CHAPTER

4

Test Techniques

Contents

Test case design techniques are the heart of testing. There are many advantages of using techniques to design test cases. They support systematic and meticulous work and make the testing specification effective and efficient; they are also extremely good for finding possible faults. Techniques are the synthesis of "best practice"—not necessarily scientifically based, but based on many testers' experiences.

Other advantages are that the design of the test cases may be repeated by others, and that it is possible to explain how test cases have been designed using techniques. This makes the test cases much more trustworthy than test cases just "picked out of the air." The test case design techniques are based on models of the system, typically in the form of requirements or design. We can therefore calculate the coverage we are obtaining for the various test design techniques.

Coverage is one of the most important ways to express what is required from our testing activities. It is worth noticing that coverage is always expressed in terms related to a specific test design technique. Having achieved a high coverage using one technique only says something about the testing with that technique, not the complete testing possible for a given object.

Test case design techniques have a few pitfalls that we need to be aware of. Even if we could obtain 100% coverage of what we set out to cover (be it requirements, or statements, or paths), faults could remain after testing simply because the code does not properly reflect what the users and customers want. Validation of the requirements before we start the dynamic testing can mitigate this risk.

There is also a pitfall in relation to value sensitivity. Even if we use an input value that gives us the coverage we want it may be a value for which incidental correctness applies. An example of this is the fact that:

2 + 2 equals 2 * 2; but 3 + 3 does not equal 3 * 3!

4.1 Specification-Based Techniques

The specification-based case design techniques are used to design test cases based on an analysis of the description of the product without reference to its internal workings. These techniques are also known as black-box tests.

These techniques focus on the functionality. They are dependent on descriptions of the expectations towards the product or system. These should be in the form of requirements specifications, but may also be in the form of, for example, user manuals and/or process descriptions. If we are lucky we get the requirements expressed in ways corresponding directly to these techniques; if not we'll have to help analysts do that during requirements documentation or do it ourselves during test design.

These test case design techniques can be used in all stages and levels of testing. The techniques can be used as a starting point in low-level tests, component testing and integration testing, where test cases can be designed based on the design and/or the requirements. These test cases can be supplied with structural or white-box tests to obtain adequate coverage.

The techniques are also very useful in high-level tests like acceptance testing and system testing, where the test cases are designed from the requirements.

The specification-based techniques have associated coverage measures, and the application of these techniques refines the coverage from requirements coverage to specific coverage items for the techniques.

The functional test case design techniques covered here are:

- Equivalence partitioning and boundary value analysis
- (Domain analysis—not part of the ISTQB syllabus)
- Decision tables
- Cause-effect graph
- State transition testing
- Classification tree method
- Pairwise testing
- Use case testing
- (Syntax testing—not part of the ISTQB syllabus)

4.1.1 Equivalence Partitioning and Boundary Value Analysis

Designing test cases is about finding the input to cover something we want to test. If we consider the number of different inputs that we can give to a product we can have anything from very few to a huge amount of possibilities.

A product may have only one button and it can be either on or off = 2 possibilities.

A field must be filled in with the name of a valid postal district = thousands of possibilities.

It can be very difficult to figure out which input to choose for our test cases. The equivalence partitioning test technique can help us handle situations with many input possibilities.

4.1.1.1 Equivalence Partitioning

The basic idea is that we partition the input or output domain into equivalence classes.

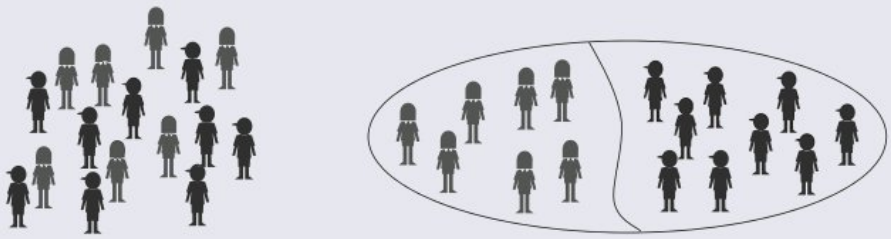
A class is a portion of the domain. The domain is said to be partitioned into classes if all members of the domain belong to exactly one class—no member belongs to more than one class and no member falls outside the classes.

The term equivalence refers to the assumption that all the members in a class behave in the same way. In this context the assumption is based on the requirements or other specifications of the product's expected behavior.

The reason for the equivalence partitioning is that all members in an equivalence class will either fail or pass the same test. One member represents all! If we select one member of a class and use that for our test case, we can assume that we have tested all the members.

Choosing test cases based on equivalence partitioning insures representative test cases.

If we take a class of pupils and the requirement says that all the girls should have an e-mail every Thurdays reminding them to bring their swimsuits, we can partition the class into a partition of girls and a partition of boys and use one representative from each class in our test cases.



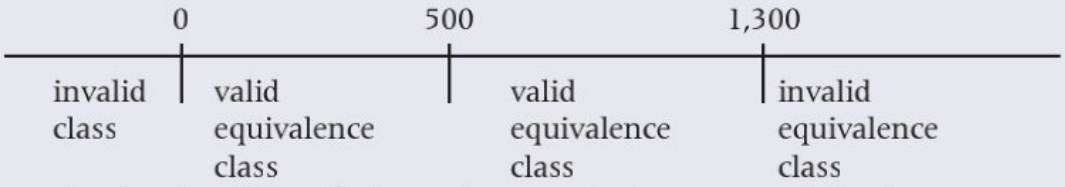
When we partition a domain into equivalence classes, we will usually get both valid and invalid classes. The invalid classes contain the members that the product should reject, or in other words members for which the product's behavior is unspecified. Test cases should be designed for both the valid and the invalid classes, though sometimes it is not possible to execute test cases based on the invalid equivalence classes.

The most common types of equivalence class partitions are intervals and sets of possibilities (unordered list or ordered lists).

Intervals may be illustrated by a requirement stating:

Income in €	Tax percentage
Up to and including 500	0
More than 500, but less than 1,300	30
1,300 or more, but less than 5,000	40

If this is all we know, we have:



Another invalid equivalence class may be inputs containing letters.

To illustrate a set of possibilities we may use the unordered list of hair colors: (blond, brown, black, red, gray). Perhaps the product can suggest an appropriate dye for these colors of hair, but none other. The valid equivalence class is the list of values; all other values belong to the invalid class, assuming we don't know how the product is going to react, if we enter one such value.

It is possible to measure the coverage of equivalence partitions. The equivalence partition coverage is measured as the percentage of equivalence partitions that have been exercised by a test.

To exercise an equivalence class we need to pick one value in the equivalence class and make a test case for this. It is quite usual to pick a value near the middle of the equivalence class, if possible, but any value will do.

Test cases for the tax percentage could be based on the input values: –5; 234; 810; and 2,207.

For the hair colors we could choose: black and green, as a valid and on invalid input value, respectively.

4.1.1.2 Boundary Value Analysis

A boundary value is the value on a boundary of an equivalence class. Boundary value analysis is hence strongly related to equivalence class partitioning. Equivalence classes of intervals have boundaries, but those of lists do not. Boundary value analysis is the process of identifying the boundary values.

The boundary values require extra attention because defects are often found on or immediately around these. Choosing test cases based on boundary value analysis insures that the test cases are effective.

For interval classes with precise boundaries, it is not difficult to identify the boundary values.

The interval given as: 0 <= income <= 500, is one equivalence class with the two boundaries 0 and 500.

If a class has an imprecise boundary (> or <) the boundary value is one increment inside the imprecise boundary.

If the above interval had been specified as: 0 <= income < 500, and the smallest increment is given as 1, we would have an equivalence class with the two boundaries 0 and 499.

The smallest increment should always be specified; otherwise we must ask or guess based on common or otherwise available information.

It is often not specified what the increment is when we are dealing with money. A reasonable guess is that if we operate in euros (€), then the smallest increment is € 0.01 or 1 cent. But beware! Tax people usually use € 1 as the smallest increment.

Sometimes we'll experience equivalence classes with open boundaries—classes where a boundary is not specified. This makes it difficult to identify a boundary value to test. In these cases we must first of all try to get the specification changed. If that is not possible we can look for information in other requirements, look for indirect or hidden boundaries, or omit the testing of the nonexisting boundary value.

Open boundaries may be seen in connection with people's incomes. If the above interval had to do with incomes we may argue that the lowest income is € 0. But what is the real upper boundary? There is no obvious upper boundary for people's income.

Because we can have both valid and invalid equivalence classes we can also have both valid and invalid boundary values, and both should be tested. Sometimes, however, the invalid ones can not be tested.

When we select boundary values for testing, we must select the boundary value and at least one value one unit inside the boundary in the equivalence class. This means that for each boundary we test two values.

In traditional testing it was also recommended to choose a value one unit

outside the equivalence class, hence testing three values for each boundary. Such a value is in fact a value on the border of the adjacent equivalence class, and some duplication could occur. But we can still choose to select three values; the choice between two or three values should be governed by a risk evaluation.

It is possible to measure the coverage of boundary values. The boundary value coverage is measured as the percentage of boundary values that have been exercised by a test.

Equivalence Partitioning and Boundary Value Analysis Test Design Template

The design of the test conditions based on equivalence class partitioning and boundary value analysis can be captured in a table such as the following one.

Table designed by Carsten Jørgensen

Test design item number:		**Traces:**	
Based on: Input/Output		**Assumptions:**	
Type	**Description**	**Tag**	**BT**

The fields in the table are:
Test design item number: Unique identifier of the test design item
Traces: References to the requirement(s) or other descriptions covered by this test design
Based on: Input/Output: Indication of which type of domain the design is based on
Assumptions: Here any assumption must be documented.
For each test condition we have the following fields:
Type: Must be one of

- VC—Valid class
- IC—Invalid class
- VB—Valid boundary value
- IB—Invalid boundary value

Remember that the invalid values should be rejected by the system.
Description: The specification of the test condition
Tag: Unique identification of the test condition
BT = Belongs to: Indicates the class a boundary value belongs to. This can be used to cross-check the boundary values.

Equivalence Partitioning and Boundary Value Analysis Test Design Examples

In this example we shall find test conditions and test cases for the testing of this user requirement.

[UR 631] The system shall allow shipments for which the price is less than or equal to € 100.

The first thing we'll do is fill in the header of the design table.

Test design item number: 11	**Traces: [UR 631]**
Based on: Input	**Assumptions:** The price cannot be negative The smallest increment is 1 cent

The next thing is identifying the valid class(es).

Type	**Description**	**Tag**	**BT**
VC	0 <= shipment price <= 100		

We then consider if there are any invalid classes. If we only have the single requirement given above, we can identify two obvious and two special invalid equivalence classes. The new rows are indicated in bold.

Type	**Description**	**Tag**	**BT**
IC	**shipment price < 0**		
VC	0 <= shipment price <= 100		
IC	**shipment price > 100**		
IC	**shipment price is empty**		
IC	**shipment price contains characters**		

Our boundary value analysis gives us two boundary values.

Type	Description	Tag	BT
IC	shipment price < 0		
IB	**shipment price = –0.01**		
VB	**shipment price = 0.00**		
VC	0 <= shipment price <= 100		
VB	**shipment price = 100.00**		
IC	shipment price > 100		
IB	**shipment price = 100.01**		
IC	shipment price is empty		
IC	shipment price contains characters		

This concludes the equivalence class partitioning and boundary value analysis for the first requirement. We will complete the table by adding tags and indicating to which classes the boundary values belong.

Test design item number: 11	**Traces: [UR 631]**
Based on: Input	**Assumptions:** The price cannot be negative The smallest increment is 1 cent

Type	Description	Tag	BT
IC	shipment price < 0	11-1	
IB	shipment price = –0.01	11-2	11-1
VB	shipment price = 0.00	11-3	11-4
VC	0 <= shipment price <= 100	11-4	
VB	shipment price = 100.00	11-5	11-4
IC	shipment price > 100	11-6	
IB	shipment price = 100.01	11-7	11-6
IC	shipment price is empty	11-8	
IC	shipment price contains characters	11-9	

We can now make low-level test cases. If we want 100% equivalence partition coverage and two value boundary value coverage for the requirement, assuming that invalid values are rejected, we get the following test cases:

Tag	Test case	Input Price=	Expected output
11-1	TC1-1	–25.00	rejection
11-2	TC1-2	0.02	rejection
11-2	TC1-3	–0.01	rejection
11-3	TC1-4	0.00	OK
11-3	TC1-5	0.01	OK
11-4	TC1-6	47.00	OK
11-5	TC1-7	99.99	OK
11-5	TC1-8	100.00	OK
11-7	TC1-9	100.01	rejection
11-7	TC1-10	100.02	rejection
11-6	TC1-11	114.00	rejection
11-8	TC1-12	" "	rejection
11-9	TC1-13	"abcd.nn"	rejection

We could choose to omit some of the test cases, especially since we actually get five different test cases covering the same equivalence class.

Ex.

In the next example we will test the following requirement:

[UR 627] The system shall allow the packing type to be specified as either "Box" or "Wrapping paper."

The test design table looks like this after the analysis.

Test design item number: 15		Traces: [UR 627]	
Based on: Input		Assumptions:	
Type	**Description**	**Tag**	**BT**
VC	"Box" "Wrapping paper"	15-1	
IC	All other texts	15-2	

This type of equivalence class does not have boundaries.

The test cases could be:

Tag	Test case	Input packing type	Expected output
15-1	TC3-1	"Box"	OK
15-2	TC3-2	"Paper"	rejection

It is only necessary to test one of the valid packing types, because it is a member of an equivalence class.

Equivalence Partitioning and Boundary Value Analysis Hints

The equivalence partitioning and boundary value analysis of requirements makes it possible for us to:

- Reduce the number of test cases, because we can argue that one value from each equivalence partition is enough.
- Find more faults because we concentrate our focus on the boundaries where the density of defects, according to all experience, is highest.

Sometimes the input or output domains we are testing do not have one-dimensional boundaries as assumed in the equivalence class partitioning and boundary value analysis as described here.

If it is not possible or feasible to partition our input or output domain in one dimension we have to use the technique called domain analysis instead.

4.1.2 Domain Analysis

Note: This technique is not part of the ISTQB syllabus, but included here because I find it interesting and occasionally useful.

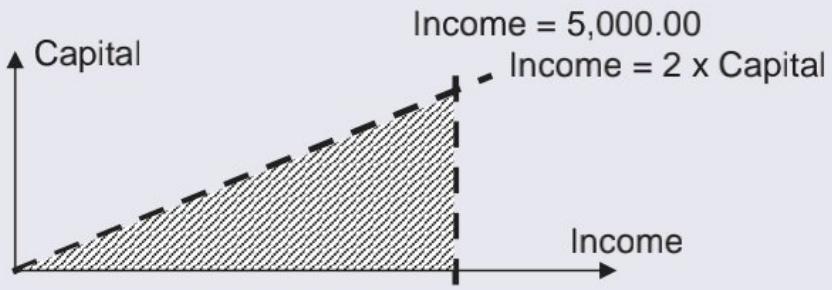
In equivalence partitioning of intervals where the boundaries are given by simple numbers, we have one-dimensional partitions. The domain analysis test case design technique is used when our input partitions are multi-dimensional, that is when a border for an equivalence partition depends on combinations of aspects or variables. If two variables are involved we have a two-dimensional domain; if three are involved, we have a three-dimensional domain, and so on.

Multidimensional partitions are called domains—hence the name of the technique, even though the principles are the same as in equivalence partitioning and boundary value analysis.

It is difficult for people to picture more than three dimensions, but in theory there is no limit to the number of dimensions we may have to handle in domain analysis. We will use two-dimensional domains in the section; the principles are the same for any number of dimensions.

For equivalence partitioning we had the example of intervals of income groups, where 0.00 <= income < 5,000.00 is tax-free. This is a one-dimensional domain. If people's capital counts in the calculation as well, so that income is only tax-free if it is also less than twice the capital held by the person in question, we have a two-dimensional domain.

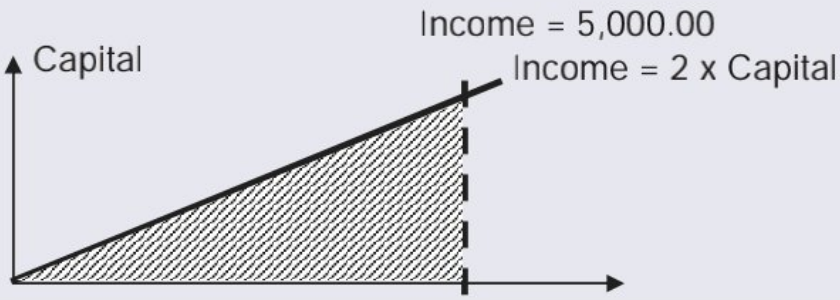
The two-dimensional domain for tax-free income is shown as the striped area (assuming that the capital and the income are both ≥ 0.00):



Borders may be either open or closed. A border is *open*, if a value on the border does not belong to the domain we are looking at. This is the case in the example where both the borders are open (income < 5,000.00 and income < 2 x capital).

A border is *closed* if a value on the border belongs to the domain we are looking at.

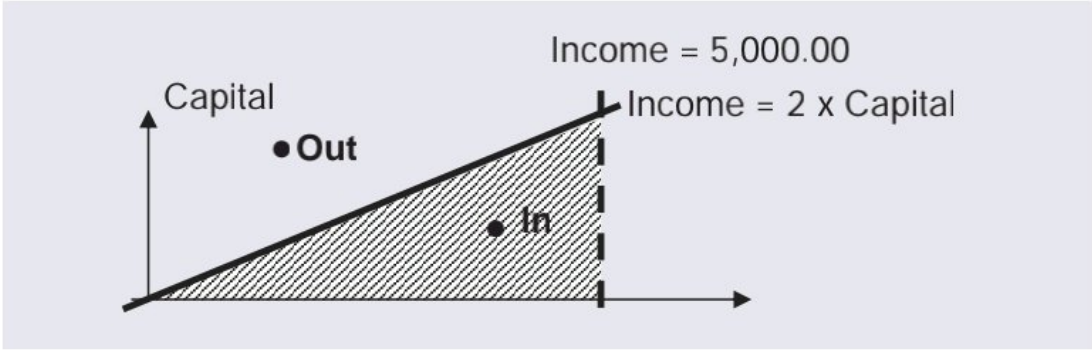
If we change the border for tax-free income to become: income <= 2 x capital, we have a closed border. In this case our domain will look like this:



In equivalence partitioning we say that a value is in a particular equivalence class—in a way we do the same for domain analysis, though here we operate with points relative to the borders:

- A point is an In point in the domain we are considering, if it is inside and not on the border
- A point is an Out point to the domain we are considering, if it is outside and not on the border (it is then in another domain)

An In point and an Out point relative to the border income <= 2 x capital are illustrated here.

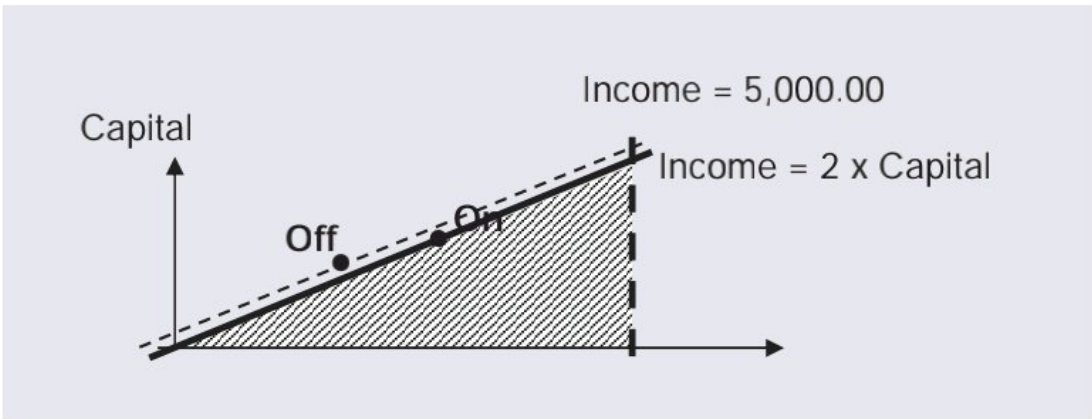


In the boundary value analysis related to equivalence partitioning described above, we operate with the boundary values on the boundary and one unit inside. In domain analysis we operate with On and Off points relative to each border.

We have:

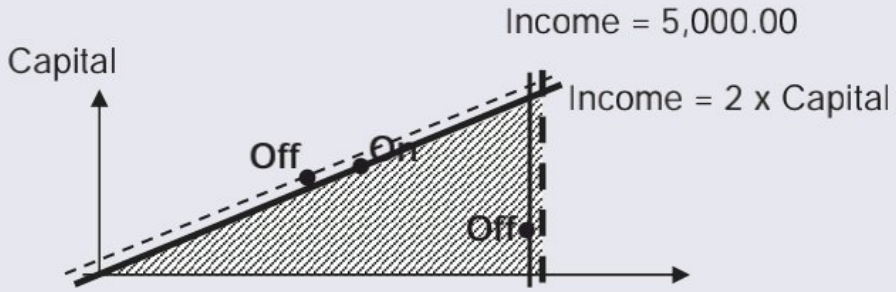
- A point is an On point if it is on the border between partitions
- A point is an Off point if it is "slightly" off the border

If the border of the domain we are looking at is closed, the Off point will be outside the domain. The "slightly" may be one unit relative to what measure we are using, so that the Off point lies on the border of the adjacent domain. This works for all practical purposes, as long as we are not working with a floating point where "one unit" is undeterminable; in this case the "slightly" will have to be far enough away from the border to ensure that the Off point is inside the adjacent domain.



If the border of the domain we are looking at is open, the Off point will be "slightly" (or perhaps one unit) inside the domain, namely, outside (or on) the closed border of the adjacent domain.

In our case with the income < 5,000.00 an On point and an Off point may look like this:



Domain Analysis Strategy

The number of test cases we can design based on a domain analysis depends on the test strategy we decide to follow. A strategy can be described as:

N-On * N-Off

where N-On is the number of On points we want to test for each border and N-Off correspondingly is the number of Off points we want to test for each border for the domains we have identified.

If we choose a 1 * 1 strategy we therefore set out to test one On point and one Off point for each border of the domains we have identified. This is what is illustrated above for the one domain we are looking at (not taking the capital>=0 border into account). If our strategy is 2 * 1, we set out to test two On points and one Off point for each of the borders for all the domains we have identified. In this case the On points will be the points where the borders cross each other, that is the extremes of the borders.

In a 1 * 1 strategy we will get two test cases for each border. If we are testing adjacent domains, we will get equivalent test cases because an Off point in one domain is an In point in the adjacent domain. These test cases will have identical expected outcomes if identical values are chosen for the low-level test cases. These duplicates need not be repeated in the test procedures for execution.

Domain Analysis Coverage

It is possible to measure the coverage for domain analysis. The coverage elements for the identified domains are the In points and the Out points. The coverage is measured as the percentage of In points and Out points that have been exercised by a test. Do not count an In point in one partition being an Out point in another partition to be tested twice.

The coverage elements for the borders are the On points and Off points. The coverage is measured as the percentage of On points and Off points that have been exercised by a test relative to what the strategy determines as the number of points to test. Again do not count duplicate points twice.

Domain Analysis Test Design Template

The design of the test conditions based on domain analysis and with the aim of getting On and Off point coverage can be captured in a table like this one.

Tag						
Border 1 condition	ON	OFF				
Border 2 condition			ON	OFF		
Border n condition					ON	OFF

Table designed by Carsten Jørgensen

The table is for one domain; and it must be expanded both in the length and width to accommodate all the borders our domain may have.

The rule is: divide and conquer.

For each of the borders involved we should:

- Test an On point
- Test an Off point

If we want In point and Out point coverage as well, we must include this explicitly in the table.

When we start to make low-level test cases we add a row for each variable to select values for. In a two-dimensional domain we will have to select values for two variables.

Tag						
Border 1 condition	ON	OFF				
Border 2 condition			ON	OFF		
Border n condition					ON	OFF
Variable X						
Variable Y						

For each column we select a value that satisfies what we want. In the first column of values we must select a value for X and a value for Y that gives us a point On border 1. We should aim at getting In points for the other borders in the column, though that is not always possible. For each of the borders we should note which kind of point we get for the selected set of values.

This selection of values can be quite difficult, especially if we have high-dimensional domains. Making the table in a spreadsheet helps a lot, and other tools are also available to help.

Domain Analysis Test Design Example

In this example we shall test this user requirement:

[UR 637] The system shall allow posting of envelopes where the longest side (l) is longer than or equal to 12 centimeters, but not longer than 75 centimeters. The smallest side (w) must be longer than or equal to 1 centimeter. The length must be twice the width and must be greater than or equal to 10 centimeters. Measures are always rounded up to the nearest centimeter. All odd envelopes are to be handled by courier.

We can rewrite this requirement to read:
length >= 12
length < 75
width >= 1
length – 2 x width >= 10
This can be entered in our template:

Tag	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
Length > = 12	ON 12	OFF						
Length < 75	12 IN		ON	OFF				
Width > = 1	1 IN				ON	OFF		
1 – 2 x w > = 10	10 ON						ON	OFF
Length	12							
Width	1							

In the table we have entered values for the first test case, namely length = 12 to get the On point for the first border condition. The simultaneous selecting of width = 1 gives the points indicated for each border in lowercase under the number.

The table fully filled in may look like this:

Tag	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
Length > = 12	ON 12	OFF 11	75 in	74 in	15 in	15 in	30 in	31 in
Length < 75	12 IN	11 in	ON 75	OFF 74	15 in	15 in	30 in	31 in
Width > = 1	1 in	1 in	30 in	30 in	ON 1	OFF 0	10 in	11 in
1 – 2 x w > = 10	10 on	9 out	15 in	14 in	13 in	15 in	ON 10	OFF 9
Length	12	11	75	74	15	15	30	31
Width	1	1	30	30	1	0	10	11

We now need to determine the expected results, and then we have our test cases ready.

4.1.3 Decision Tables

A decision table is a table showing the actions of the system depending on certain combinations of input conditions.

Decision tables are often used to express rules and regulations for embedded systems and administrative systems. They seem to have gone a little out of fashion, and that is a shame. Decision tables are brilliant for overview and also for determining if the requirements are complete.

It is often seen that what could have been expressed in a decision table is attempted to be explained in text. The text may be several paragraphs or even pages long and reformatting of the text into decision tables will often reveal holes in the requirements.

Decision tables are useful to provide an overview of combinations of inputs and the resulting output. The combinations are derived from requirements, which are expressed as something that is either true or false. If we are lucky the requirements are expressed directly in decision tables where appropriate.

Decision tables always have 2^n columns, because there are always 2^n combinations, where n is the number of input conditions.

The number of rows in decision tables depends on the number of input conditions and the number of dependent actions. There is one row for each condition and one row for each action.

Input condition 1	T	T	F	F
Input condition 2	T	F	T	F
Action 1	T	T	F	F
Action 2	T	T	T	F

The table is read one column at a time. We can, for example, see that if both input conditions are true then both actions will happen (be true).

The *coverage measure* for decision tables is the percentage of the total number of combinations of input tested in a test.

Sometimes it is not possible to obtain 100% combination coverage because it is impossible to execute a test case for a combination.

4.1.3.1 Decision Table Templates

The template to capture decision table test conditions in is the template for the decision table itself with a test design header, as shown next.

Test design item number:	**Traces:**			
Assumptions:				
	TC1	TC2		TCn
Input condition 1				
Input condition n				
Action 1				
Action n				

The fields in the table are:

Test design item number: Unique identifier of the test design item
Traces: References to the requirement(s) or other descriptions covered by this test design
Assumptions: Here any assumption must be documented

The table must have a row for each input and each action, and 2^n columns, where n is the number of input conditions. The cells are filled in with either True or False to indicate if the input conditions, respectively the actions are true or false.

The easiest way to fill out a decision table is to fill in the input condition rows first. For the first input condition half of the cells are filled with True and the second half are filled with False. In the next row half of the cells under the Ts are filled with True and the other half with False, and likewise for the Fs. Keep on like this until the Ts and Fs alternate for each cell in the last input condition row.

The values for the resulting actions must be extracted from the requirements!

4.1.3.2 Decision Table Example

Ex.

In this example we are going to test the following requirements.

[76] The system shall only calculate discounts for members.

[77] The system shall calculate a discount of 5% if the value of the purchase is less than or equal to € 100. Otherwise the discount is 10%.

[78] The system shall write the discount % on the invoice.

[79] The system must write in the invoices to nonmembers that membership gives a discount.

Test design item number: 82	**Traces: Req. [76]–[79]**
Assumptions: The validity of the input is tested elsewhere	

Note that we are only going to test the calculation and printing on the invoice, not the correct calculation of the discount.

	TC1	TC2	TC3	TC4
Purchaser is member	T	T	F	F
Value <= € 100	T	F	T	F
No discount calculated	F	F	T	T
5% discount calculated	T	F	F	F
10% discount calculated	F	T	F	F
Member message on invoice	F	F	T	T
Discount % on invoice	T	T	F	F

The test cases to execute can now be created from these test conditions and written directly into a test procedure.

Test procedure: 11			
Purpose: This test procedure tests the calculation of discount for members. Traces: Req. [76]–[79]			
Tag	**TC**	**Input**	**Expected output**
TC1	1	Choose a member and create a purchase with a value less than € 100	A discount of 5% is calculated and this is written on the invoice.
TC2	2	Choose a member and create a purchase with a value of more than € 100	A discount of 10% is calculated and this is written on the invoice.
TC3	3	Choose a nonmember and create a purchase with a value less than € 100	No discount is calculated and the "membership gives discount" statement is written on the invoice.

4.1.3.3 Collapsed Decision Tables

Sometimes it seems evident in a decision table that some conditions are without effect because one decision is decisive. For example if one condition is False an action seems to be False no matter what the values of all the other conditions are.

This could lead us to collapse the decision table, that is reduce the number of combinations by only taking one of those where the rest will give the same result. This technique is related to the condition determination testing discussed below in Section 4.2.6.

The decision as to whether to collapse a decision table or not should be based on a risk analysis.

4.1.4 Cause-Effect Graph

A cause-effect graph is a graphical way of showing inputs or stimuli (causes) with their associated outputs (effects). The graph is a result of an analysis of requirements. Test cases can be designed from the cause-effect graph.

The technique is a semiformal way of expressing certain requirements, namely requirements that are based on Boolean expressions.

The cause-effect graphing technique is used to design test cases for functions that depend on a combination of more input items.

In principle any functional requirement can be expressed as:

***f*(old state, input) → (new state, output)**

This means that a specific treatment (*f* = a function) for a given input transforms an old state of the system to a new state and produces an output.

We can also express this in a more practical way as:

***f*(ops1, ops2,..., i1, i2,..i) → (ns1, ns2,..,o1, o2..)**

where the old state is split into a number of old partial states, and the input is split into a number of input items. The same is done for the new state and the output.

The causes in the graphs are characteristics of input items or old partial states. The effects in the graphs are characteristics of output items or new partial states.

Both causes and effects have to be statements that are either True or False. True indicates that the characteristic is present; False indicates its absence.

The graph shows the connections and relationships between the causes and the effects.

4.1.4.1 Cause-Effect Graph Coverage

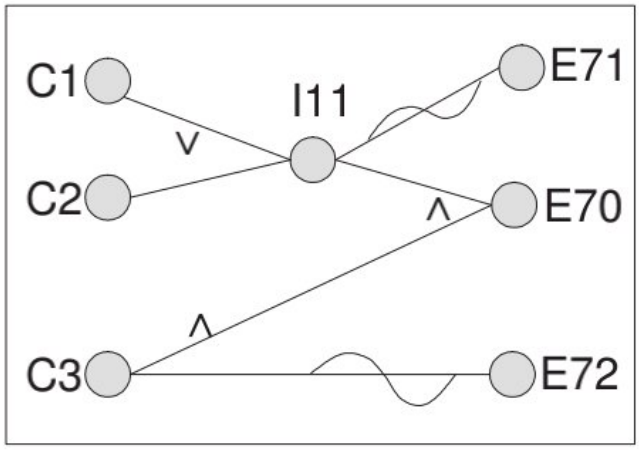
The coverage of the cause-effect graph can be measured as the percentage of all the possible combinations of inputs tested in a test suite.

4.1.4.2 Cause-Effect Graphing Process and Template

A cause-effect graph is constructed in the following way based on an analysis of selected suitable requirements:

- List and assign an ID to all causes
- List and assign an ID to all effects
- For each effect make a Boolean expression so that the effect is expressed in terms of relevant causes
- Draw the cause-effect graph

An example of a cause-effect graph is shown here.



The graph is composed of some simple building blocks:

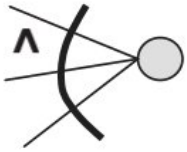
 Identified cause or effect—Must be labeled with the corresponding ID. It is a good idea to start the IDs of the causes with a C and those of the effects with an E. Intermediate causes may also be defined to make the graph simpler.

 Connection between cause(s) and effect—The connection always goes from the left to the right.

∧ This means that the causes are combined with AND, that is all causes must be True for the effect to be True.

∨ This means that the causes are combined with OR, that is only one cause needs to be True for the effect to be True.

This is a negation, meaning that a True should be understood as a False, and vice versa.



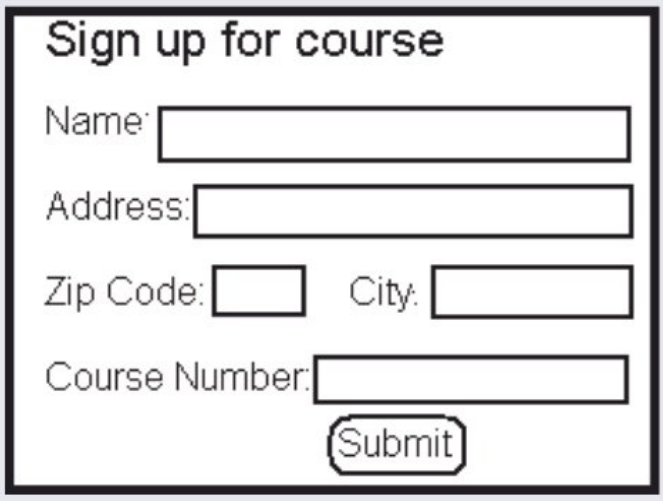
The arch shows that all the causes (to the left of the connections) must be combined with the Boolean operator; in this case the three causes must be "AND'et."

Test cases may be derived directly from the graph. The graph may also be converted into a decision table, and the test cases derived from the columns in the table.

Sometimes constraints may apply to the causes and these will have to be taken into consideration as well.

4.1.4.3 Cause-Effect Graph Example

In this example we are going to test a Web page, on which it is possible to sign up for a course. The Web page looks like this:



First we make a complete list of causes with identification. The causes are derived from a textual description of the form (not included here):

C1. Name field is filled in
C2. Name contains only letters and spaces
C3. Address field is filled in
C4. Zip code is filled in
C5. City is filled in
C6. Course number is filled in
C7. Course number exists in the system

An intermediate Boolean may be introduced here, namely I30 meaning that all fields are filled in. This is expressed as:

I30 = and (C1, C3, C4, C5, C6)

The full list of effects with identification is:

E51. Registration of delegate in system

E52. Message shown: All fields should be filled in

E53. Message shown: Only letters and spaces in name

E54. Message shown: Unknown course number

E55. Message shown: You have been registered

We must now express each effect as a Boolean expression based on the causes. They are:

E51 = and(I30, C2, C7)

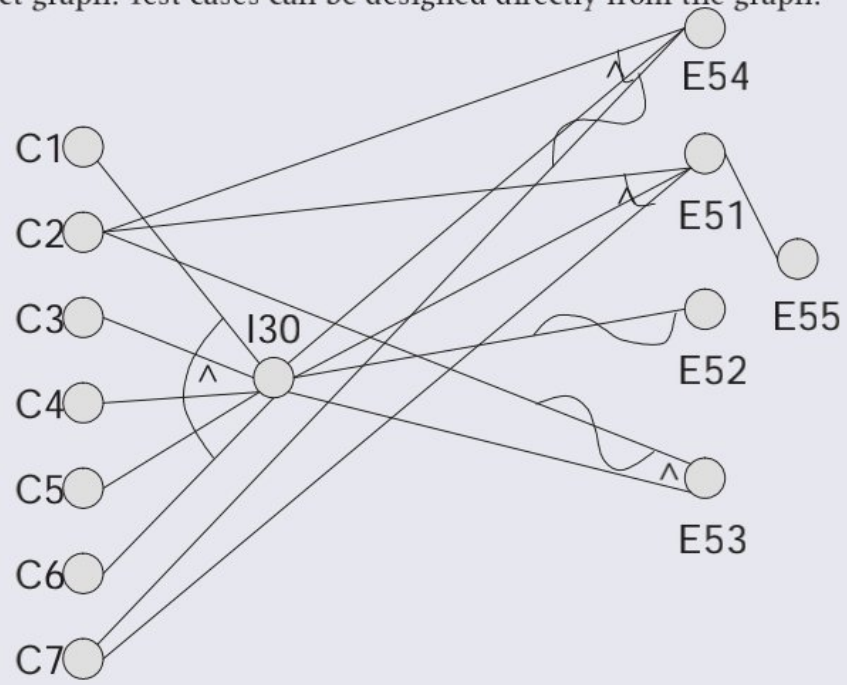
E52 = not I30

E53 = and(I30, not C2)

E54 = and(I30, C2, not C7)

E55 = E51

Drawing the causes and the effects and their relationships gives us the cause-effect graph. Test cases can be designed directly from the graph.



4.1.4.4 Cause-Effect Graph Hints

The cause-effect graph test case design technique is very suitable for people with a graphical mind.

For others it may be a help in the analysis phase and the basis for the construction of a decision table from which test cases can be described as discussed above.

All the effects can be filled into the decision table by looking at the cause-effect graph or even from the Boolean expressions directly. Fill in all combinations of True and False for the causes, and then fill in the impact each combination has on the effects.

Cause-effect graphs frequently become very large—and therefore difficult to work with. To avoid this divide the specification into workable pieces of isolated functionality.

To mitigate the size problem we can select different ways to reduce the decision table, such as removal of impossible combinations. We can also reduce the combinations by only keeping those that independently affect the outcome, like in condition determination testing, discussed in Section 4.2.6.

4.1.5 State Transition Testing

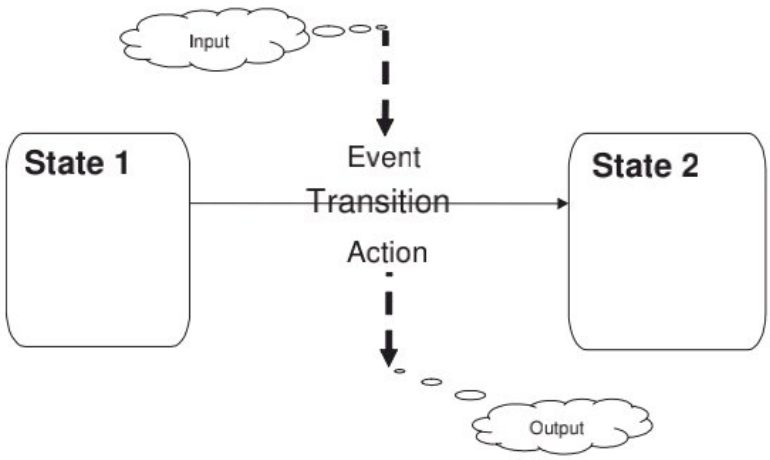
State transition testing is based on a state machine model of the test object. State machine modeling is a design technique, most often used for embedded software, but also applicable for user interface design. If we are lucky we get the state machine model as part of the specification we are going to test against. Otherwise we will have to extract it from the requirements.

Most products and software systems can be modeled as a state machine. The idea is that the system can be in a number of well-defined states. A state is the collection of all features of the system at a given point in time, including all visible data, all stored data, and any current form and field.

The transition from one state to another is initiated by an event. The system just sits there doing nothing until an event happens. An event will cause an action and the object will change into another state (or stay in the same state).

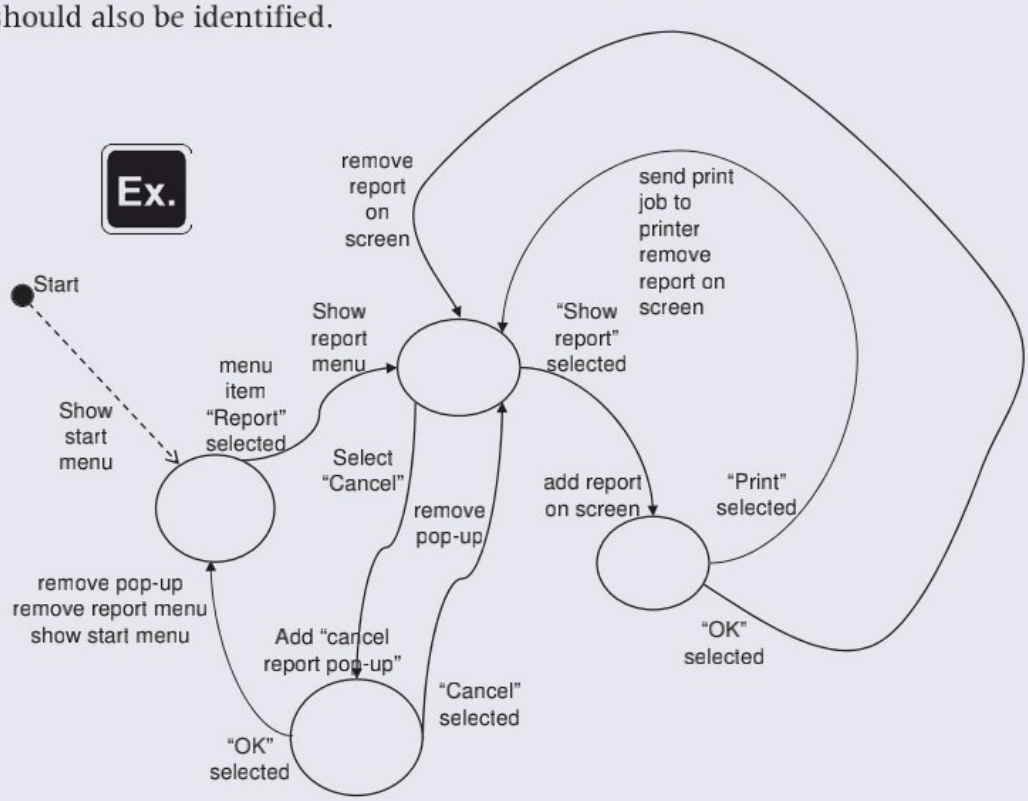
A transition = start state + event + action + end state

The principle in a state machine is illustrated next.



State machines can be depicted in many ways. The figure below shows a state machine presentation of a report printing menu, where the states are depicted as circles and the events and actions are written next to the transitions depicted as arrows.

It is a good idea to leave the states blank and just give them an identification, typically in the form of a number. The events, actions, and transitions should also be identified.



Note that the state machine has a start state. This could be a transition from another state machine describing another part of the full system.

4.1.5.1 State Transition Testing Coverage

Transitions can be performed in sequences. The smallest "sequence" is one transition at a time. The second smallest sequence is a sequence of two transitions in a row. Sequences can be of any length.

The coverage for state transition testing is measurable for different lengths of transition sequences. The state transition coverage measure is:

Chows n-switch coverage

where n = sequential transitions – 1.

We could also say that N = no. of "in-between-states."

Chows n-switch coverage is the percentages of all transition sequences of

n-1 transitions' length tested in a test suite.

State transition testing coverage is measured for valid transitions only. Valid transitions are transitions described in the model. There may, however, also be invalid, or so-called null-transitions and these should be tested as well.

4.1.5.2 State Transition Testing Templates

A number of tables are used to capture the test conditions during the analysis of state transition machines.

To obtain Chows 0-switch coverage, we need a table showing all single transitions. These transitions are test conditions and can be used directly as the basis for test cases. A simple transition table is shown next.

The fields in the table are:

Test design item number: Unique identifier of the test design item
Traces: References to the requirement(s) or other descriptions covered by this test design
Assumptions: Here any assumption must be documented
The table must have a column for each of the defined transitions (three are shown here). The information for each transition must be:
Transition: The identification of the transition
Start state: The identification of the start state (for this transition)
Input: The identification or description of the event that triggers the transition
Expected output: The identification or description of the action connected to the transition
End state: The identification of the end state (for this transition)

Test design item number:			**Traces:**
Assumptions:			
Transition			
Start state*			
Input			
Expected output			
End state*			

* the "start"and "end" states are for each specific transition (test condition) only, not the state machine

Testing to 100% Chows 0-switch coverage detects simple faults in transitions and outputs.

To achieve a higher Chows n-switch coverage we need to describe the sequences of transitions.

The table to capture test conditions for Chows 1-switch coverage is shown next.

Test design item number:			Traces:
Assumptions:			
Transition pair			
Start state*			
Input			
Expected output			
Intermediate state*			
Input			
Expected output			
End state*			

Here we have to include the intermediate state and the input to cause the second transition in each sequence. Again we need a column for each set of two transitions in sequence.

If we want an even higher Chows n-switch coverage we must describe test conditions for longer sequences of transitions.

As mentioned above we should also *test invalid transitions*. To identify these we need to complete a state table. A state table is a matrix showing the relationships between all states and events, and resulting states and actions.

A template for a state table matrix is shown below. The matrix must have a row for each defined state and a column for each input (event). In the cross-cell the corresponding end state and actions must be given.

An invalid transaction is defined as a start state where the end state and action is not defined for a specific event. This should result in the system staying in the start state and no action or a null-action being performed, but since it is not specified we cannot know for sure.

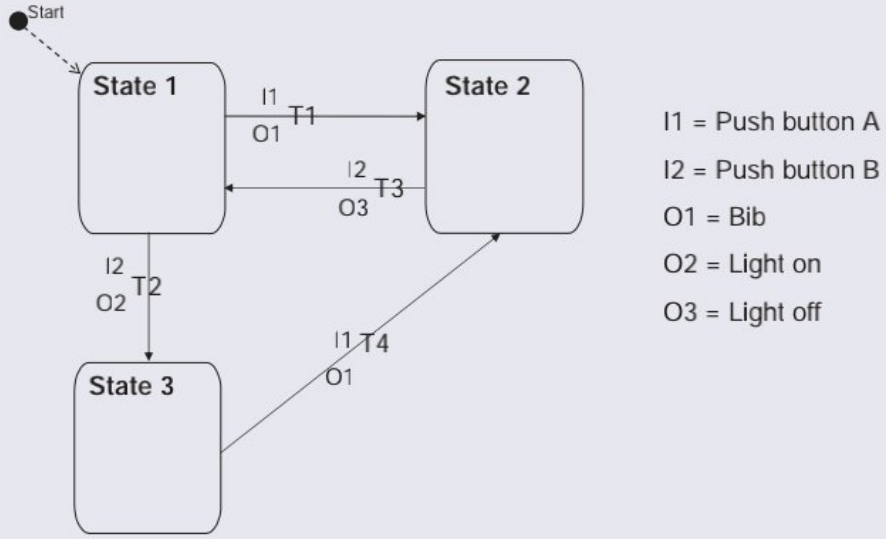
Input / Start state	
	End state / Action

The "End state / Action" for invalid transitions must be given as the identification of the start state / "N" or the like.

A test condition can be identified from this table for each of the invalid transitions.

4.1.5.3 State Transition Testing Example

In this example we are going to identify test conditions and test cases for the state machine shown here.



Don't worry about what the system is doing—that is not interesting from a testing point of view.

The drawing of the state machine shows the identification of the states, the events (inputs), the actions (outputs), and the transitions. The descriptions of the inputs and outputs are given to the right of the drawing.

First of all we have to define test conditions for all single transitions to get Chows 0-switch coverage.

Test design item number: 2.4		**Traces:** State machine 1.1		
Assumptions: None				
Transition	T1	T2	T3	T4
Start state*	S1	S1	S2	S3
Input	I1	I2	I2	I1
Expected output	O1	O2	O3	O1
End state*	S2	S3	S1	S2

Identification of sequences of two transitions to achieve Chows 1-switch coverage results in the following table.

Test design item number: 2.5		**Traces:** State machine 1.1			
Assumptions: None					
Transition pair	T1/T3	T1/T3	T3/T2	T2/T4	T4/T3
Start state*	S1	S2	S2	S1	S3
Input	I1	I2	I2	I2	I1
Expected output	O1	O3	O3	O2	O1
Intermediate state*	S2	S1	S1	S3	S2
Input	I2	I1	I2	I1	I2
Expected output	O3	O1	O2	O1	O3
End state*	S1	S2	S3	S2	S1

We will not go further in sequences.

The next thing will be to identify invalid transitions. To do this we fill in the state table. The result is:

Input / **Start state**	I1	I2
S1	S2/O2	S3/O2
S2	S2/**N**	S1/O3
S3	S2/O1	S3/**N**

We have two invalid transitions:

State 2 + Input 1 and State 3 + Input 2.

The test cases to execute can now be created from these test conditions and written directly into a test procedure.

Test procedure: 3.5			
Purpose: This test procedure tests single valid and invalid transitions. Traces: State machine 1.1			
Prerequisites: The system is in State 1			
Expected duration: 5 minutes			
Tag	TC	Input	Expected output
T1	2	Reset to state 1 + push button B	The system bibs + state 2
T2		Reset to state 1 + push button B	The light is on + state 3
IV2		Push button B again	Nothing changes
T4		Push button A	The system bibs + state 2
IV1		Push button A again	Nothing changes
T3		Push button B	The light is off + state 1

4.1.5.4 State Transition Testing Hints

We should try to avoid invalid transitions by defining the results of invalid events. This is called defensive design, and it is a design activity.

If it is not practical to define all possible state and event combinations explicitly, we should encourage the designers to define a default for truly invalid situations. It could for example be defined that all null-transitions should result in a warning.

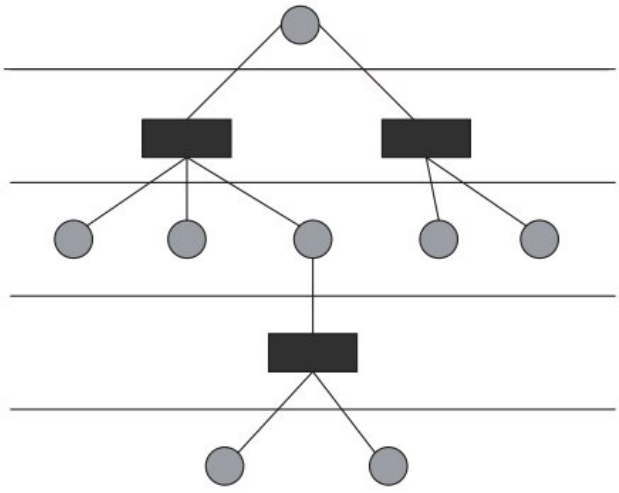
One of the difficulties of state machine is that they can become extremely complex faster and more often than you imagine. State machines can be defined in several levels to keep the complexity down, but this is a design decision.

4.1.6 Classification Tree Method

The classification tree method is a way to partition input and state domains into classes. The method is similar to equivalence partitioning, but can handle more complex situations where input or output domains can be looked at from more than one point of view.

The idea in the classification tree method is that we can partition a domain in several ways and that we can refine the partitions in a stepwise fashion. Each refinement is guided by a specific aspect or viewpoint on the domain at hand.

The result is a classification tree like the one shown here.



There are two types of nodes in the tree

- (Sub)domain nodes
- Aspect nodes

The two types must always alternate.

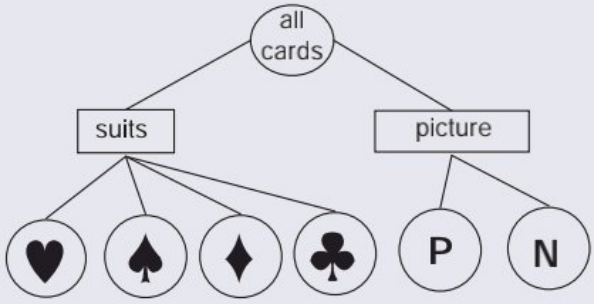
The domain is the full collection of all possible inputs and states at any given level in the tree. The state is a very broad term here; it means anything that characterizes the product at a given point in time and includes for example which window is current, which field is current, and all data relevant for the behavior both present on the screen and stored "behind the screen."

The aspect is the point of view you use when you are performing a particular partitioning of the domain you are looking at. It is very important to be aware that it is possible to look at the same domain in different ways (with different aspects) and get different subdomains as the result. This is why there can be more aspects at the same level in the classification tree and more subdomains at the same level (under the aspects) as well.

An ordinary deck of cards (without jokers) can be viewed and hence partitioned in different ways. The aspects could be:

- Suit (spades, hearts, diamonds, clubs)
- Picture or number

The classification tree resulting from this analysis would look like this:



There are a few rules that need to be observed when we make the classification tree. Under a given aspect:

- *Any* member of the domain must fit into one and only one subdomain under an aspect—it must not be possible to place a member in two or more subdomains
- *All* the members of the domain must fit into a subdomain—no member must fall outside the subdomains

If we look at the "suit" aspect above, no card can be in two suits, and all cards belong to a suit.

When we create a classification tree we start at the root domain (which is highest in the graph!). This is always the full input and state domain for the item we want to examine. We must then:

- Look at the domain and decide on the views or aspects we want to use on the domain
- For each of these aspects
  - Partition the full root domain into classes. Each class is a subdomain.
  - For each subdomain (now a full domain in its own right)
    - Decide aspects that will result in a new partitioning
    - For each aspect
    - And so on

At a certain point it is no longer possible or sensible to apply aspects to a domain. This means that we have reached a leaf of the tree. The tree is finished when all our subdomains are leaves (the lowest in the graph!).

Leaves can be reached at different levels in the classification tree. The tree does not have to be symmetric or in any other way have a predictable shape.

A leaf in a classification tree is similar to a class in an equivalence partitioning: We only need to test one member, because they are all assumed to behave in the same way.

4.1.6.1 Classification Tree Method Coverage

The coverage for a classification tree is the percentage of the total leaf classes tested in a test suite.

Leaves belonging to different aspects can be combined, so that we can reach a given coverage with fewer test cases. In areas of high risk we can also choose to test combinations of leaf classes.

4.1.6.2 Classification Tree Method Test Design Template

It is usually more practical to present a classification tree in a table rather than as a tree. A template for such a table where the test conditions are captured, is shown below.

Test design item number.:				**Traces:**				
Assumptions:								
Domain 1	**Aspect 1**		**Domain n**	**Aspect n**	**Tag**	**Tc1**		**Ten**

The fields in the table are:
Test design item number: Unique identifier of the test design item
Traces: References to the requirement(s) or other descriptions covered by this test design
Assumptions: Here any assumption must be documented.
Domain 1: A description of the (root) domain
Aspect 1: A list of the aspects defined for domain 1
For each aspect a list of subdomains are made.
For each of the subdomains new aspects are identified or the subdomain is left as a leaf.
This goes on until we have reached the leaves in all branches.
Tag: Unique identification of the leaves = test conditions
Tc1: A marking of which test cases cover the test conditions

4.1.6.3 Classification Tree Method Example

In this example we are going to test the following requirements for a small telephone book system:

(1) A person can have more than one phone number
(2) More than one person can have the same phone number
(3) There is one input field where you can type either:

- Phone number
- All names in the full name
- Some of the names in the full name

(4) A person shall be found if one or more names match
(5) One or more people shall be found if the phone number matches
(6) The output shall be either:

- An entry for each person that is found
- An error message: no person found

First we fill in the header:

Test design item number: 56	**Traces:** Req. (1) – (6)
Assumptions: None	

The next thing is to look at the root and identify the first level of aspects:

Domain 1	Aspect 1
All inputs and types of lists	
	input type
	match?
	state of list
	result

Now we must take each of these aspects one by one and filter the root domain through them. The leaf subdomains are marked in bold.

Domain 1	**Aspect 1**	**Domain 2**
All inputs and types of lists		
	input type	**pure text**
		pure number
		mixture
		empty
	match?	**no**
		yes
	state of list	**empty**
		not empty
	result	**nothing found**
		something found

We now have seven subdomains that are leaves, and three that can be further broken down by new aspect. So we find some aspects for each of the remaining subdomain, and find the subdomain for each.

The "old" leaves have been left out of the table for the time being, and the "new" leaves are marked in bold.

Domain 1	**Aspect 1**	**Domain 2**	**Aspect 2**	**Domain 3**
All inputs and types of lists				
	match?	yes	*no.of matches*	**1**
				more than 1
			type of match	name
				phone no.
	state of list	not empty	*1 name + 1 no.*	**yes**
				no

Domain 1	Aspect 1	Domain 2	Aspect 2	Domain 3
All inputs and types of lists				
	match?	yes	*no.of matches*	**1**
				more than 1
			type of match	name
				phone no.
	state of list	not empty	*1 name + 1 no.*	**yes**
				no
			1 name + many no	**yes**
				no
			many names + 1 no.	**yes**
				no
			1 name + 1 no.	**yes**
				no
	result	something found	*1 name + many no.*	**yes**
				no
			many names + 1 no.	**yes**
				no

Now we have only one subdomain, which is not a leaf left to deal with, namely a match of the name. The aspect to use here is how much of the name matches, and the subdomains are: "full name" and "part of name." These are leaves.

Before we go any further we have to control the partitioning. We need to ensure that any member of a domain fits into one and only one subdomain.

In this example we find that a string of blanks belongs to more than one subdomain, because a blank is considered to be text. To resolve that we require that pure text contains at least one letter.

Our test conditions are now defined, and we must go on defining our test data and test cases.

If we examine the classification tree we can see that we need a number of phone lists as our test data.

Telephone list 1
- Bo Hansen 4311
- Neil Smith 4210 4545
- John Raven 4545

Telephone list 2
- Bo Hansen 4311
- Neil Smith 4210 4545

Telephone list 3
- Bo Hansen 4311
- John Raven 4545

Telephone list 4
- Neil Smith 4210 4545

Telephone list 5 is empty

All the test conditions identified in the classification tree can be covered by eight test cases. We could trace the test cases to the test conditions, but since some of the test cases cover quite a few test conditions we mark the test cases that cover the conditions in the condition table here.

The test cases we can design are:

TC	Input =	Expected output
1	Telephone list 1 "Bo Hansen"	Bo Hansen 43–11
2	Telephone list 1 "4545"	Neil Smith 4545 John Raven 4545
3	Telephone list 1 "John 2"	No person found
4	Telephone list 1 "4210"	Neil Smith 4210, 4545
5	Telephone list 2 ""	No person found
6	Telephone list 3 "Raven"	John Raven 4545
7	Telephone list 4 "Bo Hansen"	No person found
8	Telephone list 5 "43"	No person found

These tables tend to get rather big, and it is normally a good idea to handle them in a spreadsheet. The full condition table is shown in Appendix 4A.

4.1.6.4 Classification Tree Method Hints

The classification tree method facilitates the test case design when it is too complicated to make equivalence class partitioning.

This is mainly the case when input is composed of more parts, and the relations between the input parts rather than the individual parts determine the outcome.

The method is useful when input can be considered as a unit, for example when the input is given via a graphical user interface. In such forms there are no sequences in the input; the user determines the sequence and the input is treated when the user decides that the form is filled in.

In other types of user interfaces, where the sequence of the input is predetermined, equivalence partitioning can be used on each individual input item.

Remember that not only what is entered by the user is input. Lists, files, database tables, and other types of data used by the system are also part of the "input."

4.1.7 Pairwise Testing

When we make test cases from a classification tree, the combinations of the leafs we get in our test cases are often more or less selected at random. Furthermore we often do not get all possible combinations tested. This may also be the case when we are testing products with other types of possible combinations of configurations (preconditions) or of inputs.

An example of a product with a number of precondition possibilities is a system that may run:

- On three different browsers
- Using two different database administration systems
- On four different operating systems

This system has limited possibilities when you think about it, but even so there are 3 x 2 x 4 = 24 different possible combinations to test.

Another example is a system with a form for entering different information about clients, both actual and potential. The following information must be supplied: the values in brackets after the information type are the possible valid values to select from:

- Size (small, medium, large)
- Business (private, civil administration, defense)
- Relevance (low, middle, high)
- Status (customer, lead, potential)

Here we have 3 x 3 x 3 x 3 = 81 possible combinations.

In some cases it may be possible and relevant to test all combinations of preconditions and/or inputs. But in some cases it is not, and then we have to have another way of designing sufficient test cases.

The pairwise test case design technique is about testing pairs of possible combinations. This reduces the number of test cases compared to testing all combinations, and experience shows that it is sufficiently effective in finding defects in most cases.

It is not always an easy task to identify all the possible pairs we can make from the combination possibilities. There are two different techniques to assist us in that task, namely orthogonal arrays and the allpairs algorithm. There is not objective evidence as to which technique is the best, but both techniques have their fans and their opponents.

4.1.7.1 Pairwise Testing Coverage

It is possible to measure the coverage of all pairs. It is simply measured as the percentage of the possible pairs that have been exercised by a test.

4.1.7.2 Orthogonal Arrays

Orthogonal arrays were first described by the Swiss mathematician Leonhard Euler, born in 1707. He introduced much of the modern terminology for mathematical analysis, including the notation for mathematical functions.

An orthogonal array is a two-dimensional array (a matrix) of values ordered in such a way that all pairwise combinations of the values are present in any two columns of the arrays.

Ex.

1	1	1
1	2	2
2	1	2
2	2	1

An example of an orthogonal array is:

Select any two columns and all the possible pairs of 1 and 2:
(1,1); (1,2); (2,1); and (2,2)
are present.

Orthogonal arrays are said to be balanced, because the number of times one possible pair is present, all the pairs will be present the same number of times (in the simple example 1 time).

An orthogonal array is mixed if not all the columns have the same range of values. We can have an orthogonal array where one column only has 1s and 2s and other columns have 1s, 2s, and 3s, for example.

The size and contents of orthogonal arrays are usually described in a general manner like:

(N, s1k1 s2k2 ... , t)

where

N = number of rows (or runs)
s = number of levels = number of different values
k = number of factors = number of columns for the corresponding s
t = strength = in any *t* columns you see each of the *st* possibilities equally often (t is usually 2 and in that case often omitted from the description)

Note that the description is often ordered so that the *s*'s are ordered in ascending order, though the actual columns in the array may be arranged differently.

This notation is very helpful when we are looking for suitable orthogonal arrays for our testing task.

The simple orthogonal array shown above can be described as: (4, 23)

An orthogonal array described as (72, 25 33 41 67) is a mixed array with 72 rows, 5 columns with 2 different values, 3 columns with 3 different values, 1 column with 4 different values, and 7 columns with 6 different values, that is 16 columns in all.

Creating orthogonal arrays is not a simple task. Many people have contributed to libraries of orthogonal arrays, and new ones are still being created. A large number of arrays in all sizes and mixtures may be found on www. research.att.com/~njas/oadir/index.html.

We can use orthogonal arrays to help us identify all the pairs of possible inputs or preconditions that we want to test. What we need to do is find a suitable array and substitute the values in this with our values. If we then design test cases corresponding to each row, we are guaranteed to have tested all the possible pairs.

The process is the following:

- Identify the inputs/preconditions (IPs) that can be combined
- For each of the IPs find and count the possible values it can have (e.g., (IP1;*n*=2); (IP2;*n*=4) and so on)
- Find out how many occurrences you have of each n, (e.g., 3 times *n* = 2, 1 times *n*= 4 and so on) (this provides you with the needed sets of sk (e.g., 23 41))
- Find an orthogonal array that has a description of at least what you need—if you cannot find a precise match, take a bigger array; this often happens, especially if we need a mixed array
- Substitute the possible values of each of the IPs with the values in the orthogonal array—if we had had to choose an array that was too big, we could just fill in the superfluous cells with valid values chosen at random
- Design test cases corresponding to each row in the orthogonal array

Orthogonal Array Example

Ex.

This example covers the system with the input possibilities and their corresponding valid values shown here:

- Size (small, medium, large)
- Business (private, civil administration, defense)
- Relevance (low, middle, high)
- Status (customer, lead, potential)

There are four input possibilities, and each of them has three possibilities, so we need at least an array of (34). One such array can be found on the Internet, namely:

1	1	1	1
1	2	2	3
1	3	3	2
2	1	2	2
2	2	3	1
2	3	1	3
3	1	3	3
3	2	1	2
3	3	2	1

We will now assign the first column to size, and substitute the values in the array with the possible values for size.

The array will look as shown below, where we have added an extra row to show which column represents which input:

size	1	1	1
small	2	2	3
small	3	3	2
small	1	2	2
medium	1	2	2
medium	3	1	3
medium	3	1	3
large	1	3	3
large	2	1	2
large	3	2	1

The fully substituted orthogonal array is shown below:

size	business	relevance	status
small	private	low	customer
small	civil administration	middle	potential
small	defense	high	lead
medium	private	middle	lead
medium	civil administration	high	customer
medium	defense	low	potential
large	private	high	potential
large	civil administration	low	lead
large	defense	middle	customer

From this table we can design the nine low-level test cases needed to get 100% pair coverage by using the values given in each row as the input values for our four fields.

We still have to derive the expected results from the requirements or other basis material—as usual the test case design technique can not provide those.

4.1.7.3 Allpairs Algorithm

James Bach has created: "a script which constructs a reasonably small set of test cases that include all pairings of each value of each of a set of parameters." The script is called Allpairs and it is available from James Bach's Web site at www.satisfice.com.

The principle of finding the pairs is different from using orthogonal arrays, but the aim is the same: to reduce the number of test cases to run when testing combinations of a number of inputs/preconditions each with a number of valid values.

In the words of James Bach: "The Allpairs script does not produce an optimal solution, but it is good enough." This is, of course, James Bach's opinion and experience, and it is not necessarily true in all contexts.

4.1.7.4 Higher-Order Combinations

The techniques described above are concerned with getting pairs of possible values for test cases. We can also choose to test higher-order combinations, such as triples or more, but that is rarely done.

4.1.7.5 Pairwise Testing Hints

When we create pairs of values as described earlier, all the values have equal weight. This means that neither orthogonal arrays nor the Allpairs algorithm takes the distribution of the values and the risks associated with individual pairs into account.

It may well be, that one particular value is way more common than any of the others—no doubt for example that there are much more private businesses than defense-related businesses around.

It may also be possible that a specific combination has a much larger risk level than the others (i.e., that the effect if a defect is not found around that combination, is much higher than for all other combinations), it could, for example, be serious for our company if all defense/leads were left out of a mailing list with invitations to a special sales event, whereas it would hardly make any difference if small/privates were missed from a general mailing campaign.

To overcome this we should supply pairwise testing with risk analysis and design more test cases around the combinations with a high risk level.

In some cases the pairs we have established to test are not actually testable. One value of one input may be prohibitive for a specific value for another input. In this case there is nothing else to do than leave that test case out and explain why a lower coverage than expected was achieved.

4.1.8 Use Case Testing

The concept of use cases was first developed by the Swedish Ivar Jacobsen in 1992 and has since made triumphal progress in the IT development world.

A use case or scenario as it is also called shows how the product interacts with one or more actors. It includes a number of actions performed by the product as results of triggers from the actor(s). An actor may be a user or another product with which the product in question has an interface.

Use cases are much used to express user requirements at an early state in the development and they are therefore excellent as a basis for acceptance testing (if they are kept up-to-date and still reflect the product as it has turned out in the end!).

Use cases should be presented in a structured textual form with a number of headings for which the relevant information must be supplied. There are many ways of structuring a use case, and it is up to each organization to define its own standard.

The following example shows a set of headings for a use case with the explanations of what to write under each.

Use case:	The name of the use case. The name should be as descriptive as possible.
Purpose:	The goal of the use case (i.e., what is achieved when it is completed).
Actor:	Who (in terms of a predefined role) is interacting with the product.
Preconditions:	Any prerequisite that must be fulfilled before the use case may be performed.
Description:	The following is a list of actions to be performed. The list should have no more than 20 steps (n); otherwise the use case should be divided.
Actor	**Product**
1.	
	n.
Postconditions:	The state in which the product and/or the actor is to be found in when the use case has been performed.
Variants and exceptions:	Any specific cases in terms of variants or exceptions must be described here. This part often constitutes the bulk of the use case.
Rules:	Any specific rules or complex calculations are described here, or references are made to, for example, standards where such issues are detailed.
Safety:	Any safety considerations are described here.
Frequency:	How often the use case must be performed.
Critical conditions:	Any conditions under which the performance of the task is especially critical (e.g., in terms of response time or volume).

Testing a use case involves testing the main flow as specified in the steps in the description. Depending on the associated risks it may also include testing the variants and exceptions.

Note that the description of the main flow is usually much shorter than the descriptions of the variants and exceptions; these may also be considerably more difficult to test, if at all possible.

As it can be seen in the template above a good use case provides a lot of useful information for testing purposes. It should in fact be possible to design our test procedures directly from the use case description.

We can get the identification of the use case for traceability purposes and

the necessary preconditions directly from the form. The high-level test cases can be extracted directly from the steps in the description, where it should be ensured that each step provides the preconditions for the next one, except for the last which provides the expected postconditions.

A use case description will rarely contain actual input values; these must be selected when we design our low-level test cases. Appropriate specification-based techniques (for example, equivalence partitioning, boundary value analysis, and pairwise testing) may be used to select the actual values to use.

Based on the description, the postconditions, and possibly the rules it should be possible to derive expected results for each test case.

The information given for variants and exceptions, safety, frequency, and critical conditions can be used for risk analysis and decisions about which variants and exceptions to test to which depths.

4.1.8.1 Use Case Testing Coverage

Since a use case is not something easily measurable, there is no coverage item defined for use case testing, and it is therefore not possible to determine the coverage.

4.1.9 Syntax Testing

Note: This technique is not part of the ISTQB syllabus, but included here because I find it is very useful.

Syntax is a set of rules, each defining the possible ways of producing a string in terms of sequences of, iterations of, or selections among other strings.

Syntax is defined for input to eliminate "garbage in."

Many of the "strings" we are surrounded by in daily life are guided by syntax.

For example

- Web addresses: www.aaaaa.aa, aaaaa.aa
  (note the difference in appearances! My word processer recognizes the first address, but not the second address, because the www is missing)
- CPR number: ddmmyy-nnnn
  (Danish Central Person Registration number)
- Credit card number: nnnn nnnn nnnn nnnn
  (my VISA card—other cards have other syntaxes)

In the examples above the rules for the different strings are expressed using for example "n" to mean that a number should be at a specific place in the string or "dd" to indicate a day number in a month.

We can set up a list of rules, defining strings as building blocks and defining a notation to express the rules applied to the building blocks in a precise and compressed way. The building blocks are usually called the elements of the entire string.

The syntax rule for the string we are defining must be given a name.

The most commonly used notation form is the Backus-Naur form. This form defines the following notations

""	elementary part, e.g	"1" "-"
|	alternative separator	"A" |"B"
[ ]	optional item(s)	[" "]
{}	iterated item	

These notations can be used to form elements and the entire sting.

An example of a syntax rule for a string called pno. could be:

pno. = 2d [" "] 2d [" "] 2d [" "] 2d

Here we have defined the elements:

2d = dig dig

dig = "0" |"1" |"2" |"3" |"4" |"5" |"6" |"7" |"8" |"9"

The means that the pno. string must consist of four sets of two digits. The digits can range from 0 to 9. The sets of two digits can be separated by blanks, but they can also not be separated.

A valid string following this syntax is a Danish telephone number: 39 62 36 48.

The way my father used to write his telephone number is, however, illegal according to this syntax: 45 940 941.

To derive test conditions we need to identify options in the syntax and test these independently. Options appear when we can choose between elementary parts or elements for a given element or for the entire string. Syntax testing does not include combinations of options as part of the technique.

There is no coverage measure for syntax testing.

To make a negative test we need to test invalid syntax as well. For this we operate with possible mutations. Examples of the most common mutations are:

- Invalid value is used for an element
- One element is substituted with another defined element
- A defined element is left out
- An extra element is added

4.1.9.1 Syntax Testing Templates

The design of the test conditions based on syntax can be captured in a table like the one shown here The fields are the standard fields in test condition templates, as described earlier.

Test design item number:	Traces:
Based on: Input / Output	Assumptions:
Tag	**Description**

4.1.9.2 Syntax Testing Example

Ex.

In this example we will test the input of a member number. The syntax defined for the member number is

member no. = type" "no" "mm"-"yy

First we list the options derived from the entire string, the elements, and the elementary parts.

	options
member no. = type" "no" "mm"-"yy	none
type = "B" | "S" | "G"	3
no = dig dig dig	none
mm = "01" | "02" | ..... | "11" | "12"	12
yy = dig dig	none
dig = "0" | "1" | "2" | ... | "8" | "9"	10
	25

We have 25 possible independent mutations. We can list them in the template like this.

Tag	Description
T1	"B" | "S" | "G"
M1	"01" | "02" | ..... | "11" | "12"
D1	"0" | "1" | "2" | ... | "8" | "9"

Since we have not included specifications of what is happening when a string with a valid syntax is entered (nor what happens if the string is invalid) we will only list the inputs for the test cases we can design from the test conditions.

The following table shows a few of the 25 possible test cases.

TC	Tag	Input
TC1	T1	B 326 04-05
TC2	T1	S 326 04-05
TC4	M1	G 326 01-05
TC5	M1	G 326 02-05
TC6	M1	G 326 03-05
TC15	M1	G 326 12-05
TC16	M1	G 111 01-11
TC17	M1	G 222 01-22
TC24	M1	G 999 01-99
TC25	M1	G 000 01-00

The number of test cases may be reduced by having a single test case cover several options. This may, however, reduce the fault correction time, if failures are encountered, because it can be more difficult to locate the fault.

To test *invalid syntax* we list the mutations we want to try. These are:

Tag	Mutation description
MU1	Invalid value – Applicable to all positions in the string
MU2	Substitute – Any two elements
MU3	Element missing – Applicable to all elements
MU4	Extra element – Anything, but may not be possible

There is an infinite number of possible test cases for the mutations. We will only list a few here:

TC	Tag	Input
TC1	MU1	F 456 02-99
TC2	MU1	B-326 02-99
TC3	MU1	B a26 02-99
TC4	MU1	B-326 02-9g
TC12	MU2	BB456 02-99
TC15	MU2	B B 02-99
TC33	MU3	B 02-99
TC34	MU3	B 456-99
TC62	MU4	BB 456 02-99

4.1.9.3 Syntax Testing Hints

The number of invalid strings to test depends on the risk related to invalid input wrongly being accepted.

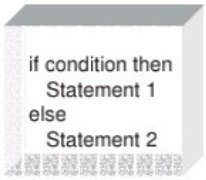
It can be a bit tricky to work with mutations, because some may be indistinguishable from correctly formed inputs if elements are identical. Some mutations may also be indistinguishable from each other, in which case they should be treated as one.

It is possible to define more mutations than those listed above, depending on the nature of the syntax.

To get an even stricter test we can use combinations of mutations. Who knows? Maybe two wrong elements at a time will cause the string to be accepted.

4.2 Structure-Based Techniques

The structural test case design techniques are used to design test cases based on an analysis of the internal structure of the component or system. These techniques are also known as white-box tests.



Traditionally the internal structure has been interpreted as the structure of the code. These techniques therefore focus on the testing of code and they are primarily used for component testing and low-level integration testing. In newer testing literature, structural testing is also applied to architecture where the structure may be a call tree, a menu structure, or a Webpage structure.

The structural test case design techniques covered here are:

- Statement testing
- Decision testing/branch testing
- Condition testing
- Multiple condition testing
- Condition determination testing
- LCSAJ (loop testing)
- Path testing
- (Interunit testing—not part of the ISTQB syllabus)

The test case design techniques in this category all require that the tester understands the structure (i.e., has some knowledge of the coding language). The tester doesn't necessarily need to be able to write code. It is like with a foreign language: You may be able to understand what is being said, but find it difficult to express yourself. This is usually not a problem anyway, because most structural testing of code is performed by programmers.

Structural testing is very often supported by tools, because the execution of components in isolation requires the use of stubs and drivers.

The test coverage can be measured for the structural test techniques. Test cases are designed to get the required coverage for the specified coverage item. If the coverage is expressed as statement coverage for example, the input to the test cases is determined using the statement test case design technique.

It is generally a good idea to start any test with specification-based, defect-based, and experience-based testing and measure the coverage achieved by executing tests designed with these techniques. This is cheaper and easier than to start off with structure-based techniques. Only if the achieved coverage is too low, should the appropriate structure-based technique be brought into action.

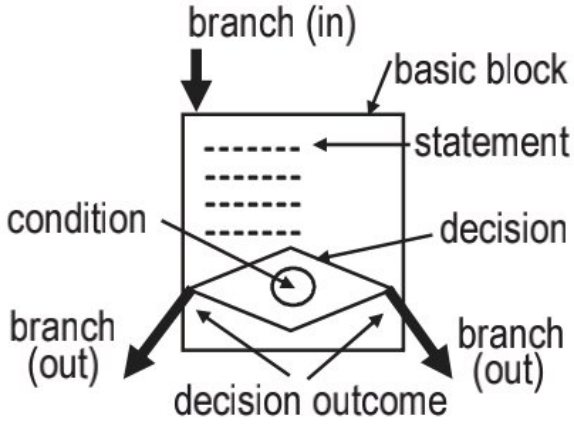
The techniques provide us with ideas for input. For the low-level test cases the concrete input values are selected. Subsequently the expected output is determined.

From where do we derive the expected output?

From the requirements—NEVER, ever from the code!

4.2.1 White-Box Concepts

Before we go any further there are some white-box concepts that need to be defined. They are illustrated in the "white-box inset" shown on the opposite page.



The first concept is that of a *statement*. An executable statement is defined as a noncomment or nonwhite space entity in a programming language, typically the smallest indivisible unit of execution.

A group of statements always executed together—or not at all—is called a *basic block*. A basic block can consist of only one statement or it can consist of many. There is no theoretical upper limit for the number of statements in a basic block.

The last statement in a basic block will always be a statement that leads to another building block, or stops the execution of the component we are working with (e.g., return) or the entire software system (end).

The most interesting is, however, when a basic block ends in a decision, that is a statement where the further flow depends on the outcome of the decision. Decision statements are for example IF ... THEN ...ELSE, FOR ..., DO WHILE..., and CASE OF....

A decision statement is also called a *branch point*. A *branch* is a (virtual) connection between basic blocks. One or more branches will lead into a basic block (except to the first), and likewise there will be one or more branches leading out of a basic block (except the last).

The braches out of a basic block are connected to the outcomes of the decision, also called *decision outcome* or branch outcome. Most decisions have two outcomes (True or False), but some have more, for example Case statements.

The last concept is that of a condition. A *condition* is a logical expression that can be evaluated to either True or False. A decision may consist of one simple condition, or a number of combined conditions.

4.2.2 Statement Testing

Statement testing is a test case design technique in which test cases are designed to execute statements.

A statement is executed in its entirety or not at all.

b = 3 + a; is one statement, whereas
if a = 2 then b = 3 + a end if; is more than one statement!

The definition of a statement is independent of how the code is actually written and what language it is written in.

In statement testing we design test cases to get a specifically required *statement coverage*. Statement coverage is the percentage of executable statements (in a component) that have been exercised by the test. Statement coverage is the weakest completion criterion we can have.

The component under test must be decomposed into the constituent statements, and we derive input for test cases from the code.

In the first example we have this small piece of code:

```
Read A;
Read B;
if A = 245 then
  Write 'Bingo';
endif;
Write B;
```

There are five statements (since "endif" doesn't count).
To get 100% statement coverage we need one test case:

TC	Input	Expected output
TC1	A = 245	

Note that we have no means of finding out what the expected output is, because the corresponding requirements (or design) are not included.

In the next example we will use this piece of code:

```
Read A;
Read B;
if A = 245 then
  Write 'Bingo';
endif;
if A < B then
  A = B;
else
  A = 0;
endif;
```

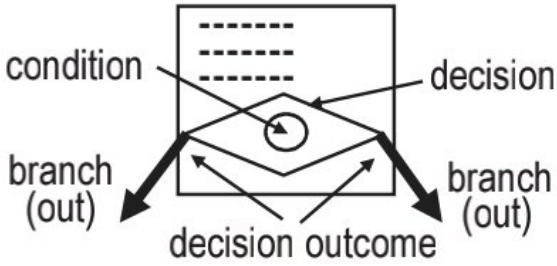
Here we have eight statements. To get 100% statement coverage we need two test cases:

TC	Input	Expected output
TC1	A = 245, B = 250	
TC2	A = 400, B = 250	

Again, we don't know what the expected results may be.

4.2.3 Decision/Branch Testing

Decision and branch testing have coexisted for many years. Experience has shown that it may be quite difficult to define branches correctly, whereas decisions and decision outcomes are much easier to define.



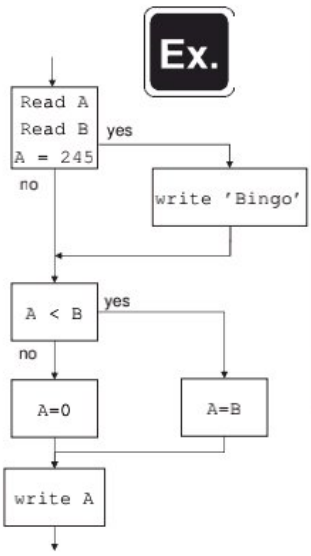
At 100% coverage branch coverage and decision coverage give identical results.

We can also see that decision outcome is defined to be equal to branch outcome, namely (according to BS7925-1): The result of a decision (which therefore determines the control flow alternative taken).

To define test cases for decision testing we have to:

- Divide the code into basic blocks,
- Identify the decisions (and hence the decision outcomes and the branches)
- Design test cases to cover the decision outcomes or branches

In most cases a decision has two outcomes (True or False), but it is possible for a decision to have more outcomes, for example, in "case of ..." statements.



Let us look at the write "Bingo" example again. The flow diagram corresponding to the code is shown here. We can see that this code has seven branches and four decision outcomes.

First we set A = 240 and B = 120.

This input covers three branches and we get branch coverage = 3/7 = 43%. It also covers four decision outcomes, and we get a decision outcome coverage of 2/4 = 50%.

Next we set A = 245 and B = 360.

This covers four more branches and two more decision outcomes, and we have now got 100% branch coverage and 100% decision outcome coverage.

It (often) requires more test cases to obtain 100% branch and decision outcome coverage than to obtain 100% statement coverage.

Decision coverage is usually measured using a software tool. Some tools can show the code and mark covered and uncovered decision outcomes by coloration of the code lines.

4.2.3.1 Other Decisions

Decisions may also be achieved by other statements than those using Boolean conditions, for example "case," "switch," or "computed goto" statements, or counting loops (implemented by "for" or "do" loops).

These should not go by untested. We have two options available for designing test cases for these, namely:

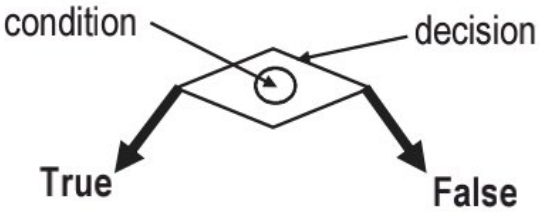
- Assume that the decision is actually implemented as an equivalent set of Boolean conditions
- Use a condition testing test case design technique as a supplement to decision testing

4.2.4 Condition Testing

A condition is a Boolean expression containing no Boolean operators, such as AND or OR. A condition is something that can be evaluated to be either TRUE or FALSE, like: "a < c."

A statement like "X OR Y" is not a condition, because OR is a Boolean operator and X and Y Boolean operands. X and Y may in themselves be conditions.

Conditions are found in decision statements.



Decision statements may have one condition like:
if (a < c) then …
They may also be composed of more conditions combined by Boolean operands, like:
if ((a=5) or ((c>d) and (c<f))) then …
or for short: if (X or (Y and Z)) then …

The condition outcome is the evaluation of a condition to be either TRUE or FALSE. In condition testing we test condition outcomes.

The *condition coverage* is the percentage of condition outcomes in every decision (in a component) that have been exercised by the test.

So to get 100% condition outcome coverage we need to get each condition to be True and False (i.e., two test cases).

In the first example with (a<c) we can design the test cases:

Test case	a	c	Outcome
1	5	7	True
2	6	2	False

In the next, more complex, example with (X or (Y and Z)) we also need to get each of the condition to be True and False.

Without going into detail about how to get X, Y, and Z to become True and False we can see, that we can still get 100% condition coverage with 2 test cases, namely for example:

Test case	X	Y	Z
1	True	True	True
2	False	False	False

A 100% condition coverage for a decision can usually be covered with two test cases regardless of the complexity of the decision statement.

The 100% condition coverage may even be achieved without getting a 100% decision outcome if the entire decision evaluates to the same for both cases! Condition testing may therefore be weaker than decision testing.

Condition testing and the condition coverage test measurement are vulnerable to Boolean expressions, which actually control decisions, being placed outside of the actual decision statement.

Ex.

Consider this example:

```
FLAG := A or (B and C);
if FLAG then
  do_something;
else
  do_something_else;
end if;
```

It may look as if we get 100% condition coverage by getting FLAG to evaluate to True and False, but in reality we need all of A, B, and C to evaluate to True and False.

To combat this we should design test cases for all Boolean expressions, not just those used directly in control flow decisions.

4.2.5 Multiple Condition Testing

With this test technique we test combinations of condition outcomes. To get 100% multiple combination coverage we must test all combinations of outcomes of all conditions.

Because there are two possible outcomes for each condition (True and False) it requires 2^n test cases, where n is the number of conditions, to get 100% coverage.

Ex.

If we take the example from above:

```
if (X or (Y and Z)) then ..
```

we have three conditions. We therefore need 23 = 8 test cases to get 100% multiple condition coverage.

The test cases we need are:

Test case	X	Y	Z
1	True	True	True
2	False	True	True
3	False	False	True
4	False	False	False
5	False	True	False
6	True	False	False
7	True	True	False
8	True	False	True

The number of test cases grows exponentially with the number of conditions! This is a very thorough test, even though it may happen that some of the test cases are impossible to execute.

Sometimes the concept of "optimized expressions" jeopardizes the measurement of this test technique. Optimized expressions mean that a compiler short-circuits the evaluation of Boolean operators. In for example the programming language C the Boolean "AND" is always short-circuited: the second operand will not be evaluated when the outcome can be determined from the first operand.

Short circuits present no obstacle to branch condition coverage or condition determination coverage, but there may be situations where it is not possible to verify multiple condition coverage.

4.2.6 Condition Determination Testing

Sometimes testing to 100% multiple condition coverage would be to go overboard in relation to the risk associated with the component.

In these cases we can use the condition determination testing technique. With this technique we should design test cases to execute branch condition outcomes that independently affect a decision outcome. This is a pragmatic compromise where we discard the conditions that do not affect the outcome.

The number of test cases needed to achieve 100% condition determination coverage depends on how the conditions are combined in the decision statements:

- As a minimum we need $n+1$ test cases
- As a maximum we need 2^n test cases

where n is the number of conditions.

Ex.

Still working with the same example as before:

if (X or (Y and Z)) then ..

we need the test cases listed in this table.

Test case	X	Y	Z
1	True	-	-
2	False	True	True
3	False	False	-
4	False	True	False

A "-" means that we don't care about what the outcome is, because it does not have impact on the result.

We can hence get 100% modified condition decision coverage or condition determination coverage with just four test cases.

4.2.7 LCSAJ (Loop Testing)

Sometimes a program needs to do the same thing a number of times with different values. Instead of having to repeat the same piece of code several times, coding languages allow loops, that is repetitive execution of the same statements with different values for the variables.

The exact decision statements that form loops depend on the coding language. The statements in a loop are called the loop body. For some types of loops the loop body will always be executed at least once; for others it may be skipped altogether depending on the conditions of the looping.


```
Read A;
B = 0;
while A <= 245 do
 B = B + 1;
 A = A + 1;
end while;
Write A;
Write B;
```

This little bit of code shows a so-called while loop.
The loop is executed as long as the value of A is less than or equal to 245, and for each loop the value of both A and B will be augmented by 1.

LCSAJ testing is a test case design technique where loops are identified and test cases developed to test linear sequences of code that start at a specific point in the code and end with a jump (or at the end of the component). Such a sequence is called a LCSAJ (Linear Code Sequence And Jump). It may also be called a DD-Path (Decision-to-Decision Paths).

An LSCAJ is defined by

- The start of the linear sequence of executable statements
- The end of the linear sequence
- The target line to which the control flow is transferred at the end of the linear sequence

The three items in an LSCAJ are usually identified by their line numbers in the source code listing.

An LCSAJ starts either from the start of a component or from a point to which control flow may jump from other than the preceding line. An LCSAJ terminates either by a specific control flow jump or by the end of the component. LCSAJs may go forwards in the code, or they may go backwards if the start point is somewhere down the code and the end point at a higher point.

LCSAJ coverage is the percentage of LCSAJs in a component that are exercised by the test.

To design the test cases using this technique we must:

- Identify each code line by its number
- List the branches (maybe with a note of the necessary conditions to satisfy them)
- From this list, find the LCSAJ start points
- Derive the LCSAJ from each of the start points

For each LCSAJ and thereby possible test case we therefore identify the start line, the end line, and the target line for the jump.

Sometimes some reformatting of the code may be needed if the coding standard used does not support this technique. The basic formatting rule that must be observed is that each branch has to leave from the end of a line and arrive at the start of a line.

From BS–7925

This LCSAJ example is taken from BS-7925. We will base it on the component shown here.

```
1.  READ (Num);
2.  WHILE NOT End of File DO
3.      Prime := TRUE;
4.      FOR Factor := 2 TO Num DIV 2 DO
5.          IF Num - (Num DIV Factor)*Factor = 0 THEN
6.              WRITE (Factor, ' is a factor of', Num);
7.              Prime := FALSE;
8.          ENDIF;
9.      ENDFOR;
10.     IF Prime = TRUE THEN
11.         WRITE (Num, ' is prime');
12.     ENDIF;
13.     READ (Num);
14.  ENDWHILE;
15.  WRITE ('End of prime number program');
```

The code lines have been defined by line numbers. The next step is to list the branches.

Branch	Type	Condition
(2 -> 3)		Requires not end of file
(2 -> 15)	: Jump	Requires end of file
(4 -> 5)		Requires the loop to execute
(4 -> 10)	: Jump	Requires the loop to be a zero-trip
(5 -> 6)		Requires the if in line 5 to be true
(5 -> 9)	: Jump	Requires the if in line 5 to be false
(9 -> 5)	: Jump	Requires a further iteration of the for loop
(9 -> 10)		Requires the for loop to have exhausted
(10 ->11)		Requires prime to be true
(10 -> 13)	: Jump	Requires prime to be false
(14 -> 2)	: Jump	Always has to take place

The start points we can identify are the lines: 1, 2, 5, 9, 10, 13, and 15. The start points are sorted here; not listed in the order they are found in the table.

We must now derive the LCSAJs from each of the start points. With start point in line 1, we get the following LCSAJs:

(1, 2, 15) (1, 4, 10) (1, 5, 9) (1, 9, 5) (1, 10, 13) (1,14, 2)

We will have to work our way through the list of starting points finding all the LCSAJs. The last one is (15, 15, exit).

When we design test cases to execute we must cover enough LCSAJs to get the coverage we want. It will almost always be so that each test case covers a number of LCSAJs.

It can also happen that some LCSAJs are impossible to execute. This must be handled when defining the completion criteria based on LCSAJ coverage.

There are a number of classic pitfalls connected with loops. One defect to watch out for in loops is the creation of an infinite loop, that is the case where a defect causes the loop to (theoretically) continue looping for ever. Examine the loop shown above and imagine what would happen if the statement "A = A + 1;" was omitted in the loop. This would be evident in such a small piece of code, but loops may be quite long and complicated, and infinite loops are seen now and again.

Another issue is the sequence of the statements in the loop body. There is almost always an issue about doing something the first time around and/or the last time around in a loop. Again consider the example above: Will the result be the same if B is set to 1 before the loop is started? And what if the condition is "<" instead of "<="?

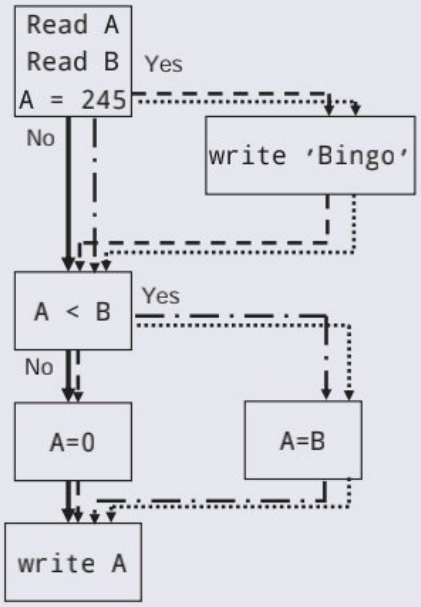
4.2.8 Path Testing

A path is a sequence of executable statements in a component from an entry point to an exit point.

Path coverage is the percentage of paths in a component exercised by the test cases.

When you execute test cases based on any of the other structure-based techniques described above, you will inevitably execute paths through the code.

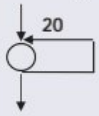
This example shows the six possible paths from beginning to end through the tiny bit of code.



This example shows the six possible paths from beginning to end through the tiny bit of code.

The number of possible paths through a component is exponentially linked to the number of decisions, especially decisions involving loops.

The control flow diagram shown here for a bit of code includes a loop that may be exercised up to 20 times. Such a loop only gives



20 + 19 + … + 1 possible different paths through the code.

In practice it may easily become impossible to obtain 100% path coverage when loops are involved.

Path testing includes a bit of error guessing. Experience shows that it can be useful to test these types of paths:

- Minimum path
- Path with no execution of any loop(s)
- Minimum path + one loop once
- Path with one loop a number of times
- Path with one loop the maximum number of times

where possible.

4.2.9 Intercomponent Testing

Note: This technique is not part of the ISTQB syllabus, but included here because I find it is very useful—especially since integration testing often is overseen and difficult to come to grips with.

The idea in structured design, object-oriented design, and most other design paradigms is that the functionality is distributed on a number of components and/or systems for ease of production and maintenance.

This means that interfaces exist between interacting components and systems. At the component level we say that components call functions in each other, or, as it is expressed in object-oriented design, use methods that other classes make available. At the system level other types of interfaces exist, such as software-to-hardware integration, software-to-network integration, or software-to-manual procedure integration.

In the following the concept of function calls will be used, but the idea is exactly the same for method usage, and system interfaces.

The intercomponent testing technique is used in integration testing where the test objects are these interfaces. The integration comes after the component or system testing and assumes that logical and other types of defects in the body of the component or system have already been found.

The intercomponent testing is based on the design of the interfaces. For each function we should be able to find a design description of:

- Input: The input parameters required by the function
- Functionality (which we are not interested in at this point)
- Output: The resulting output parameters produced by the function

What we need to test are the calls to the functions, that is the interfaces. The coverage element is calls, and the intercomponent coverage is the percentage of the total number of calls that we have covered in a specific integration test.

To identify the total number of calls made by all the components or systems being integrated, we have to count, from the design, how many times each function is called. This is called fan-in: number of calls of a specific function from other functions (or the main program). This may be calculated using a static analysis tool; see Section 9.3.4. The sum of all the fan-ins provides the total number of calls.

For software components it is very difficult to say anything in general about the values for fan-ins. In system integration we normally have a very low fan-in, often only one, for each system.

Test cases are designed to cover the calls.

We may combine the intercomponent testing with other techniques to get a more thorough coverage of the input and/or output parameters, for example equivalence partitioning and boundary value analysis for constraints on the parameters.

4.3 Defect-Based Techniques

In defect-based testing we are looking at the types of defects we might find in the product under testing. The techniques are therefore starting from previous experience, rather than the expected functionality or the structure of the test product.

The techniques may therefore be less systematic than the previously discussed techniques, since it is usually not possible to make exhaustive collections of expected defects. The determination of defect-related coverage is hence also less comprehensive: Since there is no absolute amount of expected defects, only what we have chosen or selected as expected defects, the coverage is relative to that number.

The defect-based techniques covered here are:

- Taxonomies
- (Fault injection and mutation—not part of the ISTQB syllabus)

4.3.1 Taxonomies

A taxonomy is an ordered hierarchy of names for something. In this context it is an ordered hierarchy of possible defect types.

Such a taxonomy is a sort of checklist over defects to look for, and it is used to design test cases aimed at finding out if these defects are present in the product under testing.

Many people have worked on defect taxonomies, starting with Beizer's bug taxonomy defined in the late 1980s. This taxonomy is quite comprehensive and lists possible defects in a four-level hierarchy with identification numbers. The first two levels are:

1 Requirements
- 11 Requirements incorrect
- 12 Requirements logic
- 13 Requirements, completeness
- 14 Verifiability
- 15 Presentation, documentation
- 16 Requirements changes

2 Features and Functionality
- 21 Feature/function correctness
- 22 Feature completeness
- 23 Functional case completeness
- 24 Domain bugs
- 25 User messages and diagnostics
- 26 Exception conditions mishandled

3 Structural Bugs
- 31 Control flow and sequencing
- 32 Processing

4 Data
- 41 Data definition and structure
- 42 Data access and handling

5 Implementation and Coding
- 51 Coding and typographical
- 52 Style and standards violation
- 53 Documentation

6 Integration
- 61 Internal interfaces
- 62 External interfaces, timing, throughput

7 System and Software Architecture
- 71 O/S call and use
- 72 Software architecture
- 73 Recovery and accountability
- 74 Performance
- 75 Incorrect diagnostics, exceptions
- 76 Partitions, overlay
- 77 Sysgen, environment

8 Test Definition and Execution
- 81 Test design bugs
- 82 Test execution bugs
- 83 Test documentation
- 84 Test case completeness

This taxonomy covers the entire development life cycle and may also be useful as a checklist for early static test, for example of requirements and design, as well as for static tests of the tests, specification.

Another defect taxonomy is given in IEEE 1044 in the categorization for incidents to be provided during the investigation phase of an incidents life cycle. IEEE 1044 is discussed in detail in Chapter 7. This taxonomy has up to three levels, of which only the first is provided here with the corresponding codes:

IV310	Logical problem
IV320	Computation problem
IV330	Interface/timing problem
IV340	Data-handling problem
IV350	Data problem
IV360	Documentation problem
IV380	Document quality problem
IV390	Enhancement
IV398	Failure caused by fix
IV399	Performance problem
IV400	Interoperability
IV401	Standards conformance
IV402	Other problem

Taxonomy testing is not a terribly effective test technique, especially not if the taxonomy used is a standard one, not taking the nature of the specific development process and product into account.

The taxonomies shown here, and others to be found in the testing literature are, however, very useful as starting points for making your own taxonomy or checklist of possible defects.

The coverage element for taxonomy testing is the listed defects and the coverage is calculated as the percentage of these used for designing test cases.

4.3.2 Fault Injection and Mutation

Note: This technique is not part of the ISTQB syllabus, but included here because I sometimes find it useful.

Fault (or defect) injection and mutation are a type of technique where the product under test is changed in controlled ways and then tested. This technique type is used to assess the effectiveness of the prepared test cases, rather than actually looking for defects.

Fault injection is also known as fault seeding or bebugging. In this technique defects are deliberately inserted into the source code, either by hand or by the use of tools.

The defects inserted may be inspired by a checklist or a defect taxonomy. The product is then tested, and it is determined how many of the injected defects are found and how many other defects are found. These numbers are used to estimate how many real defects are still left in the product.

Ex.

In a set of components 50 defects are injected prior to component testing. The component testing reveals

Injected defects: 26 and New defects: 83

Based on this it is estimated that there remain 77 defects in the components and more tests should be designed, if this is not acceptable.

In mutation testing so-called one-token defects, such as "<" replaced with "<=," are made in the components. For each of these defects a new version of the affected components is created and tested to see if the prepared test cases reveal the defects. If they don't they will have to be examined and corrected to find the planted defects—and one hopes, more real defects as well.

The coverage element for this type of testing technique is the injected defects and the coverage is calculated as the percentage of these found.

The drawback of the fault injection and mutation technique type is that the inserted defects are not necessarily realistic. It can also be a fairly big task to define and inject defects compared to the results to be gained.

It must also be noted that even though the technique type helps us identify more defects of the inserted types, there is a number of defects where it is of no use, for example, defects caused by omissions in the code or misunderstandings of the requirements.

Fault injection may also be applied to data, in the sense that data may be edited to be wrong compared to the expected data.

It is absolutely essential when using fault injection and mutation that a good configuration management system is in place. It must be clear what the "correct" code is and what code has been deliberately changed.

4.4 Experience-Based Testing Techniques

Faults are sly!

No matter how well we use the test case design techniques we cannot catch all the faults. There are many reasons for this.

One is that not all failures occur every time the same action is performed. Sometimes a failure only occurs when we have performed the same action several times.

Ex.

I use home banking when I pay my bills. I enter the details for a bill, hit "OK," and then the details are presented in a form for my endorsement, and I can go on to the next bill. But, if I have more than eight bills to be paid at the same time, a failure occurs. For the ninth bill the endorsement form is blank! I endorse anyway, however, because the endorsement itself still works.

Another reason why we can miss faults is something called coincidental correctness. We may happen to choose an input, for example, when choosing an input in an equivalence partition that does not reveal a fault.

An illustration of this is a test of the formula "nn" (n to the power of n). Unfortunately the programmer has misunderstood the formula and implemented it as "n+n." If we happen to choose the input value of 2, the expected result is 4 (2 x 2 = 4)—but alas, 2 +2 also equals 4.

We also need to be aware that identical functionality may or may not be implemented identically. Many are the systems where, for example, date handling has been implemented by different implementations groups for different subsystems.

Furthermore, rare or fringe situations may be overlooked or deliberately left out in the specification of the structured test cases.

The sum of all this is that systematic testing is not enough! Since faults are sly we have to attack them in unpredictable ways.

This is where the nonsystematic testing techniques come in as a valuable supplement to systematic testing techniques. The nonsystematic techniques to be discussed here are:

- Errorguessing
- Checklist-based
- Exploratory testing
- Attacks

These techniques may be used before the systematic techniques to assess test readiness by uncovering "weak" areas. This can also be used as the input to initial risk analysis. The techniques may also be used after the systematic testing as a final "mopping-up," hopefully providing extra confidence.

Nonsystematic testing techniques must NEVER be the only technique to be used.

4.4.1 Error Guessing

Error guessing is a test technique where the experience of the tester is used to anticipate what faults might be present in the test object as a result of errors made, and to design tests specifically to expose them.

This means that the tester uses his or her experience gained from the structured tests that have already been executed or from other test assignments to guess where faults may remain in the test object.

The tester has to think creatively—out of the box, over the borders, around the corners—both in relation to how the structured tests have been structured, how the test object has been produced, and the nature of the faults already found.

In relation to the testing approach we could try to find alternative approaches, and ask ourselves:

- How could this be done differently?
- What would it be completely unlikely to do?
- What assumptions may the testers, who performed the structured tests, have made?

In relation to how the test object has been produced, we could ask ourselves:

- What assumptions may the analysts or the programmers have made?
- What happens if I use a value of 0 (both input and output)?
- What about cases of "none" and "one" in lists?
- What happens if I go over the limit?
- Are there any cases of "coincidence," for example, same value twice or same value for all?

The faults we have already found can be exploited to spur new ideas. We can for example ask ourselves:

- Are there other faults like this?
- Are there any reverse faults?
- Are there any perpendicular faults?

All these questions and many more can be assembled and maintained in checklists.

The coverage for error-guessing testing is related to the tester's experience base and not easily documented.

4.4.2 Checklist-Based

Checklists can hold lists of possible faults that are known to escape the systematic test. They are formed and maintained by experience—lessons learned from previous projects; and they are a valuable asset in an organization. Checklists may be used for designing both static and dynamic tests, and they can be used as a good starting point for for example risk identification or error guessing brain storms.

Special checklists may be defect taxonomies or rules derived from standards, for example, an internal standard for user interfaces.

Coverage for checklist-based testing is related to the contents of the checklists used. The coverage for a specific test may be calculated as the percentage of the items in the list(s) covered by the test.

An example of a checklist is the list shown below for CRUD testing. The

abbreviation CRUD refers to the life cycle of data entities in a product, namely:

- Create
- Read
- Update
- Delete

CRUD testing, that is testing all the data entities according to their life cycles, is important.

The life cycles for data entities are usually not sufficiently specified in the data requirements. CRUD testing therefore starts with helping the analysts to specify sufficient requirements. The actual CRUD testing is sometimes on the verge of experience-based testing, because it may be based on previous experience of where CRUD faults appear and on related checklists.

The CRUD checklists provided below may be used as inspiration both for defining data requirements and for guiding CRUD testing.

CRUD checklist examples are shown here:

Concerning *creation of data* we may ask:

- Is it possible to create the first?
- Are the contents correct?
- Is it possible to create a new among existing?
- Are the contents correct?
- Is it possible to create the last (the highest number)?
- Are the contents correct?
- Is it possible to create more than what is allowed?
- Is it possible to create if you are not allowed to create?

Concerning the *reading of data* we may ask:

- Is it possible to read the created data?
- Is it possible to find the data in all the ways it is supposed to be found?
- Is it impossible to read data if you are not allowed?

Concerning the *updating of data* we may ask:

- Is it possible to change where it should be possible?
- Is everything saved?
- Is the change reflected everywhere?
- Is it impossible to change in places where it should be impossible?

Concerning the *deletion of data* we may ask:

- Is it possible to delete where it should be possible?
- Is everything deleted?
- Is it possible to delete all that should be deletable?
- Do all deletes have the correct cascading effects?
- Is data that should not be deleted properly protected?

4.4.3 Exploratory Testing

Sometimes it is worthwhile to search in a structured way and use the results to decide on the future course as they come in. This is the philosophy when people are looking for oil or mines, and it is the philosophy in exploratory testing.

Exploratory testing is testing 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. In other words exploratory testing is simultaneous:

- Learning
- Test design
- Test execution

Exploratory testing is an important supplement to structured testing. As with all the nonsystematic techniques it may be used before the structured test is completely designed or when the structured test has stopped.

It is important that the course of the exploratory testing is documented, so that we can recall what we have done. Imagine if drillings for oil were not documented, or if the search for mines in a field were not documented—it would be a waste of time and money. The idea in exploratory testing is not that it should not be documented, but that it should be documented as we go along.

Exploratory testing is not for kids!—nor for inexperienced testers.

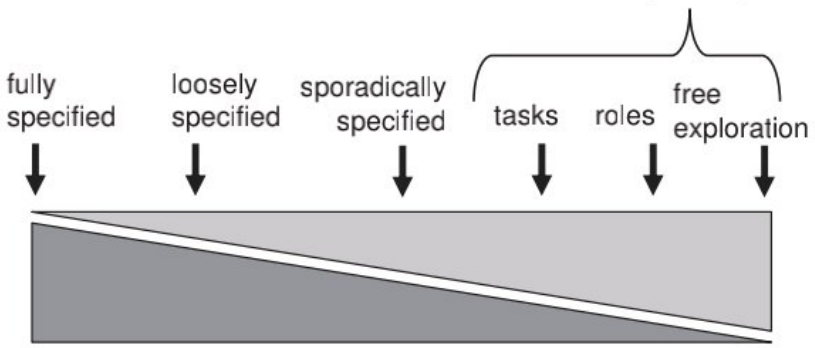
Extensive testing experience and knowledge of testing techniques and typical failures are indispensable for the performance of an effective exploratory test. It is also an advantage if the tester has some domain knowledge.

The exploratory tester needs to be able to analyze, reason, and make decisions on the fly; and at the same time have a systematic approach and be creative. The tester also needs some degree of independence in relation to the manufacturing of the system—the programmer of a system cannot perform exploratory testing on his or her "own" system.

Perhaps most importantly the exploratory tester must have an inclination towards destruction. Exploratory testing will not work if the tester is "afraid" of getting the system to fail.

4.4.3.1 Degrees of Exploratory Testing

One of the forerunners in exploratory testing is the American test guru James Bach. He has defined various degrees of exploration as illustrated in the figure below.



Furthest to the right, we have the totally free exploration. The tester simply sits before the system and starts wherever he or she feels like.

A step to the left, we find the exploratory testing guided by roles. Here the tester attacks the system under testing assuming a specific user role. This could, for example, be the role of an accountant, a nurse, a secretary, an executive manager, or any other role defined for the system. This provides a starting point and a viewpoint for the testing, which is exploratory within the framework of the role.

Even further to the left is the exploratory testing guided by a specific task. Here the tester narrows the framework for the testing even more by testing within the viewpoint of a specific task defined for a specific role for the system under testing.

On the borderline between exploratory testing and structured testing we have the sporadically specified test. Here the tester has sketched the test beforehand and takes this as the starting point and guideline for the performance of the exploratory testing.

4.4.3.2 Performing Exploratory Testing

No matter which degree of exploration we use, we have to follow the principles in the general test process. We must plan and monitor; we must specify, execute, and record; and we must check for completion.

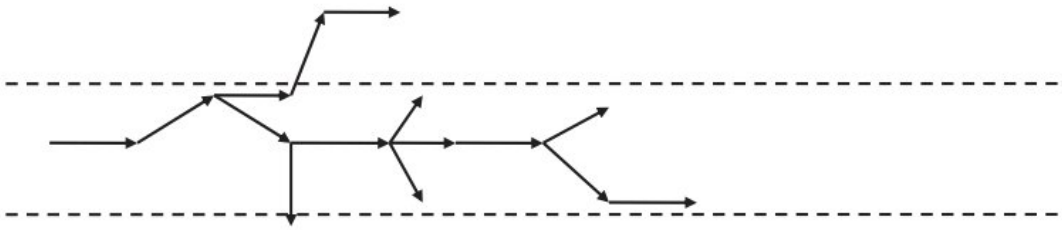
In the planning we consider what we are going to do and who is going to do it. We must choose the degree of exploration and describe the appropriate activities. The testing activities should be divided into one-hour sessions. If the sessions are shorter we risk not getting an effective flow in the exploration; if they are longer we get tired and the effectiveness goes down.

It is important to make sure that the tester or testers are protected during the sessions. There should be no phones or other interruptions to disturb the

flow of the testing.

The test specification, execution, and recording are done simultaneously during the exploratory testing session. Within the given degree of exploration the tester should allow him- or herself to get distracted—you never know what you may find.

The course of the testing session may be illustrated like this:



Stock must be taken from time to time to verify that we are on track. For each session we must:

- Take extensive notes and attach data files, screen dumps, and/or other documentation as appropriate
- Produce an overview over findings
- Reprioritize the remaining activities

The exploratory testing can stop when we have fulfilled our purpose.

4.4.3.3 Exploratory Testing Hints

There are a few weaknesses in exploratory testing, of which we need to be aware. These weaknesses are part of the reasons why exploratory testing must be a supplement only to structured and specified testing.

The *weaknesses in exploratory testing* are:

- Exploratory testing does not support automated test, and hence regression testing, very well.
- Because the exploratory testing is not specified in advance it cannot provide feedback to design before the design is actually implemented.
- Even when accompanied by extensive notes there is usually no firm documentation of test coverage for the exploratory testing.
- Exploratory testing can be very difficult to use for complex functionality; the main thread may easily be lost.
- Note taking is very difficult when working interactively.

The last weakness may be overcome by performing exploratory testing as *pair testing*. This can work really well and it has a number of benefits. Pair work sparks more ideas as the two testers inspire each other, and it enables mutual learning—even when the testers have different levels of experiences.

Extroverted people get more energy from working together, and others are less likely to interrupt a pair.

The biggest benefit is perhaps that two testers can be focused on two tasks at the time: One can follow an idea, and the other can take notes. The focuses should switch between the two testers at regular intervals.

There are of course also a *few disadvantages to pair testing*. The main one is, that a divided responsibility is no responsibility—if two people share a responsibility they carry 2% each.

As some people are extroverted, some people are introverted. Introverted people get drained for energy by working closely together with others.

If the difference in experience is too large it may cause "abandonment" by the lesser experienced, because he or she may simply opt out. On the other hand different opinions held by equally "strong" testers may block progress.

4.4.4 Attacks

The attacks technique is a form of security testing, testing how resistant a product is to those who want to break into it in various ways and for various reasons, ranging from incidental mistakes, over "fun," to serious crime.

Security testing is getting more and more powerful and sophisticated. As the market for e-commerce and e-business is growing and more and more other applications get Web access, the need for secure systems is growing. Even so, we are still constantly at least one step behind, and checklists of attacks are very valuable both to analysts defining requirements and to testers.

A product is vulnerable at the places where there is an opening into it; that is where the product has interfaces. The interfaces a product can have include, but are not limited to:

- User interface
- Operating system
- API (application programming interface)
- Data storage for example file system

Attacks are used to find areas where the product will fail due to misuse or defects in these interfaces.

James Whittaker is one of the pioneers of attacks, and he has created long lists of useful attacks. A few examples are listed here, grouped by type:

User interface attacks:

- Apply inputs that force all the error messages to occur
- Apply inputs that force the software to establish default values
- Explore allowable character sets and data types
- Overflow input buffers

Stored data attacks:

- Apply inputs using a variety of initial conditions
- Force a data structure to store too many or too few values
- Investigate alternate ways to modify internal data constraints

Media based attacks:

- Fill the file system to its capacity
- Force the media to be busy or unavailable
- Damage the media

Other ways of attacking a product may be trying to make unauthorized

- Access to and control over resources, such as restricted files and data
- Execution of programs or transactions
- Access to and control over user accounts
- Access to and control over privilege management
- Access to and control over network management facilities

The advantage of using attack-driven testing is that security holes can be closed before they are found by attackers. A pitfall is that this may create a false sense of security. There is no end to the imagination of those who want to do wrong, and we have to stay constantly alert to new weak spots. This is why checklists of any kind must be kept up-to-date with new experiences.

There is more about security testing in Sections 5.1.4 and 5.2.2.

4.5 Static Analysis

Static analysis is a testing type where, in contrast to dynamic testing, the code under static analysis is NOT executed.

Static testing, especially of code, is usually performed using tool(s), but the only thing being executed during static analysis is the tool. Static test tools are discussed in Section 9.3.4.

Traditionally static testing has been performed on code, but here it is expanded to architecture as well.

4.5.1 Static Analysis of Code

Many tools dedicated to static analysis are available on the market or as open source systems. Their capabilities vary a lot and depend very much on the coding language. Some standard development tools, such as compilers or linkers, are able to perform limited static analysis.

The static analysis techniques for code discussed here are:

- Control flow analysis
- Data flow analysis
- Compliance to standards
- Calculation of code metrics

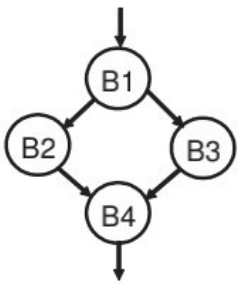
4.5.1.1 Control Flow Analysis

A control flow is an abstract representation of all possible sequence of events (paths) in the execution through a component or a system. The control flow is the basis for many of the structure-based case design techniques described earlier.

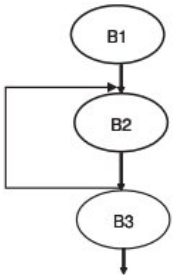
Control flow goes through basis blocs or nodes of code being executed as an entity, with an entry point in the beginning and only there, and with an exit point in the end and only there. The control goes from the starting point and is transferred from one block to another to the end.

It can be very useful to draw a control flow graph from the code (or the requirements). It gives an overview over the decisions points, branches, and paths in a piece of code, and its inherit complexity.

Two simple extracts of control flow graphs are shown here,



More basic blocks in parallel



More basic blocks in parallel

Static analysis tools may be used to draw control flow graphs of larger pieces of code. It is, however, a good idea to train drawing these, because they give a good understanding of how the code is structured.

A control flow graph does not necessarily give an idea of what the code is actually doing, but that is not important anyway. We as testers should concentrate on the structure, not the functionality.

Static analysis tools can find faults in the control flow, typically:

- "Dead" code (i.e., code that cannot be reached during execution)
- Uncalled functions and procedures

Both dead code and uncalled functions are quite often found in legacy systems. Undocumented changes and corrections have caused some code to be circumvented but not removed. The cause of the change is since forgotten, but nobody maintaining the code has had the courage (or initiative) to get the unused code removed.

Such unused areas do not present direct risks. They do however disturb maintenance, and they may unintentionally be invoked with unknown consequences.

4.5.1.2 Data Flow Analysis

Data flows through the code; that is what IT is all about. The normal life cycle for a data item, a variable, consists of the phases:

- Declaration—Space is reserved in memory for the variable value
- Definition—A value is assigned to the variable
- Use—The value of the variable is used
- Destruction—The memory set aside for the value of the variable is freed for others to use

Some static analysis tools can find anomalies in the data flow in relation to the variable life cycle.

Anomalies may, for example, be:

- Use before declaration
- Use before definition
- Redefinition before use
- Use after destruction

Any of these anomalies should set the alarm clocks ringing, as it may be a sign of something being wrong.

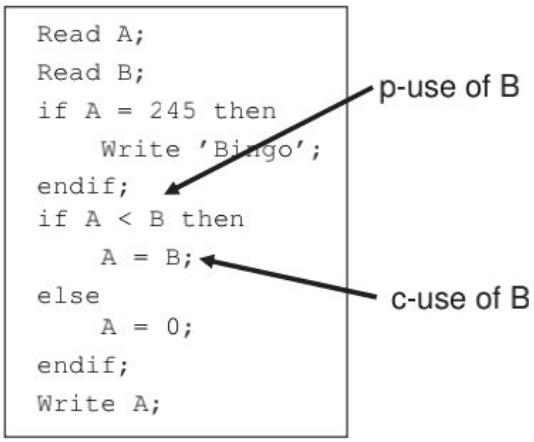
If a variable is defined and then redefined before use, is it because the first or the second definition is superfluous, or is it because a usage statement is missing?

Some programming languages permit variables to be used before they are declared; in this case the first usage will be treated as an implicit declaration.

The phases "definition" and "use" may be repeated many times during the life of a variable.

Two different kinds of usage are defined in data flow analysis:

- Computation data use = c-use = data not used in a condition
- Predicate data use = p-use = a data use associated with the decision outcome of the predicate portion of a decision statement (predicate = condition = evaluates to T or F)



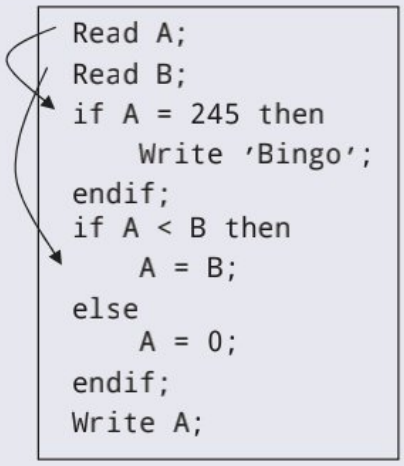
A subpath in the flow is defined to go from a point where a variable is defined, to a point where it is referenced, that is, where it is used—whatever kind of usage it is. Such a subpath is called a definition-use pair (du-pair). The pair is made up of a definition of a variable and a use of the variable.

Because there are two kinds of usages, there are three types of du-pairs, namely:

- Definition –> c-use
- Definition –> p-use
- Definition –> use (either c or p)

This example shows two du-pairs:

- One definition to p use
- One definition to c use



Data flow testing is testing in which test cases are designed based on variable usage within the code, that is, testing of du-pairs.

The coverage items for this test case design technique are the control flow sub paths and full paths through the code. The following table gives an overview of the types of coverage defined for data flow testing.

	Name	Definition
d	All-definition coverage	percentage of covered subpaths from each variable definition to some use of that definition
duc	All data definition c-use coverage	percentage of covered subpaths from each variable definition to every c-use of that definition
dup	All data definition p-use coverage	percentage of covered subpaths from each variable definition to every p-use of that definition
u	All use coverage	percentage of covered subpaths from each variable definition to every use of that definition (despite the type)
du-path	All definition use path coverage	percentage of covered "simple subpath" from each variable definition to every use of that definition

To design test cases with the data flow test technique we concentrate on one component and for that component we must:

- Number the lines, if they are not numbered already
- List the variables
- List each occurrence of each variable and assign a category (definition, p-use, or c-use)
- Identify the du-pairs and their type (c-use or p-use)
- Identify all the subpaths that satisfy the pair
- Derive test cases that satisfy the subpaths

When the final test procedures are designed from the test cases, they will of course follow a path in the control flow through the code.

There are a few pitfalls that we need to be aware of in connection with data flow analysis. Data items (variables) may be composed of a number of single variables. This is, for example, the case for arrays, which are ordered sequences of individual data items. If we ignore the constituents of composite variables like arrays and treat them as one data item, we greatly reduce the effectiveness of data flow testing. Tools will typically respond to an array or record it as a single data item rather than as a composite item with many constituents.

An important problem concerning data flow analysis is that sometimes static analysis tools will report a large number of anomalies that are not in

fact caused by anything being wrong. These "false alarms" may hide the real problems in the sheer number of alarms.

4.5.1.3 Compliance to Coding Standard

A coding standard is a guideline for the layout of the code being written in an organization.

Most static analysis tools can check for standard coding style violations, for example, missing indentation in IF statements. Some of the more sophisticated tools allow the definition of specific coding standards to check for.

It may seem trivial that coding standards should be defined and adhered to, but experience shows that it has several advantages, such as:

- Fewer faults in the code, because a good layout of the code enables the programmer to keep an overview of what he or she is writing
- Easier component testing, because a well-structured code is easier to define test cases for when using white-box techniques
- Easier maintenance, because it is faster for others to overtake code if all code is written in an identical style

There are no disadvantages in requiring adherence to a coding standard, other than the programmers having to get used to it. It is even possible to get tools to format "rough" code according to coding standards.

4.5.1.4 Calculation of Code Metrics

Static analysis tools can provide measurements of different aspects of the code like:

- Size measures—Number of lines of code, number of comments
- Complexity
- Number of nested levels
- Number of function calls—Fan-out

Lines of code (LOC) can be reported as can lines of document lines. The ratio between these may be calculated as a derived measurement.

Measurements related to code lines may be difficult to work with. The sheer definition of a "code line" is not always as simple as it may sound. Numbers of code lines may also easily be manipulated by the programmers, if the measurements are used to quantify their efficiency.

Code complexity is a measure for how "tangled" the code is. The complexity is related to the number and types of decisions in the code. It is interesting from a testing point of view because it has an impact on the test effort: the higher the complexity, the more test cases to get high decision and condition coverages.

The most commonly used complexity measure is McCabe's Cyclomatic Complexity. It was introduced by Thomas McCabe in 1976 and is often simply referred to as complexity, as CC or as McCabe's complexity.

McCabe's Cyclomatic Complexity is a measure - a single ordinal number— of soundness of components. It measures the number of linearly independent paths through the code. It is intended to be independent of language and language formats.

The original definition of McCabe's Cyclomatic Complexity is:

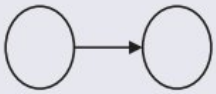
MCC = (L – N) + 2*B

where

L: Number of branches (lines in the flow graph)

N: Number of sequential blocks (basis blocks or nodes)

B: Number of broken sequences (this is used when we measure for more than one component. This is very rarely done, so B is usually = 1)



In this simple flow graph we have
L = 1 N = 2 and B = 1 and therefore
MCC = (1 – 2) + 2 x 1 = 1

For this flow graph

we have L = 4 N = 4 and B = 1, and therefore
MCC = (4 – 4) + 2 x 1 = 2

There are more simple ways of calculating McCabe's Cyclomatic Complexity:

One way is based on a count of decisions and decision outcomes. It could be expressed like this

MCC = (SUM (*branches* –1) for all decisions) + 1

IF statements always have two branches, so if we only have IF statements in the code we are looking at, we'll get

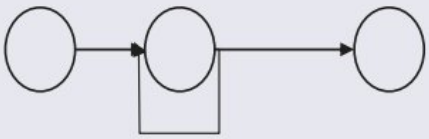
MCC = SUM(1) for all IF statements)) +1 = number of IF statements +1.

Constructions like CASE have more branches, so if we have a piece of code with two IF statements and 1 CASE statement with 10 possible outcomes, we would have MCC = 1 + 1 + 9 + 1.

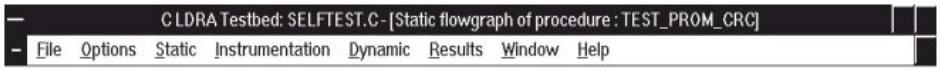
The third way of calculating McCabe's Cyclomatic Complexity is by counting so-called regions in the code. A region is a closed area found in the flow graph.

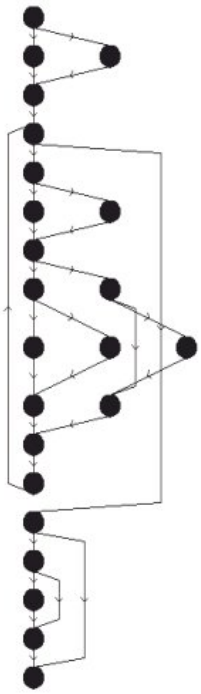
The expression is
MCC = *number of regions* + 1

In the flow graph shown here, we have 1 region and we therefore have MCC = 2.



The graph shown below is created by a static analysis tool. The tool has calculated the McCabe's CC for the component to be 9.





McCabe's Cyclomatic Complexity is a good indicator for the test effort. Mathematical analysis has shown that it gives the exact number of tests needed to test every decision point in a program for each outcome.

Some testers have experienced that there is a positive linear connection between the number of faults and the McCabe's CC (i.e., that the higher the CC the higher the number of faults). Others have found that the connection is rather that a low CC and a high CC indicate relatively fewer faults, while more faults are found in the areas where the CC is around average. Try to get your own measurements for this!

A rule of thumb says that the McCabe's CC should not be more than 10. If we find a higher CC in a component it should be revisited—maybe it could be restructured or divided. McCabe's CC measures have however been seen up to more than 3,000, so we should also use visual judgment of the control flow graphs to determine the test effort.

Cyclomatic complexity can be used for other purposes than test planning, for example, in risk analysis related to:

- Code development
- Implementation of changes in maintenance
- Reengineering of existing systems

Other complexity measures than McCabe's exist. They are, however, rarely used, except perhaps for the Halstead complexity measure, an algorithmic complexity measure.

4.5.2 Static Analysis of Architecture

In the recent testing literature the static analysis objects have been expanded to include the architecture of the product as well as the classic code object.

The code has a structure that can be analyzed, and so has the product or system as a whole. This may, for example, be:

- Structure of components in the product calling/using each other
- Menu structure of a graphical user interface for a product
- Structure of pages and other features in a Web product

The first and the second of these types may be tested with the help of a tool. We can therefore say that static analysis may be used to test these aspects of products.

4.5.2.1 Static Analysis of a Web Site

A Web site is a hierarchical structure composed of Web pages with elements such as tables, text, pictures, and links to other pages, both internally to the Web site itself and to external Web sites. The structure of a Web site is described in HTML: Hyper Text Mark-up Language.

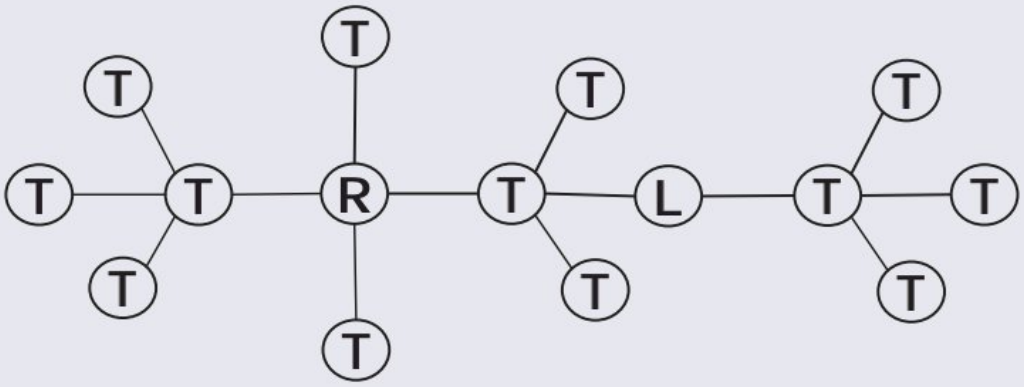
Many tools exist to analyze the HTML code and the structure and detailed composition of Web sites, that is to perform static analysis of Web sites.

The stakeholders for the static analysis information are both analysts, designers, testers, and those in charge of the maintenance of the Web sites, the Web masters.

One of the results of such an analysis is a graph showing the structure of the Web site. A graph can provide an overview of the tree structure or hierarchy of the site and show the depth, complexity, and balance of the structure.

Ex.

A very simple Web site is shown here. The R is the root, a T illustrates a non-link tag, for example a table, and an L illustrates a link.



The graphs can be very colorful and look like the most amazing fireworks. The information extracted from the graphs can be used to:

- Test whether requirements for the Web site, for example depth and balance of the Web site, have been fulfilled, or if the Web site needs to be restructured
- Estimate test effort
- Assess usability
- Asess maintainability

A rule of thumb is that the more balanced the Web site tree is, the lower the test and maintenance efforts are going to be, and the higher the usability and vice versa.

Web sites are extremely dynamic products. The static analysis of a site prior to its release is only the tip of the iceberg of testing and monitoring of a Web site. Dynamic analysis of the behavior of a site should be performed on a regular basis to identify problems like:

- Broken links
- Incomplete downloads
- Deteriorating performance
- Orphaned files

Furthermore, information may be obtained about:

- Access patterns (where do users start and where do they end their visits)
- Navigation patterns
- Activity over time
- General performance
- Page-loading speed

The Web site structure should also be monitored at regular basis, as Web sites are likely to grow and change structure in an ad hoc manner.

4.5.2.2 Call Graphs

The analysis of the architectural structure of a product is on the borderline between design and testing. In the architectural design the product is decomposed into smaller and smaller components. Components call other sub routines or functions in other components; in object-oriented design we say that classes provide methods for others to use. These dependencies or couplings can be illustrated in call graphs produced by tools.

Static architecture analysis tools can also provide measurements related to the coupling, such as:

- Fan-in: Number of calls or usages of a specific function or methods made from other functions or methods (or the main program)
- Fan-out: Number of calls or usages of functions or methods made from a specific component
- Henry and Kafura metrics: Coupling between modules
- Bowles metrics: Module and system complexity
- Troy and Zweben metrics: Modularity or coupling; complexity of structure
- Ligier metrics: Modularity of the structure chart

These metrics may be more detailed and specific depending on the architectural paradigm.

Both the call graphs and the corresponding measurements provide valuable information to the test planning. This information may for example be used to:

- Determine integration high-risk areas (high fan-out and fan-in)
- Determine detailed integration testing sequence
- Determine intercomponent testing approach

Especially in object-oriented architecture, it is relevant to know the dy-

namic dependencies, that is, how often a method is used during execution of the product. This information can be combined with the static architectural information to strengthen the test planning even more.

4.6 Dynamic Analysis

Dynamic analysis is the process of evaluating a system or a component based upon its behavior during execution.

There are two aspects of dynamic analysis, namely:

- Dynamic—Code is being executed
- Analysis—Finding out about the nature of the object and its behavior

Dynamic analysis cannot be done without tool support. Chapter 9 discusses the testing tool types.

A dynamic analysis tool instruments the code in order to catch the relevant run-time information. This means that extra code is added to the code written by the developer. The effect of this is that it is not strictly the "real" code we are analyzing. In most cases this is without any importance. The instrumentation may, however, have an adverse impact on performance, and that can pose problems if we are testing time-sensitive real-time software.

In dynamic analysis we prepare the software code we are going to analyze. The component or larger object is then executed. This execution may be execution of test cases for other test purposes, or it may be execution of specific scripts produced for the analysis.

The great advantages of dynamic analysis are that it

- Provides run-time information otherwise difficult to obtain
- Provides information as a by-product of dynamic testing
- Finds faults that it is almost impossible to find in other ways

The dynamic analysis tool reports on what is going on during execution: It provides run-time information about the behavior and state of software while it is being executed. The information we can get from this covers:

- Memory handling and memory leaks
- Pointer handling
- Coverage analysis
- Performance analysis

4.6.1 Memory Handling and Memory Leaks

Memory handling is concerned with allocation, usage, and deallocation of memory.

The tools can detect memory leaks, where memory is gradually being

filled up during extended use. This happens if we keep on allocating memory in the program and forget to deallocate memory when we no longer need it. If a program with a memory leak keeps on running we can end up with no more available memory. This will cause a failure.

Memory is automatically deallocated when the program execution stops; this is one of the reasons why memory leaks are not always detected during dynamic testing. When we test we rarely run the program for longer periods, as it might be run in real life.

The advantage of dynamic analysis in connection with memory leaks is that the analysis can detect the possibility of memory leaks long before they actually happen.

4.6.2 Pointer Handling

Pointers are used to handle dynamic allocation of memory. Instead of working with a fixed name or address of a variable we let the system find out where the actual location is, and we use a pointer to point to this location in the program.

The usage of pointers can go wrong in a number of ways. Pointers can, for example, be unassigned when used, lose their object, or point to a place in memory to which it is not supposed to point, for example beyond an array border or into some protected part of memory. A wrong pointer can cause for example part of the program or data stored in memory to be overwritten.

It may be very difficult to find the defects, underlying failures caused by wrong pointer handling. These failures have a tendency to be periodic, that is, the program may run for a long time and then suddenly start to give wrong results or even crash.

Dynamic analysis tools can identify unassigned pointers, and they can also detect faults in pointer arithmetic, for example, if the addition of two pointers results in a pointer that points to an invalid place in memory.

4.6.3 Coverage Analysis

The coverage obtained in an executed test can be measured by analysis tools. These tools provide objective measurement for some structural or white-box test coverage metrics, for example:

- Statement coverage
- Branch coverage

The tools provide objective measurements to be used in the checking against test completion criteria in a fast and reliable way.

Some tools can also deliver reports about uncovered areas. The more fancy ones produce colored reports where covered code is shown in one color and uncovered code in another. This is a great help when more test cases must be designed to obtain a higher coverage.

4.6.4 Performance Analysis

All too many products have no or insufficient performance requirements and turn out to be unable to cope with real-life volumes and loads. Performance analysis aims at measuring the performance of a product under the controlled circumstances before the product is released.

It is much better to get these aspects tested before the product breaks down when the first set of users starts using it. Tools may be used to measure what the performance is under given circumstances.

The performance testing tools can provide very useful reports based on collected information, often in graphical form. The tools can provide information about "bottle neck" areas relatively inexpensively before the product hits the real world.

Some companies have specialized in performance testing and will test products using these tools. This is a very useful alternative to investing in the tools yourself.

4.7 Choosing Testing Techniques

The big question remaining after all these descriptions of wonderful test case design techniques is: Which testing technique(s) should we use? The answer to that is: *It depends!*

There is no established consensus on which technique is the most effective. The choice depends on the circumstances, including the testers' experience and the nature of the object under testing.

With regard to the testers' experience it is evident that a test case design technique that we as testers know well and have used many times on similar occasions is a good choice. All things being equal there is no need to throw old techniques overboard. Despite the general feeling that everything is changing fast, techniques do not usually change overnight. On the other hand, we need to be aware of new research and new techniques, both in development and testing becoming available from time to time.

A little more external to the testers' direct choice is the choice guided by risk analysis. Certain techniques are sufficient for low-risk products, whereas other techniques should be used for products or areas with a higher risk exposure. This is especially the case when we are selecting between structural or white-box techniques. Testing and risk are discussed in Section 3.5.

Even further away from the testers, the choice of test techniques may be dictated by customer requirements, typically formulated in the contract. There is a tendency for these constraints to be included in the contract for high-risk products. It may also be the case for development projects contracted between organizations with a higher level of maturity. In the case of test case techniques being stipulated in a contract, the test responsible should have

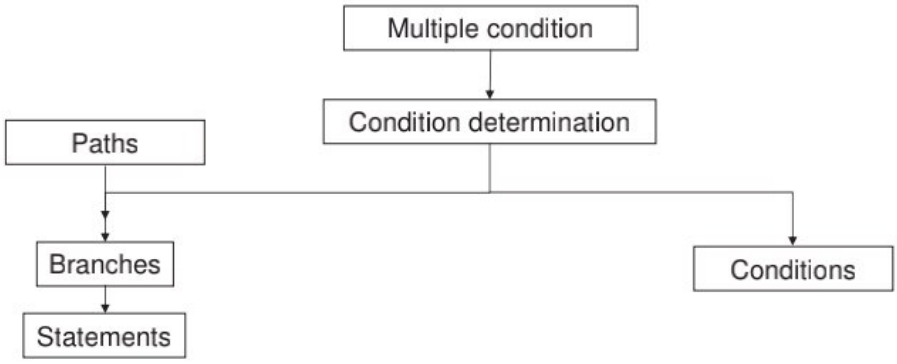
had the possibility of suggesting and accepting the choices.

Finally the choice of test case design techniques can be guided or even dictated by applicable regulatory standards.

4.7.1 Subsumes Ordering of Techniques

It is possible to define a sort of hierarchy of the structural test case design techniques based on the thoroughness of the techniques at 100% coverage.

This hierarchy is called the subsumes ordering of the techniques. The verb "subsume" means "to include in a larger class." The subsumes ordering show which techniques are included in techniques placed higher up in the order. The ordering is shown here.



The ordering can only be read downwards. We can for example see that condition determination subsumes branches. Paths also subsume branches; but we cannot say anything about the ordering of condition determination in relation to paths.

The subsumes ordering does not tell us which technique to use, but it shows the techniques' relative thoroughness. It also shows that it does not make sense to require both a 100% branch and a 100% statement coverage, because the latter will be superfluous.

4.7.2 Advice on Choosing Testing Techniques

No firm research conclusions exist about the rank in effectiveness of the functional or black-box techniques. Most of the research that has been performed is very academic and not terribly useful in "the real testing world."

One conclusion that seems to have been reached is: There is no "best" technique. The "best" depends on the nature of the product.

We do, however, know with certainty that the usage of *some technique is better than none,* and that a combination of techniques is better than just one technique.

We also know that the use of techniques supports systematic and meticulous work and that techniques are good for finding possible failures. Using test case design techniques means that they may be repeated by others with

approximately the same result, and that we are able to explain our test cases.

There is no excuse for not using some techniques.

In his book *The Art of Software Testing,* Glenford J. Meyers provides a strategy for applying techniques. He writes:

- If the specification contains combinations of input conditions, start with cause-effect graphing
- Always use boundary value analysis (input and output)
- Supply with valid and invalid equivalence classes (both for input and output)
- Round up using error guessing
- Add sufficient test cases using white-box techniques if completion criteria has not yet been reached (providing it is possible)

As mentioned earlier some research is being made into the effectiveness of different test techniques.

Stuart Reid has made a study on the techniques equivalence partitioning, boundary value analysis, and random testing on real avionics code. Based on all input for the techniques he concludes that BVA is most effective with 79% effectiveness, whereas EP only reached 33% effectiveness. Stuart Reid also concludes that some faults are difficult to find even with these techniques.

Questions

1. Why is it a good idea to use test techniques?
2. What is the other, perhaps more common, name for specification-based testing techniques?
3. What is the basic idea in equivalence partitioning?
4. Why may equivalence partitioning reduce the number of test cases?
5. What is boundary value analysis?
6. What are the coverage elements for equivalence partitioning and boundary value analysis?
7. What should happen to invalid values?
8. Where do we find most faults in connection with equivalence partitioning and boundary value analysis?
9. When is domain analysis used?
10. What are the four points defined in domain analysis?
11. What are the coverage elements for domain analysis?
12. How many columns does a decision table have depending on the number of input conditions?
13. What is the coverage element for decision tables?
14. How can you fill in a decision table?
15. What is one of the main application areas for decision tables?