	4	<4	>4	<4
Expected result	450	400	400	450	450	350

For better understanding of the example, the invalid EC's are indicated in yellow.

Output partitioning

Sometimes it is possible to not only apply input partitioning, but to also apply output partitioning. This is often perceived as a verification of the first three steps of the EP technique. During output partitioning "all" possible outputs are listed based on the specification, e.g., the error messages, and a check is carried out to see whether the expected results of the test cases already identified cover all possible outputs. If not, additional test cases are designed to cover these outputs. Test cases may also be designed to test that invalid output values cannot be induced.

For the specification example “benefit” the possible results for benefit received are 350, 400 and 450. All of these are already covered in the existing set of test cases, thus no additional test cases are needed based on the output partitioning step.

2.3 Boundary Value Analysis

Boundary value analysis: A black box test design technique in which test cases are designed based on boundary values.

Boundary Value Analysis uses a model of the component and partitions the input and output of that component, e.g., a test item (function, feature) with numeric inputs that can be divided in equivalence classes. The values at and around the boundaries of an equivalence class are referred to as boundary values. These are values at which often defects can be found. When determining the test cases, values close to these boundaries are chosen so that each boundary is tested with a minimum of two test cases, or three for full boundary value coverage (see design procedure hereafter for details). Applying boundary value analysis increases the chances of finding defects compared to a random selection from within a equivalence class.

Boundary value analysis can be applied at all test levels, but is especially recommended for component testing. One does not need a fully integrated system to test boundary values, and as always the earlier the defect is found the better. Boundary value analysis is relatively easy to apply and its defect finding capability is relatively high, but more detailed specifications are needed. When applying boundary value analysis more test cases will be created and therefore more effort, also during test execution, is needed compared to equivalence partitioning. Boundary value analysis is often considered an extension of equivalence partitioning.

Characteristics

Test levels	All, but especially recommended for component testing
Test basis	Requirements and design documents
Coverage	The coverage items are the boundaries of partitions described by the test basis. Some partitions may not have an identified boundary, for example, if a numerical partition has a lower but not an upper bound. Coverage is calculated as follows: Boundary value coverage = (number of distinct boundary values executed / total number of boundary values) * 100%
Application area	All types of software systems, especially usable for testing components (functions) with numeric equivalence classes.
Type of defects	Defects regarding the processing of the boundary values, particularly defects with “less-than” or “greater-than” logic.
Quality characteristics	Functionality, Correctness

Table 3: Characteristics Boundary Value Analysis

Applicability
This technique is applicable at any level of testing and appropriate when numeric equivalence partitions exist. Numeric is required because of the concept of being on and over a boundary. For example, a range of valid numbers is an numeric partition. A partition that consists of all rectangular objects is not an numeric partition and does not have boundary values. In addition to ranges with numbers, boundary value analysis can also be applied to numeric attributes of non-numeric variables (e.g., length), loops (including those in use cases), stored data structures, physical objects (including memory) and time-determined activities. Note that it is also possible to test for extreme boundaries (the minimum and maximum value permitted).

Limitations/Difficulties
The successfull application of this technique depends on the accurate identification of the boundaries of the equivalence partitions. The test analyst should also be aware of increments in the various valid and invalid partitions, to be able to accurately determine the values to be tested. Only numeric partitions can be used for boundary value analysis but this is not limited to a range of valid inputs (see above).

Design Procedure
When applying boundary value analysis, the equivalence classes have to be defined first (see section 2.2). When this activity is completed, the following additional rules have to be taken into account to derive boundary value analysis test cases:
- For each input that concerns a range, test cases for the end of the range and invalid input test cases for conditions just beyond the ends
- If the input is a number of valid values, test cases for the minimum and maximum number of values and one beneath and beyond these values.

As stated when applying boundary value analysis, one has to define the equivalence classes first. When a requirement specifies for example the following condition:

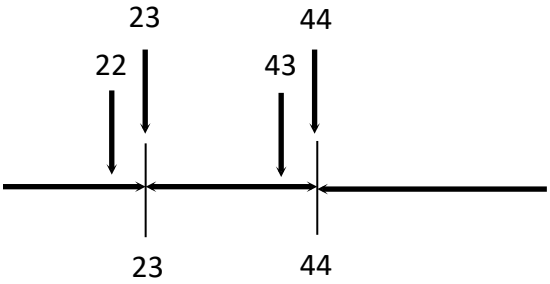
$$\text{IF } 23 \leq \text{age} < 44 \text{ THEN } ....$$

The following three distinct equivalence classes for age can be found:
- Age less than 23
- Age has a value in the range 23-43 (boundaries included)
- Age is greater or equal than 44.

Two or three values?
There are two ways to approach boundary value analysis: two value or three value testing. With two value testing, the boundary value (on the boundary) and the value that is just over the boundary (by the smallest possible increment) are used. The boundaries are defined by the maximum and minimum values in the defined equivalence partition. For three value boundary testing, the values before, on and over the boundary are used. The decision regarding whether

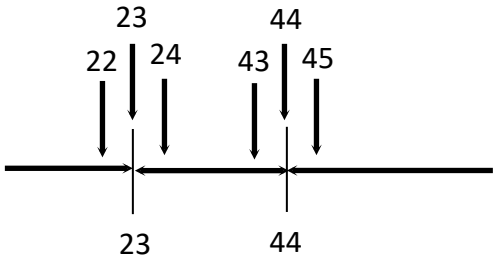
to use two or three boundary values should be based on the risk associated with the item being tested, with the three boundary values approach being used for the higher risk items.

Following the “age” example the boundary values to be selected, when taken the minimum of only two boundary test cases, for age are: 22 (invalid), 23 (valid), 43 (valid) and 44 (invalid). In this case the minimum and maximum value of each partition is tested.



For full boundary value analysis three values per boundary are selected. Thus for each identified boundary three test cases shall be identified covering the values on the boundary and an incremental distance on either side of it. This incremental distance is defined as the smallest significant value for the data type under consideration. The additional test value is then chosen just within or just without the equivalence class defined by the boundary value.

Taking the previous “age” example once again. For the lower boundary the boundary test cases will now be 22 (invalid), 23 (valid) and 24 (valid). The added boundary value test case will find an additional defect when for example ”age=23” has been implemented. This defect would not have been found when testing with only two boundary value analysis test cases. For the upper boundary the boundary values test cases are 43 (valid), 44 (invalid) and 45 (invalid). In the figure below the boundary values are shown that are to be tested for full boundary value analysis.



2.4 Decision Table Testing

Decision table testing: A black box test design technique in which test cases are designed to execute the combinations of inputs and/or stimuli (causes) shown in a decision table.

Decision table testing is a technique which aims to create test cases for “interesting” combinations of inputs. The specification is analyzed, and conditions (inputs) and results (intermediate results, messages, outputs) are identified. The conditions and results have to be stated in such a way that

they can either be true or false (boolean). Decision tables are used to test the interaction between multiple conditions. They provide a way to test all combinations of conditions and to verify that all possible combinations are processed correctly by the software under test. The objective of decision table testing is to test every, or at least every interesting, combination of conditions, relationships and constraints.

The strength of decision table testing is that it creates combinations of inputs that might not otherwise have been exercised during testing. Some research even suggests a 90% defect detection rate can be achieved with decision table testing. A disadvantage is that when there is a large number of inputs (conditions), the technique can become very complex, expensive and even unfeasible. Therefore decision table testing should only be applied on selected parts of the functionality to reduce the complexity and especially on high-risk items. Since decision table testing is a thorough technique that will test the test object in great detail, it also requires detailed specifications of good quality in order to be applicable.

Characteristics

Test levels	Component testing (when a component has decision logic), integration testing, system testing
Test basis	Requirements and design documents
Coverage	Coverage items are rules, where each rule represents a unique possible combination of inputs to the component that have been expressed as booleans. Coverage is calculated as follows: Decision table coverage = (Number of rules exercised/Total number of rules) * 100%
Application area	Test items with boolean conditions (or that can be expressed as booleans). To be used for business rules, critical components (functions), state machines and complex processing.
Typical defects	Incorrect processing based on particular combinations of conditions resulting in unexpected results. During the creation of the decision tables, defects could already be found in the specification document because of the detailed test analysis that is performed. The most common types of specification defects are omissions (there is no information regarding what should happen in a certain situation) and contradictions.
Quality characteristics	Functionality, Correctness, Interoperability

Table 4: Characteristics Decision Table Testing

Applicability

This technique is particularly applicable when the requirements are presented in the format of flow charts, tables of business rules. Decision tables are also a requirements specification

technique and some requirements may even arrive exist in a decision table format. Even when the requirements are not presented in a tabular or flow-charted format, conditions are usually found in the narrative. When designing decision tables, it is important to consider the defined conditions and their combinations, as well as those that are not explicitly defined but do exist. Only when all interacting conditions are considered a decision table is an effective tool for testing. Note that some easy to use supporting tools are available for decision table testing, making it, in many organizations, a very popular test technique.

Limitations/Difficulties
Finding all the interacting conditions can be challenging, particularly when requirements are not well-defined or do even not exist. It is not unusual while identifying conditions to discover that expected results are unknown and thus to find defects at an early stage. The technique rapidly becomes infeasible when the number of input conditions is too large. Collapsed decision tables (see hereafter) is unfortunately only part of the answer. In general when the number of input conditions is too large one needs to partition the test item, and subsequently design separate decision tables of manageable size.

Design Procedure
When using decision table testing the specifications have to be studied in detail to derive conditions and expected results. For further explanation the "benefit" specification example will again be used:

> Every person receives a benefit of 350. In addition it is determined whether a person has worked and that his/her age is higher than 40. In this case the benefit is raised by 100. Alternatively (else) for persons that are not working and have exactly 4 children the benefit is raised by 50.

Analyzing the specification above, the following conditions (inputs) are found:
- Person has worked
- Age of person is higher than 40
- Person has exactly 4 children

The following expected intermediate results (outputs) can be found:
- Benefit received of 350
- Benefit raise of 100
- Benefit raise of 50

All this information will be placed in a so-called decision table. Since decision table testing uses 100% multiple condition coverage as its main principle, all possible combinations of inputs are tested.

Multiple condition coverage: The percentage of combinations of all single condition outcomes within one statement that have been exercised by a test suite.
Multiple condition testing: A white-box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).

Therefore the number of test cases for decision table testing is 2 to the power of N, where N is the number of conditions (inputs). With 3 conditions as in the example, the number of test cases will 2^3 thus 8 test cases.

General rules when identifying the conditions for a decision table are:
- In principle only single conditions, compound conditions are split up
- Formulate the condition in a positive way (don't use NOT), making the decision table much easier to understand
- Rank the conditions by priority, first the condition that has the most impact on the results, etc. (especially important for collapsed tables – to be discussed hereafter).

The decision table based on the "benefit" example is as follows:

Test cases / Conditions/Results	1	2	3	4	5	6	7	8
Person has worked	1	1	1	1	0	0	0	0
Age of person is higher than 40	1	1	0	0	1	1	0	0
Person has exactly 4 children	1	0	1	0	1	0	1	0
Benefit received of 350	X	X	X	X	X	X	X	X
Benefit raise of 100	X	X						
Benefit raise of 50					X		X	
Expected end result	450	450	350	350	400	350	400	350

Note that sometimes combinations of conditions are infeasible, making the test case itself impossible e.g., when conditions are mutual exclusive.

Collapsed tables
Since the number of test cases grows exponentially with a larger number of inputs the decision table testing technique rapidly becomes infeasible to use. A number of authors have developed ways to bring the maximum number of test cases down to a reasonable number. The most commonly used rule for reduction of the number of test cases is the one of collapsed tables [Mors]. When all test cases are studied closely, one usually observes that there is a number of closely related test cases. Mors has developed a strategy for identifying likely 'non-relevant' test cases, which can subsequently be removed. Be aware that applying collapsed decision tables or not, should always be a risk management decision [Copeland].

The standard rule for reducing the number of test cases is that, if for two test cases only one condition (input) differs and the resulting effects (actions) are the same these test cases (columns)

can be joined. Two columns are compared that only differ one place in the table (the last condition pairs with his neighbour). It typically helps to look at the change of the Yes / No (or 1 / 0) value for a condition.

If this rule would be applied to the benefit specification example and resulting decision table, amongst others test cases, test case 1 and 2 can be joined (see decision table below). For test case 1 and 2 only one condition is different, and the result is the same. One of these two test cases can now be removed, since the chance of finding different defects with these two test cases is considered low. Following this rule also test cases 3 & 4, 5 & 7, and 6 & 8 can be combined resulting in the following collapsed decision table:

Test cases / **Conditions/Results**	**1**	**2**	**3**	**4**
Person has worked	1	1	0	0
Age of person is higher than 40	1	0	-	-
Person has exactly 4 children	-	-	1	0
Benefit received of 350	X	X	X	X
Benefit raise of 100	X			
Benefit raise of 50			X	
Expected result	450	350	400	350

Things to remember with collapsed tables:
- always delete the column on the right
- an '-' is not the same as a 'Y' or 'N'
- a column is only involved in the comparing process once (e.g., 1-2, 3-4, not 1-2, 2-3, 3-4)
- for infeasible combinations: if for two test columns only one condition differs and one test column is an infeasible combination they may be joined; the feasible column "survives".

Finally, and of course most important, it should be a risk management decision! To understand this last statement even better, let's define a decision table for the requirement: "A phone company will only accept customers that have a valid address and are older than 18 years".

The full decision table will look like this:

Test Case	**1**	**2**	**3**	**4**
Valid Address	Y	Y	N	N
Older Than 18	Y	N	Y	N
Accept	X			
Not accept		X	X	X

The collapsed decision will look like this: (check!)

Test Case	**1**	**2**	**3**	**4**
Valid Address	Y	Y	N	N
Older Than 18	Y	N	-	N
Accept	X			
Not accept		X	X	X

What happens if you have decided for the 3rd test case of the collapsed table to use 17 as your age value (as you are free to choose any value), and the developer in his program only checks for age "older than 18" to accept and has forgotten to check also on the valid address?

2.5 Cause-Effect Graphing

Cause-effect graphing: A black box test design technique in which test cases are designed from cause-effect graphs.
Cause-effect graph: A graphical representation of inputs and/or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

This specification-based test technique is based upon an analysis of the specification of the component to model its behavior by means of causes and effects. Cause-effect graphs may be generated from any source which describes the functional logic (i.e., the "rules") of a program, such as user stories or flow charts. They can be useful to gain a graphical overview of a program's logical structure and are typically used as the basis for creating decision tables. Capturing decisions as cause-effect graphs and/or decision tables enables systematic test coverage of the program's logic to be achieved. Cause-effect graphing strongly resembles decision table testing as discussed in the previous section. Mainly the way the test design is documented differs: a cause-effect graph vs. a decision table.

Characteristics

Test levels	Component testing (when a component has decision logic), integration testing, system testing
Test basis	Requirements and design documents
Coverage	Each possible cause to effect line must be tested, including the combination conditions, to achieve minimum coverage. Coverage items are rules, where each rule represents a unique possible combination of inputs to the component that have been expressed as Booleans. Coverage is calculated as follows: Cause-effect coverage = (Number of rules exercised/Total number of rules) * 100%
Application area	Test items with boolean conditions (or that can be expressed as booleans). To be used for business rules, critical components (functions), state machines and complex processing.

Typical defects	Cause-effect graphing typically find the same types of combinatorial defects as can be found with decision tables. In addition, the creation of the graphs can help to define the required level of detail in the test basis, and thus helps to improve the level of detail and quality of the test basis. As a result it supports identifying missing and ambiguous requirements.
Quality characteristics	Functionality, Correctness, Interoperability

Table 5: Characteristics Cause-Effect Graphing

Applicability
Cause-effect graphing applies in the same situations as decision table testing and also apply to the same test levels. In particular, a cause-effect graph shows condition combinations that cause results (causality), condition combinations that exclude results (not), multiple conditions that must be true to cause a result (and) and alternative conditions that can be true to cause a particular result (or). These relationships can be easier to observe in a cause-effect graph than in a narrative description.

Limitations/Difficulties
Cause-effect graphing requires additional time and effort to learn compared to some other test design techniques. It also requires tool support for creating cause-effect graphs. Cause-effect graphs have a particular notation that must be understood by the creator and reader of the graph. As a result of these limitations the uptake of cause-effect graphing is limited and many testers choose decision table testing instead.

Design Procedure
As with decision table testing, also with cause-effect graphing the specifications have to be studied in detail to derive conditions and results. For further explanation the following specification example will be used:

Car Insurance
If the car is small (less than 1.4l) or the owner lives in Bonaire, offer Cheap insurance. If the car is large (more than 2.0l) offer the Executive Options package.

Analyzing the specification above, the following causes (conditions) are found:
- C1: Small car (less than 1.4l)
- C2: Large car (more than 2.0l)
- C3: Owner live in Bonaire

The following effects (actions) can be found:
- A1: Offer cheap car insurance
- A2: Offer executive options

All this information will be used in constructing the so-called cause-effect graph. A cause-effect graph shows the relationship between the conditions and actions in a notation similar to that used by designers of hardware logic circuits. The specification “Car Insurance” is now modeled by the cause-effect graph as shown in figure 2.

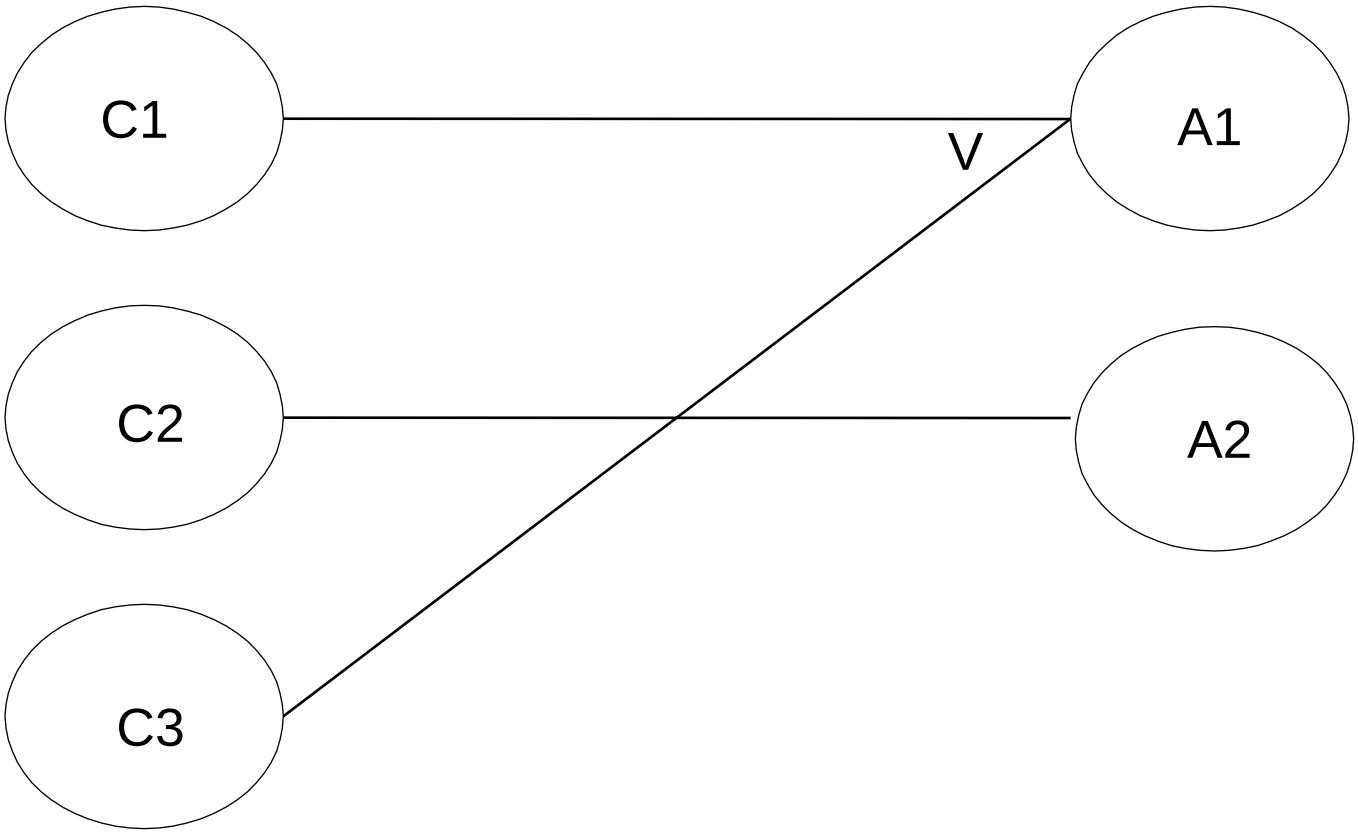


Figure 2: Cause-effect graph “Car Insurance”

Note that cause-effect graphs use standard logical symbols for AND and OR (one can observe the V symbol in the graph above to represent an AND) and a wavy line indicating NOT. The arc groups those inputs affected by a logic symbol, where there are more than two inputs involved.

The code graph is subsequently recast in the format of a decision table (see previous section). Each column of the decision table is a test case. In both cause-effect graphing and decision table testing one attempts to test all possible combinations, however sometimes a combination of conditions is infeasible. An asterisk (*) in a decision table indicates that the combination of conditions is infeasible and so no further actions are required for this test case.

The “Car Insurance” example has the following decision table:

Test cases / Causes/Effects	1	2	3	4	5	6	7	8
C1: Small car (less than 1.4l)	1	1	1	1	0	0	0	0
C2: Large car (more than 2.0l)	1	1	0	0	1	1	0	0
C3: Owner lives in Bonaire	1	0	1	0	1	0	1	0
A1: Offer cheap car insurance	*	*	X	X	X		X	
A2: Offer executive options	*	*			X	X		

Thus six test cases would be required to provide 100% cause-effect coverage, and need to be created in line with the columns in the decision table above. Of course no test cases are created for columns 1 and 2 as they are infeasible.

2.6 State Transition Testing

State transition testing: A black box test design technique in which test cases are designed to execute valid and invalid state transitions.

State transition: A transition between two states of a component or system.

State transition testing is based on the concepts of states and finite-state machines, and allows the tester to view the developing software in terms of its states, transitions between states, and the inputs and events that trigger state changes. State transition testing uses a model of the states the component may occupy, the transitions between those states, the events which cause those transitions, and the actions which may result from those transitions. Events cause the software to transition from state to state and to perform actions. Events may be qualified by conditions (sometimes called guard conditions or transition guards) which influence the transition path to be taken. The states of the model are separated, identifiable and finite in number. State transitions are tracked in either a state transition diagram that shows all the valid transitions between states in a graphical format or a state table which shows all potential transitions, both valid and invalid. A state model is typically produced for the component to identify its states, transitions, and their events and actions.

State Transition Diagrams (STD) are commonly used as state models and their notation is briefly illustrated in figure 3. Events are always caused by input. Similarly, actions are likely to cause output. The output from an action may be essential in order to identify the current state of the component. A transition is determined by the current state and an event, and is normally labeled simply with the event and action.

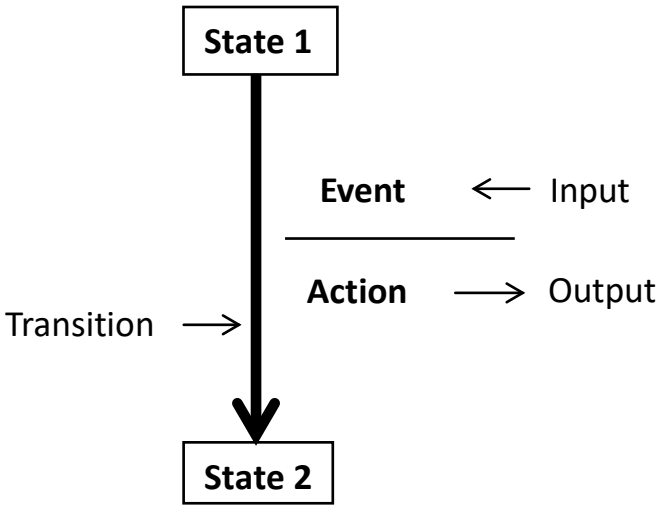


Figure 3: State Transition Diagram notation

State testing is much used within the embedded software industry and technical automation in general. However, the technique is also suitable for testing screen-dialogue flows, e.g., with websites or mobile applications.

Characteristics

Test levels	Component Testing, Integration Testing, System Testing
Test basis	State Transition Diagrams
Coverage	Coverage items are sequences of one or more transitions between states on the model. For single transitions, the coverage metric is the percentage of all valid transitions exercised during test. This is known as 0-switch coverage. For *n* transitions, the coverage measure is the percentage of all valid sequences of *n* transitions exercised during test. This is known as (N - 1) switch coverage.
Application area	State based systems, e.g., control systems, technical automation, embedded software, mobile applications and websites
Typical defects	Incorrect processing in the current state as a result of the processing that occurred in a previous state, incorrect or unsupported transitions, states with no exits and the need for states or transitions that are not specified and do not exist. Also memory issues can be found with this technique. During the creation of the state diagram, defects may be found in the specification document. The most common types of defects are omissions (there is no information regarding what should actually happen in a certain situation) and contradictions.
Quality characteristics	Functionality, Interoperability, Resource-utilization (memory)

Table 6: Characteristics State Transition Testing

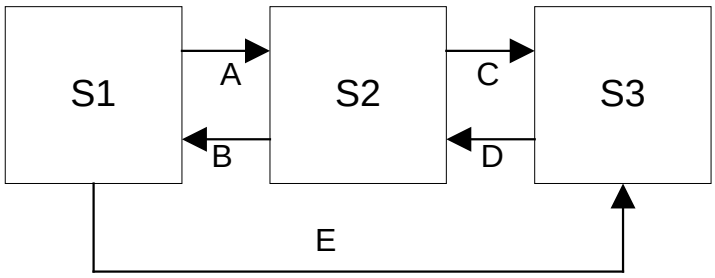
Applicability
State transition testing is applicable for any software that has defined states and has events that will cause the transitions between those states (e.g., changing screens). Some tools are available to support state transition testing. Since this technique follows a strict procedure the possibilities for automation of the test design step are high.

Limitations/Difficulties
Determining the states is often the most difficult part of defining the state table or diagram. When the software has a user interface, the various screens that are displayed for the user are often used to define the states. For embedded software, the states will be dependent upon the states that the hardware will experience.

Design Procedure
State transition diagrams show the state, the transitions, the inputs, events, actions and outputs in a special notation (see figure 3). Test cases derived from it, aim at exercising the various transitions between states. A test case may exercise any number of subsequent transitions. For each transition within a test case, the following shall be specified:

- the starting state
- the event which causes transitions to the next state
- the expected action caused from the component
- the expected next or end state.

For example, a state transition diagram (STD) with three states and 5 transitions could look like the diagram below.



When applying state transition testing the level of thoroughness can be varied by using a different level of switch coverage. Switch coverage (or N-switch) is defined by ISTQB as “the percentage of sequences of N+1 transitions that have been exercised by a test suite”.

In practice most often a distinction is made between 0-switch coverage and 1-switch coverage. When 0-switch coverage is applied every transition is executed once. This means that a test case will only go from one state to another (i.e. from S1 to S2). 100% 0-switch coverage (also called transition coverage or logical branch coverage) will therefore guarantee that every state is visited and every transition is traversed, unless the system design or the state transition model (diagram or table) are defective.

Based on the example STD the set of test cases for 0-switch coverage would be:

Test Case	**1**	**2**	**3**	**4**	**2**
Start State	S1	S1	S2	S2	S3
Input	A	E	B	C	D
End State	S2	S3	S1	S3	S2

These 5 test cases cover all possible transitions for the example STD for 0-switch coverage.

However, simply testing all transitions will find some types of state transition defects, but more may be found by testing sequences of transactions. When 1-switch coverage is deployed a test case contains two subsequent transitions that can be executed one after each other. A sequence of three successive transitions is a 2-switch, and so forth. The higher levels of switch coverage may stimulate defects that 100% 0-switch coverage would not encounter.

Using 1-switch coverage, one can extent the 0-switch tests to check for each test case what will happen if a subsequent transition is executed. This means for each test a flow of two state transitions is to be executed. This is especially useful for testing resource-utilization, e.g., memory. The test cases for 1-switch coverage would be:

Test Case	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Start State	S1	S1	S1	S2	S2	S2	S3	S3
Input	A	A	E	B	B	C	D	D
Next State	S2	S2	S3	S1	S1	S3	S2	S2
Input	B	C	D	A	E	D	B	C
End State	S1	S3	S2	S2	S3	S2	S1	S3

Note that intermediate states, and the inputs and outputs for each transition, are explicitly defined.

For any of these approaches, an even higher degree of coverage will attempt to include all invalid transitions. Coverage requirements for state transition testing must identify whether invalid transitions are included. Invalid state transition test cases will be identified using a state table (see hereafter).

A limitation of the test cases derived to achieve a level of switch coverage is that they are all designed to exercise only the valid transitions within the component. A more thorough test of the component will *also* attempt to cause invalid transitions to occur. An STD only shows the valid transitions (all transitions not shown are considered invalid). A state model that explicitly shows both valid and invalid transitions is the state table. Hereafter the example STD used earlier is represented as a state table, this again shows the five valid test cases (shaded in green) but also shows a number of invalid tests, test cases ‘where nothing should happen’ (shaded in red). Thus the state table provides an ideal means of deriving also a set of invalid test cases.

	A	B	C	D	E
S1	S2	Invalid	Invalid	Invalid	S3
S2	Invalid	S1	S3	Invalid	Invalid
S3	Invalid	Invalid	Invalid	S1	Invalid

Table 7: Example State Table

User Stories example

We will close this section with one more example for state transition testing, this time starting with three user stories for a laptop as test basis.

US1 : As a laptop user I want to startup my laptop so I can start working with it
US2 : As a laptop user I want to shut down my laptop so I can leave and take it with me
US3 : As a laptop user I want to temporarily suspend my laptop so I can have a break

The first step to transform the narrative type of user story requirements into a STD. The resulting STD is shown hereafter.

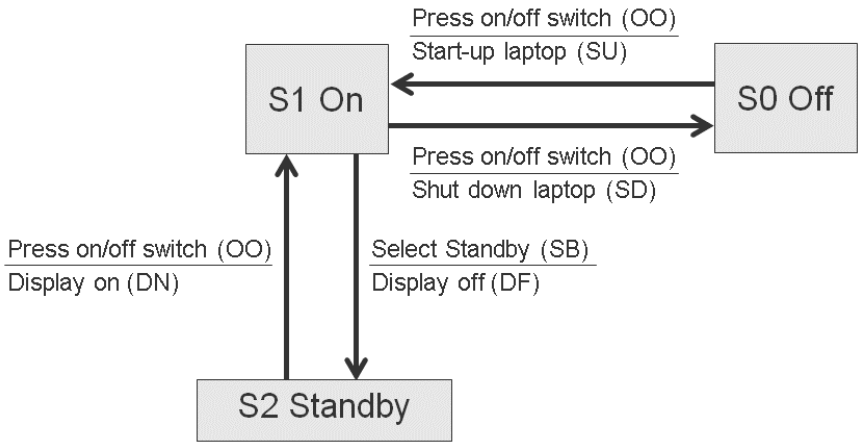


Figure 4: STD Laptop

Subsequently the standard test design procedure can again be followed. This will give the following test cases for 0-switch coverage :

Test Case	**1**	**2**	**3**	**4**
Start State	S0	S1	S2	S2
Input	OO	SB	OO	OO
Expected Output	SU	SD	DF	DN
End State	S1	S0	S2	S1

The accompanying state table, to be used for identifying the invalid test cases (shaded in red again) will be as follows:

	OO	SB
S0	S1	Invalid
S1	S0	S2
S2	S1	Invalid

2.7 Combinatorial Test Techniques

Combinatorial testing: A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple parameters and where those parameters themselves each have several values, which gives rise to more combinations than are feasible to test in the time allowed.

The objective of combinatorial testing is to provide a solution to combinatorial explosion. Combinatorial explosion occurs when a test problem can be described as a set of parameters with a number of different values and the total number of possible combinations of parameter values is too large to be feasibly tested. The main purpose of combinatorial testing is to identify a manageable set of (interesting) combinations.

In this paragraph the concept of combinatorial testing is introduced, in the two subsequent sections the two most popular combinatorial test techniques [ISTQB] are discussed: Classification Tree Method and Pairwise Testing.

Consider testing a web based system which should run on different operation systems (windows, Mac, Unix, Linux, Java, IOS), support different browsers (Internet Explorer, Google Chrome, Firefox, Safari, Android, Opera) and different types of data (gif, jpg, pdf, java script file, html, png). Testing all possible combinations would result in 6 x 6 x 6 = 216 test configurations. This would be most often be impossible to test within constraints of resources, budget and time.

The key insight underlying combinatorial testing is to test the interaction between the different parameters that effect the system, whereby not every possible parameter is a contributor to every defect. Most defects are caused by interactions between a relatively small number of parameters. The main principle behind combinatorial testing is that defects can be considered to fall in to one of the following categories:

- Single mode: in which something works or it fails
- Dual mode: in which even though two things work by themselves, they fail when paired (connected) together
- Multi-mode: in which three or more things in combination don't work together.

Empirical evidence has shown that the majority of defects in systems are either single mode or dual mode. Single modal tests are easy to scope because there will be one test for each area of functionality. The problems occur when functionalities are combined because this can lead to unmanageable numbers of tests being generated. So running all tests is not an option. If

combinations are picked at random, it is hard to determine coverage and it is possible that defects are being missed, so testing would be inefficient. For example, within pairwise testing (one of the combinatorial techniques discussed hereafter) the answer is not to attempt to test all the combinations for all values for all variables, but to test only *all pairs* of variables. This significantly reduces the number of tests that must be created and run.

Characteristics

Test levels	All, although mainly used during integration and system testing
Test basis	Requirements and design documents
Coverage	There are several levels of coverage. The lowest level of coverage is called 1-wise or singleton coverage. It requires each value of every parameter to be present in at least one of the chosen combinations (test cases). The next level of coverage is called 2-wise or pairwise coverage. It requires every pair of values of any two parameters be included in at least one combination. This idea can be generalized to n-wise coverage, which requires every sub-combination of values of any set of n parameters be included in the set of selected combinations. The higher the n, the more combinations (test cases) needed to achieve 100% coverage.
Application area	All types of systems – see hereafter in the paragraph applicability for some examples of specific situations where combinatorial testing is especially useful.
Typical defects	Defects related to the combination of specific values of multiple parameters.
Quality characteristics	Functionality, Interoperability, Portability

Table 8: Characteristics Combinatorial Test Techniques

Applicability

Combinatorial testing is most useful when testing complex configuration scenarios and/or situations where there are multiple input parameters with numerous variables per parameter that are expected to have effect on a single output condition or state. The parameters must be independent and compatible in the sense that any option for any parameter can be combined with any option for any other parameter. For example, testing an application on multiple versions of Windows, with different browser versions, and different protocols and connection speeds, then combinatorial test techniques will help to define a baseline set of test environment configurations. Another example would be testing an API that has several parameters with multiple arguments values that can be passed to those API parameters, then combinatorial will also help testers establish a baseline set of tests. Combinatorial testing can also be applied to input controls on an user interface that affect a common output state or condition. In all of these situations, combinatorial testing can be used to identify a subset of combinations, feasible in size.

For parameters with a large number of values, equivalence class partitioning, or some other selection mechanism may first be applied to each parameter individually to reduce the number of values for each parameter, before combinatorial testing is applied to reduce the set of resulting combinations.

Limitations/Difficulties
The major limitation with combinatorial test techniques is the assumption that the results of a limited set of test cases is representative of all tests and that those selected tests represent expected usage. If there is an unexpected interaction between certain variables, it may go undetected with this type of testing if that particular combination is not tested. Another problem is that these techniques are usually difficult to explain to a non-technical audience as they may not understand the logical reduction process resulting in the limited test set.

Identifying the parameters and their respective values is sometimes difficult depending on the quality of the requirements specification. Also finding the minimal set of combinations to satisfy a certain level of coverage is often difficult to do manually. Tools are usually used to find the minimum set of combinations. Some of these tools support the ability to force some (sub-) combinations to be included in or excluded from the final selection of combinations. This capability may be used by the test analyst to emphasize or de-emphasize factors based on domain knowledge or product usage information.

Note the comments on applicability and limitations/difficulties stated in the paragraph are generic for all combinatorial test techniques and therefore also apply to the ones explained hereafter: Classification Tree Method and Pairwise Testing.

2.8 Classification Tree Method

Classification tree method: A black box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains.

The Classification Tree Method (CTM) [Grochtmann] is a special approach to partition testing by partly using and improving ideas from the equivalence partitioning technique (see section 2.2) and applying it to the principles of combinatorial testing. By means of the CTM, the input domain of a test object is regarded under various aspects assessed as relevant for testing. For each aspect, disjoint and complete classifications are formed. Classes resulting from these classifications may be further classified – even recursively. The stepwise partition of the input domain by means of classification is represented graphically in the form of a tree. Subsequently, test cases are formed by combining classes of different classifications. This is done by using the tree as the head of a combination table in which the test cases are marked. Classification trees allow for some combinations to be excluded, if certain options are incompatible. When using the CTM, the most important source of information for the tester is the requirements specification of the given test object. Various tools are available to support the CTM.

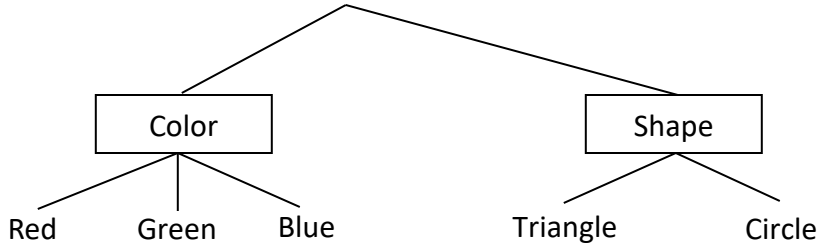
Design Procedure

Selecting test objects

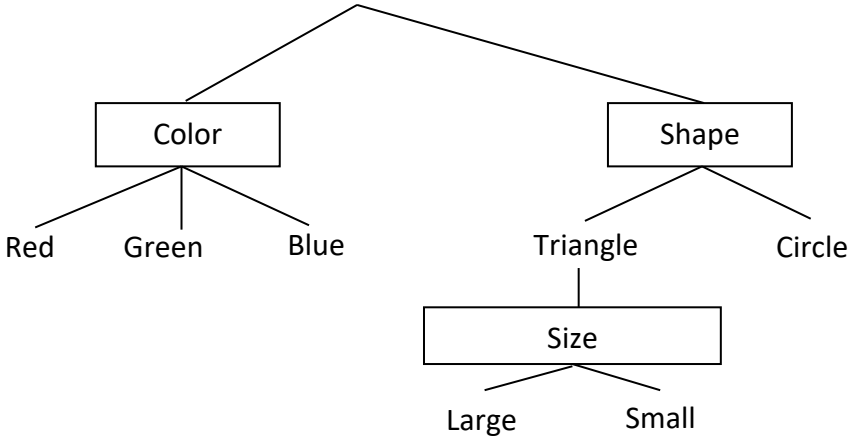
A large system cannot be tested reasonably with a single classification-tree, as such a tree would become much too large to handle. Therefore, the functionality of the system under test has to be divided into several separate test objects. This has to be done in such a way that each of the resulting test objects can be tested individually and that by testing all test objects the complete system is tested thoroughly.

Designing a classification tree

During this step, a classification tree has to be built up for each of the test objects, reflecting all test relevant aspects. To derive test cases by CTM one first identifies the aspects of the test object that influence the functional behavior of the component. As an example a camera is used that should detect different kinds of objects. These objects can vary in color and shape (the two relevant aspects in our example). The classification based on the aspect "color" leads to a partition of the input domain into, for example, the classes red, green and blue. The shape could either be triangle or circle. See figure below.



Classes itself can be further classified. In our example the triangle class has been split into two new classes: the size of triangle. See figure below.

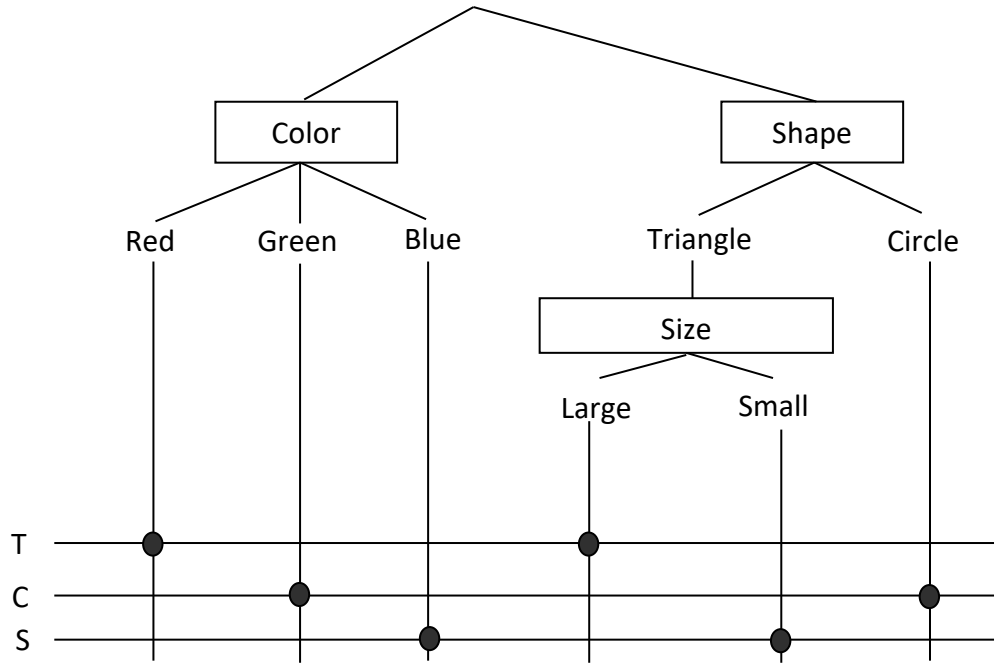


Combining classes to form test cases

When the classification-tree is finished, all the information needed for preparing test cases is available. A test case is formed by:

- Selecting a class for each aspect
- Combining these classes.

During this step, test cases are to be selected and generated that cover the most important real-life situations for each test object. Thus domain knowledge is important here. The number of test cases in principle remain for the tester to decide. However, the classification tree also provide clues to the number of test cases required. The minimum number of test cases is achieved when every leaf class is covered by at least one test case (where the leaf class is defined as a class in the classification tree not further divided into sub-classes). Since leaf classes of the same base classification cannot be combined, the minimum criterion is the largest number of leaf classes that belong to a base classification. This level of coverage is called 1-wise coverage. The minimal variant of the CTM will in our example lead to three test cases.



According to the classification-tree method the three test cases would look like:

- Test case 1: Red colored large Triangle
- Test case 2: Green colored Circle
- Test case 3: Blue colored small Triangle.

When more thorough testing is necessary based on the risks identified for the test object, more combinations of classes shall be covered by test cases. The theoretical maximal variant requires that every possible combination of the classes of all aspects must be covered by test cases. It is

basically the number of test cases that results when all permitted combinations of leaf classes are considered. In our example, the maximum criterion amounts to 9 (i.e. 3 * 3). The supporting tools for CTM also have the ability to define logical dependencies within the tree and automatically generates test cases based on a set of pre-defined (coverage) rules. It allows the test analyst to define the combinations to be tested (i.e., combinations of two values, three values, etc.).

A reasonable number of test cased obviously lies somewhere between the minimum and maximum criterion. As a rule of thumb, the total number of leaf classes provide an estimate for the number of test cases required to get sufficient test coverage. In this case that would be 6 (i.e. 3+3). The objective of the Classification Tree Method is to determine a sufficient, but limited number of test cases. So generally speaking, it is not necessary to specify a test case for each possible combination. In fact, the Classification Tree Method should enable the tester to have a good overview of the possibilities using the classification tree. As a result thorough choices can made reducing the number of tests, at least partly based on domain knowledge selecting those test cases that are most representative of real-life situations. In practical applications, this reduction of test cases is essential, since the maximum criteria can easily run into very high numbers.

2.9 Pairwise Testing

Pairwise testing: A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters.

Pairwise testing is another technique that can be used for testing combinations of values. Pairwise testing is a technique that identifies all pairs of values from the total input domain. The level of coverage achieved is referred to as 2-wise coverage. By using this technique, the number of tests can be reduced while still having confidence in the coverage. Remember that the majority of defects in systems are either single mode or dual mode.

Suppose we want to demonstrate that a new software application works correctly on PCs that use the either Windows or Linux as an operating systems, either Intel or AMD processors, and the IPv4 or IPv6 protocols. This is a total of $2 \times 2 \times 2 = 8$ possibilities but, as the table below shows, only four tests are required to test every component interacting with every other component at least once (check!). Thus with pairwise testing, four tests cover all possible pairs of values among these three parameters.

Test case	Operating System	Processor	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

Even though pairwise testing is not exhaustive, it is useful because it can check for simple, potentially problematic interactions with less tests. The reduction in test set size from eight to the four shown in the example may not be that impressive, but consider a larger example: a manufacturing automation system that has 20 controls, each with 10 possible settings providing the tester with a total of 10^{20} (100,000,000,000,000,000,000) combinations, which is far more than a tester would be able to test in a lifetime. Surprisingly, one can check all pairs of these values with only 180 tests provided they are carefully constructed. Also for pairwise testing there are a of tools available to support the test analyst in his task. These tools require the parameters and their values to be listed and will show the test cases required for covering all pairs of values.

Design Procedure

Identify the variables
Referring back to the example for the web based system at the beginning of the section on combinatorial testing, the variables are operation system, browser and type of data.

Determine the number of choices (values) for each variable
- Operating system: Windows, Mac, Unix, Linux, Java, IOS (6)
- Browser: Internet Explorer, Google Chrome, Firefox, Safari, Android, Opera (6)
- Type of data: gif, jpg, pdf, java script file, html, png (6).

Remember, multiplying 6 x 6 x 6 provides 216 combination to test. For full test coverage, each of these combinations should be tested. For more complex systems, there would be even far greater numbers. With pairwise testing all pairs of values are identified from the total input domain. This will substantially reduce the number of tests while still having confidence in the coverage.

Locate an orthogonal array
A common method of deriving pairwise tests is by using orthogonal arrays. These arrays have been used by mathematicians for centuries to solve problems. They are a set of numbers that have the properties of including all pair combinations. Let's look at an example.

	A	B	C
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Orthogonal areas have the property that any two columns that will contain all pairs of values. In the example above each of the columns covers the numbers 1, 2 and 3. Looking at columns A and C for example, we can see that this does cover all pairs: (1, 1) (1, 2) (1, 3) (2, 2) (2, 3) (2, 1) (3, 3) (3, 1) (3, 2). The next step is to use this property to derive tests. This is done by substituting the numbers in the array with the variables of the system we are testing.

The notation for orthogonal arrays is $L_x n^y$, where x = number of rows, y = number of columns and n = number of choices available within the column. So the array used in the example above is the $L_9 3^3$ array. Arrays do not always have the same number of choices for each column. For example, the $L_{64} 8^2 4^3$ array has two columns that contain 8 choices and 3 columns that contain 4 choices. The array selected must be large enough to contain all choices that the requirements define. The array can never be smaller. There are a finite number of orthogonal arrays. Therefore, the number of attributes/choices to be tested may not match exactly with an available orthogonal array. In these cases, the next largest array should be used and manipulated to produce the tests required.

If the perfect size array does not exist, choose one that is slightly bigger and apply the two following rues to deal with the "excess". The first rule deals with extra columns. If the array chosen has more columns than needed, simply delete them. The array will still be orthogonal. The second rule deals with extra values for a variable. It is tempting to delete the rows that contain these cells but don't! The array may not be orthogonal thereafter. Each row in the array exists to provide at least one pair combination that appears nowhere else in the array. If you delete a row, you lose that test case. Instead of deleting them, simply convert the extra cells to valid values.

Let's now return to our example for the web based system. Which array is needed? First, it must have three columns, one for each variable in this example. All three columns must support 6 different values. The perfect size orthogonal array would be 6^3.

Browsing through the library of possible orthogonal arrays on the internet will give $L_{36} 2^4 3^1 6^3$ as our most closely related array. We have now managed to lower the number of test cases from 216 to only 36 (!!), while still testing all pairs of values. Hereafter the $L_{36} 2^4 3^1 6^3$ array is provided in full.

1	0	0	0	0	0	1	5	5
2	0	0	0	0	1	1	1	2
3	0	0	0	0	2	0	3	1
4	0	0	0	1	0	2	4	3
5	0	0	0	1	2	3	0	2
6	0	0	0	1	2	4	2	0
7	0	0	1	1	0	4	1	4
8	0	0	1	1	1	3	3	5
9	0	0	1	1	2	5	5	3

10	0	1	0	0	1	3	4	0
11	0	1	0	0	1	5	0	4
12	0	1	0	1	0	0	2	4
13	0	1	1	0	0	2	0	0
14	0	1	1	0	1	0	1	3
15	0	1	1	0	2	1	2	1
16	0	1	1	0	2	4	4	5
17	0	1	1	1	0	5	3	2
18	0	1	1	1	1	2	5	1
19	1	0	0	0	0	5	4	1
20	1	0	0	0	2	2	3	4
21	1	0	0	1	1	0	5	0
22	1	0	1	0	0	0	0	5
23	1	0	1	0	0	3	2	3
24	1	0	1	0	1	2	2	2
25	1	0	1	0	2	5	1	0
26	1	0	1	1	1	1	4	4
27	1	0	1	1	1	4	0	1
28	1	1	0	0	0	4	5	2
29	1	1	0	0	1	4	3	3
30	1	1	0	1	0	3	1	1
31	1	1	0	1	1	5	2	5
32	1	1	0	1	2	1	0	3
33	1	1	0	1	2	2	1	5
34	1	1	1	0	2	3	5	4
35	1	1	1	1	0	1	3	0
36	1	1	1	1	2	0	4	2

Since we do not need the first 5 columns, as we are only interested in the three columns with 6 values, the first five columns will just be deleted based on the rule stated earlier in this section. Now there is a focused $L_{36}6^3$ orthogonal array remaining that is needed to test all pairs for the web based system.

1	1	5	5
2	1	1	2
3	0	3	1
4	2	4	3
5	3	0	2
6	4	2	0
7	4	1	4
8	3	3	5
9	5	5	3

10	3	4	0
11	5	0	4
12	0	2	4
13	2	0	0
14	0	1	3
15	1	2	1
16	4	4	5
17	5	3	2
18	2	5	1
19	5	4	1
20	2	3	4
21	0	5	0
22	0	0	5
23	3	2	3
24	2	2	2
25	5	1	0
26	1	4	4
27	4	0	1
28	4	5	2
29	4	3	3
30	3	1	1
31	5	2	5
32	1	0	3
33	2	1	5
34	3	5	4
35	1	3	0
36	0	4	2

Map the test problem to the array

The necessary test configurations for the web based system will be mapped to the orthogonal array. The choices for the operating system will be mapped onto column 1, those for the browser onto column 2 and those for the type of data onto column 3.

The mapping for the operating system values to the array parameters (first column) is as follows:

0: Windows	1: Mac	2: Unix
3: Linux	4: Java	5: IOS

The mapping for the browser values to the array parameters (second column) is as follows:

0: Internet Explorer	1: Google Chrome	2: Firefox
3: Safari	4: Android	5: Opera

The mapping for the type of data values to the array parameters (third column) is as follows:

0: gif	1: jpg	2: pdf

3: java script file 4: html 5: png

The resulting array for the 36 test configurations (test cases) for testing the web system is as follows:

1	Mac	Opera	Png
2	Mac	Google Chrome	Pdf
3	Windows	Safari	Jpg
4	Unix	Android	java script file
5	Linux	Internet Explorer	Pdf
6	Java	Firefox	Gif
7	Java	Google Chrome	html
8	Linux	Safari	Png
9	IOS	Opera	java script file
10	Linux	Android	Gif
11	IOS	Internet Explorer	Html
12	Windows	Firefox	Html
13	Unix	Internet Explorer	Gif
14	Windows	Google Chrome	java script file
15	Mac	Firefox	Jpg
16	Java	Android	Png
17	IOS	Safari	Pdf
18	Unix	Opera	Jpg
19	IOS	Android	Jpg
20	Unix	Safari	Html
21	Windows	Opera	Gif
22	Windows	Internet Explorer	Png
23	Linux	Firefox	java script file
24	Unix	Firefox	Pdf
25	IOS	Google Chrome	Gif
26	Mac	Android	Html
27	Java	Internet Explorer	Jpg
28	Java	Opera	Pdf
29	Java	Safari	java script file
30	Linux	Google Chrome	Jpg
31	IOS	Firefox	Png
32	Mac	Internet Explorer	java script file
33	Unix	Google Chrome	Png
34	Linux	Opera	Html
35	Mac	Safari	Gif
36	Windows	Android	Pdf

Construct the test cases
The last step to take is to construct a test cases for each row in the orthogonal array. Note that the array only specifies the input conditions. Of course a necessary part of each test case, is the expected result for that test case.

2.10 Use Case Testing

use case testing: A specification-based test design technique in which test cases are designed to execute scenarios of use cases. **use case:** A sequence of transactions in a dialogue between an actor and a component or system with a tangible result, where an actor can be a user or anything that can exchange information with the system.

A use case is a sequence of actions performed in a dialogue between a user and a system, which produces a valuable result to the system user. An use case typically has pre-conditions which need to be met for a use case to be executed successfully. For example, a cash withdrawal cannot be made without an open bank account, and a working automatic teller machine (ATM). Each use case terminates with post-conditions. Post-conditions are the results and final state of the system after the use case has been completed. An use case usually has a mainstream, most likely scenario, such as the successful withdrawal of funds from an ATM. The use case may also contain alternative branches such as the disapproval of a withdrawal because the user's bankcard is unrecognized or there is insufficient cash in the ATM.

Use case and test cases work well together. If the use cases for a system are complete, accurate, and clear, the process of deriving the test cases is straightforward. If the use cases are not in good shape, the attempt to derive test cases will help to debug the use cases. Use cases describe the "process flows" through a system based on its most likely use. The test cases derived from use cases are most useful in uncovering defects in the process flows during real-world usage of the system. Use cases are in fact very useful for designing user acceptance tests with customer/user participation. They also help uncover integration defects, caused by the interaction and interference of different system components, which individual component testing would not see.

Characteristics

Test levels	Integration testing, system testing, acceptance testing
Test basis	Requirements (Use Cases), User Manual
Coverage	No formal coverage measure
Application area	User interactive systems
Typical defects	Real-life defects, since the system is tested from a user perspective. Defect may also be found related to a mismatch

	between the software system and business processes. Typically functionality, but also usability and integration related defects.
Quality characteristics	Usability, Functionality, Suitability, Security, Interoperability

Table 9: Characteristics Use Case Testing

Design Procedure
The design of (test) use cases can be split into a number of steps.

Identification of users of the system
During this step the user groups of the system are identified and listed. Users can be human users but also other systems. The identified user groups are categorized and prioritized (mostly based on the numbers). For example, a copier may have the following users:

User	**Priority**
Walk-up user	High
Key operator	low
Operator	Medium
Service engineer	Low

Identification use cases for every user group
Users are interviewed, and documentation such as user manual, business/user processes and requirements specification are studied during this step. For every user group a list of typical use cases is identified. Every use case contains a short description (one or two sentences) and subsequently the use cases are prioritized (e.g., based on frequency). For the walk-up user of a copier the following use cases may have been identified:

Walk up user	**Priority**
Copy A4-A4	High
Copy A4-A3	Medium
Print email	Low
....	

Identify scenarios for every use case
Scenarios are identified for every use case. A use case usually has a basic, most likely, scenario. This basic scenario is sometimes referred to as the “happy day” scenario. This is how the use case is most often carried out in real-life. The use case will typically also contain alternative branches: deviation scenarios and failure scenarios. The deviation scenarios are defined as scenarios that are successful in terms of user results, but they are achieved in a different way, i.e. they have a different flow of actions than the basic scenario. Failure scenarios are defined as scenarios in which the user will encounter some problems, e.g., error messages. However, the user should in principle be able to resolve these problems by himself. For the walk-up user and related use case “Copy A4-A4” the following scenarios may have been identified:

Use case 'Copy A4-A4'	
Basic	single side – single side
Deviations	single side – double side
	Copy of a set
	Copy by using the ADF
	....
Failures	no paper in paper tray
	paper jam
	out of staples
	....

Select use cases for testing

The final step in the design process is to select the use cases which will be elaborated into full test cases. Business risk or impact is most often the important selection criteria. The selected use cases are elaborated into full test procedures based on the following template:

- name and summary
- user group
- business importance (priority)
- pre-conditions, e.g., other use cases
- basic flow
- deviation flows
- alternative flows
- influencing other use cases
- post-conditions
- additional attention points, e.g., understandability of messages, performance issues.

Hereafter you will find example of a use case associated with an ATM described as a test procedure.

Scenario	Use of ATM		
System	XYZ Bank ATM		
Objective	The goal of this use case is to test the withdrawal of cash from the ATM		
Description	**Context:** This is core functionality, so this is an important test of a frequent usage function		
	Actors: Any user may use this function	*Checked*	*Comments*
Preconditions	The ATM is on and functional. The user has a current debit card.		
Basic flow	ATM start screen is showing 1. the user inserts card 2. the user types in PIN 3. the user selects "cash with receipt" 4. the user selects £50 5. the user selects "no further service"		
Expected results	The ATM returns the card, delivers £50 and a receipt, clears the screen and displays the start screen		
Alternative flows	The user wants to have a balance printed not displayed. ATM start screen is showing 1. the user inserts card 2. the user types in PIN 3. the user selects "print balance" 4. the user selects "no further service"		
Expected results	The ATM returns the card and balance slip, clears the screen and returns to start screen		
Failure conditions	On screen instructions not clearly displayed On screen instructions not simple to follow Card not returned (see functional tests)		
Additional attention points	Note response times		

Figure 5: Example Use case Test Procedure

3.Experience-Based Techniques

3.1 Introduction

In experience-based techniques, people's knowledge, skills and background are a prime contributor to the tests being executed. The experience of both technical and business people is important, as they bring different perspectives to the testing process. Due to previous

experiences with similar systems, they may have insights into what is likely go wrong, which is of course very useful for testing. All experience-based techniques have the common characteristic that they are based on people's knowledge and experience, both of the system itself (including the knowledge of users and stakeholders) and of likely defects. Test cases are typically derived in a less systematic way, but will nevertheless be very effective.

Experience-based technique: Procedure to derive and/or select test cases based on the tester's experience, knowledge and intuition.

The experience-based tests may be effective at finding defects, but not as appropriate as other techniques to achieve specific test coverage levels or produce reusable test procedures. In cases where system documentation is poor, testing time is severely restricted or the test team has strong expertise in the system to be tested, experience-based testing is often a good alternative to using specification-based techniques. Experience-based testing may be inappropriate in systems requiring detailed test documentation, high-levels of repeatability or an ability to precisely assess test coverage. Although some ideas on coverage are presented for the experience-based techniques discussed hereafter, experience-based techniques do not have formal coverage criteria.

When using more dynamic approaches, e.g., Agile, testers normally use experience-based tests. In such situations testing is typically required to be more reactive to events, and this is less suitable with pre-planned testing approaches. In addition test execution and product evaluation are often concurrent tasks. Beware that not all experience-based tests are entirely dynamic, i.e., the tests are not always created at the same time as the tester executes the test. Sometimes, while practicing experience-based testing, testers may even implicitly use a specification-based techniques to quickly sketch some interesting test cases.

3.2 Error Guessing

Error guessing: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or system under test as a result of errors made, and to design tests specifically to expose them.

Error guessing is unstructured testing. Its value lies in the unexpected; test are executed that would otherwise not have been considered. It is a valuable add-on to specification-based test design techniques. The experience of the tester, e.g., domain and product expertise, plays an important role here. The tester has complete freedom to think of test cases on the spot and try them out. The essence of error guessing is that the tester tries to find those test cases that often are successful at finding defects – trying out these cases is the way to test them.

In principle, the only starting condition for error guessing is that the tester has an understanding of the system under test. In addition, a certain degree a stability of the system to be tested is needed. Error guessing is eminently suitable at the final stages of the test process. However, it

should (in most cases) not be used as the only technique. The benefit of scripted testing is the relative completeness that is achieved because of the nature of these techniques.

Scripted testing: Test execution carried out by following a previously documented sequence of tests.

Applicability

The error guessing technique can be used at all test levels and can focus on all quality characteristics. In practice, it is most often used at integration and system testing. It can be applied to test in high risk areas to compliment specification-based techniques. As such it also becomes a quality check on the effectiveness of the test process. Error guessing can also be applied in low risk areas where spending the resources on scripted testing is not feasible, and error guessing provides a solution to the test approach since it only uses limited resources. Using checklist (see next paragraph) with error guessing will be helpful in guiding testing and increase its effectiveness.

In addition to being used as a test technique, error guessing is also useful during product risk analysis to identify potential failure modes. It can for instance be applied to sample test initial increments and based on the results drive product risk-analysis and further testing. Error guessing can also be used effectively to test new software for common mistakes and defects before starting more rigorous and scripted testing.

Error guessing is often also fun and provides a good variation to scripted testing. Some teams use error guessing as a way to close of the week; the week ends with a two hour error guessing session.

Limitations/Difficulties

There are some clear disadvantages to error guessing such as no clear objectives, unknown coverage, no re-usable testware and as a consequence error guessing is hard to manage. Since there will be little test documentation another drawback with error guessing is the repeatability of tests executed and thus the reproducibility of the defects being found. It is possible that during error guessing defects are found whereby the test actions executed are not (exactly) known anymore afterwards. Having logging tools in place, e.g., a record & playback tool, can partly solve this issue.

As already stated, coverage is difficult to assess and varies widely with the capability and experience of the test analyst. Sometimes a defect taxonomy checklist is explicitly used with error guessing or a specific assignment (e.g., to focus on a certain test item) is provided to the tester. This will provide some level of coverage on types of defects or system areas. In the case of using a defect taxonomy checklist, error guessing is best used by an experienced tester who is familiar with the types of defects that are being targeted at.

Note that since some testers do the wildest things during error guessing, e.g., extreme large values, and as a result the test database is often not in a state to continue with regular scripted

testing thereafter. It is therefore of utmost importance that when error guessing is practiced, an efficient and effective back-up and restore procedure is available for the test database.

Coverage
When a taxonomy (or assignment) is used, coverage is determined by the appropriate types of defects listed. Without a taxonomy (or assignment), coverage is limited by the experience and knowledge of the tester and the time available and large unknown to (test) management.

Types of Defects
Since error guessing is totally open to the experience of the tester, all types of defect can be found as part of error guessing. However, typical defects are usually those defined in the particular taxonomy being targeted or "guessed" by the test analyst, that may not have been found using specification-based testing.

Procedure

Identifying weaknesses
Preparation for error guessing typically includes an activity for identifying weaknesses. During this activity it should also be determined whether it is necessary to construct a certain starting situation (e.g., an initial data set) for the error guessing test. The weaknesses often stem from errors in the mental processes of others and issues that have been forgotten. These aspects constitute the basis for the error guessing test that is to be executed. Examples include:
- Error handling, e.g., error upon error, or interruption of a process at an unexpected moment
- Illegal values, e.g., negative numbers, null values, values that too large, strings that are too long, or empty records
- Parts of the system that were subject to many change requests during the project
- Security issues
- Components claiming too many resources.

A plan is made for error guessing on the basis of identified weaknesses. A tester is given the assignment to perform error guessing on a specific part / aspect of the system or type(s) of defect during a certain period of time.

Test execution
The execution of the error guessing test depends on what needs to be tested; an interface, the processing of a certain function, a screen lay-out, etc. It may be necessary to construct an initial starting situation as indicated above. One also needs to decide whether to document the test cases and the test execution flow. If the test are undocumented and will not be added to the test set, then the initial date set may also need to be removed after execution of the error guessing test to prevent noise at other scripted tests. However, if defects are found, the corresponding test actions or test cases should be documented. This can be done in the format of a test procedure, but can also be done as part the defect report. It is also important to check the set of test cases that is already available. It is, after all, possible that the defect would have been found

on the basis of test cases already available, but that the corresponding test procedure is to be executed at a later stage.

If a tester is using error guessing on a test item that is the responsibility of another tester (a recommended practice), then any defects found should at least be shared with the other tester who will carry out a scripted test on that specific test item. The tester will also assess the extent to which existing test documentation needs to be adapted, e.g., test cases added, based on the defect(s) found.

3.3 Checklist-based testing

Checklist-based testing: An experience-based test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified.

Checklist-based testing is an experience based test technique dependent on a preplanned "to-do" list of tests composed by a tester based on previous testing experience. This list acts as an authentic guide to direct the testing process.

When applying the checklist-based test technique, the experienced test analyst uses a high-level, generalized list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified. The "to do" list then forms the checklist which enumerates the actions to be performed during the course of a test. Upon completion, the listed activities are ticked off one by one. These checklists are built based on a set of standards, experience, and other considerations. An user interface standards checklist employed as the basis for testing an application is an example of a checklist-based test.

As another example, consider a checklist for testing the image uploading functionality:

- Check for image uploading path
- Check for image uploading
- Check for image uploading with different extensions such as JPEG or BMP
- Check for uploading images with same names
- Check if the image is getting uploaded within the maximum allowable size and if not, it is necessary to verify that an error message is appearing
- Check if the bar showing the progress of image uploading is appearing or not
- Check the functionality of the cancel button at the time of image upload
- Check for multiple image uploading
- Check for good quality of uploaded image
- Check if the user is able to save the image post the uploading process.

Applicability

Checklist-based testing is used most effectively in projects with an experienced test team that is familiar with the system under test and/or familiar with the area covered by the checklist (e.g., to successfully apply an user interface checklist, the test analyst has at least to be familiar with

user interface testing but not necessarily the specific system under test). Because checklists are high-level and tend to lack the detailed steps often found in test cases and test procedures, the knowledge of the tester is used to fill in the gaps. By removing the detailed steps, checklists are low maintenance and can be applied to multiple similar releases and even systems. Checklists can be used for any level of testing. Checklists are also often used for regression testing and smoke testing.

Checklists, like the one in the example above, are like time tested guidelines for ensuring a thorough coverage which can take out most of the defects in the software product. Hence reusing the test cases based on these guidelines can help in cutting down costs incurred in missing out on important testing aspects. Checklist-based testing is also a welcome tool for companies where meeting deadlines becomes difficult and the testers are liable to miss out on executing some critical tests.

Sometimes innovative tests may be borne out of a project which can add value to the testing to next releases or other projects as well. The testers working on other projects maybe ignorant of this new set of interesting tests. This aspect is taken care of when any innovative set of tests found out in one project become part of the test checklist to be re-executed in other projects.

Checklists can also help in better integration of new testing staffing into the organization as they can use readymade guidelines to start testing on a project with confidence.

Difficulties

One of the problems when using checklist-based testing is that differences in interpretation of checklist items by testers can lead to different approaches to accomplish the tasks as mentioned in the checklist. Especially the high-level nature of the checklists can affect the reproducibility of defects. It is possible that several testers will interpret the checklists differently and will follow different approaches to fulfill the checklist items. This may cause different results, even though the same checklist is used. This can result in wider coverage but reproducibility is often sacrificed. This is especially true with complex or advanced level of tests whom are probably not suitable for a technique like checklist-based testing. The reproducibility of the defects will, as stated, typically be negatively affected.

Also today's software products are regularly in need of improvements through different upgrades. Therefore checklists have also to be upgraded and maintained on a regular basis to comprehensively cover testing of all the new aspects related to the product of similar functionalities. Checklists can be derived from more detailed test cases or lists and tend to grow over time. Maintenance is required to ensure that the checklists are covering the important aspects of the system being tested.

Finally, checklists come handy as an arrangement for some additional or last grasp testing. Its utility is questionable for holistic testing over the entirety of the software development life cycle with most software products, especially critical ones.

Coverage
The coverage is as good as the checklist but, because of the high-level nature of the checklist, the results will vary based on the test analyst who executes the checklist. Checklists may therefore also result in over-confidence regarding the level of coverage that is achieved since the actual testing depends on the tester's judgment.

Typical defects
Since checklist-based testing is totally dependent on the nature of the checklist, all types of defect can be found as part of checklist-based testing. However, typical defects found with this technique include failures resulting from varying the data used with the checklist, the sequence of steps or the general workflow during testing. Using checklists can help keep testing fresh as new combinations of data and processes are allowed during testing.

3.4 Exploratory testing

Exploratory testing: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

Exploratory testing is a powerful approach. In some situations, it can be orders of magnitude more productive than scripted testing. All testers practice some form of exploratory testing, unless they simply do not create tests at all. Among the hardest things to explain is something that everyone already knows. We all know how to listen, how to read, how to think, and how to tell anecdotes about the events in our lives. As adults, we do these things every day. Yet the level of any of these skills, possessed by the average person, may not be adequate for certain special situations. Psychotherapists must be *expert* listeners and lawyers *expert* readers; research scientists must scour their thinking for errors and journalists report stories that transcend parlour anecdote. So it is with exploratory testing (ET): the simultaneous design and execution of tests. This is a simple concept. But the fact that it can be described in a sentence can make it seem like something not worth describing. Its highly situational structure can make it seem, to the casual observer, that it has no structure at all. However, exploratory testing can be as disciplined as any other intellectual activity. Many organizations practice a formalised process of exploratory testing, and session-based test management is a method specifically designed to make exploratory testing auditable and measurable on a large scale.

What makes exploratory testing interesting is that it when a tester has the skills to *listen*, *read, think* and *report,* rigorously and effectively, without the use of pre-scripted tests, the exploratory approach to testing can be many times as productive as the scripted variety (predefined test procedures, whether manual or automated). And when properly supervised and chartered, even testers without special skills can produce useful results that would not have been anticipated by a script. Of course, tests may be worth reducing to a repeatable scripted form for a variety of good reasons. There may be special accountability requirements, perhaps, or maybe there are certain tests that must be executed in just the same way, every time, in order to serve as a kind of benchmark. In some contexts, test objectives are achieved better through a more scripted

approach; in other contexts, test objectives will benefit more from the ability to create and improve tests as tests are being executed. Most situations benefit from a mix of scripted and exploratory approaches.

Cem Kaner who has introduced the term Exploratory Testing (ET) to the testing industry describes it as any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests, and where the following conditions apply:

- The tester is not required to use or follow any particular test materials or procedures
- the tester is not required to produce materials or procedures that enable test re-use by another tester or management review of the details of the work done [Kaner].

Exploratory testing is an interactive test process. It is a free-form process in some ways and has much in common with informal experience-based test techniques such as error guessing. However, unlike traditional informal testing, this procedure consists of specific tasks, objectives, and deliverables that make it a systematic process.

The puzzle analogy

Have you ever solved a jigsaw puzzle? If so, you have practised exploratory testing. Consider what happens in the process. You pick up a piece and scan the jumble of unconnected pieces for one that goes with it. Each glance at a new piece is a test case ("Does this piece connect to that piece? No? How about if it is turned around? Well, it almost fits but now the picture doesn't match..."). You may choose to perform your jigsaw testing process more rigorously, perhaps by concentrating on border pieces first, or on certain shapes, or on some attribute of the picture on the cover of the box. Still, can you imagine what it would be like to design and document all your jigsaw "test cases" before you began to assemble the puzzle, or before you knew anything about the kind of picture formed by the puzzle? When we solve a jigsaw puzzle, we change how we work as we learn about the puzzle and see the picture form. This is a general lesson about puzzles: *the puzzle changes the puzzling.* The specifics of the puzzle, as they emerge through the process of solving that puzzle, affect our tactics for solving it. This truth is at the heart of any exploratory testing,

Exploratory testers ask, "What's the most powerful test I can perform, right now?" The power of a test is how much useful information it reveals about the product. The product and many factors around it change continuously throughout the course of the project, or even from moment to moment during a test session. The power of exploratory tests can be optimised throughout the test process, whereas test procedures, because they don't change, tend to become less powerful over time. They fade for many reasons, but the major reason is that once a scripted test has been executed for the first time and not found a defect, the chance that you will find a defect on its second execution is, in most circumstances, substantially lower than if you ran a new test instead.

Applicability

In general, ET is called for in any situation where it is not obvious what the next test should be, or when you want to go beyond the obvious tests. More specifically, pure exploratory testing fits in any of the following situations:

- You need to provide rapid feedback on a new product or feature
- You need to learn the product quickly
- You have already tested using scripts and seek to diversify the testing
- You want to find the single most important bug in the shortest time
- You want to check the work of another tester by doing a brief independent investigation
- You want to investigate and isolate a particular defect
- You want to investigate the status of a particular risk, in order to evaluate the need for scripted tests in that area.

Limitation/Difficulties

ET is powerful because of how the information flows backward from executing testing to re-designing them. Whenever that feedback loop is weak, or when the loop is particularly long, slow, or expensive, ET loses that power. In these situations, pre-scripted tests may be the preferred option. Another place to use scripted tests is in any part of our testing that will be subjected to extreme retrospective scrutiny. But do not just settle for scripted testing just because they please the auditors. Consider using a combined exploratory and scripted approach, and get the best of both worlds.

In addition to the ones already mentioned, there are other situations where applying ET may not be the preferred option:

- Testing detailed and complex calculation; in such circumstances test cases with expected results are required and may take some time to develop
- Testing most critical features; in some safety-critical areas assurance is needed on the coverage achieved
- Testing reliability / performance; these type of testing often need a lot of preparation, e.g., designing operational profiles, defining test cases and creating test data
- Testware is important; in ET less re-usable is build, if this is important, e.g., as a basis for test automation, more scripted testing will be needed
- Test automation, e.g., as a basis regression testing, is an important objective
- Testers are less skilled; of course ET builds on the skills of the testers as any of the experienced-based techniques.

As with error guessing, since with ET there will be less test documentation compared to scripted testing, repeatability of tests executed and thus the reproducibility of the defects being found could be a problem. However, since some test documentation is produced with ET, this usually is less of a problem than with error guessing.

Finally with ET, test design as an early static test technique debugging the requirements is lost. The defects found in the test basis while doing test design are often by themselves already a justification for a structured test design phase using specification-based techniques.

Coverage
Charters are created to specify objectives and tasks. Exploratory sessions are then planned based on the charters to achieve those objectives. The charter may also identify where to focus the testing effort, what is in and out of scope of the test session, Sometimes the scope is defined in terms of requirements, allowing for traceability from test charters to requirements and tracking requirements coverage with ET. A session may be also used to focus on particular defect types or other potentially problematic areas that can be addressed without the formality of scripted testing.

Type of defects
In principle all types of defects can be found with exploratory testing. However, typical defects found with exploratory testing are scenario-based issues that were missed during scripted functional testing, issues that fall between functional boundaries, and workflow related issues. Security issues are also sometimes uncovered during exploratory testing.

Practicing exploratory testing
The external structure is easy enough to describe. Over a period of time, a tester interacts with a product to fulfil a test mission, and reports results. The basic external elements of ET thus are: time, tester, product, mission, and reporting. The mission is fulfilled through a continuous cycle focusing on the mission, conceiving questions about the product that if answered would also allow the tester to satisfy the mission, designing tests to answer those questions, and executing tests to get the answers. Often the tests don't fully answer the questions, so tests are adjusted to keep trying (in other words, exploration). The exploratory tester is at any time ready to report on status and results.

Test charters
An exploratory test session often begins with a charter, which states the mission and perhaps some of the tactics to be used. The charter may be chosen by the tester himself, or assigned by the test lead or test manager. Most often charters are documented. In some organizations, test cases and procedures are documented on a high-level only, as a consequence they essentially serve as charters for exploratory testing.

Test charter: A statement of test objectives, and possibly test ideas about how to test. Test charters are used in exploratory testing.

Test charters can take a format of only one sentence, such as the examples hereafter. However often they are a one page summary describing the testing that needs to be done on an item or feature. These one page test charters commonly describe the following topics under a number of headings (see figure 6 for an example):

- *What* – What needs to be tested and is within the scope of the test charter, e.g., bullet list and numbering in Word. Both through menu and right mouse click
- *What not* – What is explicitly not within scope, e.g., non-functionals since they are already tested by a dedicated other test

- *Why* – What are the questions that need to be answered, e.g., to verify that the bullet lists are consistent and to check the correct numbering. The "Why" can also be used to reference specific requirement that need to be covered.
- *How* – This is usually the result of a short brainstorm and defined in free format test ideas that can be used later, e.g., use a word document, use .dot file, new/existing doc. using right mouse button, menu bar / imported doc.
- *Expected Problems* –This resemble the "How" but changes the mindset of the brainstorm stakeholders, what is likely to fail?, where can one find defects?, etc.
- *Reference* – A reference to relevant documentation, e.g., diagrams or models

Some examples of one sentence test charters mentioned earlier:

- Define workflows through XYZ and try each one. The flows should represent realistic scenarios of use, and they should collectively encompass each primary function of the product
- Test all fields that allow data entry (you know the drill: function, stress, and limits, please)
- Run XYZ with browser ABC and report any defects.
- Check the user-interface against Windows interface style guide.

Test Charter Search Engine

- *What*: Search Engine to look up other sources of information in the company (list of sample information sources: A, B, C etc.). Standard and Advanced search must be tested.
- *Why*: To test the search feature with single information sources and multiple sources, to see that the retrieved information is presented consistently and according to standard, and that the retrieved information is correct.
- *How*: Search from the WEB portal as well as continue searching in the result list (advanced search – refining the search).
- *Expected problems*: Some information not found, Not possible to navigate to information found (jumping between information sources), Information found not presented consistently independent of sources.
- *References*: Requirement specification section x.11.

Figure 6: Example Test Charter

Note that typically test charters are ambiguous. They are intended to communicate the mission of a test session to testers who have already been trained in the expectations, vocabulary, techniques and tools used by the organization. Remember, in ET we make maximum use of skill, rather than attempting to represent every action in written form.

In pure exploratory testing, the only result that comes from an ET session is a set of defect reports. However, ET sessions may also results in a set of written notes that are reviewed by the test manager. It may also result in updated test materials or new test data. If you think about it, most written test procedures were probably created through a process of some sort of exploratory testing.

The outer trappings, inputs and outputs to exploratory testing are worth looking at, but it is the inner structure of ET that matters most - the part that occurs inside the mind of the tester. That's where ET succeeds or fails; where the excellent explorer is distinguished from the amateur. This is a complex subject, but here are some of the basics:

- *Test Design:* An exploratory tester is first and foremost a test designer. Anyone can design a test accidentally; the excellent exploratory tester is able to craft tests that systematically explore the product. That requires skills such as the ability to analyse a product, evaluate risk, use tools, and think critically, among others.
- *Careful Observation:* Excellent exploratory testers are more careful observers than novices, or for that matter, experienced scripted testers. The scripted tester need only observe what the script tells him to observe. The exploratory tester must watch for *anything* unusual or mysterious. Exploratory testers must also be careful to distinguish observation from inference, even under pressure, lest they allow preconceived assumptions to blind them to important tests or product behaviour.
- *Critical Thinking:* Excellent exploratory testers are able to review and explain their logic, looking for errors in their own thinking. This is especially important when reporting the status of a session of exploratory tests, or investigating a defect.
- *Diverse Ideas:* Excellent exploratory testers produce more and better ideas than novices. They may make use of heuristics to accomplish this. Heuristics are mental devices such as guidelines, generic checklists, mnemonics, or rules of thumb. The Satisfice Heuristic Test Strategy Model (http://www.satisfice.com) is an example of a set of heuristics for rapid generation of diverse ideas. James Whittaker's set of attacks is another [Whittaker]. The diversity of tester temperaments and backgrounds on a team can also be harnessed by savvy exploratory testers through the process of group brainstorming to produce better test ideas.
- *Rich Resources:* Excellent exploratory testers build a deep inventory of tools, information sources, test data, and friends to draw upon. While testing, they remain alert for opportunities to apply those resources to the testing at hand.

Procedure

The fact that exploratory testing is hard to manage is often cited as a big disadvantage. To solve this issue an approach to ET has been developed called: session-based test management. The approach has been developed based on discussions between ET experts, but especially taking into account contributions from ET practitioners. A systematic process has been defined that shows ET can be applied in a structured way [Veenendaal]. This process is reflected in figure 7 and briefly explained hereafter.

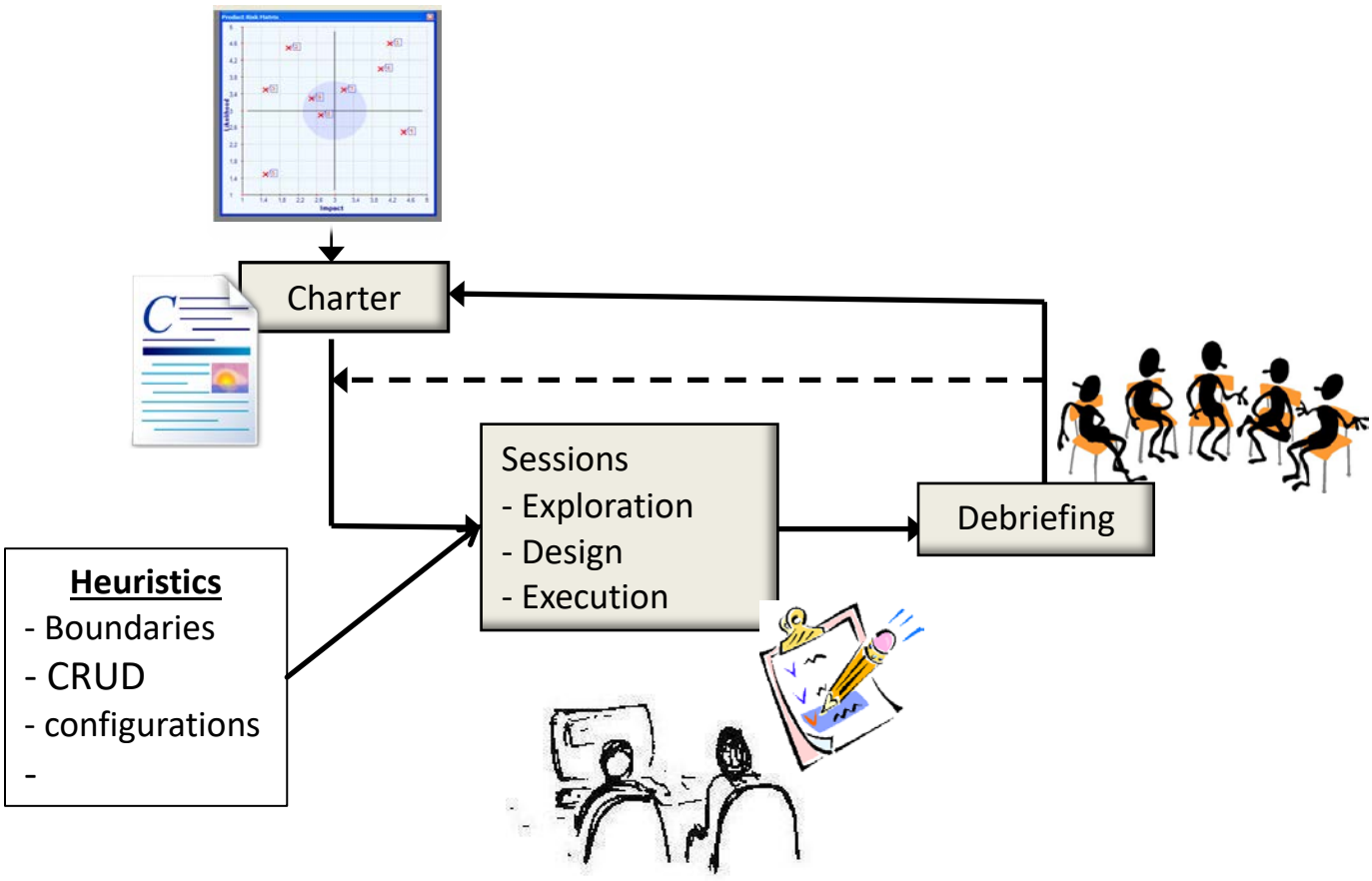


Figure 7: Process flow Exploratory Testing

Since ET is only one of the many ways to test a system, also a project applying ET as a test technique will start by executing a product risk assessment. The results will be used to make clear choices regarding test coverage and test depth required and to decide on which parts of the system under test ET will be applied. Subsequently the preparation is started by establishing test charters for specific test items. Test charters will often be produced using brainstorm sessions with various stakeholders to document the most important attention points and test ideas for the exploratory tests to be executed (refer to figure 6 for an example of a test charter). A test charter can even be perceived as a high level test design document.

On the basis of test charters, test session are performed during which the tester becomes acquainted with the new product and at the same time designs and executes test cases. Test charters are used to guide the test sessions. A session is an uninterrupted period of time between two hours and one day. During the test session, working in pairs, the test actions are documented (along defined guidelines) in the form of notes. This makes the tests reusable and defect reproduceable to a certain degree. Re-testing a test item can then take place partly based on the notes of a previous session if required. The notes can also be used as a basis for developing more detailed testware. The notes will commonly contain details of the tests executed, a list of defects found and other observations and a general conclusion. Test sessions are typically performed by

two person that stimulate each other in thinking of interesting test cases. When working in pairs one tester will execute the tests, while the other tester makes notes. Whilst the sessions takes place guided by a test charter, also so-called heuristics and checklist of common defects are used as a supporting tool.

When the test session has ended a debriefing takes place involving fellow testers and other interested parties. Experiences with the product, e.g., new product risks identified, are discussed, and test ideas and defects found are exchanged. In the context of exchanging ideas the question "What is the most important defect you have encountered today?" may be asked. At the end of the debriefing session the team decides what are the important items or features that should be tested next. A new test session will start based on a test charter and the next exploratory test session begins. Note how much a debriefing session resembles a daily stand-up daily meeting commonly used in Agile projects, again showing that ET has a perfect fit within these projects.

References

[Copeland] L. Copeland (2003), *A Practitioner's Guide to Software Test Design*, Artech House

[Cohn] M. Cohn (2004), *User Stories Applied: For Agile For Agile Software Development,* Pearson Education

[Foundations] R. Black, E. van Veenendaal and D. Graham (2012), *Foundations of Software Testing – ISTQB Certification 3rd edition,* Cengage Learning,

[Grochtmann] M. Grotchmann and K. Grimm (1993), Classification Trees for Partition Testing, in: *Software Testing, Verification and Reliability,* Vol. 3, no. 2, June 1993

[ISTQB] ISTQB (2016), *Worldwide Software Testing Practices Report 2015-2016*, International Software Testing Qualifications Board

[Kaner] C. Kaner, J. Falk and H. Nguyen (1999), *Testing Computer Software – 2nd edition*, John Wiley and Sons

[Kit] E. Kit (1995), *Software Testing in the Real World,* Addison-Wesley, London

[Mors] N.P.M. Mors (1993), *Decision Tables* (in Dutch), Lansa Publishing BV

[Myers] G.J. Myers (1979), *The Art of Software Testing*, Wiley-Interscience, New York

[Veenendaal] E. van Veenendaal (2004), Exploratory Testing: Meaningful or Meaningless (in Dutch), in: *Software Release Magazine*, Year 9, November 2004, No. 7

[Whittaker] J. A. Whittaker (2002), *How to Break Software: A Practical Guide to Testing,* Pearson Education

The Complete Book “The Testing Practitioner”

The book **The Testing Practitioner** has been written as a background book to the ISTQB Advanced syllabi. The Testing Practitioner provides a comprehensive description of the state-of-art in software testing and in addition addresses a number of challenges and topics for the test practitioner. Offering insights from leading experts in testing, each chapter in this book has been extensively reviewed for technical content, assuring that it is accurate and time-worthy. The seven section of The Testing Practitioner cover materials found essential for test engineers and test managers working in real-world businesses. Following the syllabi this book deals with test principles, test process, test management, risk management, inspections and reviews, test techniques (both functional and non-functional), test process improvement, tools and people issues. Some of the specific topics included are test planning, risk based testing, completion criteria, estimation, static analysis, exploratory testing, performance testing, usability testing and data driven testing.

This book is intended to meet the practical needs of both test engineers and test managers, especially those preparing for the ISTQB Advanced Test Analyst or Test Manager exam. A great reference for the new and experienced test practitioner applying test principles and structured testing to software development. This book may not contain all the answers you need, but is will surely set you of in the right direction.

Order on-line via Amazon

The Author

Drs. Erik van Veenendaal, CISA (www.erikvanveenendaal.nl), is a leading international consultant and trainer, and a recognized expert in the area of software testing and requirement engineering. Erik is the (co-)author of numerous papers and a number of books on software quality and testing. He is a regular speaker, e.g., running a tutorial on test design techniques, at both national and international testing conferences and a leading international trainer in the field of software testing.

Since its foundation in 2002, Erik has been strongly involved in the International Software Testing Qualifications Board (ISTQB). From 2005 to 2009, he was the vice president of the ISTQB organization; he currently is the president for the Curaçao Testing Qualifications Board (CTQB).

Erik one of the core developers of the TMap test methodology and the TMMi test improvement model, and currently the CEO of the TMMi Foundation. For his major contribution to the field of testing, Erik received the European Testing Excellence Award (2007) and the ISTQB International Testing Excellence Award (2015). You can follow Erik on twitter via @ErikvVeenendaal.

Enjoyed this eBook and want to read more?
Check out our extensive library on Huddle.

EuroSTAR Software Testing | Huddle

www.eurostarsoftwaretesting.com