Test Case Analysis and Design 6

Key Topics

Functional and non-functional requirements
Requirement traceability
Black box testing
White box testing
Statement coverage
Branch coverage
Equivalence partitioning
Boundary value analysis
Experienced-based testing
Decision tables
State transition testing
Use-case testing

6.1 Introduction

Test case analysis and design are concerned with analysing the requirements to determine the test conditions and designing the test cases (using various techniques). The requirements and test conditions are used to specify the test cases, where each test case includes input, a procedure for carrying out the test, and the expected results. The quality of the testing is influenced by the quality of the test cases, and they need to be designed to cover the requirements. Traceability of the

G. O'Regan, *Concise Guide to Software Testing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-28494-7_6

test cases to the requirements ensures that the test cases are sufficient to verify that all of the requirements have been implemented and tested.

The user requirements specify what the customer wants and define *what* the software system is required to do, as distinct from *how* this is to be done. They are determined from discussions with the customer to determine their actual needs, and they are then refined into the *system requirements*, which state the *functional* and *non-functional* requirements of the system. The requirements must be precise and unambiguous to ensure that all stakeholders are clear on what is (and what is not) to be delivered, and prototyping may be employed to clarify the requirements and to assist in their definition.

Requirement verification is concerned with ensuring that the requirements are properly implemented (i.e. *building it right*). In other words, it is concerned with ensuring that the requirements are properly addressed in the design and implementation, and a traceability matrix and testing are often employed as part of the verification activities.

Requirement validation (i.e. *building the right system*) is concerned with ensuring that the right requirements are defined, and that they are precise, complete, consistent, and realizable and reflect the actual needs of the customer. The validation is done by the stakeholders, and it may include several reviews of the requirements (and prototype), reviews of the design, and user acceptance testing.

The software design of the system is a blueprint of the solution of the system to be developed. It is concerned with the high-level architecture of the system, as well as the detailed design that describes the algorithms and functionality of the individual programs. The system architecture may include hardware such as personal computers and servers, as well as the definition of the subsystems with the various software modules and their interfaces. The choice of the architecture of the system is a key design decision, as it affects the performance and maintainability of the system.

The role of software testing is to reduce the risk of defects being present in the software and to increase confidence in its correctness. Test case design is concerned with the design and specification of the test cases to verify and validate the requirements and design. The traceability matrix is an effective way of verifying that the test cases cover all of the requirements and design. It involves mapping the requirements and design to the unit test cases; the system test cases; and the UAT test cases, and the traceability matrix provides a crisp summary of how the requirements have been implemented and tested.

Several types of testing are performed during the project, including unit, integration, system, regression, performance, and user acceptance testing. The software developers perform the unit testing, and the objective is to verify the correctness of a module. This type of testing is termed *"white box"* testing (also known as structural testing) and is based on knowledge of the internals of the software module. White box testing typically involves checking that every path in a module has been tested, and it involves defining and executing test cases to ensure code and branch coverage.

The objective of *"black box"* testing (also called specification-based testing) is to verify the functionality of a module (or feature or the complete system itself), and knowledge of the internals of the software module is not required. There are several specification-based techniques, which are used including use-case testing, equivalence partitioning, boundary value analysis, decision tables, and state transition testing.

6.2 Requirement Engineering

The process of determining the requirements for a proposed system involves discussions with the relevant stakeholders to determine their needs and to explicitly define what functionality the system should provide, as well as any hardware and performance constraints.

The specification of the requirements needs to be precise and unambiguous to ensure that all parties involved share a common understanding of the system and fully agree on what is to be developed and tested. A feasibility study may be needed to demonstrate that the requirements are feasible, and may be implemented within the defined schedule and cost constraints.

The requirements are the foundation for the system, and it is therefore essential that the requirements are *complete* (all services required by the user are defined), *consistent* (requirements should not contradict one another), and *unambiguous* (the requirements are clear and definite in meaning).

Prototyping may be employed to assist in the definition and validation of the requirements, and a suitable prototype will include key parts of the system. It will allow users to give early feedback on the proposed system and on the extent to which it meets their needs. Prototyping is useful in clarifying the requirements and helps to reduce the risk of implementing the incorrect solution.

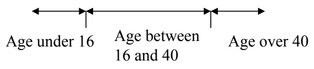
We distinguish between the user (or business) requirements and the system requirements. The *user requirements* are the high-level requirements for the system (they tend to be high-level statements in a natural language with diagrams and tables), whereas the *system requirements* are a more detailed description of what the system is to do.

The system requirements include the functional and non-functional requirements. A *functional requirement* is a statement about the functionality of the system: i.e. a description of the behaviour of the system and how it should respond to particular inputs. A *non-functional requirement* is a constraint on the functionality of the system (e.g. a timing, performance, reliability, availability, portability, usability, safety, security, dependability, or a hardware constraint).

It is essential that the functional and non-functional requirements are stated precisely, and the *non-functional requirements are often quantitatively specified* so that it may be objectively determined (by testing) whether they are satisfied or not. Further, it is essential that the non-functional requirements are satisfied, as otherwise the delivered system may be unusable or unacceptable to the client. The non-functional requirements often affect the overall architecture of the system, rather than the individual components of the system.

6.3 Test Case Design Techniques

Test design involves defining the test conditions and test cases from the requirements and design documents, and establishing traceability to them to ensure that the test conditions/test cases are sufficient to verify the requirements/design. Each test case includes a test objective, input to the test case, the test procedure, and the expected results (Fig. 6.1).



The test objective may be to find defects, obtain information on the quality of the software, provide confidence in the quality of the software, or prevent defects. The requirements/test conditions are the starting point for the specification of the test case, and the inputs are then identified. The test procedure is sufficiently detailed so that no expertise is required for execution of the test case, and test repeatability is guaranteed.

The formality of the testing depends on several factors including the maturity of the organization, the context of testing, the time constraints, and personnel involved. It is important that the testing is independent of the software development group. There are several test design techniques used for white box/black box testing (Table 6.1), which are discussed in more detail later in the chapter.

6.3.1 Black Box Testing

The objective of black box testing (also called *specification-based testing*) is to verify that the functionality of a module (or feature or the complete system itself)

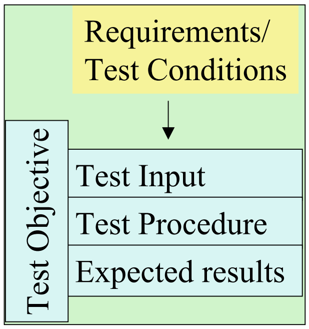


Fig. 6.1 Test case specification

Table 6.1 Test design techniques

Type	Sub-type	Description
Black box testing (specification-based testing)	*Overview* Equivalence partitioning Boundary value Analysis Decision table State transition Use-case testing	Based on functional and non-functional requirements Use of equivalence classes for similar behaviour to reduce the number of test cases This is based on the fact that many defects occur on the borders (boundaries) Determine causes and effects; create decision table and test cases Determine the states and events and valid and invalid transitions. Prepare test cases to verify correct behaviour at transitions Determine users and their interactions (scenarios). Test cases are based on users and their interactions
White box testing (structure-based testing)	*Overview* Statement testing Decision testing	Test conditions/test cases are based on an analysis of the internal structure of the software. It is an indication of test coverage Goal is to achieve 100% statement coverage in testing Decision coverage is the percentage of decision outcomes that have been exercised in testing
Experienced-based testing	*Overview* Error guessing Exploratory testing	Testing is based on knowledge and experience of the tester Experience/knowledge of tester is used to anticipate what defects might be present in the software Tests are designed and executed at the same time (sometimes, they may not be recorded)

satisfies the requirements, and knowledge of the internals of the software module is not required. There are several popular specification-based techniques such as equivalence partitioning, boundary value analysis, decision tables, state transition testing, and use-case testing.

Equivalence Partitioning

Equivalence partitioning involves dividing the test input data into a set of classes and selecting one input value from each class. The input selected is representative of the class, and this approach reduces an extremely large number of test cases (if all input data is used) to a manageable number of effective test cases.

For example, consider an application program that accepts an input range from 1 to 100. Then, this leads to three equivalence classes with one for valid input between 1 and 100, and two classes for invalid input (one for any data value below 1 and one for any value above 100).

Boundary Value Analysis

Boundary value analysis is an extension to equivalence partitioning and is based on the fact that many defects occur on the boundaries of the input domain (lower/upper values), rather than in the centre of the data range. Each boundary has a valid boundary value and an invalid boundary value, and the test cases are based on both valid and invalid boundary values.

For each input that concerns a range of values, the test cases are based on the boundaries and values just outside the boundaries. For example, the exact boundary values of the input range 1–100 are 1 and 100, and values just outside the boundaries are 0, 2, 99, and 101. For a range $\{r_1 \ldots r_2\}$, the exact boundary values are r_1, r_2, and the values just outside the boundaries are $r_1 - 1$, $r_1 + 1$, $r_2 - 1$, and $r_2 + 1$.

Decision Tables

A decision table (sometimes referred to as a cause–effect table) is a systematic way of stating business rules, and it provides a useful way to deal with a combination of different inputs and states. It is useful for testers when exploring combinations of different inputs and states that must correctly implement the business rules.

The use of a decision table in test design involves studying the specification, determining causes (including triggers) and effects (including actions), creating the decision table, and defining the test cases. The causes are required to be single conditions only with compound conditions split up, and the conditions are formulated in a positive way. All the combinations of true and false for the conditions are listed (if there are n conditions, then since there are two choices for each condition, there are 2^n combinations for n of them), and the corresponding action(s) to be carried out for that combination defined. A combination is sometimes referred to as a rule.

The analysis of the combinations may lead to combinations that were not mentioned in the original specification, and this is generally interpreted as that an error message should be displayed (i.e. another action may need to be added to display an error message). Further, the analysis of the combinations is useful in identifying omissions or ambiguities in the specification.

For example, consider a credit card system where there are three conditions on the discount received depending on whether you are a new customer, an existing customer with a loyalty card, or if you have a coupon. A new customer will receive a discount of 12% for today only, whereas an existing customer with a loyalty card will get a discount of 10%, and a person who has a coupon receives a discount of 15% today (but this offer cannot be used with the new customer discount).

The rules defining the discounts available today are summarized in the decision table in Table 6.2.

Each column of the decision table is formed from a combination of the conditions and represents the business rules (R1–R8). The approach to testing is generally to test each rule separately (if dealing with a small number of combinations),

Table 6.2 Decision table with business rules

Conditions	R1	R2	R3	R4	R5	R6	R7	R8
New customer	T	T	T	T	F	F	F	F
Existing/loyalty	T	T	F	F	T	T	F	F
Coupon	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X*	X*	15	12	25	10	15	0

*X means that this is an invalid combination

and if there are a large number of combinations then a representative sample is chosen (this may involve prioritization of the combinations). There would generally be one test for each rule of the decision table, and this helps in finding defects.

The combinations of rules R1 and R2 should not arise, and an X for the discount indicates this (i.e. you cannot both be a new customer and have a loyalty card, and so there should be an error message). The discount for R3 is 15% [i.e. the maximum of the new customer discount (12%) and the coupon discount (15%)]. The discount for R5 is 25% [i.e. the sum of the existing customer discount (10%) and the coupon discount (15%)]. Rules R4, R6, and R7 have only one type of discount, and rule R8 had no discount (i.e. 0%).

State Transition Testing

State transition testing is employed when part of the system may be described with a finite state machine. That is, the system may be in a finite number of states, and the transitions from one state to another are determined by the rules of the machine. A state transition model has several parts including:

– States of the system
– Transitions from one state to another
– Events that cause a transaction
– Actions that result from a transition.

A state diagram is used to illustrate the finite state machine model of the system, where the states are shown as circles, the transitions are shown as arrows between circles, and events are shown as text near the transitions. State transition testing is useful at identifying defects at transitions and in identifying behaviour when none should be present.

The state diagram in Fig. 6.2 is a simplified model of PIN authentication for an ATM. It has five states and four possible events (Card Inserted, Enter PIN, Invalid PIN, and Valid PIN).

The test cases are then derived from typical scenarios such as the normal situation where the PIN is entered correctly, and then a second scenario where an incorrect PIN is entered. Test conditions may be derived from a state graph in various ways, as each state and each transition may be treated as a test condition.

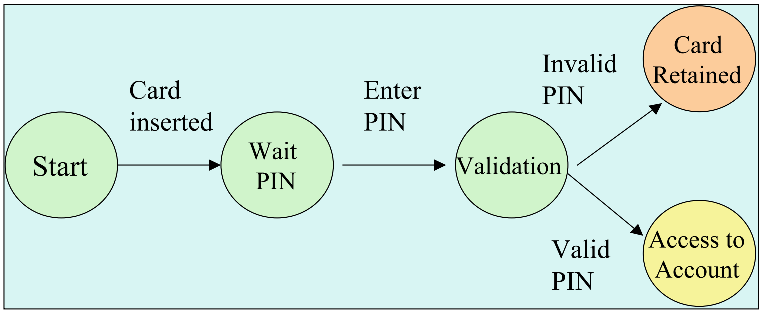


Fig. 6.2 State diagram for PIN authentication

A state transition table shows the state that a finite state machine will move to from its current state and the input. It is useful in designing test cases for *n*-switch coverage, where the state transition table consists of the start state, the inputs, the expected output, and end state. The testing of all valid transitions is known as "0-switch" coverage. It is also possible to consider transition pairs (1-switch coverage) or triples (2-switch coverage), as well as coverage of *n* transitions (*n*-1-switch coverage).

Use-Case Testing

A use-case diagram models the dynamic aspects of the system, and it shows a set of use cases and actors and their relationships. It describes scenarios (or sequences of actions) in the system from the user's viewpoint (actor) and shows how the actor interacts with the system. An actor represents the set of roles that a user can play, and the actor may be human or an automated system.

Use-case testing is a form of testing that identifies test cases to exercise the whole system based on the interactions that a user has with the system. Each use case has a basic path/flow as well as failure scenarios.

A use-case diagram shows a set of use cases, with each use case representing a functional requirement. Use cases are employed to model the visible services that the system provides within the context of its environment, and for specifying the requirements of the system as a black box. Each use case carries out some work that is of value to the actor, and the behaviour of the use case is described by the flow of events in text. The description includes the main flow of events for the use case and the exceptional flow of events.

Use cases provide a way for the end users and developers to share a common understanding of the system (Jacobson et al. 1999). They may be applied to all or part of the system (subsystem), and the use cases are the basis for development and testing. The use cases describe the process flow through a system based on the likely use of the system, and the test cases derived from the use cases are very useful in finding real-world defects that users are most likely to find.

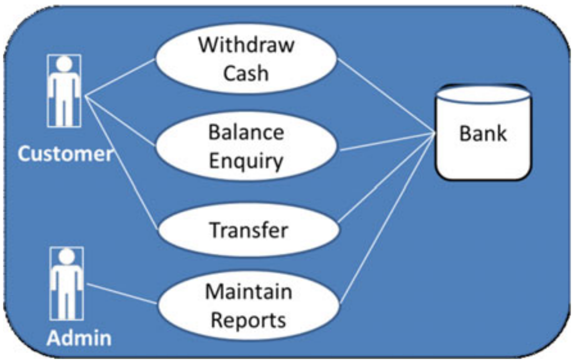


Fig. 6.3 Use-case diagram of ATM

An ellipse is used to represent a use case graphically, and Fig. 6.3 presents a simple example of use cases in an ATM application. The typical user operations at an ATM include the balance inquiry operation, cash withdrawal, and the transfer of funds from one account to another. The actors for the system include "customer" and "admin", and these actors have different needs and expectations of the system.

The behaviour from the user's viewpoint is described, and the use cases include "withdraw cash", "balance enquiry", and "transfer", whereas the behaviour for the admin actor is to "maintain/reports". The use-case view includes the actors who are performing the sequence of actions.

Each use case must specify the preconditions that must be satisfied for the use case to work and the post-conditions that must be satisfied following the execution of the use case. Each use case has a most likely scenario and possibly additional scenarios covering special cases or exceptional conditions. Use-case testing involves testing the main scenario and one for each alternate scenario.

The advantages of use-case testing are that the main tasks/functions of the system are being tested, and that the testing is user-focused and designed to find real-world defects.

6.3.2 White Box Testing

White Box Testing (also called *structure-based techniques*) involves the design of test conditions and test cases based on an analysis of the internal structure of the software. It may be used to measure the coverage of existing test cases as well as to derive further test cases to increase coverage. We distinguish between *statement coverage* (where 100% statement coverage means that every statement is executed at least once) and *decision coverage* (*branch coverage*), which has the additional requirement that all possible outcomes of every decision will be tested. There are several types of white box testing including statement testing and decision testing.

Statement Testing

A statement is a syntactic entity in a programming language that expresses an action to be performed. A program is a collection of one or more statements, and a statement may include expressions. The objective of statement coverage is to determine the percentage of statements executed during testing and to determine where the code has not been executed due to blocked tests. The objective is to process and execute every line of code, and the statement coverage is given by the following formula:

$$\text{Statement Coverage} = \frac{\text{Number Statements Executed}}{\text{Total Number of Statements}} \times 100\%$$

Statement coverage tests cannot test false conditions in a statement (i.e. if the condition part of an if statement is false, then the statement is not executed). For example, consider the following code fragments written in a C like syntax.

Fragment A	Fragment B
1. x = 2;	1. x = 3;
2. y = 3;	2. y = 2;
3. if (x > y)	3. if (x > y)
4. x = x + 1	4. x = x + 1

There are four statements in each fragment. However, three statements are executed for fragment one (i.e. the statement coverage is 75%), whereas four statements are executed for fragment two (i.e. the statement coverage is 100%).

Decision Testing

Decision testing (also called branch testing) covers both the true and the false conditions as distinct from statement testing, which covers only true conditions. A decision is a point in the program where the control flow has two or more alternative routes, and decision coverage (branch coverage) is the percentage of decision outcomes that have been executed by the suite of tests.

$$\text{Decision Coverage} = \frac{\text{Number Decision Outcomes executed}}{\text{Total Number of Decision Outcomes}} \times 100\%$$

A branch is the outcome of a decision, and branch coverage measures the extent to which decision outcomes have been tested. It is applicable to statements such as *if statements*, *while statements*, *repeat statements*, and *case statements*. The branch coverage metric is given by the formula above. Decision testing measures the extent to which all branches in the software code have been reached, and 100% decision coverage guarantees 100% statement coverage (but not the other way around). The following code fragment illustrates decision testing.

```
1. if (x > y)
2.       x = x + 1
3. else
4.    x = x − 1
```

Then, we need two test cases to achieve 100% decision coverage, and the following two test cases will suffice:

Test Sets

T1: x is 3 and y is 1
T2: x is 3 and y is 4

6.3.3 Experienced-Based Testing

The approach in experienced-based testing is to use the knowledge and experience of the tester to prepare test conditions and test cases. The personnel involved have previous experience of similar projects and are familiar with the software and its environment. They often have an insight into what could go wrong with the software, which is potentially very useful in identifying defects during testing.

These techniques are generally used after the completion of black box and white box testing, rather than as a replacement for these activities. The two main experienced-based techniques are error guessing and exploratory testing.

Error Guessing
This approach uses the experience of the tester to anticipate which defects might be present in the software. Often, the form of testing proceeds without any formal test documentation and relies on the knowledge and expertise of the tester. The success of this type of testing is dependent on the expertise and skills of the tester, as experienced testers will know where the defects are most likely to be. However, the disadvantage with error testing is that it is not repeatable as such, and it is random in the sense that it is trying things out rather than the normal planned activities.

Exploratory Testing
Exploratory testing is concerned with exploring the software and determining what works and does not work. The tester decides what to test next and where to spend the limited amount of time available, and tests are often designed and executed at the same time. The objective is to spend the minimal time in planning and the maximum time in execution.

Test design and execution activities are performed in parallel without formally documenting the test conditions and test cases. This type of testing is often performed after the more formal testing has been completed as a check to ensure that the most serious defects have been identified in the software.

6.4 Test Case Specification

Several types of testing that may be performed were described in Table 3.1, and there is often a separate test plan for unit, system, and UAT. The unit tests are based on the software design, the system tests are based on the system requirements, and the UAT tests are based on the user requirements. The dedicated test plans generally include a planning section as well as the test scripts for the testing (Tables 6.3 and 6.4).

Each of these test plans contains test scripts (e.g. the unit test plan contains the unit test scripts and so on), and the test scripts are traceable to the design (for the unit tests) and for the system requirements (for the system test scripts). The unit tests are more focused on white box testing, whereas the system test and UAT tests are focused on black box testing.

Each test script contains an objective(s) and the procedure by which the test is carried out. A test script generally includes:

– Test case ID
– Test type (e.g. unit, system, UAT)
– Objective(s)/description
– Test script steps (for each objective)
– Expected results
– Actual results
– Tested by.

Regression testing involves carrying out a subset of the defined tests to verify that the core functionality of the software remains in place following changes to the system (e.g. correction of defects or addition of new functionality).

The test plans are often described in a test document, but they could also be defined using a test management tool (see Chap. 10 for a discussion of tools for software testing). The dedicated test plan will detail the planning specific to the type of testing being conducted including the testing to be performed, the approach, the resources and training required, the planned preparation dates, the test environment

Table 6.3 Planning section in dedicated test plan

Description (including objectives)	
Approach	
Resources and responsibilities	
Training required	
Preparation dates	
Testing dates	
Test environment	
Testing tools	
Entry criteria	
Exit criteria	
Features to test	

Table 6.4 Template for test case

TEST CASE TEMPLATE				
Title			**Req ID**	
Author			**Test Case ID**	
Date			**Test Type**	
Objective	**Objective Description**			
#	Enter context of Objective # here			
#	Enter context of Objective # here			
#	Enter context of Objective # here			
#	Enter context of Objective # here			
#	Enter context of Objective # here			
Test Objective #	Enter Test Objective			
Test Input	Enter Input for test			
Test Script Step(s)	Enter Test Script Step(s) here			
Expected Results	Enter Expected Results of test			
Actual Results	Enter Actual Results of test			
Data Location	Enter Location of Test Data here			
	Name	**Test Date**	**Test Status**	**Defect No.**
Tested By				
Reviewed By				
Approved By				
Test Objective #	Enter Test Objective			
Test Input	Enter input for test			
Test Script Step(s)	Enter Test Script Step(s) here			
Expected Results	Enter Expected Results of test			
Actual Results	Enter Actual Results of test			
Data Location	Enter Location of Test Data here			
	Name	**Test Date**	**Test Status**	**Defect No.**
Tested By				
Reviewed By				
Approved By				

and test tools required, and the entry and exit criteria. It will generally include risks and assumptions, as well as a traceability matrix that maps the test cases to the requirements or design.

The test plan will contain the test scripts to carry out the testing, where a test script defines the steps required to carry out a particular test (Table 6.4). Each test script is based on the test conditions that have previously been defined, and may contain several test objectives (or subtests). Each test objective includes a description, the inputs to the test case, the test procedure (or steps required to carry out the test), the expected output, the actual results, and whether the test passes or fails. The results of the test scripts will be recorded including details of who carried out the tests.

Table 3.1 describes the types of testing that may be carried out during the project such as unit testing, component testing, system testing, performance testing, load/stress testing, browser compatibility testing, usability testing, security testing, regression testing, test simulation, and acceptance testing.

6.5 Requirement Traceability

Requirement traceability was briefly discussed in Sect. 3.11 and provides a way to verify that all of the defined requirements for the project have been implemented and tested. One way to do this is to consider each requirement number and to go through every part of the design document to find where the requirement is being implemented in the design, and similarly to go through the test documents and find any reference to the requirement number to show where it is being tested. This would demonstrate that the particular requirement number has been implemented and tested.

A more effective way to do this is to employ a traceability matrix (Table 6.5), which may be employed to map the user requirements to the system requirements; the system requirements to the design; the design to the unit test cases; the system test cases; and the UAT test cases. The matrix provides a crisp summary of how the requirements have been implemented and tested.

The traceability of the requirements is *bidirectional*, and the traceability matrix may be maintained as a separate document or as part of the requirement document. The basic idea is that a mapping between the requirement numbers and the sections of the design or test plan is defined, and this provides confidence that all of the requirements have been implemented and tested.

Requirements will usually be numbered, and a single requirement number may map on to several sections of the design or to several test cases: i.e. the mapping may be *one to many*. The traceability matrix provides the mapping between the individual requirement numbers and the sections in the design or test plan corresponding to the particular requirement number.

Table 6.5 Sample trace matrix

Requirement No.	Sections in design	Test cases in test plan
Rl.l	D1.4, D1.5, D3.2	T1.2, T1.7
R1.2	D1.8, D8.3	T1.4
R1.3	D2.2	T1.3
…	…	…
R1.50	D20.1, D30.4	T20.1, T24.2

It is essential to keep the traceability matrix up to date during the project and especially after changes to the requirements. The traceability matrix is useful in determining the impacts of a proposed change to the requirements, as it enables the impacts on other requirements and project deliverables to be easily determined.

6.6 Review Questions

1. What is the difference between a functional and non-functional requirement?
2. What is the difference between requirement verification and validation?
3. Explain the difference between black box testing and white box testing.
4. Describe the main specification-based techniques used in black box testing.
5. Describe the main structure-based techniques used in white box testing.
6. Describe the main experienced-based testing techniques.
7. What is the purpose of requirement traceability?
8. Explain the difference between statement coverage and branch coverage.
9. Explain use-case testing.
10. Explain equivalence partitioning and boundary value analysis.

6.7 Summary

Test analysis and design are concerned with analysing the requirements to determine the test conditions and designing the test cases (using various techniques) for the testing. The requirements and test conditions are used to specify the test cases, where each test case includes input, the procedure for carrying out the test, and the expected results. The quality of the testing is influenced by the quality of the test cases. Traceability of the test cases to the requirements is essential in ensuring that the testing is sufficient to verify that all of the requirements have been implemented.

The user requirements specify what the customer wants and define what the software system is required to do, as distinct from how this is to be done. The requirements are the foundation for the system, and the process of determining the requirements, analysing and validating them, and managing them throughout the project lifecycle is termed requirement engineering.

The objective of black box testing is to verify that the functionality of a module (or feature or the complete system itself) satisfies the requirements, and knowledge of the internals of the software module is not required. There are several popular specification-based techniques such as equivalence partitioning, boundary value analysis, decision tables, state transition testing, and use-case testing.

White box involves the design of test conditions and test cases based on an analysis of the internal structure of the software. It may be used to measure the coverage of existing test cases, and white box testing includes statement testing and decision testing.

The approach in experienced-based testing is to use the knowledge and experience of the tester to prepare test conditions and test cases. The personnel involved often have an insight into what could go wrong with the software, which is useful in identifying defects.

The objective of requirement traceability is to verify that all of the defined requirements for the project have been implemented and tested. The traceability matrix provides a crisp summary of how the requirements have been implemented and tested.

Reference

Jacobson I, Booch G, Rumbaugh J (1999) The unified software modelling language user guide. Addison-Wesley, Boston