Test design techniques

What is Test analysis? or How to identify the test conditions?

Test analysis: identifying test conditions

- Test analysis is the process of looking at something that can be used to derive test information. This basis for the tests is called the **test basis**.
- The test basis is the information we need in order to start the test analysis and create our own test cases. Basically it's a documentation on which test cases are based, such as requirements, design specifications, product risk analysis, architecture and interfaces.
- We can use the test basis documents to understand what the system should do once built. The test basis includes whatever the tests are based on. Sometimes tests can be based on experienced user's knowledge of the system which may not be documented.
- From testing perspective we look at the test basis in order to see what could be tested. These are the test conditions. A **test condition** is simply something that we could test.
- While identifying the test conditions we want to identify as many conditions as we can and then we select about which one to take forward and combine into test cases. We could call them **test possibilities.**
- As we know that testing everything is an impractical goal, which is known as exhaustive testing. We cannot test everything we have to select a subset of all possible tests. In practice the subset we select may be a very small subset and yet it has to have a high probability of finding most of the defects in a system. Hence we need some intelligent thought process to guide our selection called **test techniques.** The test conditions that are chosen will depend on the test strategy or detailed test approach. For example, they might be based on risk, models of the system, etc.
- Once we have identified a list of test conditions, it is important to prioritize them, so that the most important test conditions are identified. Test conditions can be identified for **test data** as well as for test inputs and test outcomes, for example, different types of record, different sizes of records or fields in a record. Test conditions are documented in the IEEE 829 document called a Test Design Specification.

What is Test design? or How to specify test cases?

Test design: specifying test cases

- Basically test design is the act of creating and writing test suites for testing a software.
- Test analysis and identifying test conditions gives us a generic idea for testing which covers quite a large range of possibilities. But when we come to make a test case we need to be very specific. In fact now we need the exact and detailed specific input. But just having some

values to input to the system is not a test, if you don't know what the system is supposed to do with the inputs, you will not be able to tell that whether your test has passed or failed.

- Test cases can be documented as described in the **IEEE 829 Standard for Test Documentation.**
- One of the most important aspects of a test is that it checks that the system does what it is supposed to do. Copeland says 'At its core, testing is the process of comparing "what is" with "what ought to be" '. [Copeland, 2003]. If we simply put in some inputs and think that was fun, I guess the system is probably OK because it didn't crash, but are we actually testing it? We don't think so. You have observed that the system does what the system does but this is not a test. Boris Beizer refers to this as **'kiddie testing'** [Beizer, 1990]. We may not know what the right answer is in detail every time, and we can still get some benefit from this approach at times, but it isn't really testing. In order to know what the system *should* do, we need to have a source of information about the correct behavior of the system – this is called an **'oracle'** or **a test oracle.**
- Once a given input value has been chosen, the tester needs to determine what the expected result of entering that input would be and document it as part of the test case. Expected results include information displayed on a screen in response to an input. If we don't decide on the expected results before we run a test then there might be a chance that we will notice that there is something wildly wrong. However, we would probably not notice small differences in calculations, or results that seemed to look OK. So we would conclude that the test had passed, when in fact the software has not given the correct result. Small differences in one calculation can add up to something very major later on, for example if results are multiplied by a large factor. Hence, ideally expected results should be predicted before the test is run.

What is Test implementation? or How to specifying test procedures or scripts?

The document that describes the steps to be taken in running a set of tests and specifies the executable order of the tests is called a **test procedure** in IEEE 829, and is also known as a **test script.** When test Procedure Specification is prepared then it is implemented and is called Test implementation. Test script is also used to describe the instructions to a test execution tool. An automation script is written in a programming language that the tool can understand. (This is an automated test procedure.). We will study about it in the chapter no. 6.

The tests that are intended to be run manually rather than using a test execution tool can be called as manual test script. The test procedures, or test scripts, are then formed into a test execution schedule that specifies which procedures are to be run first – a kind of **superscript.**

Writing the test procedure is another opportunity to prioritize the tests, to ensure that the best testing is done in the time available. A good rule of thumb is 'Find the scary stuff first'. However the definition of what is 'scary' depends on the business, system or project and depends up on the risk of the project.

What are the categories of test design techniques?

A test design technique basically helps us to select a good set of tests from the total number of all possible tests for a given system. There are many different types of software testing technique, each with its own strengths and weaknesses. Each individual technique is good at finding particular types of defect and relatively poor at finding other types.

For example, a technique that explores the upper and lower limits of a single input range is more likely to find boundary value defects than defects associated with combinations of inputs. Similarly, testing performed at different stages in the software development life cycle will find different types of defects; component testing is more likely to find coding logic defects than system design defects.

Referring to the figure 4.1 each testing technique falls into one of a number of different categories. Broadly speaking there are two main categories:

1. **Static technique**
2. **Dynamic technique**

    - *Specification-based (black-box, also known as behavioral techniques)*
    - *Structure-based (white-box or structural techniques)*
    - *Experience- based*

Dynamic techniques are subdivided into three more categories: specification-based (black-box, also known as behavioral techniques), structure-based (white-box or structural techniques) and experience- based. Specification-based techniques include both functional and nonfunctional techniques (i.e. quality characteristics).

What is Static testing technique?

- Static testing is the testing of the software work products manually, or with a set of tools, but they are **not executed**.
- It starts early in the Life cycle and so it is done during the verification process.
- **It does not need computer as the testing of program is done without executing the program.** For example: reviewing, walk through, inspection, etc.
- Most static testing techniques can be used to 'test' any form of document including source code, design documents and models, functional specifications and requirement specifications.

What is Dynamic testing technique?

- This testing technique needs computer for testing.

- It is done during Validation process.
- The software is tested by executing it on computer. Ex: Unit testing, integration testing, system testing.

What is black-box, Specification-based, also known as behavioral testing techniques?

- Specification-based testing technique is also known as **'black-box'** or input/output driven testing techniques because they view the software as a black-box with inputs and outputs.
- The testers have no knowledge of how the system or component is structured inside the box. In black-box testing the tester is concentrating on what the software does, not how it does it.
- The definition mentions both functional and non-functional testing. Functional testing is concerned with what the system does its features or functions. Non-functional testing is concerned with examining how well the system does. Non-functional testing like performance, usability, portability, maintainability, etc.
- Specification-based techniques are appropriate at all levels of testing (component testing through to acceptance testing) where a specification exists. For example, when performing system or acceptance testing, the requirements specification or functional specification may form the basis of the tests.

- There are four specification-based or black-box technique:

    - Equivalence partitioning
    - Boundary value analysis
    - Decision tables
    - State transition testing

What is Equivalence partitioning in Software testing?

- Equivalence partitioning (EP) is a specification-based or black-box technique.
- It can be applied at any level of testing and is often a good technique to use first.
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes – the two terms mean exactly the same thing.
- In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

For example, a savings account in a bank has a different rate of interest depending on the balance in the account. In order to test the software that calculates the interest due, we can identify the ranges of balance values that earn the different rates of interest. For example, 3% rate of interest is given if the balance in the account is in the range of $0 to $100, 5% rate of interest is given if the balance in the account is in the range of $100 to $1000, and 7% rate of interest is given if the balance in the account is $1000 and above, we would initially identify three valid equivalence partitions and one invalid partition as shown below.

In the above example we have identified four partitions, even though the specification mentioned only three. This shows a very important task of the tester that is a tester should not only test what is in the specification, but should also think about things that haven't been specified. In this case we have thought of the situation where the balance is less than zero. We haven't (yet) identified an invalid partition on the right, but this would also be a good thing to consider. In order to identify where the 7% partition ends, we would need to know what the maximum balance is for this account (which may not be easy to find out). In our example we have left this open for the time being. Note that non-numeric input is also an invalid partition (e.g. the letter 'a') but we discuss only the numeric partitions for now. We have made an assumption here about what the smallest difference is between two values. We have assumed two decimal places, i.e. $100.00, but we could have assumed zero decimal places (i.e. $100) or more than two decimal places (e.g. $100.0000) In any case it is a good idea to state your assumptions – then other people can see them and let you know if they are correct or not. When designing the test cases for this software we would ensure that all the three valid equivalence partitions are covered once, and we would also test the invalid partition at least once. So for example, we might choose to calculate the interest on balances of-$10.00, $50.00, $260.00 and $1348.00. If we hadn't specifically identified these partitions, it is possible that at least one of them could have been missed at the expense of testing another one several times over. Note that we could also apply equivalence partitioning to outputs as well. In this case we have three interest rates: 3%, 5% and 7%, plus the error message for the invalid partition (or partitions). In this example, the output partitions line up exactly with the input partitions. How would someone test this without thinking about the partitions? A inexperienced tester (let's call him Robbin) might have thought that a good set of tests would be to test every $50. That would give the following tests: $50.00, $100.00, $150.00, $200.00, $250.00, … say up to $800.00 (then Robbin would have got tired of it and thought that enough tests had been carried out). But look at what Robbin has tested: only two out of four partitions! So if the system does not correctly handle a negative balance or a balance of $1000 or more, he would not have found these defects – so the naive approach is less effective than equivalence partitioning. At the same time, Robbin has four times more tests (16 tests versus our four tests using equivalence partitions), so he is also much less efficient. This is why we say that using techniques such as this makes testing both more effective and more efficient. Note that when we say a partition is 'invalid', it doesn't mean that it represents a value that cannot be entered by a user or a value that the user isn't supposed to enter. It just means that it is not one of the expected inputs for this particular field.

- Equivalence partitioning (EP) is a specification-based or black-box technique.
- It can be applied at any level of testing and is often a good technique to use first.
- The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence partitioning'. **Equivalence partitions** are also known as equivalence classes – the two terms mean exactly the same thing.

- In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

For example, a savings account in a bank has a different rate of interest depending on the balance in the account. In order to test the software that calculates the interest due, we can identify the ranges of balance values that earn the different rates of interest. For example, 3% rate of interest is given if the balance in the account is in the range of $0 to $100, 5% rate of interest is given if the balance in the account is in the range of $100 to $1000, and 7% rate of interest is given if the balance in the account is $1000 and above, we would initially identify three valid equivalence partitions and one invalid partition as shown below.

Invalid partition	Valid (for 3% interest)		Valid (for 5%)		Valid (for 7%)
-$0.01	$0.00	$100.00	$100.01	$999.99	$1000.00

In the above example we have identified four partitions, even though the specification mentioned only three. This shows a very important task of the tester that is a tester should not only test what is in the specification, but should also think about things that haven't been specified. In this case we have thought of the situation where the balance is less than zero. We haven't (yet) identified an invalid partition on the right, but this would also be a good thing to consider. In order to identify where the 7% partition ends, we would need to know what the maximum balance is for this account (which may not be easy to find out). In our example we have left this open for the time being. Note that non-numeric input is also an invalid partition (e.g. the letter 'a') but we discuss only the numeric partitions for now. We have made an assumption here about what the smallest difference is between two values. We have assumed two decimal places, i.e. $100.00, but we could have assumed zero decimal places (i.e. $100) or more than two decimal places (e.g. $100.0000) In any case it is a good idea to state your assumptions – then other people can see them and let you know if they are correct or not. When designing the test cases for this software we would ensure that all the three valid equivalence partitions are covered once, and we would also test the invalid partition at least once. So for example, we might choose to calculate the interest on balances of-$10.00, $50.00, $260.00 and $1348.00. If we hadn't specifically identified these partitions, it is possible that at least one of them could have been missed at the expense of testing another one several times over. Note that we could also apply equivalence partitioning to outputs as well. In this case we have three interest rates: 3%, 5% and 7%, plus the error message for the invalid partition (or partitions). In this example, the output partitions line up exactly with the input partitions. How would someone test this without thinking about the partitions? A inexperienced tester (let's call him Robbin) might have thought that a good set of tests would be to test every $50. That would give the following tests: $50.00, $100.00, $150.00, $200.00, $250.00, … say up to $800.00 (then Robbin would have got tired of it and thought that enough tests had been carried out). But look at what Robbin has tested: only two out of four partitions! So if the system does not correctly handle a negative balance or a balance of $1000 or more, he would not have found these defects – so the naive approach is less effective than equivalence partitioning. At the same time, Robbin has four times more tests (16 tests versus our four tests using equivalence partitions), so he is also much less efficient. This is why we

say that using techniques such as this makes testing both more effective and more efficient. Note that when we say a partition is 'invalid', it doesn't mean that it represents a value that cannot be entered by a user or a value that the user isn't supposed to enter. It just means that it is not one of the expected inputs for this particular field.

What is Boundary value analysis in software testing?

- Boundary value analysis (BVA) is based on testing at the boundaries between partitions.
- Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).
- As an example, consider a printer that has an input option of the number of copies to be made, from 1 to 99. To apply boundary value analysis, we will take the minimum and maximum (boundary) values from the valid partition (1 and 99 in this case) together with the first or last value respectively in each of the invalid partitions adjacent to the valid partition (0 and 100 in this case). In this example we would have three equivalence partitioning tests (one from each of the three partitions) and four boundary value tests. Consider the bank system described in the previous section in equivalence partitioning.

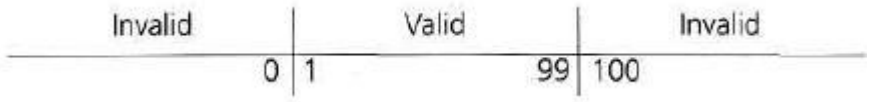
Because the boundary values are defined as those values on the edge of a partition, we have identified the following boundary values: -$0.01 (an invalid boundary value because it is at the edge of an invalid partition), $0.00, $100.00, $100.01, $999.99 and $1000.00, all valid boundary values. So by applying boundary value analysis we will have six tests for boundary values. Compare what our naive tester Robbin had done: he did actually hit one of the boundary values ($100) though it was more by accident than design. So in addition to testing only half of the partitions, Robbin has only tested one sixth of the boundaries (so he will be less effective at finding any boundary defects). If we consider all of our tests for both equivalence partitioning and boundary value analysis, the techniques give us a total of nine tests, compared to the 16 that Robbie had, so we are still considerably more efficient as well as being over three times more effective (testing four partitions and six boundaries, so 10 conditions in total compared to three). By showing the values in the table, we can see that no maximum has been specified for the 7% interest rate. We would now want to know what the maximum value is for an account balance, so that we can test that boundary. This is called an 'open boundary', because one of the sides of the partition is left open, i.e. not defined. But that doesn't mean we can ignore it, we should still try to test it, but the question is how?

Open boundaries are very difficult to test, but there are different ways to approach them. Actually the best solution to the problem is to find out what the boundary should be specified as! One approach is to go back to the specification to see if a maximum has been stated somewhere else for a balance amount. If so, then we know what our boundary value is. Another approach might be to investigate other related areas of the system. For example, the field that holds the account balance figure may be only six figures plus two decimal figures. This would give a maximum account balance of $999 999.99 so we could use that as our maximum boundary value. If we still not able to find anything about what this boundary should be, then we probably need to use an intuitive or experience-based approach to check it by entering various large values trying to make it fail.

We can consider another example of Boundary value analysis where we can apply it to the whole of a string of characters (e.g. a name or address). The number of characters in the string is a partition, e.g. between 1 and 30 characters is the valid partition with valid boundaries of 1 and 30. The invalid boundaries would be 0 characters (null, just hit the Return key) and 31 characters. Both of these should produce an error message.

While testing why it is important to do both equivalence partitioning and boundary value analysis?

Technically, because every boundary is in some partition, if you did only boundary value analysis you would also have tested every equivalence partition. However, this approach may cause problems if that value fails – was it only the boundary value that failed or did the whole partition fail? Also by testing only boundaries we would probably not give the users much confidence as we are using extreme values rather than normal values. The boundaries may be more difficult (and therefore more costly) to set up as well. For example, in the printer copies example described earlier we identified the following boundary values:



Suppose we test only the valid boundary values 1 and 99 and nothing in between. If both tests pass, this seems to indicate that all the values in between should also work. However, suppose that one page prints correctly, but 99 pages do not. Now we don't know whether any set of more than one page works, so the first thing we would do would be to test for say 10 pages, i.e. a value from the equivalence partition. We recommend that you test the partitions separately from boundaries – this means choosing partition values that are NOT boundary values. However, if you use the three-value boundary value approach, then you would have valid boundary values of 1, 2, 98 and 99, so having a separate equivalence value in addition to the extra two boundary values would not give much additional benefit. But notice that one equivalence value, e.g. 10, replaces both of the extra two boundary values (2 and 98). This is why equivalence partitioning with two-value boundary value analysis is more efficient than three-value boundary value analysis.

What is Decision table in software testing?

The techniques of equivalence partitioning and boundary value analysis are often applied to specific situations or inputs. However, if different combinations of inputs result in different actions being taken, this can be more difficult to show using equivalence partitioning and boundary value analysis, which tend to be more focused on the user interface. The other two specification-based software testing techniques, decision tables and state transition testing are more focused on business logic or business rules.

A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table (Myers describes this as a combinatorial logic network [Myers, 1979]). However, most people find it more useful just to use the table described in [Copeland, 2003].

- Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers.
- Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.
- It helps the developers to do a better job can also lead to better relationships with them. Testing combinations can be a challenge, as the number of combinations can often be huge. Testing all combinations may be impractical if not impossible. We have to be satisfied with testing just a small subset of combinations but making the choice of which combinations to test and which to leave out is also important. If you do not have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

How to Use decision tables for test designing?

The first task is to identify a suitable function or subsystem which reacts according to a combination of inputs or events. The system should not contain too many inputs otherwise the number of combinations will become unmanageable. It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time. Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations of True and False for each of the aspects.

Let us consider an example of a loan application, where you can enter the amount of the monthly repayment or the number of years you want to take to pay it back (the term of the loan). If you enter both, the system will make a compromise between the two if they conflict. The two conditions are the loan amount and the term, so we put them in a table (see Table 4.2).

TABLE 4.2 Empty decision table:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered:				
Term of loan has been Entered:				

Next we will identify all of the combinations of True and False (see Table 4.3). With two conditions, each of which can be True or False, we will have four combinations (two to the power of the number of things to be combined). Note that if we have three things to combine, we will have eight combinations, with four things, there are 16, etc. This is why it is good to tackle small sets of combinations at a time. In order to keep track of which combinations we have, we will alternate True and False on the bottom row, put two Trues and then two Falses on the row above the bottom row, etc., so the top row will have all Trues and then all Falses (and this principle applies to all such tables).

TABLE 4.3 Decision table with input combinations:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been *entered:*	T	T	F	F
Term of loan has been entered:	T	F	T	F

In the next step we will now identify the correct outcome for each combination (see Table 4.4). In this example, we can enter one or both of the two fields. Each combination is sometimes referred to as a rule.

TABLE 4.4 Decision table with combinations and outcomes:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been *entered:*	T	T	F	F
Term of loan has been entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:	Y	Y		
Process term:	Y		Y	

At this point, we may realize that we hadn't thought about what happens if the customer doesn't enter anything in either of the two fields. The table has highlighted a combination that was not mentioned in the specification for this example. We could assume that this combination should result in an error message, so we need to add another action (see Table 4.5). This highlights the strength of this technique to discover omissions and ambiguities in specifications. It is not unusual for some combinations to be omitted from specifications; therefore this is also a valuable technique to use when reviewing the test basis.

TABLE 4 . 5 Decision table with additional outcomes:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been *entered:*	T	T	F	F
Term of loan has been entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:	Y	Y		
Process term:	Y		Y	
Error message:				Y

Now, we make slight change in this example, so that the customer is not allowed to enter both repayment and term. Now the outcome of our table will change, because there should also be an error message if both are entered, so it will look like Table 4.6.

TABLE 4 . 6 Decision table with changed outcomes:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**

Repayment amount has been entered:	T	T	F	F
Term of loan has been entered:	T	F	T	F
Actions/Outcomes				
Process loan amount:		Y		
Process term:			Y	
Error message:	Y			Y

You might notice now that there is only one 'Yes' in each column, i.e. our actions are mutually exclusive – only one action occurs for each combination of conditions. We could represent this in a different way by listing the actions in the cell of one row, as shown in Table 4.7. Note that if more than one action results from any of the combinations, then it would be better to show them as separate rows rather than combining them into one row.

TABLE 4.7 Decision table with outcomes in one row:

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered:	T	T	F	F
Term of loan has been entered:	T	F	T	F
Actions/Outcomes:				
Result:	Error message	Process loan amount	Process term	Error message

The final step of this technique is to write test cases to exercise each of the four rules in our table.

Credit card example:

Let's take another example. If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today, second if you are an existing customer and you hold a loyalty card, you get a 10% discount and third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable. This is shown in Table 4.8.

TABLE 4.8 Decision table for credit card example

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

In Table 4.8, the conditions and actions are listed in the left hand column. All the other columns in the decision table each represent a separate rule, one for each combination of conditions. We may

choose to test each rule/combination and if there are only a few this will usually be the case. However, if the number of rules/combinations is large we are more likely to sample them by selecting a rich subset for testing.

Now let's see the decision table for credit card shown above:

- Note that we have put X for the discount for two of the columns (Rules 1 and 2) – this means that this combination should not occur. You cannot be both a new customer and also holding a loyalty card as per the conditions mentioned above. Hence there should be an error message stating this.
- We have made an assumption in Rule 3. Since the coupon has a greater discount than the new customer discount, we assume that the customer will choose 20% rather than 15%. We cannot add them, since the coupon cannot be used with the 'new customer' discount as stated in the condition above. The 20% action is an assumption on our part, and we should check that this assumption (and any other assumptions that we make) is correct, by asking the person who wrote the specification or the users.
- For Rule 5, however, we can add the discounts; since both the coupon and the loyalty card discount should apply (that's our assumption).
- Rules 4, 6 and 7 have only one type of discount and Rule 8 has no discount, so 0%.

If we are applying this technique thoroughly, we would have one test for each column or rule of our decision table. The advantage of doing this is that we may test a combination of things that otherwise we might not have tested and that could find a defect. However, if we have a lot of combinations, it may not be possible or sensible to test every combination. If we are time-constrained, we may not have time to test all combinations. Don't just assume that all combinations need to be tested. It is always better to prioritize and test the most important combinations. Having the full table helps us to decide which combinations we should test and which not to test this time. In the example above all the conditions are binary, i.e. they have only two possible values: True or False (or we can say Yes or No).

What is State transition testing in software testing?

- State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based.
- Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.
- A finite state system is often shown as a **state diagram** (see Figure 4.2).
- One of the advantages of the state transition technique is that the model can be as detailed or as abstract as you need it to be. Where a part of the system is more important (that is, requires more testing) a greater depth of detail can be modeled. Where the system is less important (requires less testing), the model can use a single state to signify what would otherwise be a series of different states.
- A **state transition model has four basic parts:**

- *The states that the software may occupy (open/closed or funded/insufficient funds);*
- *The transitions from one state to another (not all transitions are allowed);*
- *The events that cause a transition (closing a file or withdrawing money);*
- *The actions that result from a transition (an error message or being given your cash).*

Hence we can see that in any given state, one event can cause only one action, but that the same event – from a different state – may cause a different action and a different end state.

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but it may refuse to give you the money because of your insufficient balance. This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Let us consider another example of a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

We will look first at test cases that execute valid state transitions.
Figure 4.2 below, shows an example of entering a Personal Identity Number (PIN) to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here – there would also be a time-out from 'wait for PIN' and from the three tries which would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transition from the 'eat card' state back to the start state. We have not specified all the possible events either – there would be a 'cancel' option from 'wait for PIN' and from the three tries, which would also go back to the start state and eject the card.

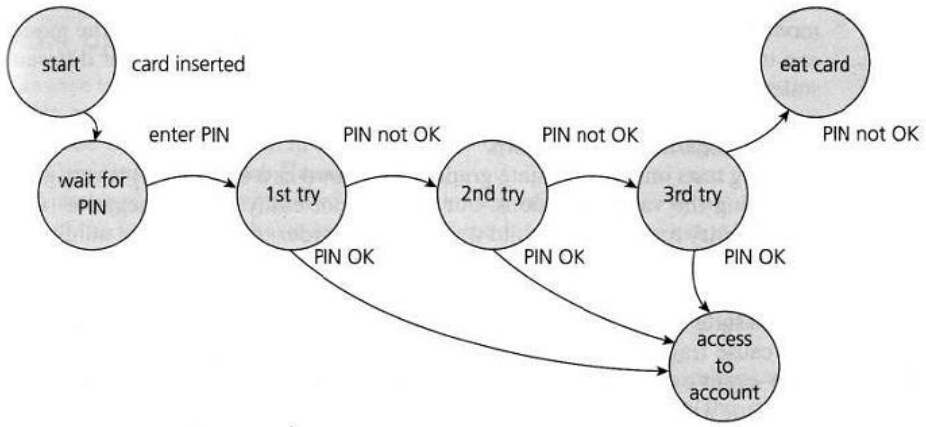


FIGURE 4.2 State diagram for PIN entry

In deriving test cases, we may start with a typical scenario.

- First test case here would be the normal situation, where the correct PIN is entered the first time.
- A second test (to visit every state) would be to enter an incorrect PIN each time, so that the system eats the card.
- A third test we can do where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.
- Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). So there could be a transition from 'access account' which just goes back to 'access account' for an action such as 'request balance'.

Test conditions can be derived from the state graph in various ways. Each state can be noted as a test condition, as can each transition. However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

We need to be able to identify the coverage of a set of tests in terms of transitions. We can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is l-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much we have tested (covered) will discuss in a white-box perspective. However, state transition testing is regarded as a black-box technique.

What is Use case testing in software testing?

- Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish. They are described by Ivar Jacobson in his book *Object-Oriented Software Engineering: A Use Case Driven Approach* [Jacobson, 1992].
- A use case is a description of a particular use of the system by an actor (a user of the system). Each use case describes the interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user).
- Actors are generally people but they may also be other systems.
- Use cases are a sequence of steps that describe the interactions between the actor and the system. Use cases are defined in terms of the actor, not the system, describing what the actor does and what the actor sees rather than what inputs the system expects and what the system's outputs.
- They often use the language and terms of the business rather than technical terms, especially when the actor is a business user.
- They serve as the foundation for developing test cases mostly at the system and acceptance testing levels.
- Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components. Used in this way, the actor may be something that the system interfaces to such as a communication link or sub-system.

- Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system (i.e. the defects that the users are most likely to come across when first using the system).
- Each use case usually has a mainstream (or most likely) scenario and sometimes additional alternative branches (covering, for example, special cases or exceptional conditions).
- Each use case must specify any preconditions that need to be met for the use case to work.
- Use cases must also specify post conditions that are observable results and a description of the final state of the system after the use case has been executed successfully.

The ATM PIN example is shown below in Figure 4.3. We show successful and unsuccessful scenarios. In this diagram we can see the interactions between the A (actor – in this case it is a human being) and S (system). From step 1 to step 5 that is success scenario it shows that the card and pin both got validated and allows Actor to access the account. But in extensions there can be three other cases that is 2a, 4a, 4b which is shown in the diagram below.

For use case testing, we would have a test of the success scenario and one testing for each extension. In this example, we may give extension 4b a higher priority than 4a from a security point of view.

System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

What is white-box or Structure-based or structural testing techniques?

- Structure-based testing technique is also known as **'white-box'** or 'glass-box' testing technique because here the testers require knowledge of how the software is implemented, how it works.

- In white-box testing the tester is concentrating on how the software does it. For example, a structural technique may be concerned with exercising loops in the software.
- Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.
- Structure-based techniques can also be used at all levels of testing. Developers use structure-based techniques in component testing and component integration testing, especially where there is good tool support for code coverage.
- Structure-based techniques are also used in system and acceptance testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

What is Experience- based testing technique?

- In **experience-based techniques,** people's knowledge, skills and background are of prime importance to the test conditions and test cases.
- The experience of both technical and business people is required, as they bring different perspectives to the test analysis and design process. Because of the previous experience with similar systems, they may have an idea as what could go wrong, which is very useful for testing.
- Experience-based techniques go together with specification-based and structure-based techniques, and are also used when there is no specification, or if the specification is inadequate or out of date.
- This may be the only type of technique used for low-risk systems, but this approach may be particularly useful under extreme time pressure – in fact this is one of the factors leading to exploratory testing.

What is Error guessing in software testing?

- The Error guessing is a technique where the experienced and good testers are encouraged to think of situations in which the software may not be able to cope. Some people seem to be naturally good at testing and others are good testers because they have a lot of experience either as a tester or working with a particular system and so are able to find out its weaknesses. This is why an error guessing approach, used after more formal techniques have been applied to some extent, can be very effective. It also saves a lot of time because of the assumptions and guessing made by the experienced testers to find out the defects which otherwise won't be able to find.
- The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to be.
- This is why an error guessing approach, used after more formal techniques have been applied to some extent, can be very effective. In using more formal techniques, the tester is likely to gain a better understanding of the system, what it does and how it works. With this better understanding, he or she is likely to be better at guessing ways in which the system may not work properly.
- Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required). If anyone ever says of a system or the environment in which it is to operate 'That could never happen', it

might be a good idea to test that condition, as such assumptions about what will and will not happen in the live environment are often the cause of failures.

- A structured approach to the error-guessing technique is to list possible defects or failures and to design tests that attempt to produce them. These defect and failure lists can be built based on the tester's own experience or that of other people, available defect and failure data, and from common knowledge about why software fails.

What is Exploratory testing in software testing?

- As its name implies, exploratory testing is about exploring, finding out about the software, what it does, what it doesn't do, what works and what doesn't work. The tester is constantly making decisions about what to test next and where to spend the (limited) time. This is an approach that is most useful when there are no or poor specifications and when time is severely limited.
- Exploratory testing is a hands-on approach in which testers are involved in minimum planning and maximum test execution.
- The planning involves the creation of a test charter, a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort, the objectives and possible approaches to be used.
- The test design and test execution activities are performed in parallel typically without formally documenting the test conditions, test cases or test scripts. This does not mean that other, more formal testing techniques will not be used. For example, the tester may decide to us boundary value analysis but will think through and test the most important boundary values without necessarily writing them down. Some notes will be written during the exploratory-testing session, so that a report can be produced afterwards.
- Test logging is undertaken as test execution is performed, documenting the key aspects of what is tested, any defects found and any thoughts about possible further testing.
- It can also serve to complement other, more formal testing, helping to establish greater confidence in the software. **In** this way, exploratory testing can be used as a check on the formal test process by helping to ensure that the most serious defects have been found.
- Exploratory testing is described in [Kaner, 2002] and [Copeland, 2003] Other ways of testing in an exploratory way ('attacks') are described in [Whittaker, 2002].

What is Structure-based technique in software testing?

- Structure-based techniques serve two purposes: test coverage measurement and structural test case design.
- They are often used first to assess the amount of testing performed by tests derived from specification-based techniques, i.e. to assess coverage.
- They are then used to design additional tests with the aim of increasing the test coverage.

- Structure-based test design techniques are a good way of generating additional test cases that are different from existing tests.
- They can help ensure more breadth of testing, in the sense that test cases that achieve 100% **coverage** in any measure will be exercising all parts of the software from the point of view of the items being covered.

What is test coverage in software testing? It's advantages and disadvantages

Test coverage measures the amount of testing performed by a set of test. Wherever we can count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage and is known as test coverage.

The basic coverage measure is where the 'coverage item' is whatever we have been able to count and see whether a test has exercised or used this item.

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

There is danger in using a coverage measure. But, 100% coverage does *not* mean 100% tested. Coverage techniques measure only one dimension of a multi-dimensional concept. Two different test cases may achieve exactly the same coverage but the input data of one may find an error that the input data of the other doesn't.

Benefit of code coverage measurement:

- It creates additional test cases to increase coverage
- It helps in finding areas of a program not exercised by a set of test cases
- It helps in determining a quantitative measure of code coverage, which indirectly measure the quality of the application or product.

Drawback of code coverage measurement:

- One drawback of code coverage measurement is that it measures coverage of what has been written, i.e. the code itself; it cannot say anything about the software that has *not* been written.
- If a specified function has not been implemented or a function was omitted from the specification, then structure-based techniques cannot say anything about them it only looks at a structure which is already there.

Where to apply this test coverage in software testing?

The answer of the question is that test coverage can be used in any level of the testing. Test coverage can be measured based on a number of different structural elements in a system or component. Coverage can be measured at component testing level, integration-testing level or at system- or acceptance-testing levels. For example, at system or acceptance level, the coverage items may be requirements, menu options, screens, or typical business transactions. At integration level, we could measure coverage of interfaces or specific interactions that have been tested.

We can also measure coverage for each of the specification-based techniques or black-box testing:

• EP: percentage of equivalence partitions exercised (we could measure valid and invalid partition coverage separately if this makes sense);
• BVA: percentage of boundaries exercised (we could also separate valid and invalid boundaries if we wished);
• Decision tables: percentage of business rules or decision table columns tested;
• State transition testing: there are a number of possible coverage measures:

– Percentage of states visited
– Percentage of (valid) transitions exercised (this is known as Chow's 0- switch coverage)
– Percentage of pairs of valid transitions exercised ('transition pairs' or
Chow's 1-switch coverage) – and longer series of transitions, such as transition triples, quadruples, etc.
– Percentage of invalid transitions exercised (from the state table).

The coverage measures for specification-based techniques would apply at whichever test level the technique has been used (e.g. system or component level).

Why to measure code coverage?

- To know whether we have enough testing in place
- To maintain the test quality over the life cycle of a project
- To know how well our tests, actually test our code

How we can measure the coverage?

Coverage measurement of code is best done **by using tools** and there are a number of such tools on the market. These tools can helps in:

- Increasing the quality and productivity of testing.
- They increase quality by ensuring that more structural aspects are tested, so defects on those structural paths can be found.
- They increase productivity and efficiency by highlighting tests that may be redundant, i.e. testing the same structure with different data (as there is possibility of finding the defects by testing the same structure with different data).

What are the types of coverage?

There are many types of test coverage. Test coverage can be used in any level of the testing. Test coverage can be measured based on a number of different structural elements in a system or component. Coverage can be measured at component testing level, integration-testing level or at system- or acceptance-testing levels. For example, at system or acceptance level, the coverage items may be requirements, menu options, screens, or typical business transactions. At integration level, we could measure coverage of interfaces or specific interactions that have been tested.

We can also measure coverage for each of the specification-based techniques:

- EP: percentage of equivalence partitions exercised (we could measure valid and invalid partition coverage separately if this makes sense);
- BVA: percentage of boundaries exercised (we could also separate valid and invalid boundaries if we wished);
- Decision tables: percentage of business rules or decision table columns tested;
- State transition testing: there are a number of possible coverage measures:

    - Percentage of states visited
    - Percentage of (valid) transitions exercised (this is known as Chow's 0- switch coverage)
    - Percentage of pairs of valid transitions exercised ('transition pairs' or
    - Chow's 1-switch coverage) – and longer series of transitions, such as transition triples, quadruples, etc.
    - Percentage of invalid transitions exercised (from the state table).

The coverage measures for specification-based techniques would apply at whichever test level the technique has been used (e.g. system or component level).

The different types of coverage are:
1) Statement coverage
2) Decision coverage
3) Condition coverage

What is Statement coverage? Advantages and disadvantages

- The statement coverage is also known as line coverage or segment coverage.
- The statement coverage **covers only the true conditions.**
- Through statement coverage we can identify the statements executed and where the code is not executed because of blockage.
- In this process each and every line of code needs to be checked and executed

Advantage of statement coverage:

- It verifies what the written code is expected to do and not to do
- It measures the quality of code written

- It checks the flow of different paths in the program and it also ensure that whether those path are tested or not.

Disadvantage of statement coverage:

- It cannot test the false conditions.
- It does not report that whether the loop reaches its termination condition.
- It does not understand the logical operators.

The statement coverage can be calculated as shown below:

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

To understand the statement coverage in a better way let us take an example which is basically a pseudo-code. It is not any specific programming language, but should be readable and understandable to you, even if you have not done any programming yourself.

Consider code sample 4.1 :

```
READ X
READ Y
I F X>Y THEN Z = 0
ENDIF
```

Code sample 4.1

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable A contains a value that is greater than the value of variable Y, for example, X = 12 and Y = 10. Note that here we are doing structural test *design* first, since we are choosing our input values in order ensure statement coverage.

Now, let's take another example where we will measure the coverage first. In order to simplify the example, we will regard each line as a statement. A statement may be on a single line, or it may be spread over several lines. One line may contain more than one statement, just one statement, or only part of a statement. Some statements can contain other statements inside them. In code sample 4.2, we have two read statements, one assignment statement, and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

```
1 READ X
2 READ Y
3 Z =X + 2*Y
4 IF Z> 50 THEN
5 PRINT large Z
6 ENDIF
```

Code sample 4.2

Although it isn't completely correct, we have numbered each line and will regard each line as a statement. Let's analyze the coverage of a set of tests on our six-statement program:

TEST SET 1
Test 1_1: X= 2, Y = 3
Test 1_2: X =0, Y = 25
Test 1_3: X =47, Y = 1

Which statements have we covered?

- In Test 1_1, the value of Z will be 8, so we will cover the statements on lines 1 to 4 and line 6.
- In Test 1_2, the value of Z will be 50, so we will cover exactly the same statements as Test 1_1.
- In Test 1_3, the value of Z will be 49, so again we will cover the same statements.

Since we have covered five out of six statements, we have 83% statement coverage (with three tests). What test would we need in order to cover statement 5, the one statement that we haven't exercised yet? How about this one:

Test 1_4: X = 20, Y = 25

This time the value of Z is 70, so we will print 'Large Z' and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

Note that Test 1_4 on its own is more effective which helps in achieving 100% statement coverage, than the first three tests together. Just taking Test 1_4 on its own is also more efficient than the set of four tests, since it has used only one test instead of four. Being more effective and more efficient is the mark of a good test technique.

What is Decision coverage? Its advantages and disadvantages

- Decision coverage also known as branch coverage or all-edges coverage.
- It **covers both the true and false conditions** unlikely the statement coverage.
- A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage
- A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement. With an IF statement, the exit can either be TRUE or FALSE, depending on the value of the logical condition that comes after IF.

Advantages of decision coverage:

- To validate that all the branches in the code are reached
- To ensure that no branches lead to any abnormality of the program's operation
- It eliminate problems that occur with statement coverage testing

Disadvantages of decision coverage:

- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

The decision coverage can be calculated as given below:

$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$

In the previous section we saw that just one test case was required to achieve 100% statement coverage. However, decision coverage requires each decision to have had both a True and False outcome. Therefore, to achieve 100% decision coverage, a second test case is necessary where A is less than or equal to B which ensures that the decision statement 'IF A > B' has a False outcome. So one test is sufficient for 100% statement coverage, but two tests are needed for 100% decision coverage. It is really very important to note that **100% decision coverage guarantees 100% statement coverage, but *not* the other way around.**

```
1 READ A
2 READ B
3 C = A – 2 *B
4 IFC <0THEN
5 PRINT "C negative"
6 ENDIF
```

Code sample 4.3

Let's suppose that we already have the following test, which gives us 100% statement coverage for code sample 4.3.

TEST SET 2 Test 2_1: A = 20, B = 15

The value of C is -10, so the condition 'C < 0′ is True, so we will print 'C negative' and we have executed the True outcome from that decision statement. But we have not executed the False outcome of the decision statement. What other test would we need to exercise the False outcome and to achieve 100% decision coverage?

Before we answer that question, let's have a look at another way to represent this code. Sometimes the decision structure is easier to see in a control flow diagram (see Figure 4.4).

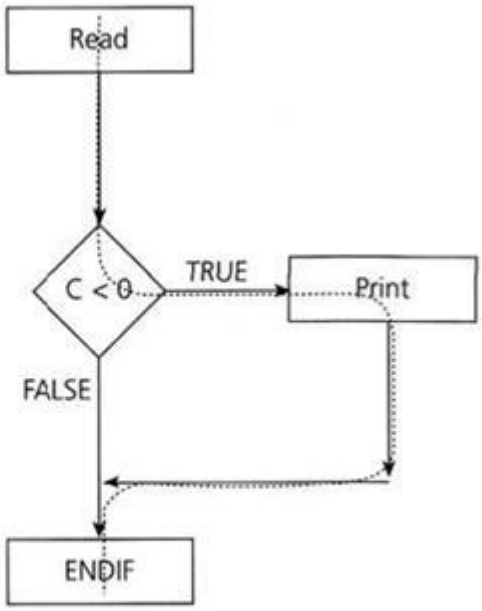


FIGURE 4.4 Control flow diagram for code sample 4.3

The dotted line shows where Test 2_1 has gone and clearly shows that we haven't yet had a test that takes the False exit from the IF statement.
Let's modify our existing test set by adding another test:

TEST SET 2
Test 2_1: A = 20, B = 15
Test 2_2: A = 10, B = 2

This now covers both of the decision outcomes, True (with Test 2_1) and False (with Test 2_2). If we were to draw the path taken by Test 2_2, it would be a straight line from the read statement down the False exit and through the ENDIF. We could also have chosen other numbers to achieve either the True or False outcomes.

What is Condition coverage?

- This is closely related to decision coverage but has better sensitivity to the control flow.
- However, full condition coverage does not guarantee full decision coverage.
- Condition coverage reports the true or false outcome of each condition.
- Condition coverage measures the conditions independently of each other.

Other control-flow code-coverage measures include linear code sequence and jump (LCSAJ) coverage, multiple condition coverage (also known as condition combination coverage) and condition determination coverage (also known as multiple condition decision coverage or modified condition decision coverage, MCDC). This technique requires the coverage of all conditions that can affect or determine the decision outcome.

How to choose that which testing technique is best?

How to choose that which technique is best? This is the wrong question!
Each technique is good in its own way in finding out the certain kind of defect, and not as good for finding out the other kind of defects. For example, one of the benefits of structure-based techniques is that they can find out the defects or things in the code that aren't supposed to be there, such as 'Trojan horses' or other malicious code. However, if there are parts of the specification that are missing from the code, only specification-based techniques will find that, structure-based techniques can only test what is there. If there are things missing from the specification and from the code, then only experience based techniques would find them.
Hence, each individual technique is aimed at particular types of defect. For example, state transition testing is unlikely to find boundary defects.

So, how to choose which testing technique is best, decision will be based on a number of factors, both internal and external.

The **internal factors** that influence the decisions about which technique to use are:

- **Models used in developing the system-** Since testing techniques are based on models used to develop that system, will to some extent govern which testing techniques can be used. For example, if the specification contains a state transition diagram, state transition testing would be a good technique to use.

- **Testers knowledge and their experience -** How much testers know about the system and about testing techniques will clearly influence their choice of testing techniques. This knowledge will in itself be influenced by their experience of testing and of the system under test.

- **Similar type of defects -** Knowledge of the similar kind of defects will be very helpful in choosing testing techniques (since each technique is good at finding a particular type of defect). This knowledge could be gained through experience of testing a previous version of the system and previous levels of testing on the current version.

- **Test objective -** If the test objective is simply to gain confidence that the software will cope with typical operational tasks then use cases would be a sensible approach. If the objective is for very thorough testing then more rigorous and detailed techniques (including structure-based techniques) should be chosen.

- **Documentation –** Whether or not documentation (e.g. a requirements specification) exists and whether or not it is up to date will affect the choice of testing techniques. The content and style of the documentation will also influence the choice of techniques (for example, if decision tables or state graphs have been used then the associated test techniques should be used).

- **Life cycle model used -** A sequential life cycle model will lend itself to the use of more formal techniques whereas an iterative life cycle model may be better suited to using an exploratory testing approach.

The **external factors** that influence the decisions about which technique to use are:

- **Risk assessment -** The greater the risk (e.g. safety-critical systems), the greater the need for more thorough and more formal testing. Commercial risk may be influenced by quality issues (so more thorough testing would be appropriate) or by time-to-market issues (so exploratory testing would be a more appropriate choice).

- **Customer and contractual requirements -** Sometimes contracts specify particular testing techniques to use (most commonly statement or branch coverage).

- **Type of system used -** The type of system (e.g. embedded, graphical, financial, etc.) will influence the choice of techniques. For example, a financial application involving many calculations would benefit from boundary value analysis.

- **Regulatory requirements -** Some industries have regulatory standards or guidelines that govern the testing techniques used. For example, the aircraft industry requires the use of equivalence partitioning, boundary value analysis and state transition testing for high integrity systems together with statement, decision or modified condition decision coverage depending on the level of software integrity required.

- **Time and budget of the project -** Ultimately how much time there is available will always affect the choice of testing techniques. When more time is available we can afford to select more techniques and when time is severely limited we will be limited to those that we know have a good chance of helping us find just the most important defects.