Practical Guide to Software System Testing

© Dr Kelvin Ross
# Course Structure

<table>
<thead>
<tr>
<th>Time</th>
<th>Day 1</th>
<th>Day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1) Testing Fundamentals</td>
<td>Recap</td>
</tr>
<tr>
<td>10:15 –10:35</td>
<td>Morning Tea and Discussion</td>
<td>Morning Tea and Discussion</td>
</tr>
<tr>
<td>10:30 –12:00</td>
<td>(2) Testing Processes</td>
<td>(6) Test Automation</td>
</tr>
<tr>
<td>12:00 –13:00</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>13:00 –14:40</td>
<td>Forum – Test Strategy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3) Black Box Testing</td>
<td></td>
</tr>
<tr>
<td>14:40 –15:00</td>
<td>Afternoon Tea and Discussion</td>
<td>Afternoon Tea and Discussion</td>
</tr>
<tr>
<td>15:00 –17:00</td>
<td>Exercise – Black Box Testing</td>
<td>Close</td>
</tr>
<tr>
<td></td>
<td>(4) Testing Infrastructure</td>
<td></td>
</tr>
</tbody>
</table>
## Contents

1. **TESTING FUNDAMENTALS** .................................................................................. 1  
   1.1 Testing Introduction ......................................................................................... 2  
   1.2 Testing Myths .................................................................................................. 4  
   1.3 A Brief Testing Exposé ................................................................................... 5  
   1.4 Purpose of Testing .......................................................................................... 7  
   1.5 Why Exhaustive Testing is Not Possible ....................................................... 8  
   1.6 Understanding Software Defects .................................................................... 11  
   1.7 Costs Associated with Testing and Defects ................................................... 12  
   1.8 Risks-Based Testing ....................................................................................... 18  
   1.9 Careers in Software Testing .......................................................................... 21  

2. **TESTING PROCESSES** .................................................................................... 23  
   2.1 The Testing Process ......................................................................................... 24  
   2.2 Kinds of Testing .............................................................................................. 27  
   2.3 Testing Strategy ............................................................................................... 33  
   2.4 Test Plans ........................................................................................................ 40  

3. **BLACK BOX TESTING** .................................................................................... 42  
   3.1 Purpose of Black-Box Testing ....................................................................... 43  
   3.2 Systematically Decomposing Requirements into Tests .............................. 44  
   3.3 Black-Box Testing Techniques ...................................................................... 46
1. Testing Fundamentals

3.4 Black Box Test Coverage Metrics...........................................74
3.5 Test Oracles ........................................................................75
3.6 True Oracle..........................................................................76
    Sampling Oracle ..................................................................76
3.8 Heuristic Oracle ..................................................................77

4. Testing Infrastructure ...............................................................79
4.1 Test Environments ..................................................................80
4.2 Test Documentation ..................................................................84
4.3 Configuration Management and Change Control ....................89

5. White Box Testing ....................................................................92
5.1 Purpose of White-Box Testing ................................................93
5.2 Statement, Branch and Path Coverage .....................................94
5.3 Loop Testing .........................................................................102
5.4 Branch Condition Testing ......................................................104
5.5 White-Box Techniques Conclusions ........................................107
5.6 Complexity Metrics ..............................................................108
5.7 Lines of Code ........................................................................108

6. Incident Tracking ......................................................................111
6.1 Purpose of Incident Tracking ..................................................112
6.2 Incident Tracking Activities ...................................................112
6.3 Incident Life Cycles ...............................................................117
6.4 Incident Status Reports ..........................................................118
6.5 Incident Tracking System Functional Requirements .................128

7. Automated Test Tools ...............................................................129
7.1 Introduction ..........................................................................130
7.2 Types of Testing Tools ..........................................................131
7.3 Automated Test Execution ......................................................132
7.4 Test Capture/Playback Tools ..................................................133
7.5 Why Test Automation ...........................................................134
7.6 Regression Testing ..............................................................135
7.7 What To Automate ...............................................................136
7.8 Efficiency of Automated Testing ............................................138
7.9 Pros and Cons of Automated Testing .......................................140
7.10 Adoption Process ................................................................142
7.11 Automation Readiness ........................................................143
8. TEST MANAGEMENT ...................................................................................... 145
  8.1 COMPOSITION OF THE TEST GROUP..................................................... 146
  8.2 ESTIMATING EFFORT AND COST .......................................................... 148
  8.3 SCHEDULING ACTIVITY ....................................................................... 149
  8.4 MONITORING PROGRESS ..................................................................... 151
  8.5 ALLOCATING RESOURCES .................................................................... 153
  8.6 TEST COMPLETION CRITERIA................................................................. 154

FURTHER READING ......................................................................................... 155

INDEX ............................................................................................................. 159
1. Testing Fundamentals

- introduction to software system testing
- testing myths
- testing expose
- why exhaustive testing is not possible
- sources of software defects
- costs associated with testing
- risk-based testing
- testing careers
1. Testing Fundamentals

1.1 Testing Introduction

Testing is all about finding defects in a system.

The impact of defects can be far reaching

- In critical systems (e.g. mission-, security-, safety-, financially-, environmentally-critical) the impact of defects and the failures they cause can be catastrophic

Noteworthy examples include
- London Ambulance Service
- Ariane 5
- Therac 25
- Year 2000

- Even for non-critical system, significant impact can result from disgruntled users
  - Intel floating point

Testing is primarily a defect detection mechanism, its purpose is to find defects that are already in the system.

Testing must also be balanced with defect prevention within the development process, its purpose to prevent defects being introduced in the first place. Defect prevention techniques include building prototypes, performing reviews of the code.

In testing, the number of bugs discovered is always proportional to the number of bugs left in the system, e.g. defect removal ratio of 80%.

To remove a greater proportion of defects will require exponentially increasing effort. Target of 100% defect removal is not feasible.

Testing is more than just running a few tests to see if the application is alive. Testing involves developing hundreds and in some cases thousands of test cases to test an application comprehensive. Testing is more than just running test cases, it involves significant effort and coordination:

- prior to testing, when planning and designing the tests to run
- following testing, when recording the results of tests

EXPERIENCE REPORT

I dealt with one person who thought that you should be able to test a system with two or three test cases. There was 30 seconds of stunned silence when I said in many cases my clients had developed many thousands of test cases and in one case had spent more than 20000 hours testing their application.
1.1.1 Goals of Course

- How to define a test strategy for your project
- How to carefully design the tests for your system
- How to build a test environment
- How to track defects to ensure they are resolved appropriately
- How to choose and adopt automated test tools
- How to manage testing resources
1.2 Testing Myths

Testing is mind-numbingly boring – appropriate for trained gorillas

Not necessarily, replaying tests yes. Designing effective tests is quite analytical, automated test bed creation even more analytical.

Testing is used to show a system has zero defects

Cannot show that the system is free of defects, only that it contains defects

No method (including formal proof) can show zero defects, as we can never safely assume a set of requirements is correct. At best we can verify against requirements.

Testing is best done once the system is in the field and being used.

The costs of fixing defects is significantly more expensive than finding and fixing them by testing.

Tests need only be run once or twice.

Tests will be repeated many times.

Tests need to be repeated after changes are made, as even minor modifications can lead serious defects being introduced.

Some testing processes will repeat the same test many times before the system is commissioned, e.g. system test, factory acceptance test, site acceptance test.

Programmers can do all the testing, after all they are they ones that wrote the code

Testing done by programmers can often be too subjective, and can have emotional attachments to their own code.

Testing done by testers independent of the developer of the code often take a fresh view of the code.

Developers quite often carry incorrect design assumptions across into the testing.
1.3 A Brief Testing Exposé

Is this the way you approach testing.

Consider the simple calculator.

The developer writing the code for the calculator decides to start testing after producing a first cut.

The tester starts with the addition operation:

\[
\begin{align*}
1 + 1 &= 2 \\
1 + 999 &= 1000 \\
\text{and so on.}
\end{align*}
\]

everything seems to be going well!

When testing subtraction the user finds they have not considered subtraction that results in a negative number.

\[
1 - 999 \text{ doesn't come out to } -998
\]

The developer goes in and fixes the code, and adds negative numbers.

The go back and retest

\[
1 - 999 = -998
\]

and then other cases they haven’t considered so far.

Once the system is in operation a user discovers that calculator does not support the first operand of addition having a negative value. If we’re lucky the system crashes, a worse case scenario is that the incorrect result goes unnoticed and causes other downstream problems.

It has been some time since the developer last looked at the code. They finally track down the cause of the problem. The defect is that the addition logic is not conditional on whether the result in the accumulator is positive or negative.

The problem has cost the user in wasted time and frustration to discover there was a problem with the calculator and not with slippery fingers. It may have cost the user financial damages through down-stream problems. The problem has cost the developer, there is the time to interpret the users’ cryptic account of what is going wrong, the time to trace the problem back to the defect being in the addition function. There is also the cost of the developer having to send out patches to all users to correct the problem, and the lost credibility that comes with public bugs.

The problem went unnoticed as after the logic was initially updated to include negative numbers in the accumulator, but the other operations were never retested to determine if they worked correctly. Retesting is not feasible as no documentation was completed initially, and the effort to redesign the test is excessive.

In any event, it would be unlikely that the initial tests might have picked up the problem, as the initial specifications for addition did not include negative accumulator values. Without documentation of the test carried out it is difficult to determine what has been tested and what additional tests should be provided to test new areas of functionality.

There is a fair chance that a specification for the calculator functions would not have existed. In this scenario it would be difficult to determine what to test and what
represents correct operation, e.g. a specification should have stated what are the ranges of possible values to the addition function and what are values that could constitute test cases.

In the above scenario, testing is ad-hoc, ignoring any logical approach or method. During this course, we strive to replace this approach with one that makes testing more effective and efficient.

After they correct the discovered problem, how will the user go back and test before reissuing the calculator to the users?? Most developers don't learn from their catastrophes.
1.4 Purpose of Testing

Many long-winded arguments as to whether testing is to find bugs or demonstrate correctness.

The most logical viewpoint is that testing is to find defects.

“Testing is the process of executing a program with the intent of finding errors”

Glen Myers

Many have proposed that testing is to demonstrate the system is correct.

“Testing can only prove the presence of bugs, never their absence”

E. W. Dijkstra

Testing is the process of discovering errors in a software system. Testing success is finding an error, the more critical the more successful we are. Not finding an error is theoretically a failed test case.

The role of the tester is to protect their organisation from releasing software that is defective. However, as the tester cannot prove that there are no defects, their job is to attempt to find defects and provide them as evidence that the software is faulty. Somewhat like the justice system – innocent until proven guilty.

Based on the evidence provided by the testers, it is up to management to decide whether they can live with the level of identified defects, or to postpone release until the defects are resolved.

As we cannot prove the system correct, management may question why test at all if we are only going to find defects. The reason is that it is more economical to test to find defects than it is to risk it and fix the defects as they are discovered by the customers.

Testers are quite often grouped in with Quality Assurance. This is not appropriate, as quality assurance is looking at improving the whole development process to make it run more smoothly and effectively. Testers are more used by quality assurance as quality control, to check that certain levels of quality are observed as a result of improvements that have been made to the process.
1.5 Why Exhaustive Testing is NOT Possible

Many managers will propose that a software system should be tested for all combinations to prove that the software works.

This is not possible as the number of combinations supported in nearly all programs can be considered, for all practical purposes, infinite.

A comprehensive discussion of why it is impossible to test software for all combinations, the reader should refer to [Kan96c]. Here we provide a few examples as to why exhaustive testing is not feasible.

1.5.1 Can’t Test All Inputs

Consider the calculator that takes an signed 10 digit number as input (for one operand).

There are $10^{10}$ positive inputs and $10^{10} - 1$ negative inputs for the one parameter that constitute valid inputs for the first operand. Finite but very large.

However the tester must also consider invalid inputs. What about handling of alphabetic characters typed from a keyboard. What about if the number is in scientific notations, what about if it was an expression (e.g. $1 + 52$). All of a sudden we are dealing with potential infinite possibilities for the one input.

It gets worse when there are multiple inputs, as it is a combinatorial explosion. Say there were n inputs with m possible values for each input, there are $m^n$ combinations of inputs. For instance, 3 inputs, each with (not many) 1000 possible values = 1 billion combinations of input.

1.5.2 Can’t Test All Timing

Some systems respond to timing issues. For example, on my mobile phone if I delay pressing a key when entering a text string, the entry will move to the next character position to be entered rather than scrolling through the available characters for this position.

When testing with timing it is impossible to test for all possible delays, as time is continuous, i.e. between any two points in time there is always another point that could be chosen.

1.5.3 Can’t Test All Paths

Some people may argue that you can test just the paths of a program to overcome the problems above.

Refer to program structure below.

Plot all paths by beginning at START and finishing at END. At X can either go to END or loop back to A up to 19 times.

5 paths from A to X:
- ABCX
- ABDEGX
Only performing a single iteration, i.e. not returning along loop at X, results in 5 paths.

Two iterations of the decision logic, i.e. returning along loop at X the first time only, results in 25 paths (5 the first x 5 the second).

Three iterations of the decision logic has $25 = 5^3$. If the decision logic is executed 20 times then there are $5^{20}$ paths.

All combinations of loop options results in $5 + 5^2 + 5^3 + \ldots + 5^{20} = 10^{14}$ (100 trillion) paths.

The logic above is quite often found in straight-forward programs. For example, it can be as simple as a program that reads up to 20 records from a data file, stopping
if it runs out of records, and processing each record read according to it satisfying particular conditions.

```plaintext
Count := 0;
REPEAT
  Count := Count + 1;
  READ Record
  IF Record <> EOF THEN
    IF ShortRecord(Record) THEN
      /* Short Record */
      IF EarlyShortRecord(Record) THEN
        ProcessEarlyShort(Record)
      ELSE
        ProcessLateShort(Record)
      ENDIF
    ELSE
      /* Long Record */
      IF EarlyLongRecord(Record) THEN
        ProcessEarlyLong(Record)
      ELSE
        ProcessLateLong(Record)
      ENDIF
    ELSE
      WRITE "EOF Reached"
    ENDIF
  ENDIF
UNTIL Record = EOF OR COUNT > 20
```

If you could write, execute and verify a test case of each path of this simple program every second, it would take more than 3 million years to complete the tests.
1.6 Understanding Software Defects

Kaner, Falk and Nguyen [Kan93a] describe 13 major categories of software defects:

- User interface errors – the system provides something that is different from
- Error handling – the way the errors are recognised and treated may be in error
- Boundary-related errors – the treatment of values at the edges of their ranges may be incorrect
- Calculation errors – arithmetic and logic calculations may be incorrect
- Initial and later states – the function fails the first time it is used but not later, or vice-versa
- Control flow errors – the choice of what is done next is not appropriate for the current state
- Errors in handling or interpreting data – passing and converting data between systems (and even separate components of the system) may introduce errors
- Race conditions – when two events could be processed, one is always accepted prior to the other and things work fine, however eventually the other event may be processed first and unexpected or incorrect results are produced
- Load conditions – as the system is pushed to maximum limits problems start to occur, e.g. arrays overflow, disks full
- Hardware – interfacing with devices may not operate correctly under certain conditions, e.g. device unavailable
- Source and version control – out-of-date programs may be used where correct revisions are available
- Documentation – the user does not observe operation described in manuals
- Testing errors – the tester makes mistakes during testing and thinks the system is behaving incorrectly
1.7 Costs Associated with Testing and Defects

For a detailed discussion of tradeoffs between quality costs see [Kan93a] or [Kan96c].

1.7.1 Software Development is Likely to Fail

Previous software development projects have a high risk of failure to deliver on-time and within budget.

In 1994, the Standish Group examined 8380 Industry Software Project Developments to identify the scope of software project failures. They found that 53% were "challenged" (cost overruns, late delivery and/or missing capabilities); 31% were canceled during development; and 16% were successfully completed (ontime, within budget, full capability).


For large companies:

- 9% were successful
- 61.5% were challenged
  - Over-budget by 178%
  - Late by 230%
  - Missing 58% of capabilities
- 29.5% cancelled
1.7.2 Costs of Testing

Defect Prevention Costs | Defect Appraisal Costs
--- | ---
- Staff training | - Design review
- Requirements analysis | - Code inspection
- Early prototyping | - Glass box testing
- Fault-tolerant design | - Black box testing
- Defensive programming | - Training testers
- Usability analysis | - Beta testing
- Clear specification | - Test automation
- Accurate internal documentation | - Usability testing
- Evaluation of the reliability of development tools (before buying them) or of other potential components of the product | - Pre-release out-of-box testing by customer service staff

Many attribute the cost of testing to be around 25% of the development cost. Some projects will allocate testing resources where the cost can be as much as 80% of the development budget (for reasons discussed below).

1.7.3 Costs of Defects

Especially for software developers the costs can be extensive:
- Technical support calls
- Preparation of support answer books
- Investigation of customer complaints
- Refunds and recalls
- Coding / testing of interim bug fix releases
- Shipping of updated product
- Added expense of supporting multiple versions of the product in the field
- PR work to soften drafts of harsh reviews
- Lost sales
- Lost customer goodwill
- Discounts to resellers to encourage them to keep selling the product
- Warranty costs
- Liability costs
- Government investigations
- Penalties
- All other costs imposed by law
**Internal Costs**
- Bug fixes
- Regression testing
- Wasted in-house user time
- Wasted tester time
- Wasted writer time
- Wasted marketer time
- Wasted advertisements
- Direct cost of late shipment
- Opportunity cost of late shipment

**External Costs**
- Wasted time
- Lost data
- Lost business
- Embarrassment
- Frustrated employees quit
- Demos or presentations to potential customers fail because of the software
- Failure when attempting other tasks that can only be done once
- Cost of replacing product
- Cost of reconfiguring the system
- Cost of recovery software
- Cost of tech support
- Injury / death

---

**EXPERIENCE REPORT**

At a recent meeting somebody was telling us about an electronic circuit he had developed to regulate voltage supplied to light bulbs on (waterway) navigation buoys for the Queensland Government. He mentioned that through the use of solar power, the voltage would often oversupply the light bulbs and they would tend to burn out much quicker.

The issue was the cost of light bulb replacement was extremely expensive, in the order of $12,000 per light bulb, due to the need for a vessel, staffing, and means of recovery of the navigation buoy.

Similar costs can be attributed to maintenance in software projects, and it quickly becomes clear how immediate costs of testing can result in significant downstream savings through reductions in maintenance.
Boehm's cost studies [Boe76a] reported that, for one Air Force computer:

- initial software development costs were $75 per instruction
- maintenance costs were $4000 per instruction

Studies by Martin and McClure [Mar83a] summarised relative costs at each development stage, as shown in the following chart. In this study, testing accounted for 45% of the initial development costs. Testing is also an integral part of maintenance as well, but this has not been isolated.

**Relative Cost per Development Stage [Mar83a]**

![Relative Cost per Development Stage Chart]

### 1.7.4 Balancing the Costs

See figure below. Feigenbaum (1991) estimated that the typical company spends 5 to 10 cents of every quality cost dollar on prevention, another 20 to 25 cents on appraisal, and the remaining 65 to 75 cents on internal and external failure costs.
Need to balance the costs, so that excessive amounts are not drawn by internal/external failures. The way to do this is to compare the cost of removing defects in relation to fixing the defects that remain in the system.

As soon as it becomes more expensive to test for the defects than it is to correct them afterwards, then it is time to cut down on testing – usually you will be spending a substantial sum on testing before this becomes the case.

Defects can be assumed to have a fixed cost, therefore the total cost of fixing defects increases linearly on the number of defects left in the system.

Testing effort, however, increases exponentially in relation to increases in proportion of defects removed. A goal of 100% defect removal is infeasible, as exhaustive testing is not possible (as discussed previously).

These costs could be plotted on a graph as shown in the figure below. The total defect repair cost reduces linearly as the proportion of defects remaining is lowered, whereas the cost to test to remove defects increases exponentially as the proportion of defects removed increases.

This graph would need to be correlated against your own cost allocations, based on experience and estimation, or internal measurements and data analysis.

The point at which the two lines cost is the breakeven point. Prior to this point for projects that higher criticality, the defect cost is increased, which indicates more resource can be allocated to achieve a higher defect removal proportion. See figure below.
Test Effort vs. Failure Cost (Varying)

- Testing Effort
- Lower Integrity Failures
- Medium Integrity Failures
- Higher Integrity Failures

Defect Reduction Ratio

Cost
1.8 Risks-Based Testing

As pointed out in earlier sections, it is impossible to test everything. Furthermore, testing must be carried out within limited budgets and schedules.

Whenever there is not enough time and resources to test everything, decisions must be made to prioritise what is tested.

To prioritise we must decide what are the most important parts of the system (those parts that have greatest impact if not operating correctly).

To prioritise testing we compare risks of system failures: the likelihood of incurring damage if a defect in the system causes the system to operate incorrectly or unexpectedly.

**Defect**

A specific cause of a failure, e.g. incorrect parameters to write pay-cheque function call.

A defect can also be called a bug, error, fault

**Failure**

Inability of the system to perform its intended function, e.g. prints pay cheques with the wrong value

**Risk**

The chance of incurring a certain cost over a given period resulting from a failure

Risk is a product of a failure's:

- severity of impact
- likelihood of occurrence

For example, a 50% chance of issuing cheques that total more than a $1M in overpayment during the lifetime of the system

To prioritise testing of the system based on risk, the tester must break the system into the smaller components, and then determine the risk of each component. Those components with higher risk receive higher priority for testing, those of lower risk receive lower priority.

The tester can arbitrarily designate what the components of the system should be, as it is merely a way of focusing testing effort on particular areas. For example, they may be functions of the system, or even subsystems of the overall system.

1.8.1 The Risk Matrix

Risk is the product of severity of impact and likelihood of occurrence.

- Severity represents the impact and cost incurred when the failure is encountered
- Likelihood represents the probability that the failure leads to detrimental impact. It is a combination of the probability of the problem being present in the system (which could be based on the confidence of the correctness of the system) and the probability that when the failure occurs the impact and cost is realised (which may be dependent on other external events occurring).
The highest risk areas are those with highest impact severity combined with highest likelihood of occurrence. Whereas, the lowest risk is lowest severity combined with lowest likelihood of occurrence.

Risk levels can be graded proportionally according to increases in severity and likelihood.

It is easiest to think of the combination of severity and likelihood to derive a risk level as a matrix.

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Very High</th>
<th>High</th>
<th>Low</th>
<th>Very Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very High</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>High</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Low</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Very Low</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

Each subcomponent can be assigned a risk level according to the risk matrix. For example, a subcomponent is associated with failures of very high likelihood and impact can be assigned risk level A.

The risk matrix provides the basis for prioritising tests.

- Testing should first focus on class A risks: those with highest likelihood and priority.
- If time permits, after class A risks, attention should be given to class B risks.
- In tight schedules and budgets there probably will not be time to test class C risks.

### 1.8.2 Breakdown of Risk-Based Testing

According to Collard [Col97a], the breakdown of test cases using risk-based test priorities fall into three categories:

1. Critical test cases, that have reasonable probability of finding major defects. Typically contains 10-15% of all likely test cases.
2. Important test cases, that are not critical, but still worth the effort if time and resources are available. Typically contains 15-25% of all likely test cases.
3. Low-yield test cases, that have very low probability of finding significant errors. Typically contains 60-75% of all likely test cases. These cases are not worth the effort if time and resources are scarce.

Collard further reports that Bell Communications Research applies risk-prioritisation to testing, and concluded that the most important 15% of test cases find over 75% of the defects.
1.8.3 Allocating Effort According to Risk

With tight schedules and resources, the amount of effort required to establish test coverage for the proposed error should also be taken into account. Test coverage levels may be allocated according to the level of risk and the level of effort required, as shown in the following tables.

Coverage with generous schedule and resources may be allocated according to the following table:

<table>
<thead>
<tr>
<th>Risk</th>
<th>Test Cost</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>A</td>
<td>100%</td>
<td>85%</td>
<td>70%</td>
</tr>
<tr>
<td>B</td>
<td>50%</td>
<td>40%</td>
<td>30%</td>
</tr>
<tr>
<td>C</td>
<td>30%</td>
<td>15%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Coverage with limited schedule and may be allocated differently, such as according to the following table:

<table>
<thead>
<tr>
<th>Risk</th>
<th>Test Cost</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>A</td>
<td>85%</td>
<td>60%</td>
<td>40%</td>
</tr>
<tr>
<td>B</td>
<td>40%</td>
<td>25%</td>
<td>0%</td>
</tr>
<tr>
<td>C</td>
<td>15%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
1.9 Careers in Software Testing

In the past testing has been considered an additional task performed by the developers, and had very little emphasis applied to it as a skill. Testing is now recognised as a profession, that requires substantial experience and professional training. Due to increased demand for software quality, testing is establishing itself a very healthy career path, commanding high salaries.

1.9.1 Skills in Demand

What employers are looking for (see attachment of recent positions vacant clippings)

- **General**: Strong analytical focus; Good communications; QA background

- **Methods experience**: Development of test plans; Creation and maintenance of test environments; Test standards; Test documentation (e.g. test cases and procedures)

- **Testing approaches**: Integration; Acceptance; Stress/volume; Regression; Functional; End-to-end; GUI

- **Market**: Banking/Finance; Shrink Wrapped/Commercial Products; Telecomms; Internet; Year 2000

- **Test tools**: Capture/playback tools (e.g. SQA; WinRunner); Load testing tools (e.g. LoadRunner)

- **Environment skills**: Mainframe (e.g. MVS, JCL); Client/Server (e.g. Win/NT, Unix)

- **Application skills**: Documentation (e.g. Office, Excel, Word, Lotus Notes); Database, (e.g. Oracle, Access, 3GL, 4GL, SQL, RDBMS); Programming (e.g. C/C++, VB, OO)

Collard [Col97a] proposes that testers skills fall into three groups:

- Functional subject matter expertise
- Technology base
- QA and testing techniques
1.9.2 Personality of Testers

Positive attributes

- Planned, systematic and meticulous – logical approach to testing
- Crusader – like to set high standards for the project
- Persistent – don’t give up easily
- Practical – realise what can be achieved in limited time and budget
- Analytical – take and intuitive approach to digging up errors
- Moralist - strive for quality and success, recognise the costs of poor quality

Negative attributes

- Little empathy for the developers – don’t realise the emotional attachment developers have with their “offspring”
- Lack of diplomacy – create conflict with developers with “in your face” attitude. Testers offer cheer in the developers presence when they find a defect, and follow that up with quotes of defect statistics and bug counts.
- Skeptical – take developers information with a grain of salt
- Stubborn – immovable on some proposals

Many developers run and hide from testers when they spot them in the corridor:

- Believe they are going to be hassled about a problem
- Afraid to discuss issues regarding their current development in case it is later used against them
2. Testing Processes

- software testing and software development processes
- kinds of testing
- test strategy and planning
2. The Testing Process

2.1 The Testing Process

2.1.1 The Traditional Lifecycle

Traditional to leave testing until after the coding has been done and there are executable components available to test.

Problems with this approach is that testing is being left to too late in the process.

- Simply designing the tests provides insights into the system that often discovers bugs before the tests are run.
- Usually the phases up to coding will overrun (85% of software projects are delivered late or not at all). In this scenario there will be pressure to get the product out of the door as soon as possible after coding. The schedule pressure then placed on testing will usually lead to failure to complete an adequate test.

2.1.2 Parallel Testing Lifecycle

Only test execution needs to wait until after coding. Testing planning and test case design can occur in parallel with development. The act of planning and designing the tests to be applied will discover bugs, e.g. confusing requirements will be identified.
2.1.3 Defect Prevention Lifecycle

It is also important to augment the development and testing processes with defect prevention techniques, to improve the processes so that the bugs are not introduced in the first place.
2.1.4 The Development V-Model

Verification and validation processes propose the software development V-model. This process considers development and testing to be a V. The left-side of the V is the development, and the right side is testing. At each point down the left side of the V there is a corresponding test that checks that the development step is correct.

The tests proposed at each level can be planned and designed in parallel to lower level activities being conducted. This way as we come up the right side of the V the testing is ready to commence.
2.2 Kinds of Testing

There are many different kinds of testing that focus on finding different kinds of software errors.

2.2.1 Static and Dynamic Testing

Static Testing is examining the component or system without executing the code.

Static testing may not only examine the code, but also specification, design and user documents. Examinations include:

- Review meetings, such as design walkthroughs, or formal inspections
- Peer review, such as desk checking
- Running syntax and type checkers as part of the compilation process, or other code analysis tools

Dynamic testing, on the other hand, is executing the code to test, without necessarily examining the code itself.

In this course we mainly discuss dynamic testing.

2.2.2 Functional (Black-Box) Testing

Testing based on the requirements of the system or component under test. Quite often these are derived directly from functional requirements documents, where each functional requirement is turned into functional test criteria.

Testing coverage at functional testing is usually achieved by building a trace matrix between requirements and test criteria that test that requirement. Those requirements that are not covered properly are those that do not have sufficient test criteria assigned.

Where function requirements documents are not available, the tester must interpret the requirements of the system by observing the operational behaviour of the system or component under test.

Black-box testing is so named as it IS NOT POSSIBLE to look inside the box to determine the design and implementation structures of the components you are testing.

2.2.3 Structural (White-Box) Testing

Testing based on the design and implementation structures of the system or component under test.

The structures provide information about flow of control and information that can be used to derive test cases. E.g. if-then-else statements in a program indicate that two kinds of result are computed based on whether a certain condition is true or false, both of which should be tested.
Testing coverage for structural testing is usually achieved by recording those paths of a program that have been executed when test have been applied. Checking coverage of paths may include checking that every statement in the program have been executed during the test of a component or system, i.e. there a no lines in the code that have not been run at least once during test.

White-box testing is so named as it IS POSSIBLE to look inside the box to determine the design and implementation structures of the components you are testing.

### 2.2.4 Module (Unit) Testing

Module Testing is applied at the level of each module of the system under test.

Module testing is usually done by the developers themselves, before the module is handed over for integration with other modules.

Usually the testing at this level is structural (white-box) testing.

Module Testing is sometimes called Unit Testing representing the lowest level component available for testing.

Note in some situations, unit testing is also used to refer to testing of logical or function units that have integrated a number of modules. We consider this to be integration testing.

### 2.2.5 Integration Testing

Integration Testing is an interim level of testing applied between module testing and system testing to test the interaction and consistency of an integrated subsystem.

Integration testing is applied incrementally as modules are assembled into larger subsystems. Testing is applied to subsystems which are assembled either:

- Bottom-up - assembles up from the lowest level modules and replaces higher level modules with test harnesses to drive the units under test.
- Top-down – assembles down from the highest level modules replacing lower level modules with test stubs to simulate interfaces

Integration testing is performed so a partial system level test can be conducted without having to wait for all the components to be available, and thus problems can be fixed before affecting downstream development on other components. Thus integration testing is usually done as soon as identifiable logical subsystems are complete.

Integration testing is also performed as some tests cannot be carried out on a fully integrated system, and requires integration with special test harnesses and stubs to properly test a component. It also helps to focus testing to particular components and to isolate the problems that will be discovered.

Usually the testing at this level is a mixture of functional (black-box) and structural (white-box) testing.
2.2.6 Interface Testing

Interfaces are the sources of many errors, as often there is a misunderstanding of the way components and subsystems should work together because they are developed by different people.

Interface testing focuses specifically on the way components and subsystems are linked and work together.

Interface testing can be applied to internal interfaces between subcomponents of a system (e.g. between separate modules), as well as to external interfaces to other systems (e.g. data interchanges to other applications).

2.2.7 System Testing

System testing represents testing after a complete system has been assembled.

System testing is usually performed after all module, integration and unit testing have been successfully applied.

2.2.8 Acceptance Testing

Demonstrates to the customer that predefined acceptance criteria have been met by the system.

Typically used as a mechanism to handover the system.

The customer may write the acceptance test criteria and request that these be executed by the developer, or the developer may produce the acceptance testing criteria which are to be approved by the customer.

The customer may demand to:
- Run the tests themselves
- Witness tests run by the developer
- Inspect documented test results

2.2.9 Performance, Load and Stress Testing

Performance is concerned with assessing timing and memory usage aspects of the system's operation.

Performance testing may be concerned with checking that an operation completes within a fixed deadline, or that only a fixed size of memory is allocated.

Load testing increases the system to peak levels of operation and checks whether this has any detrimental effect on operation. For example, running with maximum levels of concurrent users to determine whether there is any impact on network performance, or increasing the database to maximum size to determine whether transactions are still feasible.

Stress testing is concerned with degrading the performance of the system to determine whether this has any detrimental effect on operation. For example, disconnecting a redundant network server to determine if remaining servers can maintain operation.
These kinds of testing are very closely related, but it is important to note their subtle differences.

2.2.10 Usability Testing
Usage testing considers human factors in the way that users interact with the system. This testing considers the way in which information is presented to the user, such as layout of data and buttons, or colours used to highlight information, e.g. errors and alarms.

2.2.11 Reliability Testing
Reliability testing establishes how reliable the system will be when the system is operating.

Usually reliability testing is used to establish a Mean Time Between Failure (MTBF) which is the expected duration between failures of the system, such as the system crashing and not being operational.

The MTBF is established by generating a sequence of tests that reflects the operational profile of a system. The user first needs to construct a statistical model of the operational profile and then build the test sequence by randomly selecting each test case in the sequence from the statistical distribution.

This is sometimes called Statistical Testing.

2.2.12 Security Testing
Tests that access control is limited appropriately:
- Password control
- Authority levels
- File access privileges
- Server access

Examines questions like could an unauthorised person gain access to or modify functions or data within the system.

2.2.13 Backup/Recovery Testing
Testing that the backup and recovery mechanisms of a system are stable and sufficient.

2.2.14 Installation Testing
Testing that the system has been installed and configured correctly for the installation site.

2.2.15 Alpha and Beta Testing
Alpha and beta testing represents use of the system by actual users. Testing is informal and represents typical use of the system.
Alpha testing is usually conducted by users who are internal to the development organisation, such as another development group in the same company. It is somewhat like a pilot installation of the system.

Beta testing is testing carried out by external participating sites where other organisations are using the system on a trial basis. Usually, participating beta testers sign an agreement that waives damages, and is usually limited to non-critical applications.

Beta testing is employed quite often on projects that could affect a large number (potentially millions) of users. Beta testing can be very expensive, but insignificant when compared to the cost of dealing with problems handed out to the complete customer base.

### Beta Testing Case Studies

Microsoft, and other shrink-wrapped software vendors, use beta testing extensively to release trials of their products, before distributing them to millions of users. These beta testers provide their services for free or in return for free or discounted software.

The Hong Kong Jockey Club systems support billions of dollars in betting transactions every year. They beta test their systems by having many thousands of their employees attend mock horse racing days to exercise the new systems installed at the race track. These beta testers are paid to attend the mock racing days, and are provided funny money "to gamble with.

A local computer game manufacturer, Auran, used the internet to collect feedback from more than 1000 volunteer beta testers of their product "Dark Reign". The software automatically provided feedback which profiled how the system was being used by each beta tester, and how they could fine tune their game. They had over 10,000 applicants for beta testers within one week, all in return for a Dark Reign poster signed by the developers of the system.

### 2.2.16 Parallel Testing

The system under test is put into operation in parallel with the existing system that it is replacing. Results computed in the new system are compared to those results computed in the existing system to discover discrepancies.

After a suitable parallel test period, and providing no significant problems are encountered, the existing system is replaced with the system under test.

### 2.2.17 Regression Testing

Regression testing is applied after changes have been made to a system. The operation of the new version of the system is compared to the previous version to determine if there are any unexpected differences.

Regression is applied as changing software, for example to fix known defects or to add new functions, has a very high likelihood of introducing new defects. Some studies have predicted that for every 6 lines of code modified a new defect will be added.
Regression testing has become a commonly applied form of testing due to the introduction of capture/playback test tools. Testers use these tools to capture the operation of a version of a system under test, and then the operation can be played back automatically on later versions of the system and any differences in behaviour are reported.

2.2.18 Mutation Testing

Mutation testing is used to evaluate the effectiveness of the testing applied to a system.

Mutation testing introduces modifications to the system under test that would constitute defects. The tests previously applied to the system are then reapplied to the modified system to check that the tests applied would pick up the defects introduced.

A variant of this approach is error seeding, which deliberately introduces defects into the system before testing begins. This is used as a motivation technique for the testers to discover defects that are known to exist. However, as with any techniques that could be used to evaluate performance, care must be taken not to lower morale with measurements that could be used against a particular developer or tester.

This method has been used in a number of scientific studies to measure the effectiveness of different testing techniques.
2.3 Testing Strategy

So what kind of testing is appropriate for your system. A test strategy balances the corporate and technical requirements of the project. A test strategy must consider:

1. what risks are most critical to your system
2. where in the development process defects are introduced that contribute to these risks
3. propose testing approaches to detect these defects

2.3.1 Test Strategy Cube

Perry [Per95a] presents an approach to developing a test strategy using a test strategy cube (see below).

The axes of the cube are:
The process in completing the test strategy cube is:
1. select and rank test factors
2. identify system development phases
3. identify the risks associated with development
4. place risks in matrix
5. assign test factors to mitigate those risks

### 2.3.1.1 Test Factors

Test factors proposed by Perry [Per95a] to be used in the test strategy cube are described in the table below:

<table>
<thead>
<tr>
<th>Test Factor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Assurance that the data entered, processed and output by the system is accurate and complete.</td>
</tr>
<tr>
<td>Authorisation</td>
<td>Assurance that data is processed in accordance with the intents of management.</td>
</tr>
<tr>
<td>File Integrity</td>
<td>Assurance that the data entered into the system will be returned unaltered.</td>
</tr>
<tr>
<td>Audit Trail</td>
<td>The capability to substantiate the processing that has occurred.</td>
</tr>
<tr>
<td>Continuity of processing</td>
<td>The ability to sustain processing in the event problems occur.</td>
</tr>
<tr>
<td>Service levels</td>
<td>Assurance that the desired results will be available within a time frame acceptable to the user.</td>
</tr>
<tr>
<td>Access Control</td>
<td>Assurance that the system resources will be protected against accidental and intentional modification, destruction, misuse and disclosure.</td>
</tr>
<tr>
<td>Compliance</td>
<td>Assurance that the system is designed in accordance with the organisational strategy, policies, procedures, and standards.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Assurance that the system will perform its intended function with the required precision over an extended period of time.</td>
</tr>
<tr>
<td>Ease of use</td>
<td>The extent of effort required to learn, operate, prepare input for, and interpret output from the system.</td>
</tr>
<tr>
<td>Maintainable</td>
<td>The effort required to locate and fix an error in an operational system.</td>
</tr>
<tr>
<td>Portable</td>
<td>The effort required to transfer a program from one hardware configuration and/or software system environment to another.</td>
</tr>
<tr>
<td>Coupling</td>
<td>The effort required to interconnect components within a system and</td>
</tr>
</tbody>
</table>
2. Testing Processes

2.3.1.2 Test Tactics

Test tactics are allocated to mitigate risks at different phases. The following examples are proposed by Perry [Per95a] to be used in the test strategy cube:

**Correctness test factor**
- The user has fully defined the functional specifications.
- The developed design conforms to the user requirements.
- The developed program conforms to the system design specifications.
- Functional testing ensures that the requirements are properly implemented.
- The proper programs and data are placed into production.
- The user-defined requirement changes are properly implemented into the operational system.

**Authorization test factor**
- The rules governing the authorization of transactions that enable the transaction to be processed are defined.
- The application is designed to identify and enforce the authorization rules.
- The application programs implement the authorization rules design.
- The system is compliance tested to ensure that the authorization rules are properly executed.
- Unauthorized data changes are prohibited during the installation process.
- The method and rules for authorization are preserved during maintenance.

**File integrity test factor**
- Requirements for file integrity are defined.
- The design provides for the controls to ensure the integrity of the file.
- The specified file integrity controls are implemented.
- The file integrity functions are tested to ensure they perform properly.
- The integrity of production files is verified prior to placing those files into a production status.
- The integrity of the files is preserved during the maintenance phase.

**Audit trail test factor**
- The requirements to reconstruct processing are defined.
- The audit trail and information in the audit trail needed to reconstruct processing are designed.
- The audit trail requirements are incorporated into the system.
- The audit trail functions are tested to ensure the appropriate data is saved.
- The audit trail of events during installation is recorded.
- The audit trail requirements are updated during systems maintenance.

### Continuity of processing test factor
- The impact of each system failure has been defined.
- A contingency plan has been designed to address the identified failures.
- The contingency plan and procedures have been written.
- Recovery testing verifies that the contingency plan functions properly.
- The integrity of the previous systems is assured until the integrity of the new system has been verified.
- The contingency plan is updated and tested as systems requirements change.

### Service level test factor
- The desired service level for all aspects of the system is defined.
- The method to achieve the predefined service levels is incorporated into the systems design.
- The programs and manual systems are designed to achieve the specified service level.
- Stress testing is conducted to ensure that the system can achieve the desired service level when both normal and above normal volumes of data are processed.
- A fail-safe plan is used during installation to ensure service will not be disrupted.
- The predefined service level is preserved as the system is maintained.

### Access control test factor
- The access to the system has been defined including those individuals/programs authorized access and the resources they are authorized.
- The procedures to enforce the access rules are designed.
- The defined security procedures are implemented. Compliance tests are utilized to ensure that the security procedures function in production.
- Access to the computer resources is controlled during installation. The procedures controlling access are preserved as the system is updated.

### Compliance test factor
- The system requirements are defined and documented in compliance with the system development methodology.
- The system design is performed in accordance with the system design
methodology.

- The programs are constructed and documented in compliance with the programming methodology.
- The testing is performed in compliance with the test methodology.
- The integration of the application system in a production environment is performed in accordance with the installation methodology.
- Maintenance is performed in compliance with the system maintenance methodology.

**Reliability test factor**

- The level of accuracy and completeness expected in the operational environment is established.
- The data integrity controls to achieve the established processing tolerances are designed.
- Data integrity controls are implemented in accordance with the design.
- Manual, regression, and functional tests are performed to ensure the data integrity controls work.
- The accuracy and completeness of the system installation is verified.
- The accuracy requirements are maintained as the applications are updated.

**Ease of use test factor**

- The usability specifications for the application system are defined.
- The system design attempts to optimize the usability of the implemented requirements.
- The programs conform to the design in order to optimize the ease of use.
- The interrelationship between the manual and automated system is tested to ensure the application is easy to use.
- The usability instructions are properly prepared and disseminated to the appropriate individuals. As the system is maintained, the ease of use is preserved.

**Maintainable test factor**

- The desired level of maintainability of the system is specified.
- The design is developed to achieve the desired level of maintainability.
- The programs are coded and designed to achieve the desired level of maintainability.
- The system is inspected to ensure that it is prepared to be maintainable.
- The application system documentation is complete.
- Maintainability is preserved as the system is updated.

**Portable test factor**

- The portability in moving the application system from one piece of hardware or
2. Testing Processes

- The design is prepared to achieve the desired level of portability.
- The programs are designed and implemented to conform with the portability design.
- The implemented system is subjected to a disaster test to ensure that it is portable.
- The documentation is complete to facilitate portability.
- Portability is preserved as the system is maintained.

### Coupling test factor
- The interface between the system being developed and other related systems is defined.
- The design takes into account the interface requirements.
- The designed programs conform to the interface design specifications.
- Functional and regression testing are performed to ensure that the interface between systems functions properly.
- The interface between systems is coordinated prior to the new system being placed into production.
- The interface between systems is preserved during the systems maintenance process.

### Performance test factor
- The performance criteria are established.
- The design specifications are prepared to ensure that the desired level of performance is achieved.
- The programs are designed and implemented to achieve the performance criteria.
- The system is compliance tested to ensure that the desired performance levels are achieved.
- The performance is monitored during the installation phase.
- The desired level of performance is preserved during systems maintenance.

### Ease of operation test factor
- The operational needs are incorporated into the system design.
- The operational needs are communicated to operations.
- The needed operating procedures are developed.
- The operational procedures are tested to ensure they achieve the desired level of operational usability.
- The operating procedures are implemented during the installation phase.
2. Testing Processes

- The changes to the operational system are reflected in changes to operating procedures.

2.3.2 Test Activities

Test activities are assigned to implement tactics identified above. A test strategy must closely fit with other development, and verification and validation strategies. It is common that significant changes are made to accommodate verification and validation (testing) issues within the development process, for example, synchronisation points in the processes to coordinate on shared activities, or implementing hooks in the code to be used for testing purposes.

The role of test strategy is not to manage how a single testing activity is implemented. This kind of information is contained in a test plan, whereas the test strategy proposes how different testing activities can be planned to cover all important corporate and technical test requirements.

It is quite common to develop a test strategy group for a project. This group consists of the managers responsible for implementing different phases of testing, and may also have members of the development team contributing to the strategy [KR3].

Testing strategy coordinates the separate testing activities applied to a system.
2.4 \{XE "test plan" \r "indTestPlan" \} Test Plans

A test plan describes, for a testing activity, the components to be tested, the testing approach, any tools required, the resources and schedule, and the risks of the testing activity.

As with other plans “failing to plan is planning to fail”.

The test plan is used to predict solutions up-front to problems that need to be solved later in the process.

2.4.1 Test Plan Objectives

The main objectives of the test plan are:

- to identify the items that are subject to testing
- to communicate, at a reasonably high level, the extent of testing
- to define the roles and responsibilities for test activities
- to provide an accurate estimate of effort required to complete the testing defined in the plan
- to define the infrastructure and support required (capital purchases, access required, etc.)

Level of Detail in Test Plans

One client made some complaints about the test plans we prepared for them. They said that the extent of the test requirements were not detailed enough as they did not precisely define all the steps involved in each test. We only provided high level test requirements describing what aspect of the system was being tested.

They were correct in the fact that the test plans were not at such a low level of detail. This was intentional! To provide such a level of detail is at a stage of defining the test cases and test procedures. This is test design. There is significant cost involved in test design, beyond that assigned to preparation of the plan.

The purpose of the test plan is to identify at a high level what is required so that activities can be resourced and scheduled. The plan is an agreement on the extent of testing to be completed. Planning is done before significant effort is further expended preparing the test designs.

In this case, the client’s complaints also included that the testing proposed was too expensive! To reduce the cost, the extent of the testing had to be reduced. Imagine the waste had we moved to test design and prepared the low level detail for each test, only later to realise that some of the tests we defined were to be eliminated to save money.

The level of our test plan provided sufficient detail to decide what tests were to be carried out and what was to be omitted. We knew what we were in for …
2.4.2 IEEE Standard Test Plans

The IEEE Standard [IEEE83a] identifies the main components of a test plan according to the structure of the test plan document:

- **Identifier** – unique identifier assigned to plan
- **Introduction** – summarise what's in the plan, what are the main issues that a reader should look at in more detail if they were to pick up the plan, and provide references to other documents
- **Test Items** – identifies the components to be tested, including specific versions or variants
- **Features to be Tested** – those aspects of the system that will undergo test
- **Features not to be Tested** – aspects of system not being tested and reasons why they can be ignored
- **Approach** – describes the general approach to testing each feature to be tested
- **Item Pass/Fail Criteria** – criteria to determine whether each test item passed or fails testing
- **Suspension Criteria and Resumption Criteria** – identifies the conditions under which testing can be suspended, and what testing activities are to be repeated if testing resumes
- **Test Deliverables** – describes the documentation that represents all the testing activities applied to the test items covered under this plan
- **Testing Tasks** – identifies all the tasks required to complete testing, including any dependencies between tasks, or special skills required for tasks
- **Environmental Needs** – describes the test environment, including any special hardware facilities, software facilities, and support tools.
- **Responsibilities** – groups responsible for managing, designing, preparing, executing, witnessing, checking and resolving.
- **Staffing and Training Needs** – specifies who is available to carry out the testing tasks, their skill level requirements, and any special training needs.
- **Schedule** – proposed test milestones and events, and proposals for task coordination and effort estimations.
- **Risks and Contingencies** – identifies any high risk assumptions of the plan, and contingencies for each risk proposed
- **Approvals** – sign-off requirements for the plan

2.4.3 Test Plan Issues

Testers can get frustrated trying to complete the test plans before the details of the system that they are testing have been finalised. In these scenarios, "To Be Defined" or "TBD" can be used as a placeholder for parts of the plan that are not yet known. This terminology also provides a simple mechanism for searching for parts of the plan that still require development.
3. Black Box Testing

- purpose of black-box testing
- a systematic approach to decomposing system requirements into tests
- black-box test design techniques for defining test cases, e.g. functional analysis, equivalence partitioning, boundary value analysis, cause-effect graphing
- test coverage metrics
- test oracles as mechanisms to capture and compare test results
3.1 Purpose of Black-Box Testing

Black-box testing is proposing tests without detailed knowledge of the internal structure of the system or component under test. It is based on the specification of what the system's requirements.

Many other replacement terms are used for black-box testing, including:

- specification-based testing
- input/output testing
- functional testing

Black-box testing is typically performed as the development lifecycle nears a completely integrated system, such as during integration testing, interface testing, system testing and acceptance testing. At this level the components of the system are integrated sufficiently as to demonstrate that complete requirements are fulfilled.

The types of errors that are most commonly found in black-box testing, include:

- incorrect or missing functions
- interface errors, in the way different functions interface together, the way the system interfaces with data files and data structures, or the way the system interfaces with other systems, such as through a network
- load and performance errors
- initialisation and termination errors
3.2 Systematically Decomposing Requirements into Tests

Most testers, at the start of the testing project, are confronted with the problem of deciding what test cases they will execute to thoroughly test their system.

Initially the tester is overwhelmed, as they are looking at an empty test set that must be populated, which, for the average system, will need to be filled with many thousands of test cases to test adequately.

As with any huge task, the key is to break the task down into smaller manageable activities. This is where test design fits in, decomposing the task of testing a system into smaller manageable activities, ultimately to a level that corresponds to establishment of an individual test case. Of course, test design is also the mechanism used for assuring yourself that you have sufficient test cases to cover all appropriate aspects of your system.

Designing what test cases are required is a labour-intensive activity. No tool can automatically determine what test cases are needed for your system, as each system is different, and test tools do not know what constitutes correct (or incorrect) operation. Test design requires the tester’s experience, reasoning and intuition.

3.2.1 Specification to Guide Testing

The specification or a model of your system is the initial starting point for test design. The system specification or model may be a functional specification, performance or security specification, user scenario specification, or a specification of the hazards and risks of the system. Whatever the case may be, the specification describes the criteria against which test as it defines correct or acceptable operation.

In many cases, particularly with legacy systems, there may be little or no documentation available to use as a system specification. Even where documentation does exist, there is a high risk that it has not been kept up-to-date after year of modification to the system. It is essential that knowledgeable end-users of the system are included in test design, as a substitute for missing or out-of-date specifications. If current documentation is inadequate for specification purposes, then at least a simple form of specification should be created from which the top-level set of test objectives can be derived. In many cases, a significant part of the test design technique involves formulating a specification to test against.

3.2.2 Decomposing Test Objectives

Test design will focus on a set of specification components to test. High-level test objectives are proposed for these specification components. Each test objective is then systematically decomposed into either other test objectives or test cases using test design techniques.

Once you have decomposed your test objectives to single components for a given test criteria, there are many kinds of test design techniques that can be selected depending on the type of testing you are applying, and many of these techniques are appearing in standards [BCS97a]. Techniques include

- Equivalence Class Partitioning
- Boundary Value Analysis
- State Transitions Testing
- Cause-Effect Graphing

Decomposing the testing task involves incrementally breaking down your system into test objectives until sets of test cases are identified.

Documenting your test designs is crucial. Test designs are normally represented in a document called a Test Design Specification, for which standard formats are available [IEEE83a]. The Test Design Specification provides an audit trail that traces the design from specification components through to applicable test cases. The document records the transformation of test objectives and justifies that each test objective has been adequately decomposed at the subsequent level. In many cases, test design documents are required to justify to other parties that each requirement has been fulfilled.

Systematic test design is the key to decomposing your huge complex testing task. Once you have competed the difficult task of deciding what test cases you need to complete your tests, executing the test cases is relatively straight-forward.
3.3 Black-Box Testing Techniques

Presented below are different techniques that can be employed to test a component in an attempt to discover defects. The component level considered is typically that of a single function.

Many of the techniques and examples are extracted from the British Computer Society's Standard for Component Testing [BCS97a]. This standard provides excellent guidance for test design techniques, and is being proposed for international standard status.

3.3.1 Functional Analysis

Functional analysis is a very general technique that is probably the most widely used approach in industry. The technique is based on the simple assumption that the system is expected to perform certain functions, and test cases are proposed to the functions are performed as expected.

The approach first needs to specification of what functions are expected to be provided by the system. This information is typically provided in functional specifications of the system. However in many cases, for example legacy systems, such specification may not be available or be so far out-of-date that they no longer correctly reflect the system's functions.

In these situations, the user must build a functional specification. Quite often a good starting point for such a specification is the menu structure provided by the system, or the user documentation provided with the system.

For example, consider the following case study which builds function tests for a shire council’s mainframe application.

3.3.1.1 Shire Council Application

The shire council has a core information system that supports:

- Finance, Accounting and Budgeting
- Inventory Management
- Purchasing
- Services
- Records Management

As usual, there is no up-to-date documentation available for the system.

The system has been developed for some time, and there are a number of application end-users that are trained in its operation.

3.3.1.2 Decomposing the system into functions

Begin by dividing the system into function groups for each main functional category.

For example:

- Operator Functions
- System Administration Functions
• Installation/Decommissioning Functions
• Security Functions
• Recovery/Backup Functions
• Interface Functions
• Etc.

Taking each functional category in turn, develop a hierarchy of function groups that define the types of functions performed for that category.

There is limited guidance for this step, as the groups are dependent on the type of system. Good starting points for the function group hierarchy include:

• System menus
• User Manual section headings / table of contents

The main menu provides initial function groups for operator functions:

Sub-menus further define the function group hierarchy:

For each function group, identify the set of functions that belong in the group. If the function can be further decomposed into sub-functions it should be a function group.
As with the function group hierarchy, good starting points for the function group hierarchy include:

- Lowest level menus
- User Manual section paragraphs

Ultimately a menu may link to particular functions rather than groups of functions. After analysis the tester can develop a hierarchy of function groups and functions:

### 3.3.1.3 Defining the Functions

For each function, provide a description:

- What the function is supposed to do
- How the function is supposed to work
The detail of description is dependent on users of the function model

- For the previous example, it may be acceptable to a small group just to have “Accessed from menu option 210.” But this may not be acceptable for longer term documentation requirements.

- Other groups may need considerably more detail.

More detail will be added required in test design techniques, it is either provided up-front, or as required during test design.

The following screens show the sequence of interaction when using the purchase order function:

Screen 1:

Screen 2:
3. Black Box Testing

Screen 3:

Screen 4:

Screen 5:
Data flow model:

```
<table>
<thead>
<tr>
<th>Form Inputs</th>
<th>Form Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supplier Number</td>
<td>Purchase Order Number</td>
</tr>
<tr>
<td>- Identifier for supplier of products</td>
<td>- Unique code generated for purchase order entry</td>
</tr>
<tr>
<td>Responsibility</td>
<td>Default Purchasing Officer</td>
</tr>
<tr>
<td>- Budget allocation code</td>
<td>- Code of purchasing officer taken from log-in</td>
</tr>
<tr>
<td>Location</td>
<td>Default Location</td>
</tr>
<tr>
<td>- Intended storage location for products</td>
<td>- Code of store location taken from log-in</td>
</tr>
<tr>
<td>Order Date</td>
<td>Default Delivery Site Details</td>
</tr>
<tr>
<td>- Date on which the order is raised</td>
<td>- Delivery site derived from location</td>
</tr>
<tr>
<td>Reference</td>
<td>Order Total</td>
</tr>
<tr>
<td>- Text field for arbitrary reference code</td>
<td>- Sum of ordered product subtotals</td>
</tr>
<tr>
<td>Delivery Code</td>
<td><strong>For each ordered product</strong></td>
</tr>
<tr>
<td>- Identifier for delivery address</td>
<td>Default Product Price</td>
</tr>
<tr>
<td>Delivery Details</td>
<td>- Price per product unit taken from supplier’s product records</td>
</tr>
<tr>
<td>- Details of delivery address</td>
<td>Default Product Description</td>
</tr>
<tr>
<td>Carrier</td>
<td>- Description of product taken from supplier’s product records</td>
</tr>
<tr>
<td>- Method of delivery</td>
<td></td>
</tr>
<tr>
<td>Deliver</td>
<td></td>
</tr>
<tr>
<td>- Special instructions for delivery</td>
<td></td>
</tr>
<tr>
<td>Contact</td>
<td></td>
</tr>
<tr>
<td>- Person responsible for enquiries</td>
<td></td>
</tr>
</tbody>
</table>
```

Data flow for the purchase order entry function.
<table>
<thead>
<tr>
<th>Information</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchasing Officer</td>
<td>Person responsible for issuing purchase order</td>
</tr>
<tr>
<td>Payment Terms</td>
<td>When payment is required</td>
</tr>
<tr>
<td>Sales Tax</td>
<td>Code for sales tax applicable to purchase order</td>
</tr>
<tr>
<td>Print Order Option</td>
<td>Y/N to print order</td>
</tr>
<tr>
<td>Confirmation Order Option</td>
<td>???</td>
</tr>
<tr>
<td>Print Order Prices Option</td>
<td>Y/N to include prices on printed order</td>
</tr>
<tr>
<td>Batch Print Option</td>
<td>Y/N to print at end-of-day batch or now on on-line printer</td>
</tr>
</tbody>
</table>

**For each ordered product**

<table>
<thead>
<tr>
<th>Information</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Code</td>
<td>Identifier for ordered product</td>
</tr>
<tr>
<td>Description</td>
<td>Description of ordered product</td>
</tr>
<tr>
<td>Quantity</td>
<td>Number of units ordered</td>
</tr>
<tr>
<td>Price</td>
<td>Price per unit</td>
</tr>
<tr>
<td>Unit</td>
<td>How units are sold, e.g. each, dozen</td>
</tr>
<tr>
<td>Due Date</td>
<td>Date before which ordered products should be received</td>
</tr>
<tr>
<td>Tax Class</td>
<td>Code for tax applicable to ordered product</td>
</tr>
</tbody>
</table>
3.3.1.4 Test Design using Functional Analysis
Analyses each function separately to define a set of test criteria
- Analyse each function’s input and output processing to determine what tests are required

Focus on each function to analyse the function’s:
- Criteria
- Outputs
- Inputs
- Internal Conditions
- Internal State

**Function’s Criteria**
Define criteria that determine whether the function is performing correctly

<table>
<thead>
<tr>
<th>No.</th>
<th>Criteria</th>
<th>Test Objective / Case Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Purchase order recorded for a supplier for a selection of products of different quantities</td>
<td>PUR-POE-01-001</td>
</tr>
<tr>
<td>2.</td>
<td>Products recorded in purchase order are added to products awaiting delivery in the inventory stock</td>
<td>PUR-POE-01-002</td>
</tr>
</tbody>
</table>

**Function’s Outputs**
Identify those outputs that must be produced by the system in order to support the function
- Identify how values are captured
- Identify how values are determined to be correct

<table>
<thead>
<tr>
<th>No.</th>
<th>Output</th>
<th>Accessibility</th>
<th>Correctness</th>
<th>Test Objective / Case Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Unique purchase order number generated at start of purchase order entry.</td>
<td>Observable in purchase order entry form</td>
<td>Check new order added without overwriting other purchase orders</td>
<td>PUR-POE-02-001</td>
</tr>
<tr>
<td>2.</td>
<td>Default unit price of product derived from supplier’s price list</td>
<td>Observable in purchase order entry form</td>
<td>Price corresponds to unit price in supplier’s product records</td>
<td>PUR-POE-02-002</td>
</tr>
<tr>
<td>3.</td>
<td>Default description of product derived from supplier’s price list</td>
<td>Observable in purchase order entry form</td>
<td>Same as recorded in supplier’s price list for item code</td>
<td>PUR-POE-02-003</td>
</tr>
<tr>
<td>4.</td>
<td>Purchase order total amount derived from sum of product order subtotals</td>
<td>Observable in purchase order entry form</td>
<td>Sum of product order subtotals</td>
<td>PUR-POE-02-004</td>
</tr>
<tr>
<td>5.</td>
<td>New order with appropriate products and quantities included purchase order records</td>
<td>Observable in purchase order records using query</td>
<td>Purchase order record details correspond to those entered</td>
<td>PUR-POE-01-001</td>
</tr>
<tr>
<td>6.</td>
<td>Products order quantities included in inventory stock level after completion</td>
<td>Check stock levels for ordered product before and after purchase order</td>
<td>Stock level increased by ordered quantity</td>
<td>PUR-POE-01-002</td>
</tr>
</tbody>
</table>
### Function’s Inputs
Identify those inputs required to generate the outputs for each function.

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Requirements</th>
<th>Test Objective / Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Supplier Number</td>
<td>Supplier number used to generate a purchase order. The purchase order is recorded for that supplier</td>
<td>PUR-POE-01-001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PUR-POE-03-001</td>
</tr>
</tbody>
</table>

Inputs not yet covered: all except Supplier Number

### Function’s Internal Conditions
Identify internal conditions under which outputs are expected to be produced.

<table>
<thead>
<tr>
<th>No.</th>
<th>Internal Condition</th>
<th>Effect</th>
<th>Test Objective / Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Purchase order within budget allowance</td>
<td>Purchase order can be completed</td>
<td>PUR-POE-04-001</td>
</tr>
</tbody>
</table>

Only the one internal condition considered

### Identify how invalid inputs are treated

<table>
<thead>
<tr>
<th>No.</th>
<th>Input</th>
<th>Treatment</th>
<th>Test Objective / Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Supplier Number</td>
<td>If supplier number does not exist in supplier detail records then an error message should be printed, and abort adding purchase order.</td>
<td>PUR-POE-03-002</td>
</tr>
<tr>
<td>2</td>
<td>Supplier Number</td>
<td>If supplier number is not entered then an error message should be printed, and abort adding purchase order.</td>
<td>PUR-POE-03-003</td>
</tr>
</tbody>
</table>

Inputs not yet covered: all except Supplier Number

<table>
<thead>
<tr>
<th>No.</th>
<th>Condition</th>
<th>Treatment</th>
<th>Test Objective / Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Purchase order not within budget allowance</td>
<td>Purchase order cannot be approved and error message is raised. The user is then passed back to purchase order entry to allow editing or cancellation</td>
<td>PUR-POE-04-002</td>
</tr>
</tbody>
</table>

Only the one internal condition considered
Function’s Internal Conditions

Identify how the internal state changes after receiving each input.

- How can the internal state be observed externally to check for correct change to state

<table>
<thead>
<tr>
<th>No.</th>
<th>Internal State</th>
<th>Accessibility</th>
<th>Correctness</th>
<th>Test Objective / Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Completed purchase order with batch print option chosen is indicated as not yet printed in records</td>
<td>Accessible from purchase order records</td>
<td>Not yet printed flag should indicate it is awaiting print</td>
<td>PUR-POE-05-001</td>
</tr>
</tbody>
</table>

Not yet complete: only one internal state issue considered
3.3.2 Use Cases\textsuperscript{[KR8]}

Collard provides an useful article on deriving tests from use cases (see [Col99a]).

A use case is a sequence of actions performed by a system, which together produce results required by users of the system. It defines process flows through a system based on its likely use.

Tests derived from use cases help uncover defects in process flows during actual use of the system. The tester may discover that it is not possible to transition to a different part of the system when using the system as defined by the use case.

Use cases also involve interaction or different features and functions of the system. For this reason, tests derived from use cases will help uncover integration errors.

3.3.2.1 Use Case Definition

Each use case has:

- **preconditions**, which need to be met for the use case to work successfully
- **postconditions**, define the conditions in which the use case terminates. The postconditions identify the observable results and final state of the system after the use case has completed.
- **flow of events**, which defines user actions and system responses to those actions. It is comprised of a **normal scenario**, which defines the mainstream of most likely behaviour of the system for the use case, and **alternative branches**, which provide other paths that may be taken through the use case.

Use cases may also have shared or common pieces, such as threads of use cases reused across several use cases.

A sample use case for a selecting products for a purchase order is shown below.

3.3.2.2 Deriving Test Cases

Deriving test cases from use cases is relatively straight-forward, and consists of choosing paths that traverse through the use case. Paths exist not only for the normal path but also for the alternative branches.

For each test case the path exercised can be identified, and then input and expected results can be defined. A single path can give rise to many different test cases, and other black-box test design techniques, such as equivalence partitioning and boundary value analysis (described in later sections), should be used to derive further test conditions.

A large number of test cases can be derived using this approach, and it requires judgement to prevent an explosion of test cases. As usual, selecting candidate test cases should be based on the associated risk, which involves the impact of failure, the likelihood of finding errors, the frequency of use, and complexity of the use case.
**Use Case Name:** Select Product

**Use Case Description**

Selecting the product to be ordered helps build a purchase order (PO) and adds a new line item to the PO. A PO can contain several line items, each line item on the PO orders a quantity of one specific product, from the vendor to whom the PO is addressed.

**Preconditions**

An open PO must already be in existence, with its status set to the value "in progress", and this PO must have a valid Vendor ID number and a unique PO number. The user must have entered a valid user ID# and password, and be authorised to access the PO and to add or modify a line item.

The number of line items on the existing PO must not equal twenty-five (For this sample, the organisation's business policy limits the number of lines per PO to twenty five or less.) The product number must be unique on each existing line item (another business policy - no duplicate products within one PO).

The system must be able to access the correct product list for the vendor.

**Postconditions**

The same PO must still exist and still be open. A new line has been added to the PO, with the next sequential line item number assigned, and the correct product has been selected for this line item (unless the use case was aborted).

The system must be left in the correct state to be able to move on the next steps after the product selection.

The product number cannot duplicate an existing product number on another line item on the same PO. The product selected or entered must be a valid one for this vendor.

Alternatively, the original PO must remain unchanged from its original state if a product was not selected (for example, because the process was aborted).

Error messages are issued, if appropriate, and after the use case has been exercised the system is waiting in a state ready to accept the next input command.

Use case for selecting a product for a purchase order.
An example test case is as follows:

<table>
<thead>
<tr>
<th>Test Case Name:</th>
<th>Select product - normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case Name:</td>
<td>Select product</td>
</tr>
<tr>
<td>Use Case Path to be Exercised:</td>
<td>[U1, S1.1, U2.1, S2, U3.1, S3.1, U4, S4.1]</td>
</tr>
<tr>
<td>Input Data:</td>
<td>Product ID# A01045 - Matador Stapler</td>
</tr>
<tr>
<td>Initial Condition:</td>
<td>PO# 18723 is already open and displayed for vendor ID# 67</td>
</tr>
<tr>
<td></td>
<td>User ID# 12 is authorised to work on this PO</td>
</tr>
<tr>
<td></td>
<td>3 line items already attached to this PO.</td>
</tr>
<tr>
<td>Expected Results:</td>
<td>PO# 18723 is still open and displayed for Vendor ID#67</td>
</tr>
<tr>
<td></td>
<td>User ID#12 is still authorised to work on this PO.</td>
</tr>
<tr>
<td></td>
<td>Eight line items are now attached to this PO.</td>
</tr>
<tr>
<td></td>
<td>New line item has been established for Matador Stapler.</td>
</tr>
<tr>
<td></td>
<td>New line item has been assigned line item number 8.</td>
</tr>
<tr>
<td></td>
<td>System is ready to proceed to next activity.</td>
</tr>
</tbody>
</table>

Test case for normal scenario of selecting a product for a purchase order.

3.3.2.3 Negative Test Cases

Negative test cases consider the handling of invalid data, as well as scenarios in which the precondition has not been satisfied. There are kinds of negative test cases that can be derived:

1. Alternative branches using invalid user actions. An example is provided below.

<table>
<thead>
<tr>
<th>Test Case Name:</th>
<th>Select product - enter invalid product code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case Name:</td>
<td>Select product</td>
</tr>
<tr>
<td>Use Case Path to be Exercised:</td>
<td>[U1, S1.1, U2.2, S3.2]</td>
</tr>
<tr>
<td>Input Data:</td>
<td>Product ID# A01055 - Matador Stapler (correct ID# is actually A01045)</td>
</tr>
<tr>
<td>Initial Condition:</td>
<td>PO# 18723 is already open and displayed for vendor ID# 67</td>
</tr>
<tr>
<td></td>
<td>User ID# 12 is authorised to work on this PO</td>
</tr>
<tr>
<td></td>
<td>Three line items already attached to this PO.</td>
</tr>
<tr>
<td></td>
<td>Product ID# A1055 is not a valid product number for this vendor.</td>
</tr>
<tr>
<td>Expected Results:</td>
<td>PO# 18723 is still open and displayed for Vendor ID#67</td>
</tr>
<tr>
<td></td>
<td>User ID#12 is still authorised to work on this PO.</td>
</tr>
<tr>
<td></td>
<td>The same three line items are still attached to this PO.</td>
</tr>
<tr>
<td></td>
<td>No new line item has been established.</td>
</tr>
<tr>
<td></td>
<td>Error message displayed: &quot;Invalid Product ID#: please re-enter or abort&quot;.</td>
</tr>
<tr>
<td></td>
<td>System is ready to proceed to next activity.</td>
</tr>
</tbody>
</table>

2. Attempting inputs not listed in the test case. This includes attempting to violate the preconditions, e.g. trying the use case on POs that do not have status "in progress", such as adding a new item to a "closed" PO.
3.3.2.4 **Conclusion**

Deriving test cases will inevitably discover errors in the use cases as well. We may discover that we can't abort without providing a correct purchase order number.

Use cases and test cases work well together in two ways:

1. If the use cases for a system are complete, accurate and clear, the process of deriving test cases is straight-forward.
2. If the use cases are not in good shape, deriving test cases will help to debug the test cases.

In other material, this approach is also referred to as transaction flow testing. A transaction flow is similar to a flow chart, but instead of using low level internal details in the flow chart, it provides a high level functional description from a user's perspective. In the same way as for use cases above we derive test cases to exercise different paths available through the transaction flow. See Chapter 3, Section 4 of [Bei84a] for further detail.
3.3.3 Equivalence Partitioning

Equivalence partitioning is a standardised technique in [BCS97a].

Equivalence partitioning is based on the premise that inputs and outputs of a component can be partitioned into classes, which according to the component's specification, will be treated similarly (equivalently) by the component. This assumption is that similar inputs will evoke similar responses.

A single value in an equivalence partition is assumed to be representative of all other values in the partition. This is used to reduce the problem that it is not possible to test every input value. The aim of equivalence testing is to select values that have equivalent processing, one that we can assume if a test passes with the representative value, it should pass with all other values in the same partition. That is we assume that similar

Some equivalence partitions may include combinations of the following:

- valid vs. invalid input and output values
- numeric values with negative, positive and 0 values
- strings that are empty or non-empty
- lists that are empty or non-empty
- data files that exist or not, are readable/writable or not
- date years that are pre-2000 or post 2000, leap years or non-leap years (a special case is 29 February 2000 which has special processing of its own)
- dates that are in 28, 29, 30 or 31 day months
- days on workdays or weekends
- times inside/outside office-hours
- type of data file, e.g. text, formatted data, graphics, video or sound
- file source/destination, e.g. hard drive, floppy drive, CD-ROM, network

3.3.3.1 Example

The following example is extracted from [BCS97a].

Consider a function, `generate_grading`, with the following specification:

"The function is passed an exam mark (out of 75) and a coursework mark (out of 25), from which it generates a grade for the course in the range ‘A’ to ‘D’. The grade is calculated from the overall mark, which is calculated as the sum of exam and c/w marks, as follows:

- greater than or equal to 70 – ‘A’
- greater than or equal to 50, but less than 70 – ‘B’
- greater than or equal to 30, but less than 50 – ‘C’
- less than 30 – ‘D’

Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers."
3.3.3.2 Analysis of Partitions

The tester provides a model of the component under test that partitions the input and output values of the component. The inputs and outputs are derived from the specifications of the component’s behaviour.

A partition is a set of values, chosen in such a way that all values in the partition are expected to be treated the same way by the component (i.e. they have equivalent processing).

Partitions for both valid and invalid values should be chosen.

For the `generate_grading` function, two inputs are identified:

- exam mark

\[
\begin{align*}
0 \leq \text{exam mark} \leq 75
\end{align*}
\]

- coursework mark

\[
\begin{align*}
0 \leq \text{coursework mark} \leq 75
\end{align*}
\]

Less obvious equivalence partitions would include non-integer values for inputs. For example:

- exam mark = real number
- exam mark = alphabetic
- coursework mark = real number
- coursework mark = alphabetic

Next, outputs of the `generate_grading` function are considered:

- grade

\[
\begin{align*}
\text{FM} & : \text{exam mark + c/w mark} < 0 \\
\text{D} & : 0 \leq \text{exam mark + c/w mark} < 30 \\
\text{C} & : 30 \leq \text{exam mark + c/w mark} < 50 \\
\text{B} & : 50 \leq \text{exam mark + c/w mark} < 70 \\
\text{A} & : 70 \leq \text{exam mark + c/w mark} < 100 \\
\text{FM} & : \text{exam mark + c/w mark} > 100
\end{align*}
\]
Equivalence partitions may also be considered for invalid outputs. It is difficult to identify unspecified outputs, but they must be considered as if we can cause one to occur then we have identified a defect in either the component, its specification, or both.

For example, for grade outputs we may propose the following invalid outputs.
- \( \text{grade} = 'E' \)
- \( \text{grade} = 'A+' \)
- \( \text{grade} = 'null' \)

In this example, we have proposed 19 equivalence partitions.

In developing equivalence partitions, the tester must exercise subjective choice. For example, the additional invalid inputs and invalid outputs. Due to the subjectivity, different testers will arrive at different equivalence partitions.

### 3.3.3.3 Design of Test Cases

Test cases are designed to exercise partitions.

A test case comprises the following:
- The inputs to the component
- The partitions exercised
- The expected outcome of the test case

Two approaches to developing test cases to exercise partitions are available:
1. Separate test cases are generated for each partition on a one-to-one basis
2. A minimal set of test cases is generated to cover all partitions. The same test case may be repeated for different test cases.

When developing test cases, the corresponding input or output value is varied to exercise the partition. Other input and output values, not related to the partition being exercised, are set to an arbitrary value.

### One-to-one Test Cases for Partitions

The test cases for the exam mark input partitions are as follows:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>44</td>
<td>-10</td>
<td>93</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>59</td>
<td>5</td>
<td>108</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>0 ≤ e ≤ 75</td>
<td>e &lt; 0</td>
<td>e &gt; 75</td>
</tr>
<tr>
<td>Expected output</td>
<td>'B'</td>
<td>'FM'</td>
<td>'FM'</td>
</tr>
</tbody>
</table>

An arbitrary value of 15 has been used for coursework mark inputs.

The test cases for the coursework mark input partitions are as follows:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>8</td>
<td>-15</td>
<td>47</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>48</td>
<td>25</td>
<td>87</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>0 ≤ c ≤ 25</td>
<td>c &lt; 0</td>
<td>c &gt; 25</td>
</tr>
<tr>
<td>Expected output</td>
<td>'C'</td>
<td>'FM'</td>
<td>'FM'</td>
</tr>
</tbody>
</table>
An arbitrary value of 40 has been used for exam mark inputs.

The test cases for other invalid input partitions are as follows:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>48.7</td>
<td>‘q’</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>15</td>
<td>15</td>
<td>12.76</td>
<td>‘g’</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>63.7</td>
<td>?</td>
<td>52.76</td>
<td>?</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>real</td>
<td>alpha</td>
<td>real</td>
<td>alpha</td>
</tr>
<tr>
<td>Expected output</td>
<td>‘FM’</td>
<td>‘FM’</td>
<td>‘FM’</td>
<td>‘FM’</td>
</tr>
</tbody>
</table>

The test cases for valid outputs partitions are as follows:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>-10</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>-10</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>-20</td>
<td>17</td>
<td>45</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>t &lt; 0</td>
<td>0 ≤ t &lt; 30</td>
<td>30 ≤ t &lt; 50</td>
</tr>
<tr>
<td>Expected output</td>
<td>‘FM’</td>
<td>‘D’</td>
<td>‘C’</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>44</td>
<td>60</td>
<td>80</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>22</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>66</td>
<td>80</td>
<td>110</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>50 ≤ t &lt; 70</td>
<td>70 ≤ t ≤ 100</td>
<td>t &gt; 100</td>
</tr>
<tr>
<td>Expected output</td>
<td>‘B’</td>
<td>‘A’</td>
<td>‘FM’</td>
</tr>
</tbody>
</table>

The input values of exam mark and coursework mark have been derived from the total mark.

Finally, invalid outputs partitions are considered:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>-10</td>
<td>100</td>
<td>null</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>0</td>
<td>10</td>
<td>null</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>-10</td>
<td>110</td>
<td>?</td>
</tr>
<tr>
<td>Partition tested (of exam mark)</td>
<td>‘E’</td>
<td>‘A+’</td>
<td>null’</td>
</tr>
<tr>
<td>Expected output</td>
<td>‘FM’</td>
<td>‘FM’</td>
<td>‘FM’</td>
</tr>
</tbody>
</table>

**Minimal Test Cases for Multiple Partitions**

In many cases above test cases are similar, but a targeting different equivalence partitions. It is possible to develop single test cases that exercise multiple partitions at the same time.

This approach enables the tester to reduce the number of test cases required to cover all the equivalence partitions.

For example, consider the following test case:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>60</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>20</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>80</td>
</tr>
</tbody>
</table>
3. Black Box Testing

Expected output | ‘A’
---|---
The above test case exercises three partitions:
1. \(0 \leq \text{exam mark} \leq 75\)
2. \(0 \leq \text{coursework mark} \leq 25\)
3. \(\text{grade result} = ‘A’ : 70 \leq \text{exam mark} + \text{coursework mark} \leq 100\)

Similarly, test cases can be developed to exercise multiple invalid value partitions:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>-10</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>-15</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>-25</td>
</tr>
<tr>
<td>Expected output</td>
<td>‘FM’</td>
</tr>
</tbody>
</table>

The above test case exercises another three partitions:
1. \(\text{exam mark} < 0\)
2. \(\text{coursework mark} < 0\)
3. \(\text{grade result} = ‘FM’ : \text{exam mark} + \text{coursework mark} < 0\)

Comparison of One-to-one and Minimalist Approaches

The disadvantage of the one-to-one approach is it requires more test cases. However the identification of partitions is more time consuming than the generation and execution of test cases. Any savings in reducing the number of test cases is relatively small compared with the cost of apply the technique to propose partitions.

The disadvantage of the minimalist approach is the difficulty of determining the cause in the event of a failure. This is due to many different partitions being exercised at once. This is more a problem of making debugging more difficult, rather than affecting the testing process.
3.3.4 Boundary Value Analysis

Boundary value analysis is a standardised technique in [BCS97a].

Boundary value analysis extends equivalence partitioning to include values around the edges of the partitions.

As with equivalence partitioning, we assume that sets of values are treated similarly by components. However, developers are prone to making errors in the treatment of values on the boundaries of these partitions.

For example, elements of a list may be processed similarly, and they may be grouped into a single equivalence partition. However, in processing the elements, the developer may not have correct processing for either the first or last element of the list.

Boundary-values are usually the limits of the equivalence classes. Examples include:

- Monday and Sunday for weekdays
- January and December for months
- 32767 and -32768 for 16-bit integers
- top-left and bottom-right cursor position on a screen
- first line and last line of a printed report
- 1 January 2000 for two digit year processing
- strings of one character and maximum length strings

Test cases are selected to exercise values both on and either side of the boundary. Values either side of the boundary are selected at an incremental distance from the boundary, the increment being the smallest significant value for the data type under consideration (e.g. increment of 1 for integers, $0.01 for dollars).

3.3.4.1 Example

Consider again generate_grading function, used previously as an example for the equivalence partitioning technique above. This example is extracted from [BCS97a].

Initially equivalence partitions are identified (these are the same in the previous equivalence partitioning technique), and then these are used to propose boundary values.

For example, the exam mark partitions give rise to boundary value tests that exercise exam marks of -1, 0, 1, 74, 75, and 76:
Test cases based on the boundary values for the exam mark are:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>74</td>
<td>75</td>
<td>76</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>89</td>
<td>90</td>
<td>91</td>
</tr>
<tr>
<td>Boundary value (exam mark)</td>
<td>0</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An arbitrary output of 15 has been used for coursework mark inputs.

Similarly coursework marks give rise to boundary values of 0 and 25, which give rise to the test cases:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input (exam mark)</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>64</td>
<td>65</td>
<td>66</td>
</tr>
<tr>
<td>Boundary value (coursework mark)</td>
<td>0</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected output</td>
<td>‘FM’</td>
<td>‘C’</td>
<td>‘C’</td>
<td>‘B’</td>
<td>‘B’</td>
<td>‘FM’</td>
</tr>
</tbody>
</table>

An arbitrary value of 40 has been used for exam mark inputs.

In the equivalence partitioning example we also considered non-integer and non-numeric partitions of exam and coursework marks, namely real numbers and alphabetic values. Even those these are valid equivalence partitions, they have no identifiable boundaries, and therefore no boundary values will be considered for deriving test cases.

Equivalence partitions for the grade result are also considered. The boundaries of grade result partitions are 0, 30, 50, 70, and 100:
Test cases based on the boundary values for the grade output are:

<table>
<thead>
<tr>
<th>Test Case</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input (exam mark)</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>15</td>
<td>6</td>
<td>24</td>
<td>50</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>25</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>Boundary value (total mark)</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Expected output</td>
<td>F'M'</td>
<td>D'</td>
<td>D'</td>
<td>D'</td>
<td>C'</td>
<td>C'</td>
<td>C'</td>
<td>B'</td>
<td>B'</td>
<td></td>
</tr>
</tbody>
</table>

Invalid partitions for grade results used in the equivalence partitioning example, (i.e. 'E', 'A+' and null), do not have identifiable boundaries, and no test cases will be proposed.

Note that many of the identified partitions were bounded on one side only, namely:

- exam mark > 75
- exam mark < 0
- coursework mark > 25
- coursework mark < 0
- exam mark + coursework mark > 100
- exam mark + coursework mark < 0

These partitions could be assumed to be bound by the data type used for the input or output. For example, 16 bit integers have boundaries of 32767 and -32768.

Thus we could propose another 18 test cases, for instance for exam mark we have:

<table>
<thead>
<tr>
<th>Test Case</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input (exam mark)</td>
<td>49</td>
<td>45</td>
<td>71</td>
<td>74</td>
<td>75</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input (coursework mark)</td>
<td>20</td>
<td>25</td>
<td>0</td>
<td>25</td>
<td>25</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total mark (as calculated)</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary value (total mark)</td>
<td></td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expected output</td>
<td>B'</td>
<td>A'</td>
<td>A'</td>
<td>A'</td>
<td>A'</td>
<td>F'M'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similar test cases would be provided for coursework marks, and grade result (based on total mark boundary values).
The function has the following specification:

> If there are sufficient funds available in the account or the new balance would be within the authorised overdraft limit then the debit is processed. If the new balance would exceed the authorised overdraft limit then the debit is not processed and if it is a postal account it is suspended. Letters are sent out for all transactions on postal accounts and for non-postal accounts if there are insufficient funds available (i.e. the account would no longer be in credit).

### 3.3.5.2 Analysis of Cause-Effects Graph

The conditions are:

- C1 → new balance in credit
- C2 → new balance in overdraft, but within authorised limit
- C3 → account is postal

The actions are:

- A1 → process debit
- A2 → suspend account
- A3 → send out letter
The following cause-effect graph shows the relationships between conditions and actions:

![Cause-effect graph for cheque debit function.](image)

The cause-effect graph is then reformulated as a decision table. All true and false combinations for input conditions are proposed, and true and false values are allocated to actions (* is used for combinations of input conditions that are infeasible and consequently there are no actions possible). The result is shown in the following decision table:

<table>
<thead>
<tr>
<th>Rule</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: new balance in credit</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C2: new balance overdraft,</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>but within authorised limit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3: account is postal</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>A1: process debit</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>A2: suspend account</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>A3: send out letter</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

The feasible combinations of inputs are then covered by test cases:

<table>
<thead>
<tr>
<th>test case</th>
<th>causes</th>
<th>effects</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>account type</td>
<td>overdraft limit</td>
</tr>
<tr>
<td>1</td>
<td>'c'</td>
<td>$100</td>
</tr>
<tr>
<td>2</td>
<td>'p'</td>
<td>$1500</td>
</tr>
<tr>
<td>3</td>
<td>'c'</td>
<td>$250</td>
</tr>
<tr>
<td>4</td>
<td>'p'</td>
<td>$750</td>
</tr>
<tr>
<td>5</td>
<td>'c'</td>
<td>$1000</td>
</tr>
<tr>
<td>6</td>
<td>'p'</td>
<td>$500</td>
</tr>
</tbody>
</table>
3.3.6 State Transition Testing

State transition testing is a standardised technique in [BCS97a].
State transition testing uses a model of the system comprising:

- the states the program may occupy
- the transitions between those states
- the events which cause those transitions
- the actions which may result

The model is typically represented as a state transition diagram.

Test cases are designed to exercise the valid transitions between states. Additional test cases may also be designed to test that unspecified transitions cannot be induced.

3.3.6.1 Example

Consider the following example state transition diagram that handles input requests for a display mode of a time display device.

The state transition diagram consists of:

- states, e.g. displaying time (S1)
- transitions, e.g. between S1 and S3
- events that cause transitions, e.g. "reset" during state S1 will cause a transition to S3
• actions that result from the transition, e.g. during the transition from S1 to S3 as a result of event ‘reset’, the action ‘display time’ will occur.

### 3.3.6.2 Test Cases for Valid Transitions

Test cases are designed to exercise valid transitions.

For each test case, the following are specified:

- the starting state
- the input(s)
- the expected outputs
- the expected final state

For the example above this provides the following 6 test cases:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start State</td>
<td>S1</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S2</td>
<td>S4</td>
</tr>
<tr>
<td>Input</td>
<td>CM</td>
<td>R</td>
<td>TS</td>
<td>CM</td>
<td>R</td>
<td>DS</td>
</tr>
<tr>
<td>Expected Output</td>
<td>D</td>
<td>AT</td>
<td>T</td>
<td>T</td>
<td>AD</td>
<td>D</td>
</tr>
<tr>
<td>Finish State</td>
<td>S2</td>
<td>S3</td>
<td>S1</td>
<td>S1</td>
<td>S4</td>
<td>S2</td>
</tr>
</tbody>
</table>

The above set of test cases achieves 0-switch coverage [Cho78a].

Other levels of switch coverage are achieved by joining longer sequences of transitions:

- 1-switch coverage is achieved by looking at the result of performing a sequence of two valid transitions for each test
- N-switch coverage is achieved by looking at the result of performing a sequence of N+1 valid transitions for each test

### 3.3.6.3 Test Cases for Invalid Transitions

State transition tests designed for switch coverage only test for valid sequences of transitions. Comprehensive testing will also try to cause invalid transitions to occur.

The software transition diagram above only shows valid transitions. A state transition model that explicitly shows invalid transitions is a state table.

For example, the state table for the time display device is shown below.

<table>
<thead>
<tr>
<th></th>
<th>CM</th>
<th>R</th>
<th>TS</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S2/D</td>
<td>S3/AT</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>S2</td>
<td>S1/T</td>
<td>S4/AD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S3</td>
<td>-</td>
<td>-</td>
<td>S1/T</td>
<td>-</td>
</tr>
<tr>
<td>S4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>S2/D</td>
</tr>
</tbody>
</table>
Cells of the state table shown as -, represent the null transition, where any transition that can be induced will represent a failure.

Tests are proposed as shown for valid transitions to test these invalid transitions. The state table is ideal for identifying the test cases. There are 16 tests corresponding to the cells of the state table.
3.3.7 Other Techniques

There are many more black-box test design techniques. The following briefly identifies a few more:

Syntax Testing

Uses a model of the formally designed syntax of the inputs to the system. Tests are derived for valid and invalid representations according to the syntax model.

The model represents the syntax using a number of rules that define how a valid language term is built up from iterations, sequence and selections of symbols or other terms of the language.

Such models are often provided for programming languages, and can be found at the back of programming textbooks and manuals.

Test cases are derived from the rules using a number of predefined criteria.

See Appendix B.5 in [BCS97a] for more detail.

Cross-Functional Testing

Cross-functional testing uses a matrix of interactions between features of a system. The matrix has X and Y axes being the features of the system, and the cells indicate which component is being updated by one feature and then used in another.

Tests are designed from the matrix to show that these interactions between features follows that defined in the matrix.

See Chapters 3 and 8 of [Col97a] for further detail.
3.4 Black Box Test Coverage Metrics

A number of different coverage metrics can be formulated for black-box techniques:

- Requirement coverage – percentage of functional requirements covered by (and traced to) test cases
- Equivalence partitioning – percentage of equivalence partitions covered by test cases
- Boundary value analysis – percentage of boundary values covered by test cases
- Cause effect graph – percentage of unique possible combinations covered by test cases
- Decision table – percentage of unique possible combinations covered by test cases
- State transition testing – 0-, 1- and N-switch coverage

Quite often it is necessary to build up trace matrices to record coverage, e.g. a trace matrix is kept between functional requirements in a specification to the test cases.

Coverage metrics are best used as a guide. For further reading, see Kaner's article discussing testing negligence and its relation to coverage metrics [Kan96a].
3.5 Test Oracles\[KR9]\[
Capture and comparison of results is one key to successful software testing. Test oracles\{ XE "test:oracles" \} \{ XE "oracles" \} are an mechanism used to calculate the expected result of a test.

Test oracles are commonly the tester calculating or checking the result by hand.

If the test case output involves complex calculations, then programs may be written to compute or check the output. Care must be taken to ensure that the same approach is not used for developing the test oracle as used in developing the system operation tested by the oracle, otherwise errors may remain undetected.

If different results are obtained for a test case, the tester must first determine whether the system under test is incorrect or whether the oracle is incorrect.

Douglas Hoffman [Hof99a] categorised a number of different alternatives for implementing oracles:

<table>
<thead>
<tr>
<th>True Oracle{ XE &quot;oracles:truese&quot; }</th>
<th>Heuristic Oracle{ XE &quot;heuristic:truese&quot; }</th>
<th>Sampling Oracle{ XE &quot;oracles:sampling&quot; }</th>
<th>Consistent Oracle{ XE &quot;oracles:consistent&quot; }</th>
<th>No Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td><strong>Definition</strong></td>
<td><strong>Definition</strong></td>
<td><strong>Definition</strong></td>
<td><strong>Definition</strong></td>
</tr>
<tr>
<td>• Independent generation of all expected results</td>
<td>• Verifies some values, as well as consistency of remaining values</td>
<td>• Selects a specific collection of inputs or results</td>
<td>• Verifies current run results with a previous run (Regression Test)</td>
<td>• Doesn't check correctness of results (only that some results were produced)</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td>• No encountered errors go undetected</td>
<td>• Faster and easier than True Oracle</td>
<td>• Can select easily computed or recognised results</td>
<td>• Fastest method using an oracle</td>
<td>• Can run any amount of data (limited only by the time the SUT takes)</td>
</tr>
<tr>
<td>• Much less expensive to create and use</td>
<td>• Much less expensive to create and use</td>
<td>• Can manually verify with only simple oracle</td>
<td>• Verification is straightforward</td>
<td></td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
<td><strong>Disadvantages</strong></td>
</tr>
<tr>
<td>• Expensive to implement</td>
<td>• Expensive to implement</td>
<td>• Can miss systematic errors</td>
<td>• Original run may include undetected errors</td>
<td>• Only spectacular failures are noticed</td>
</tr>
<tr>
<td>• Complex and often time-consuming when run</td>
<td>• Complex and often time-consuming when run</td>
<td>• May miss systematic and specific errors</td>
<td>• Original run may include undetected errors</td>
<td></td>
</tr>
<tr>
<td>• Often &quot;trains the software to pass the test&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consider a function to compute a sine wave.
3.6 True Oracle

Using a true oracle the tester would select points on the X axis and calculate the sine at each point selected. The calculation should be performed using an independent algorithm, as errors may go undetected if the system under test and the oracle share the same mechanisms for calculating results.

3.7 Sampling Oracle

Using a sampling oracle the tester would select specific points on the X axis where the sine result is known without calculation. For example, the outcome of the sine function is precisely known at 0, 90, 180, 270 and 360 degrees.
3.8 Heuristic Oracle

Using a heuristic oracle (see paper by Hoffman [Hof99a]) the tester would analyse the inputs and results of the sine function to determine if there are predictable relationships between them. The tester would then provide a number of test cases using the sampling oracle approach above, and then the remainder of the tests would be based on consistency of inputs and results with the predicted relationships.

For example, the sampling oracle defined tests at 0, 90, 180, 270 and 360 degrees. In addition, the tester notices that between 0 and 90 degrees, the result of sine(X) only increases as X increases. Similarly, between 90 and 180 degrees, sine(X) always decreases as X increases, and so on for 180 to 270 and 270 to 360 degrees.

The heuristic oracle for sine is very simple as we need only check for increases and decreases depending on the range. This example oracle will trap a number of different errors, e.g. dips, scale or discontinuity of the function.
Some kinds of errors may go undetected using a heuristic oracle. For example, rounding errors, or constant change of the sine function may go undetected.

Using heuristic oracles, a tester looks for simple predictable relationships between inputs and results that they can easily check. Most complex algorithms contain simple relationships. Selected ranges may be selected for the heuristic, but the heuristic must hold for all inputs and results within the range. If there are exceptions it becomes more complicated to check whether the exceptions apply, and the advantages are lost.

It is best to define to groups, what can know to expect, and what we can't determine separately. The former are tested using a sampling oracle, and the latter using a heuristic oracle.

When looking for patterns, consider reordering data. For instance, two equivalent sets of elements can be sorted and compared to check that they contain matching items. For instance, where new database items are created using an ID that is the next increment, sorting the items by ID may reveal any breaks in the sequence to reveal missing items.

Blocks of information can be represented by the starting value and count of elements. Variations are possible by using fixed increments between values or using simple patterns for changing increments. For example, to test packets of data transmitted over a network, packets can be randomly generated with varying lengths, and starting values, but fixed increments. It is then simple to check the integrity of the received packet, by only understanding the packet start value, length and increment parameters, without recording the complete packet contents as an expected result.
4. Testing Infrastructure

- developing environments for testing
- test documentation
- configuration management and change control
4.1 Test Environments

Test environments\{XE "test:environments"\} are the facilities built up around the system under test to assist in the testing process.

The test environment includes:

- the system or components under test
- test documentation, describing the tests to be performed and recording results
- test harnesses and test stubs, used to replace parts of the system that are not yet available or cannot be used to control the system for the necessary tests
- test oracles, used to predict or check outputs of test cases
- test data, including data files that are used as inputs in test cases

Much of the test environment may be provided by automated testing facilities. These, however, are not discussed here, and will be addressed in later sections.

There are a number of test characteristics to be considered when performing tests and building a test environment:

Testability\{XE "testability"

How easily can the software system be tested. The developers need to consider how the system can be tested during design of the system.

Testability must consider both controllability and observability.

In many cases, tests will not be able to be performed at the higher-levels, such as system testing, and control of certain tests can only be done at unit testing. For example, invoking error conditions, such as network parity error detection.

Controllability\{XE "controllability"

Testing is limited by the facilities available for to control the system to provide the inputs and invoke the conditions necessary to exercise test cases.

The system must be controllable to be able to put the system in a state where the test can be carried out.

Also it must be possible to return the system to a state in which other tests can be carried out.

Observability

When test case are run there must be some mechanism to check the results of the test case are correct.

For example, the user may observe the display to check the output, or they may check a data file that is output using an automated test oracle.

Predictability\{XE "predictability"

Predictability is the expected output of the test case can be determined in advance of the test case being executed.

Test cases that are not predictable may result in operation that is not known whether it is correct.

Often this is achieved by controlling the environment in which a test is executed to limit other systems interfering with results of the test. For instance, this may involve providing a test
stub for interfaces to provide constant interface data under which the same behaviour can always be observed.

Repeatability
XE "repeatability"

Test cases need to be repeatable. The test should return the same result each time the test case is executed. We do not mean that the same value is returned each time, but we expect the test case always returns either a pass result or a fail result.

The tester will need to repeat the test case at different times and the procedures should be sufficiently defined that the do not have to consider different alternatives each time. More importantly, the developer will want to see reproducible tests when trying to debug an incident brought about by a failed test case.

The environment needs to be controlled so that the complete environment can be rolled back to perform a test. For example, this may included resetting databases and input files.

4.1.1 Test Toolkit

4.1.1.1 Test Harnesses

A test harness (XE “test:harness”) or test driver (XE “test:driver”) is supporting code and data used to provide an environment for testing components of your system.

Test harnesses are typically used during unit/module test and are typically created by the developer of the code.

Some test harnesses provide a simple user interface for the testing. Others are more sophisticated and are designed to fully automate the testing to provide repeatability when the code is modified.

Building test harnesses may account for 50% of the total code produced. Test harnesses are a valuable asset, and should be controlled in the same way the code is managed, e.g. with proper change control, configuration management, review and maintenance procedures.

4.1.1.2 Test Stubs

Test stubs (XE “test:stub”) are dummy components that stand in for unfinished components during unit/module testing and integration testing.

Test stubs quite often code routines that have no or limited internal processing, i.e. just function or procedure headers. If a return value is required from a routine, then code may be provided to always return a specific value for the purposes of testing (e.g. a boolean function always returning true for any input). More sophisticated stubs may request the tester to provide a value to be returned by the routine, or read the return value from a data file.

Often, simulators (XE “simulator”) may be used as test stubs to mimic other interfaced systems. Simulators are commonly used in testing of real-time systems, where the system provide continuous data that resembles operational profiles of interfaced equipment or network traffic.
In some cases test stubs are created by taking the code for a low level module and modifying it slightly to provide specific control to achieve the required test criteria. For example, a file I/O driver module may be modified to inject file I/O errors at specific times, and thus enable the error handling of I/O errors to be analysed.[KR12]

4.1.1.3 Test Oracles

**Test oracles** are facilities provided to calculate the expected output for a test case or to check that the actual output for the test case is correct. Test oracles are covered in Section 3.5.

4.1.2 Test Environment Independence

It is very important to maintain independence between the test environment and the production environment wherever possible.

The main reasons for this are:

- To prevent testing from corrupting the production environment. The main purpose of testing is to break the system. However, the last thing we want to do is break the production environment.

  Testing will also degrade performance of the test environment as it will increase the load on the environment. This may be a problem during normal business hours if system performance is critical.

- To prevent the production environment from influencing testing. The outcome of tests could be adversely affected by operations performed within the production environment. Either the expected results may change, or certain production events may alter the procedures required to run the tests (e.g. acknowledge mail message)

- Testing typically requires significant reconfiguration of the system. For instance, new databases may need to be installed, or different network equipment needs to be (de)commissioned. Having to do this frequently is wasting time and money.

Where practicable, separate equipment should be provided.

Avoid the testing night shift. It is more expensive, lowers staff morale, and if anything goes wrong there is often not enough time to get production back on-line before business hours anyhow. Also you may not restore the system back to its required production configuration.

Where practicable, avoid the use of production data:

- Changes to production data affect the tests. Test may no longer have the results expected as inputs to the test have changed.

- The large volume of production data makes it difficult to detect errors or track down the cause of errors. For example, imagine trying to check that a record is inserted into a report in the correct order if there are 10,000 production data records already in the report. What if there were 1,000,000 records.

Managers will push to have production data used. However, it is not the “real” properties that is important for testing. The different values in the production data may not provide sufficient coverage.
Even for performance and load testing, better control of testing is achieved if test data is used rather than production data.
4.2 Test Documentation

Serious software projects are a communications exercise, as there are many interested parties involved in the testing process.

Any communication dependent activity, such as the testing process, is dependent on good documentation.

Test documentation, however, must be balanced. Documentation becomes a tradeoff between a big wall of paper that can be measured in the metres of documentation, such as the case in some defence projects, and traceable decision support that guides testers on what activities are performed next and how they are to perform them.

Test documentation should record:

- what needs to be done
- what has been accomplished

Test documentation also serves as protection against litigation. The courts tend to look favourably upon organisations that have comprehensive documentation, which indicates a rigorous process has been followed. However, documentation that reveals defects in the system, but then which have gone on to be ignored could be the noose that the layers use to hang you.

Documentation need not necessarily be written documents. For instance, test documentation in the form of data used by test tools can form effective test documentation.

Test documentation is most effective when

- traceable – the reader can trace the development of information and concepts through the test documentation, beginning from the formulation of an approach in the test plans, through the design of the test cases, to the result of the executing the tests recorded in test logs.
- auditable – the test documentation can be evaluated or review to determine whether the testing is consistent and complete
As with any product of the software development process, however, there is significant effort required to keep the documentation up-to-date. Whenever the software requirements, design or code is modified the test documentation should be modified accordingly. Otherwise, the test documentation becomes rapidly out-of-date and no longer serves can be used to describe the testing performed or what is required if testing is to be repeated for any parts of the system.

4.2.1 Objectives of Test Documentation

Before test execution commences:

- Define overall testing strategy for a project
- Plan how each level of testing is to be conducted
- Identify what tests are to be carried out
- Define how each test is to be performed

After test execution is conducted:

- Record execution of each test
- Track incidents that arise during testing
- Summarise outcomes of testing activities and make recommendations regarding quality and acceptability

4.2.2 Test Documentation Standard

IEEE provides comprehensive guidance on the structure of test documentation [IEEE83a].

The IEEE proposes 7 different kinds of test documents:

- Test plan
- Test design specification
- Test case specification
- Test procedure specification
- Test item transmittal
- Test log
- Test summary report

4.2.2.1 Test Plan

See Section 2.4.2.

4.2.2.2 Test Design Specification

Main sections:

1. Identifier
2. Features to be tested
3. Approach refinements
4. Test identification
5. Feature pass/fail criteria

4.2.2.3 **Test Case Specification**
Main sections:
1. Identifier
2. Test items
3. Input specifications
4. Output specifications
5. Environmental needs
6. Special procedural requirements
7. Inter-case dependencies

4.2.2.4 **Test Procedure Specification**
Main sections:
1. Identifier
2. Purpose
3. Special requirements
4. Procedure steps
   4.1 Log
   4.2 Setup
   4.3 Start
   4.4 Proceed
   4.5 Measure
   4.6 Shutdown
   4.7 Restart
4. Testing Infrastructure

4.8 Stop
4.9 Wrap-up
4.10 Contingencies

4.2.2.5 Test Item Transmittal Report
Main sections:
1. Identifier
2. Transmittal items
3. Location
4. Status
5. Approvals

4.2.2.6 Test Log
Main sections:
1. Identifier
2. Description
3. Activity and event entries
   3.1 Execution description
   3.2 Procedure results
   3.3 Environmental information
   3.4 Anomalous events
   3.5 Incident report identifiers

4.2.2.7 Test Incident Report
Main sections:
1. Identifier
2. Summary
3. Incident description
4. Impact

4.2.2.8 Test Summary Report
Main sections:
1. Identifier
2. Summary
3. Variances
4. Comprehensive assessment
5. Summary of results
6. Evaluation
7. Summary of activities
8. Approvals
4.3 Configuration Management and Change Control

Effective management of the configurations of components used in testing and control of how changes are applied is crucial to the success of a testing project.

Configuration Management (CM) is the management of the evolution of components that make up a software development and applies equally to testing as the design and implementation of the product. CM provides the means for managing integrity and traceability throughout the development artifacts produced during software development and testing.

CM is responsible for managing the change process and tracking changes to ensure the configuration of the product is accurately known at any time.

CM is used to reduce many problems associated with developing software systems:

- Inconsistent/Inappropriate configurations released to testers/customers
- Testing an out-of-date component
- Testing the wrong component
- Components changing mid-way through testing
- Inability to track faults back to their originating system configurations
- Untested components used in subsequent development

4.3.1 CM Planning

There are four main activities in defining what CM should be employed:

- Identification
- Control
- Status Accounting
- Audit and Review

These are described in following sections.

4.3.1.1 CM Identification

Identification of all the components and their versions, baselines and configurations which represent the systems:

- Specifications, designs, code
- User manuals, operational instructions, installation procedures
- Inspection reports, change requests, problem statements
- Test plans, stubs, harnesses, test data

Identify the configurations of different components that make up a product. For example, a parts list describing the versions of subcomponents used.

Identify the attributes that can be assigned to components. For example, whether a variant for a specific platform, such as Windows vs. Unix specific variants.
Define how the components can be stored in a repository. For example, what directory hierarchies are used for supporting projects and the different kinds of information in a project.

4.3.1.2 CM Control

CM control describes how changes to components of software development and testing activities are controlled. This includes:

- How to access the components. For example, components for release are only found in a particular repository or directory on the network.
- Who can access components. For example, who has the authority to hand components over to testers.
- When components can be accessed. For example, the preconditions to be satisfied before release of the system, such as passing all tests. Similarly, we might indicate that a configuration of a system under test is frozen and no changes are allowed until the system is released from test.
- Assessment of the impact of proposed changes. For example, having reviews of proposed changes made and approval granted prior to changes being made.

CM control is established through defined procedures, for example:

- Configuration Control Boards - to review proposed changes before they are made, and to review whether the configuration is suitable for release
- Change requests/authorizations - formal records of change proposals

4.3.1.3 CM Status Accounting

Reporting functions for assessing state of the development configuration

Indication of criteria, such as:

- When development phases are complete. For example, reporting criteria that indicate whether the project manager can assume the system testing phase is complete, such as statuses of the test summary reports
- Components tested. For instance, a report of all those modules which have undergone unit testing.
- Work pending. For example, notification that a system configuration has changed and that testing has not been applied to the new configuration.
- Changes requested. For example, an indication of what changes remain pending for a component so the project manager may whether the testing should commence or wait until changes made.

4.3.1.4 CM Audit and Review

CM audit and review verifies whether:

- The configuration is complete. For example, all components necessary for testing are included.
- The configuration is a consistent set of parts. For example, the specifications packaged with a system ready for test are the correct version.
4.3.2 CM Tools

Turnkey/base systems

- **Repositories/version management** – supports storing of component and versions; provide structuring mechanisms for defining hierarchies of components and configurations of subcomponents

- **Build support** – supports constructing different builds of the system, (many based on makefile concept)

- **Change request systems** – support lodging of change requests and release of configuration according to a change approval process

- **Workflow/team management** – support different areas of development where developers and testers can work without interference through changes made by other developers; define when components can be accessed and by whom

Customised systems

- Specifically tailored to organisation needs/process

- Integrated across different CM activities

4.3.3 Tailoring CM

Different organizations have different CM requirements. Even within the same organisation needs differ and requirements have to be integrated.

Levels of CM:

- **Organisational Level** – how the company as a whole manages configuration

- **Project Level** – projects may have different kinds of configurations and different processes

- **Application Level** – how different groups in a project might vary, such as testers needs versus developers

- **Individual Level** – individuals have their own preferences

4.3.4 Main Considerations for Testing

- Define what components need to be placed under version and change management for testing

- Define criteria for development handing over components for testing

- Define how testing is isolated from changes to components

- Define mechanisms for testing to be able to specifically identify versions of components or systems under test

- Define what test documentation needs to be placed under version and change management and conditions under which changes are allowed.
5. White Box Testing

- statement, branch and path coverage
- loop testing
- decision coverage
- pros and cons of white-box testing
- complexity metrics
5.1 Purpose of White-Box Testing

A common goal of tester’s is to exercise every path through a program. However, as we saw it is infeasible to do this. Recall the simple program with a trillion paths (see Section 1.5.3).

Rather than attempting to test every conceivable path, white-box testing techniques attempt to provide mechanisms for selecting particular paths to test.

Quite often white-box testing techniques are associated with test coverage metrics, which measure the percentage of paths of the selected type that are exercised by test cases.

It is common in projects that target coverage levels are set to guide the developers and testers on how thoroughly a program must be tested.

White-box testing is applied at the source code (program) level. Hence the term white-box, meaning that we can look into the code to find out exactly what logic is being used to control the application under test. In black-box testing we do not have the ability to look at the actual logic being used, and must use specification or our intuition to come up with the test cases.

When looking at the source code during white-box testing, we use the code’s structure to define what kinds of tests are required. In many cases, tests are proposed to exercise certain paths through the code and to pass through certain parts of the code’s structure.
5.2 Statement, Branch and Path Coverage

Statement coverage, branch coverage and path coverage are white-box testing techniques that utilise the logical flow through a program to propose test cases. By logical flow we mean, the way in which certain parts of a program may be executed as we run the program.

Logical flow of a program can be represented by a flow graph. For instance, the following shows the flow graph derived from a sample program.

```
if A then
  if B then
    D
  else
    E
  end if;
else
  if C then
    F
  else
    G
  end if;
end if
```

The flow graph is made up of:
- **Nodes**, representing statements (or subprograms) that may be visited during execution of the program
- **Edges**, representing the way in which the logic allows the program to pass from one statement (or subprogram) to another. Edges are directional.
- **Branch nodes**, nodes which have more than one edge exiting itself.
- **Branch edges**, edges which exit a branch.
- **Paths**, the possible ways to move from one node to another by travelling along edges.

Execution of a test case causes the program to execute certain statements, which corresponds to taking a specific path through the flow graph.

Different paths can be taken by vary the conditions to cause a different branch edge to be taken from a branch node. This corresponds to varying the test case values execute different parts of the program.

Branch, statement and path coverage is derived from the execution path of a program in ways that correspond to visiting nodes, edges and paths within the flow graph.
5.2.1 Statement Coverage

Statement coverage is determined by assessing the proportion of statements visited by the set of proposed test cases. 100% statement coverage is where every statement in the program is visited by at least one test.

Statement coverage corresponds to visiting the nodes of the graph. 100% coverage is where all nodes are visited by the paths taken by the test cases.

There are 10 nodes in the sample flow graph.

A, B, D, H, K traces one execution path through the program. It visits 5 of the 10 nodes, thus 50% coverage.

Actual statement coverage levels may differ from node coverage levels depending on the way your graph is defined, e.g. depending on how many statements are grouped as one node, etc.

5.2.2 Branch Coverage

Branch coverage is determined by assessing the proportion of decision branches exercised by the set of proposed test cases. 100% branch coverage is where every decision branch in the program is visited by at least one test.

Branch coverage corresponds to visiting the branch edges of the graph. 100% coverage is where all branch edges are visited by the paths taken by the test cases.

There are 6 branch edges in the sample flow graph.

A, B, D, H, K traces one execution path through the program. It visits 2 of the 6 branch edges, thus 33% coverage.
5.2.3 Path Coverage

Path coverage is determined by assessing the proportion of execution paths through a program exercised by the set of proposed test cases. 100% path coverage is where every path in the program is visited by at least one test.

Path coverage corresponds to visiting the paths through the graph. 100% coverage is where all paths are taken by the test cases.

There are 4 paths in the sample flow graph.

A, B, D, H, K traces one execution path through the program. It visits 1 of the 4 paths, thus 25% coverage.

5.2.4 Statement, Branch and Path Coverage Differences

It is possible to have 100% statement coverage without 100% branch coverage. All nodes are visited, without visiting all branch edges.

Example corresponds to program:

```plaintext
if A then B;
C
```

Note that there is no else part.

It is possible to have 100% branch coverage without 100% path coverage. All branch edges are visited, without visiting all paths.

Example corresponds to program:

```plaintext
if A then B else C;
if D then E else F;
```

Two paths visit all branch edges, but there are 4 possible paths through the graph.

Where 100% path coverage is achieved, 100% branch coverage is
achieved by default. Where 100% branch coverage is achieved, 100% statement coverage is achieved by default.

5.2.5 Designing Tests for Coverage
Steps
1. Analyse source code to derive flow graph
2. Propose test paths to achieve coverage from the flow graph
3. Evaluate test conditions to achieve each path
4. Propose input and output values based on conditions

5.2.6 Case Study
Consider the following program that performs a Binary Search.

```plaintext
procedure Binary_Search (Key : ELEM; T : ELEM_ARRAY;
    Found : in out BOOLEAN; L : in out ELEM_INDEX ) is
    - - Assume that T'FIRST and T'LAST are both
    - - than or equal to zero and T'LAST >= T'FIRST
    Bot : ELEM_INDEX := T'FIRST;
    Top : ELEM_INDEX := T'LAST;
    Mid : ELEM_INDEX;
begin
    L := (T'FIRST + T'LAST) mod 2;
    Found := T(L) = Key;
    while Bot <= Top and not Found loop
        Mid := (Top + Bot) mod 2;
        if T( Mid ) = Key then
            Found := true;
            L := Mid;
        elsif T( Mid ) < Key then
            Bot := Mid + 1;
        else
            Top := Mid – 1;
        end if;
    end loop;
end Binary_Search;
```

The program works by first choosing the middle point of the array. It then compares the value at that point to the key. If it matches then the program exits and reports the key found at the middle position.

If no match is made at the middle position then the value at the middle position is compared to the key.
• If it is less than the key, then the key will only be found above the middle position and the bottom, middle and top values of the array and changed and the search continued based on these new settings.

• If it is greater than the key, then the key must be in the lower half of the array, and the bottom, middle and top settings are changed to look in that part of the array repeating the search.

The following illustrates the steps in searching.

Start:

Key = 6

\[ T = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array} \]

Iteration 0:

Key = 6

\[ T = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array} \]

\[ \text{Bot} \quad \text{L} \quad \text{Top} \]

\[ T(L) = 5 \]

Found = FALSE

Iteration 1:

Key = 6

\[ T = \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array} \]

\[ \text{L, Mid} \quad \text{Bot} \quad \text{Top} \]

\[ T(\text{Mid}) = 5 \]

Found = FALSE
Iteration 2:

The flow graph for this program is as follows:

Key = 6
T(Mid) = 8
Found = FALSE

Iteration 3:

Exit Program (Found = TRUE; L = 6):
The flow graph for this program is as follows:
Note that in the flow graph above, some statements have been merged into single nodes.

5.2.6.1 Propose Coverage Paths

One path can provide 100% statement and branch coverage (100% path coverage is not feasible):


This path corresponds to the example illustrated above.

In most cases, it is unlikely that you can choose one case that will work this way. Some combinations of paths will not be achievable within the one test case.

For example, the following path would give 100% statement and branch coverage based on the flow graph:


However, once you sit down and try to turn this in to a test case you realise that this path is not possible. The B, C, D, J iteration at the start would mean that no further iterations of the loop would occur. In fact, the first iteration of the loop will never satisfy T(Mid) = Key as this fact is discounted before entering the loop (i.e. T(L) = Key).

In general, it is better to come up with simple test cases that look at different areas of coverage separately.

For example, start with a simple single iteration of the loop, such as a test that has to move the mid point once to the left before finding the key in the array. Inputs for this test can be specified quite simply, and upon analysis will result in coverage using the path


This leaves only a path that visits node F to achieve 100% path coverage. This could be a similar test case, that instead looks to the once to the right before finding the key in the array:

- A, B, C, E, F, H, J, B, C, D, J, B, I

It is much simpler to come up with these simpler cases, than it is to consider the one long path that achieves 100% coverage in one hit.

5.2.6.2 Analysing Paths for Test Case Values

It is possible to work backwards from a path to obtain a test case, but it takes some analysis. However, this may be required when your guesses at test cases still aren’t covering the path you require that fulfil coverage requirements.

To analyse the path, you need to develop an understanding of the conditions in the program that force that path to be taken, and you must be able to trace the path through the program and record the decisions that are made to stay on the path.

Let’s assume that you propose a path that doesn’t enter the loop:

- A, B, I

In this path the loop is not executed and at node B the loop condition fails on the first attempt. Thus we can deduce that the loop condition
Bot <= Top and not Found was FALSE. This means that for this path to be taken we must have test inputs that either result in:

- Bot > Top – the input array is empty
- Found – the key was found on first access, i.e. the middle of the array was the key

In either case, the tester would need to build input values that satisfy the required condition to follow the path. However, it gets more difficult as the paths get longer. Test case inputs and expected outputs for the paths suggested above, namely:

- A, B, C, E, F, H, J, B, C, D, J, B, I

are left as an exercise.

As you can see, deducing test case values is difficult, and is somewhat similar to debugging. However, by going through this process you will potentially find problems, even before you execute a single test case on the running program.

5.2.7 Review of Coverage Example

Difficult to determine what paths are feasible and test case values to satisfy paths. This requires a solid understanding of the program logic to develop the inputs required to progress down a certain path. Easier to have a number of smaller paths, which together give the coverage level required.

Typically, the statement, branch and path coverage is not conducted by first analysing the program. It is more typical to apply some test cases, such as those proposed by black-box testing techniques, analyse the statement, branch and path coverage achieved, and then analyse the program to deduce additional test cases required using a white-box approach.

Sufficiency of test cases is questionable using these coverage techniques. For example, haven’t covered when the array doesn’t have a value which is equal to the key, and didn’t consider when the array is empty (i.e. Bot > Top). Didn’t cover loops in detail.
5.3 Loop Testing
Loops easily lead to programs in which it is not feasible to test all paths. A combinatorial explosion of paths are required whenever loops are introduced. Refer to example in Section 1.5.
To decide what paths of a loop to test there a number of guidelines.

5.3.1 Basis Path Testing
Basis paths provide mechanism to tackle path testing involving loops
Zero-path
  Represents short circuit of the loop
One-path
  Represent a loop in which only one iteration is performed

Basis paths represents atomic components of all paths. All possible paths are combinations and sequences of basis paths.
In testing loops based on basis paths, the tester should aim to exercise each basis path with at least one test. Both zero-path and one-path basis paths should be covered.
Combinations and sequences of basis paths do not need to be exercised by a test. They are assumed to be covered by combinations and sequences of already tested basis paths.

5.3.1.1 Basis Path Example
Again looking at our binary search program:
The basis paths for this program consist of the following:

**Zero-paths**
- B, C

**One-paths**
- B, C, D, J
- B, C, E, F, H, J
- B, C, E, G, H, J

Tests would be developed to exercise these paths. This would follow a similar approach as to designing tests to provide statement, branch or path coverage.

### 5.3.1.2 Basis Path Testing Issues

Basis path testing doesn’t consider:

- Proper loop termination (e.g. reaching maximum value). The paths considered need only consider one pass through the loop. This may miss problems to do with proper loop termination.
- Switching between conditions used in each iteration. When passing from one iteration of the loop to the next, problems introduced through changing the conditions between the iteration may be missed.

### 5.3.2 Beizer’s Loop Tests [Bei95a]

Beizer [Bei95a] proposes a number of other loop iteration test categories:

- **Bypass**: any value that causes loop to be exited immediately
- **Once**: values that cause the loop to be executed exactly once
- **Twice**: values that cause the loop to be executed exactly twice
- **Typical**: a typical number of iterations
- **Max**: the maximum number of allowed iterations
- **Max + 1**: one more than the maximum allowed
- **Max - 1**: one less than the maximum allowed
- **Min**: the minimum number of iterations required
- **Min + 1**: one more than the minimum required
- **Min - 1**: one less than the minimum required
- **Null**: one with a null or empty value for number of iterations
- **Negative**: one with a negative value for number of iterations

Some cases may overlap, or not apply to each loop, e.g. Bypass with Min/Null/Negative, etc.
5.4 Branch Condition Testing

Identifies individual boolean operands within decisions. These decisions are typically conditions used within if-then-else or while (loop) statements.

Tests exercise individual and combinations of boolean operand values

Example, and if-then-else statement:

```
if A or (B and C) then
  do_something
else
  do_something_else
```

In this case the decision is A or (B and C). The boolean operands are A, B and C.

5.4.1 Branch Condition Coverage

Branch condition coverage has a set of tests which have tested each operand for both its TRUE and FALSE outcome.

For the example decision above, A or (B and C):

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Weaknesses

- Can often be achieved without exercising both decision outcomes
  In the above example, the outcome of the decision is TRUE in both cases.
- Can often be achieved with just two cases.
  For example:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

5.4.2 Branch Condition Combination Coverage

Branch condition combination coverage looks at covering all combinations of TRUE and FALSE values for all operands.

For the example decision above, A or (B and C):

<table>
<thead>
<tr>
<th>Test Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>4</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>5</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>6</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>7</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
5. White Box Testing

5.4.3 Modified Condition Decision Coverage

Pragmatic compromise that requires fewer test cases than branch condition combination.

Used widely in avionics software. Required by aviation standard RTCA/DO-178B.

Intended to show that each boolean operand can independently affect the decision outcome. Each operand is changed between values true and false, while keeping the other operands some selected fixed value. The outcome should be shown to be different for the changed operand. This may require searching for the fixed values to assign to the other operands to make this happen.

For example, decision outcome = A or (B and C)

Changing the value of operand A, and keeping B and C the same:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>A2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Changing the value of operand B, and keeping A and C the same:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>B2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Changing the value of operand C, and keeping A and B the same:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>C2</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

100% coverage requires:

- minimum of n+1 test cases
5. White Box Testing

- maximum of 2n test cases

Using example above, minimal tests: A1 & B1 same, B2 & C1 same

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (A1, B1)</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2 (A2)</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>3 (B2, C1)</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>4 (C2)</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

5.4.4 Condition Coverage Issues

In program source code, the placement of boolean conditions may be outside the decision point. For example:

\[
\text{FLAG ::= A or (B and C)}
\]

\[
\text{if FLAG then \ldots}
\]

In this case, a variation is to test for all boolean expressions. For above this means testing for boolean expressions:

- A or (B and C)
- FLAG

Some compilers introduce optimisation that may make it difficult to show coverage for all operands. For example, C and C++ have short-circuit operators `&&` and `||` that do not work exactly the same as a conventional `and` and `or` (they may skip analysing the second operand).

Other decision structures may introduce difficulties. For example, case and switch statements.
5.5 White-Box Techniques Conclusions

White-box testing is useful for analysing particular paths in programs to exercise them with test cases.

In most cases white-box testing gives rise to a large number of test cases. This is because the tests are to cover a large number of execution paths through the code.

White-box testing is typically used in conjunction with black-box techniques. After applying black-box tests, analysis is made of white-box test coverage levels. This shows up areas in which black-box testing may have been inadequate, basically parts of the program that were not executed by black-box tests. Additional test cases may then be proposed to boost the level of coverage.

The techniques can be applied to specification which have similar characteristics of program flow and decisions. For instance, statement, branch and path coverage could be applied to flow charts or transaction flow diagrams, or decision coverage could be applied to specification truth tables.

5.5.1 Pros and Cons of White-Box Testing

Advantages

- Easier to identify decisions, and equivalence classes that are being used in the logic than with black-box testing.
- More objective than black-box testing, different people deriving test cases will arrive at corresponding results.
- Automated tools available to propose paths to test and to monitor coverage as the code is tested.

Disadvantages

- Incorrect code gives rise to tests that do not conform to expected operation as well.
- Can't detect wrong function being implemented, or missing functions or code.
- Need to have a good understanding of the code to understand the meaning of each test path through the code.
- Some paths are difficult to visit.
- Doesn't consider interfacing issues.
- Need access to the code.
- Impossible to test spaghetti code or pacincko code as there are excessive numbers of paths.
- Problems with multiple processes, threads, interrupts, event processing.
- Dynamically changing data and code is difficult to analyse.
- Only practical at low levels, for whole systems the number of paths through a whole program is incredibly large.
5.6 Complexity Metrics

Many complexity metrics can be used to provide a basis to determine how thoroughly a program should be tested.

The assumption used in complexity metrics is how difficult will it be to develop or analyse a portion of a program. Such assumptions have a close correlation to the number of defects that will be attributed to that part of the program. The more complex it is, the more defects you could expect to find.

Typically complexity metrics are based on combination of size of the program and complexity of the logic used to control the program.

Complexity metrics should be used as a guide to those areas of the system that should be scrutinised more thoroughly (apply more testing).

Complexity metrics can be used as an input for risk-based testing. High complexity would lead us to assume higher likelihood of failure. By breaking down a system to programs we can use the programs complexity as a component of the likelihood factor used in calculating risk levels.

In defining the risk level, the tester still needs to assess the impact of the failures for each program, as well as the likelihood of the defect turning into a failure. When considering likelihood, are there other external events before things really go wrong, for instance, is the program in question only used in rare circumstances.

Higher complexity also leads to a higher chance of finding errors in the program, as they are more likely to exist. This means that testing in high complexity areas will typically have higher defect discovery productivity.

The following sections go through a number of complexity metrics used in industry practice.

You may develop your own complexity metrics based on your own experience. For instance, in Year 2000 testing, you may count the number of date processing routines used by each function. Then when conducting your Year 2000 tests this information will tell you which function may have greater use of date processing.

5.7 Lines of Code

Simple metric: count number of lines of code in program and use count as measure of complexity.

Scientific studies [Lip82a]:
- Small programs have error rate of 1.3% to 1.8%
- Large programs have error rate increasing from 2.7% to 3.2%

Effectiveness
- Not too accurate based on error assumptions
- Better than a sheer guess

Variations
- What to include? Comments, declarations
- Weigh the listing, punched-hole metrics (sarcastic analysis)
5. White Box Testing

5.7.1 McCabe’s Complexity Metrics
McCabe’s metric counts the number of decision statements in a program (and adds one)
If module A has 6 if-then-else statements then it has complexity of 7
Complexity is additive. Module A has complexity of 7 and module B has complexity of 9, their combination is complexity of 16
Case statements
• complexity of one less than number of choices
• conversion to if-then-else statements
Refinements
• Myers’[Mye79a] proposal for complex decisions, each sub-decision is a decision in its own right, e.g. (not EOF) AND (Count >= 20), has 2 separate decisions rather than 1
Assumptions
• Simple to count
• Relates to the number of paths through a program
Scientific studies
• 23% of routines with complexity > 10 account for 53% of errors; routines of complexity > 10 had 21% more errors per lines of code than those < 10 [Wal79a]
• More scrutiny to programs with complexity greater than 10
Effectiveness
• Better than lines of code
• Doesn’t distinguish between different types of control flow, e.g. CASE statement, loops, if statement nesting

5.7.2 Halstead’s Metrics
Halstead’s metric is based on the use of operators (e.g. keywords) and operands (e.g. variable names, data base objects) in a program
Vocabulary
\[ n_1 = \text{number of distinct operators in program} \]
\[ n_2 = \text{number of distinct operands in program} \]
Halstead program length: \[ H = n_1 \log_2 n_1 + n_2 \log_2 n_2 \]
Length
\[ N_1 = \text{program operator count} \]
\[ N_2 = \text{program operand count} \]
Bug prediction
\[ B = (N_1 + N_2) \frac{\log_2 (n_1 + n_2)}{3000} \]
Example
- accesses 75 data-base objects a total of 1300 times
- uses 150 operators a total of 1200 times
- $B = \frac{(1300 + 1200) \log_2 (75 + 150)}{3000} = 6.5$ bugs

Also predictions for time and effort

Scientific studies
- Shown to be twice as good as lines of code
- Not improved by augmenting with lines of code or McCabe's complexity
6. Incident Tracking

- life cycle of an incident
- incident reports
- incident tracking systems
- incident status reporting
- incident analysis
6.1 Purpose of Incident Tracking

- Mechanism for identifying and recording potential defects in systems under test
- Ensuring that problems and defects are addressed appropriately and in a timely manner
- Provide mechanisms to monitor defects and discover areas in which improvements could be made for a better quality product

6.1.1 What is an “Incident”

Any event or unexpected behaviour of a system under test that requires further investigation

Sometimes called:
- Defect
- Error
- Problem
- Anomaly
- Opportunity!!

6.2 Incident Tracking Activities

6.2.1 Process
### 6.2.2 Incident Report

Fundamental object in incident tracking

**Example from [Kan93a]**

<table>
<thead>
<tr>
<th>COMPANY’S NAME</th>
<th>CONFIDENTIAL</th>
<th>PROBLEM REPORT # ____</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM _______</td>
<td>RELEASE ____</td>
<td>VERSION ____</td>
</tr>
</tbody>
</table>

**REPORT TYPE (1-6)**

| 1 - Coding error | 4 – Documentation |
| 2 - Design issue  | 5 – Hardware      |
| 3 - Suggestion   | 6 – Query         |

**SEVERITY (1-3)**

| 1 – Fatal | 2 – Serious | 3 – Minor |

**ATTACHMENTS (Y/N)**

If yes, describe

**PROBLEM SUMMARY**

__________________________________________________________________________

**CAN YOU REPRODUCE THE PROBLEM (Y/N)?**

**PROBLEM AND HOW TO REPRODUCE IT**

__________________________________________________________________________

__________________________________________________________________________

**SUGGESTED FIX (optional)**

__________________________________________________________________________

__________________________________________________________________________

**REPORTED BY** __________________________  **DATE __ / __ / __**

**ITEMS BELOW ARE FOR USE ONLY BY THE DEVELOPMENT TEAM**

**FUNCTIONAL AREA** __________________________  **ASSIGNED TO** __________________________

**COMMENTS**

__________________________________________________________________________

__________________________________________________________________________

**STATUS (1-2)**

| 1 - Open | 2 - Closed |

**PRIORITY (1-5)**

**RESOLUTION (1-9)**

| 1 - Pending | 2 - Fixed | 3 - Irreproducible |
| 4 - Deferred | 5 - As designed | 6 - Can't be fixed |
| 7 - Withdrawn by reporter | 8 - Need more info | 9 - Disagree with suggestion |

**RESOLUTION VERSION**

**RESOLVED BY**  **DATE __ / __ / __**

**RESOLUTION TESTED BY**  **DATE __ / __ / __**

**TREAT AS DEFERRED (Y/N)**
### 6.2.3 Resolution Outcomes for Incident Reports

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pending</td>
<td>Incident yet to be resolved</td>
</tr>
<tr>
<td>Fixed</td>
<td>When the incident has been eliminated</td>
</tr>
<tr>
<td>Irreproducible</td>
<td>When the incident cannot be recreated</td>
</tr>
<tr>
<td>Deferred</td>
<td>Incident is acknowledged, but not to be fixed until the next release</td>
</tr>
<tr>
<td>As designed</td>
<td>The incident is not an error, but reflects the intended operation of the system</td>
</tr>
<tr>
<td>Withdrawn by originator</td>
<td>The originator feels that the incident report should never have been raised.</td>
</tr>
<tr>
<td>Need more info</td>
<td>The originator is requested to provide more info</td>
</tr>
<tr>
<td>Disagree with suggestion</td>
<td>No change to the design will be made</td>
</tr>
<tr>
<td>Duplicate</td>
<td>Closes incident as being duplicated by another incident report. Cross-references should be provided.</td>
</tr>
</tbody>
</table>

### 6.2.4 Activities

![Activity Diagram]

- Identification
- Assessment
- Resolution
- Validation
- Management
- Analysis
6.2.4.1 Identification
While under test or during general use the system may exhibit behaviour that is unexpected

- the user (hereafter called the originator) can raise an incident report
- formally initiates an assessment of the unexpected behaviour

6.2.4.2 Assessment
Each incident raised is assessed

- to determine whether it is to be considered a defect that requires resolution
- to determine the impact of unexpected behaviour

6.2.4.3 Resolution
The system is modified to remove the defect and resolve the incident

May involve modification to:

- code
- data / configuration
- documentation

6.2.4.4 Validation
After modification the repaired system undergoes a retest:

- to check that the incident was resolved
- to check that other problems related to the incident have been considered in the resolution
- to check that the resolution has not unexpectedly affected other parts of the system

6.2.4.5 Management
Overall generation and assessment of reports summarising collective defect data

The purposes of the report include:

- Ensuring that all defects are addressed as soon as practicable.
- Estimating the progress of defect resolution
- Estimating effectiveness of the testing process
- Estimating the reliability of aspects of the system under test

6.2.4.6 Analysis
Each defect is analysed to determine

- how the defect may have originally been inserted into the system
• why the defect may have been previously overlooked

Purpose is to assess defects to determine whether the development process could be improved to help prevent similar defects being introduced in the future.
6.3 Incident Life Cycles
The following sections describe stages in the life cycle of an incident report (derived from [Kan93a]).

6.3.1 Report the incident
Write incident report
- Find incident
- Investigate in enough detail to provide clear description
Enter incident report
- Entered directly; or
- Entered by someone else

6.3.2 Send incident report to project manager
Incident reports made available for project manager review
Normal actions
1. Evaluate
2. Set priority
3. Add comments
4. Allocate to programmer(s)
Alternatives actions
- Return to originator for more detail
- Mark as Deferred, As-Designed or Irreproducible

6.3.3 Send incident report to programmers
Programmer investigates incident with a goal to:
- Fix the cause of the incident
- Explain why the incident shouldn’t be fixed - may ask for more info

6.3.4 When the incident is (allegedly) fixed
Programmer has marked the incident as fixed
Submit for retest
- If passes retest then confirm fixed
- If fails retest then set status to pending
- If other problems found in retest raise new problem reports and cross-reference
6.3.5 Deferrals and the appeal process
The incident is recognised, but choice made not to fix yet
Explanation for deferral provided in comments
Referral review meetings should be conducted to review referred problems prior to releases
  • Include marketing, tech support, etc.

6.3.6 Incidents that aren’t being addressed
Some incident reports get lost, other deliberately set aside, some assigned a low priority and forgotten
Circulate summary reports regularly to remind of not-yet-resolved problems
Review summary reports in detail prior to release

6.4 Incident Status Reports
Reports state
  • how many incidents found through the project
  • how many still outstanding
  • how many deferred compared to how many fixed
  • how many found by testers and how many by others
  • progress each week as well as cumulative total

Help managers evaluate:
  • quality of programming effort
  • current reliability of the product
  • effectiveness of the testing effort
  • rate of discovery of new incidents compared to rate of fixing incidents (to project likely completion date)
6.4.1 Incidents Raised (Count)

6.4.2 Incidents Raised - Trends

Rayleigh Curve
- Expected shape
- Tests being executed steadily
- Tests spread evenly across system
- No changes during test
- Learning curve at start
- Slow continuous decline

Reverse Rayleigh Curve
- Problems in starting testing
- Little defects early on
  - Premature confidence in system
- Lots of defects later
  - Panic to meet deadline

Squiggle Curve
- Simultaneous testing and modification
- Later increases are incidents resulting from modifications introduced
6. Incident Tracking

Late Increase
- Near end of schedule
- Motivation by testers to find extra problems prior to release

6.4.3 Incident Raised - Variable Effort

6.4.4 Actual Project Data
6.4.5 Incidents by Testing Type

6.4.6 Effectiveness by Function
6.4.7 Incident Closure

The closer the incidents fixed curve to the incidents raised curve the more efficiently defects are being fixed after being raised

- Sufficient resources assigned to debugging and maintenance
- Effective debugging and maintainability of the code

Actual Project Data
## 6.4.8 Summary Reports

Weekly summary of new incident reports

### Weekly Report - New Incidents

**17/6/98**

**System:** Test Studio  
**Release:** 1.0.3

<table>
<thead>
<tr>
<th>Component</th>
<th>TS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>moderate</td>
<td>TS108</td>
</tr>
<tr>
<td>Configuration</td>
<td>moderate</td>
<td>TS106</td>
</tr>
<tr>
<td>Generic Editor</td>
<td>major</td>
<td>TS103</td>
</tr>
<tr>
<td></td>
<td>critical</td>
<td>TS104</td>
</tr>
<tr>
<td></td>
<td>minor</td>
<td>TS105</td>
</tr>
<tr>
<td></td>
<td>major</td>
<td>TS110</td>
</tr>
<tr>
<td>Generic Reviewer</td>
<td>critical</td>
<td>TS107</td>
</tr>
<tr>
<td>Class Query Viewer</td>
<td>major</td>
<td>TS109</td>
</tr>
</tbody>
</table>

### Weekly Report - Incident Status

**17/6/98**

**System:** Test Studio  
**Release:** 1.0.3  
**Last report dated:** 10/6/98

<table>
<thead>
<tr>
<th>Outstanding Incidents</th>
<th>Now</th>
<th>Last Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>34</td>
<td>44</td>
</tr>
<tr>
<td>Major</td>
<td>88</td>
<td>109</td>
</tr>
<tr>
<td>Moderate</td>
<td>79</td>
<td>82</td>
</tr>
<tr>
<td>Minor</td>
<td>55</td>
<td>71</td>
</tr>
</tbody>
</table>

- No. found since last report: 47
- No. fixed since last report: 55
- No. deferred since last report: 2
- Total no. deferred: 45
### Weekly summary of outstanding incidents

**Weekly Report - Outstanding Incidents**  
17/6/98

**System:** Test Studio  
**Release:** 1.0.3

<table>
<thead>
<tr>
<th>Severity</th>
<th>Date</th>
<th>Incident ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>3/6/98</td>
<td>TS104</td>
<td>Unsaved fields when creating new incident report</td>
</tr>
<tr>
<td></td>
<td>13/6/98</td>
<td>TS107</td>
<td>Review operations don't change state</td>
</tr>
<tr>
<td>Major</td>
<td>3/6/98</td>
<td>TS103</td>
<td>80 character string types only supports 48 characters</td>
</tr>
<tr>
<td></td>
<td>15/6/98</td>
<td>TS109</td>
<td>Performance of class query viewers are slow when loaded up</td>
</tr>
<tr>
<td></td>
<td>17/6/98</td>
<td>TS110</td>
<td>Slow response on cross-reference queries under load</td>
</tr>
<tr>
<td>Moderate</td>
<td>15/6/98</td>
<td>TS108</td>
<td>Windows keep getting located further down and right of the last time and eventually start sliding off page.</td>
</tr>
<tr>
<td></td>
<td>12/6/98</td>
<td>TS106</td>
<td>Type definitions project specific</td>
</tr>
<tr>
<td>Minor</td>
<td>8/6/98</td>
<td>TS105</td>
<td>Object Field Description Undisplayed</td>
</tr>
</tbody>
</table>

**End of test cycle**

### Test Cycle Complete

**System:** Test Studio  
**Release:** 1.0.3

<table>
<thead>
<tr>
<th>Severity</th>
<th>Unresolved incidents before this version</th>
<th>New Incidents</th>
<th>Resolved Incidents</th>
<th>Remaining Incidents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>26</td>
<td>18</td>
<td>28</td>
<td>16</td>
</tr>
<tr>
<td>Major</td>
<td>65</td>
<td>49</td>
<td>62</td>
<td>52</td>
</tr>
<tr>
<td>Moderate</td>
<td>55</td>
<td>38</td>
<td>42</td>
<td>51</td>
</tr>
<tr>
<td>Minor</td>
<td>41</td>
<td>28</td>
<td>18</td>
<td>51</td>
</tr>
</tbody>
</table>

**Resolution in this cycle:**

- Fixed: 91
- Irreproducible: 12
- Deferred: 26
- Other: 21
6.4.9 Incident Analysis

Defect type (sample)

Phase introduced (sample)
Root cause (sample)

Detection deferral (sample)
6.5 Incident Tracking System Functional Requirements

- Support activities involved in capturing, assessing, resolving and analysing defects.
- Maintain a defect repository, and record history of defect resolution.
- Provide capability to query and browse through defects
- Report defect statistics and compile evidence for defect predictions
- Provide capability to add notes, comments and attachments to defects
- Support coordination of defect activities amongst different project staff
- Link defects to other project attributes such as system versions, test cases, subcomponents, developers, testers, system documentation, etc.
- Integrate with version control tools
- Provide external interfaces, such as email, internet, etc.
7. Automated Test Tools

- Types of tools
- Demo of test tool
- Pros and cons of test automation
- Short term V's long-term efficiency
- Test tool adoption
- Regression testing
7. Automated Test Tools

7.1 Introduction

What is Automated Testing

- Tools used to streamline the process and documentation
- Tools used to streamline the execution of tests
- Tools used to streamline the gathering of metrics

Offers significant improvements in productivity

- in managing the processes
- in minimising human involvement
- in replicating work

Automated testing is the fastest growing area of the test industry
7.2 Types of Testing Tools

Test automation tools
- Record tests and replays them to compare results
- Script together test procedures

Test management tools
- plan, design and document tests

Test analysis tools
- Analyse code and specifications for test coverage, path and complexity analysis

Others
- Syntax checkers, code scanning tools, debuggers, test data generation tools, etc.
7.3 Automated Test Execution

What is a test?
- Comparison of actual results against expected results

Test automation supports
- recording test cases
- rerunning test cases
- comparing results
- reporting outcome

Test Capture / Playback most common
7.4 Test Capture/Playback Tools

7.4.1 Demo
Two types of tool
- Host system (green screen)
  Will demo using TALC2000 from Tallecom Software
- Client-server system (GUI)
  Will demo using SQA Robot from Rational Software

Aim
- To illustrate types of support
- To demonstrate test construction concepts
  - Test creation
  - Test validations
  - Test execution
  - Test results
  - Test composition
  - Regression test
7.5 Why Test Automation

- Tight deadline
- Tests repeated many times
- Testing can be run 24 hours a day
  - Not just business hours
- Faster and more accurate
  - Eliminates human tester inconsistency
- Documented results
  - Full audit trail
  - Logging of errors
- Tests scripts will be an asset
  - Not just an expense
- Quicker test turn-around
- Enables Incremental process
7.6 Regression Testing

- Testing date related changes is not enough
  - Up to 60% of all system modifications inadvertently introduce new defects
    (Software Engineering Institute – Bellcore Study)
    For every 6 lines of code modified a new error will be introduced
- Must consider impact on functionality for each change
  - May not be related to change made
- Manual Testing limits regression testing carried out
  - Only about 10% of test cases will be rerun
- Automation shown to improve cost and effectiveness of retest
7.7 What To Automate

- Easy cases first
- More complex cases as skills build
- Some tests will remain manual
- Always be practical

7.7.1 Splitting Test Sets
Look at building two sets of tests

- Sanity tests
  - Run before the full testing is started to determine if the system is ready to go through full testing
  - Run quickly, less than an hour
  - Checks that no unexpected surprises are encountered
- Full testing
  - Complete set of tests
  - May take many hours or even days to complete

7.7.2 Key Automation Requirements
[Pet95a]

Maintainability
- Resilience to changes in system.
  - Robust with respect to window position movements, button position, etc.
- Anticipate user interface changes
  - Modularise so that changed parts can be swapped easily

Reliability
- False positives (intolerable)
- False negatives (minor amount acceptable)

Error Recovery
- Unattended testing requires handling of errors that leave system in unexpected state
- Restore system to base state and continue testing
  - Test case independence - not dependent on earlier tests
7.7.3 Control of Test Environment

Automated testing doesn't support variation in system operation, after all that's what the tools are designed to detect.

Important to have good control of test data and application

- Isolate system under test from other systems that cause unexpected behaviour
- Control input databases and files to avoid variability in test data
  - Avoid use of production databases and files as these change from day to day. New records added and may prevent previous tests from running
7.8 Efficiency of Automated Testing

Improvements in productivity are gained through

Duplication
- A test can be duplicated/parameterised for different cases
- Useful for load and stress testing to duplicate large quantity of tests

Repetition
- The same test is repeated many times
- Useful for regression testing to ensure modifications have not adversely affected the system

7.8.1 Manual vs. Automated Effort

Many studies have shown varying relationships between effort

<table>
<thead>
<tr>
<th>Activity</th>
<th>Manual Test Time (time units)</th>
<th>Automated Test Time (time units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Test Preparation</td>
<td>1</td>
<td>2 - 3</td>
</tr>
<tr>
<td>Test Execution</td>
<td>1</td>
<td>.1 - .2</td>
</tr>
<tr>
<td>Test Maintenance</td>
<td>1</td>
<td>2 - 3</td>
</tr>
<tr>
<td>Re-Test</td>
<td>1</td>
<td>.1 - .2</td>
</tr>
<tr>
<td>Test Results Checking</td>
<td>1</td>
<td>.1 - .2</td>
</tr>
<tr>
<td>Test Documentation</td>
<td>1</td>
<td>.1 - .2</td>
</tr>
</tbody>
</table>

Effort Scenario
- One 30 minute test rerun 3 times

<table>
<thead>
<tr>
<th>Activity</th>
<th>Manual Test Time (minutes)</th>
<th>Automated Test Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Test Preparation</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>Test Execution (1st time)</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>Test Maintenance</td>
<td>5 x 3</td>
<td>15 x 3</td>
</tr>
<tr>
<td>Re-Test (3 times)</td>
<td>30 x 3</td>
<td>6 x 3</td>
</tr>
<tr>
<td>Test Results Checking</td>
<td>10 x 4</td>
<td>5 x 4</td>
</tr>
<tr>
<td>(all 4 tests)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Documentation</td>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>215</td>
<td>157</td>
</tr>
</tbody>
</table>

Net saving is 58 minutes or 27%.

Quality Assurance Institute study [QAQ95a]
- 1750 test cases finding 700 errors
## Test Steps

<table>
<thead>
<tr>
<th>Test Steps</th>
<th>Manual Testing</th>
<th>Automated Testing</th>
<th>Percent Improvement with Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Plan Development</td>
<td>32</td>
<td>40</td>
<td>-25%</td>
</tr>
<tr>
<td>Test Case Development</td>
<td>262</td>
<td>117</td>
<td>55%</td>
</tr>
<tr>
<td>Test Execution</td>
<td>466</td>
<td>23</td>
<td>95%</td>
</tr>
<tr>
<td>Test Results Analysis</td>
<td>117</td>
<td>58</td>
<td>50%</td>
</tr>
<tr>
<td>Error Status/Correction Monitoring</td>
<td>117</td>
<td>23</td>
<td>80%</td>
</tr>
<tr>
<td>Report Creation</td>
<td>96</td>
<td>16</td>
<td>83%</td>
</tr>
<tr>
<td>Total Duration (Hours)</td>
<td>1090</td>
<td>277</td>
<td>75%</td>
</tr>
</tbody>
</table>
7.9 Pros and Cons of Automated Testing

Advantages

- Enables more thorough testing
  - Improved regression testing
- Time to develop new/improved tests
  - Shorten duration for testing
  - Improve time to market
- Improve productivity of scarce resources
  - Testers are hard to find and expensive
  - Improves morale
- Reduces tester error and oversights
  - Not detecting wrong output
  - Sloppy fingers in keystrokes
- Provides an detailed test log and audit trail
  - Records test execution precisely for debugging process

Disadvantages

- Time and effort to setup tests initially
  - Must be run 2 - 3 times to recover costs
  - Pressures to drop automation when deadline is close
- Maintenance of test cases
  - Modification is system behaviour must be matched
  - Avoid volatile features
- Learning curve
  - May be several weeks before proficiency is achieved
- Doesn’t eliminate manual testing
  - Typical to not automate more than 50 - 75% of test cases (depending on environment)
- Easy to focus on tests that are simple to automate
  - Avoid coverage of difficult high risk cases
- Investment cost
  - Can be as high as $30000 for a single user license
- Technical limitations
  - Specific to particular environment, e.g. Unix, PC
  - Application type, e.g. GUI, text only, networking, web
  - Changes in timing and responses
• Ease-of-use
  – Some are programmer oriented and not suited for end-users

• Lack of stability and support
  – Vendors not quick in responding to bugs
  – Lack of experienced users in the marketplace
7.10 Adoption Process

Many tools are bought and left on the shelf. They become very expensive pieces of office furniture.

Steps towards adoption:

1. Assess needs and get high-level management commitment
   - Make clear pros and cons

2. Contact vendors and assess high-level capability of tools
   - Tools are specific to different platforms or languages
   - Some tools require re-linking libraries

3. Assess costs and budget commitment
   - Tools range in price from $500 to $30000 per seat
   - Training $2000 per day up (some $5000 per day)

4. Evaluate products for own environment
   - Define specific technical criteria for own requirements
   - Common for 14, 30, 90 day evaluation licenses

5. Contact reference sites
   - Quite often they are selected by the vendor and will be favourable
   - Try requests on email lists and news groups

6. Negotiate investment
   - Factor in total price of licenses, maintenance and support, training, consulting and contract staff

7. Initial purchase and pilot
   - Assess use on single project
   - Establish processes, performance and estimation indicators
   - Removed from critical path of project

8. Broader use and additional investment
7.11 Automation Readiness

Many organisations will leap to take advantage of productivity improvements without being first being ready. However, before tools can be successful, a level of testing maturity must be in place in your organisation.

Make sure your organisation avoids the pitfalls by ensuring the following are satisfied:

- Identified a need for automated testing
  - Can you cost justify the effort to move to automation
  - Do you have metrics that indicate the level of testing already being undertaken, and to gauge any improvements
  - Are you testers complaining of repetitiveness of executing tests manually

- Organisational support
  - Is there sufficient resource and budget to purchase tools, undertake training, and to conduct an evaluation
  - Supportive and understanding management essential

- Have established testing expertise already in place
  - Test automation doesn’t tell you what tests to run
    Still have to design tests, doesn’t tell you what tests to implement
  - Requires process that is stable, well-defined and well-managed
  - Can’t turn chaos into order
  - Need supporting processes in place, e.g. configuration management of test data

- Invest in sufficient training to proficiently use the tool
  - Often the testers fight with their own lack of knowledge
  - Budgeting for ongoing training is also important for new staff

- Run a pilot project
  - Avoid immediately employing in critical path
  - Give testers a chance to experiment
  - Productivity not evident straight-away (lessons to be learned)

- Stability of application
  - Functionality is frequently changing and it is infeasible to create tests that do not require constant maintenance

- Specialised skills and support available
  - Existing testers may not have programming skill required to control some tools
  - Contractors and consultants with experience available

- Avoid fear of organisational change
- Many developers resent change in practices as they fear personal/career impact
- Frequently undermine tool adequacy
8. Test Management

♦ composition of the test group
♦ estimating effort and cost
♦ scheduling activity and monitoring progress
♦ allocating resources
♦ test completion criteria
8.1 Composition of the Test Group

The testing group is comprised of:

**Test manager**
- To manage the testing project

**Application end-users**
- Provide domain expertise

**Software developers**
- To create harnesses, stubs, simulators, oracles, etc

**Test tool / environment experts**
- Specific tool and environment support for the test group

**Test consultants**
- To advise on strategy, methodology

8.1.1 Independent Testing

Testing group is most effective when independent of the development group.

Focus of development is to deliver product on time and in budget.

Independent test group responsibility is to protect the organisation from releasing faulty software.

In many cases the independent test group can reduce the pressure on development to release when they are not ready, e.g. functions not stable yet

<table>
<thead>
<tr>
<th>Differences</th>
<th>Developer</th>
<th>Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding</td>
<td>Has a deep understanding of the system (perhaps already misinterpreting requirements)</td>
<td>Must spend significant time learning the system</td>
</tr>
<tr>
<td>Viewpoint</td>
<td>Will test primarily based on the implementation knowledge of the system (less likely to find requirements errors). i.e. looks at the testing based in what they have built (structural view)</td>
<td>Usually tests independent of the implementation structure supported. i.e. looks at the testing based on what must be built (functional view).</td>
</tr>
<tr>
<td>Drive</td>
<td>Driven by delivery schedule</td>
<td>Driven to break the system</td>
</tr>
</tbody>
</table>
8.1.2 Tester Morale

- Minimise conflict with developers
  - Describe problems in great detail
  - Avoid developer finding excuse to ignore problem
- Don’t just focus on getting tasks done to schedule
  - Praise quality work
- Develop a group culture that values testing as a professional activity
- Shield your group
  - Deal with problematic developers or managers personally rather than leaving the tester to take the blame
- Don’t let project manager beat your test group into tight schedules or unworkable approaches
8.2 Estimating Effort and Cost

8.2.1 Estimation methods
Boehm [Boe81a] discusses a number of software engineering estimation methods that apply equally to software testing:

Algorithmic models
- Mathematical function over a number of variables considered to be cost drivers
- Algorithms extracted from data analysis
  - May be simple as a graph mapping project function points against time spent testing

✓ objective, not influenced by emotion, calibrated on previous experience
✗ previous data not always representative of future projects, e.g. new application, new platform
✗ unable to deal with exceptional conditions, e.g. greater effectiveness of team

Expert judgement
- consult one or more experts, who, using their experience and understanding of the project, arrive at an estimate
- may use different estimation techniques
  - group consensus, delphi, wideband delphi

✓ can identify differences between past and current projects, and exceptional conditions
✗ subjective, suffers from biases, optimism, pessimism, unfamiliarity with actual project

Parkinsonian Estimation
- Work expands to fill the available volume - Parkinson's law (1957)

✓ in some cases, remarkably accurate - usually where estimate has left good deal of extra time and money
✗ generally inaccurate - tends to reinforce poor development practice
Price-To-Win Estimation

- Price that will win a competitive bid, or release before competitor

✗ Typical to run out of time and money before project complete - software compromised, long hours, frustration

- Estimation techniques have made it difficult for customers and developers to differentiate between legitimate and price-to-win estimates

Top-Down Estimation

- An overall estimate is derived for the project, using above methods
- The overall estimate is split up amongst the various components of the project

✔ Focuses at the system level and incorporates other hidden costs, e.g. documentation, configuration management

✗ Misses low level tasks likely to escalate costs

Bottom-Up Estimation

- Estimates are derived for lower-level components
- Costs are summed to achieve cost for entire project
- Complementary to top-down testing
- Requires development of a detailed work plan (Work Breakdown Structure)

✔ Tends to focus just on the costs of the components, without hidden costs

✔ More balance in many estimates (win some - lose some)

✗ More effort required to complete estimate

8.3 Scheduling Activity

Define the tasks to be undertaken

- Define the inputs and outputs
- Define the techniques and tools to be used

Identify dependencies

- What predecessor tasks are to be complete prior to commencement
- Define the entry and exit criteria
Identify resources required

- Who is responsible for coordinating the activity
- What other groups must participate
- Are any special skills and training required
- What quantity is required
- Are there limitations on availability

Identify critical paths through the schedule

- Mark tasks on critical path for careful monitoring as they may cause schedule to slip

Analyse risks in schedule and propose contingencies

- Important to schedule with close synchronisation with developers
- Many development activities will impact the testing process
  - Late delivery of design documentation will slow test case design
  - Late delivery of code will delay test execution

Key scheduling objectives [KFN93a]:

- Provide predictability to the project manager
- Identify opportunities to pull in or protect the project schedule
- Be fair to your staff
- Maximise productivity
8.4 Monitoring Progress

Either
- estimate percent of tasks completed
- count number of predetermined milestones completed

You can determine average times for how long things take
- You can predict how long a task is likely to take

There is variations in how long the same task takes
- You can provide limits around your predictions

8.4.1 Test Measurements
Example measurements:
- Average number of cycles of testing
- Duration of the typical cycle of testing
- Bugs reported per tester-day
- Hours per test of a given type
- Average number of tests for a test plan of a given type
- Number of tests per bug
- Number of bugs per 1000 lines of code
- Number of tests executed

Careful records must be kept

Noise in the data

8.4.2 Alternative Approaches

Collect as much data as possible and then try to determine useful metrics
- Overwhelmed by data
- not enough time to collect data that might not be used
Decide what you want to do with metrics and then collect the data to define the metric

- Goal, Question, Metric paradigm
  - **Goal** - define the ultimate purpose of the metric
  - **Question** - define what sort of metric questions would need to be answered
  - **Metric** - actual measurements and calculations that are made

### 8.4.3 Performance Metrics

Many managers plan to use metrics to assess individual tester performance:

- No of tests run
- No of bugs discovered, either high / low

Many of these figures are influenced by external factors, e.g. developers

- If the software is delivered late, little time is left to achieve optimum proficiency and run tests and find a large number of bugs

Can be influenced by testers

- A large number of insignificant bugs may be raised
8.5 Allocating Resources

Allocate the majority of testing resources to those parts of the system that have greatest risk.

Allocate testing resources based on estimates of where most errors are likely to be found, using:

- lines of code, module size, function size
- complexity metrics (McCabe’s and Halstead’s metrics)
- analysis of modules where most errors have been found
- emphasis on new modules and functions (not reused modules)
- designers knowledge of weakest part of the system
8.6 **Test Completion Criteria**[KR23]

Any combination of some or all of the following:

- All black-box test cases run
- White-box test coverage targets met
- Rate of fault discovery goes below a target value
- Target percentage of all faults in the system are found
- Measured reliability of the system achieves its target value
- Test phase time and resources are exhausted
  - Most common applied in real-world
Further Reading


Presents a critical analysis of capture/playback style of test automation. Presents a very negative view that would easily (and perhaps unnecessarily) scare people off, but does present some pitfalls to be avoided.


Very useful reading defining a set of test design techniques and associated coverage metrics. Also provides guidelines for the applications of the techniques with small worked examples.


One of Beizer's earlier books. Provides lots of valuable information. A recommended general testing reference.
Further Reading


Provides comprehensive analysis of different techniques that can be used for black-box testing. The book considers the techniques at quite a technical level and presents a mathematical basis for the way the techniques are applied. As usual, Beizer's books are quite well structured and this book is quite useable as a technical reference book.


An excellent text that describes many facets of testing client/server systems. Provides good overviews of commercial testing tools that are used for varying kinds of testing that can be applied. Also discussed many related issues, such as testing SQL interfaces and security.


Quite valuable article describing a practical approach for deriving test cases from use cases.


Most comprehensive course notes I've ever encountered. Very thorough explanation of concepts, detailed examples and lots of industry statistics.


Insightful summary of how test oracles can be used to calculate the expected outcome of tests.
   Industry standard the proposes the structure of test documents, such as test plans, test design specifications, test cases, test procedures and test logs.

   Describes the application of Configuration Management to the management of software engineering projects.

   Comprehensive guidelines on testing software systems. Strong practical focus.

http://www.kaner.com/coverage.htm
   Discusses legal aspects of what constitutes negligence and how it relates to software testing and the use of coverage metrics to measure completeness of testing.


   Detailed explanation of why exhaustive testing is not possible. Presented with concrete examples that can be used to convince management.


   Early classic text on software testing. Presents quite a few of the black-box and white-box testing techniques.
Further Reading


Comprehensive explanation of techniques that can be used for system testing. Presents quite rigorous approaches to testing, accompanied by many checklists and data that could be used directly within projects.

[Pet95a] B. Pettichord, Success with Test Automation, BMC Software (unsure of source - contact bret_pettichord@bmc.com)

Describes a number of principles for test automation based on author's experience at BMC software. Lots of practical suggestions.


[Ros96a] L. L. Rose, Getting the Most Out of an Automated Test Tool, Rational Technical Paper

Presents some useful advice on how automated test tools can be applied to different types of testing, e.g. unit testing, stress testing, etc.

Index

Note: This index is still not quite complete, but I have included interim results for your convenience.

A
acceptance testing ......................... 29
alpha testing................................. 30

B
backup/recovery testing.................... 30
beta testing .................................. 30
black-box testing ......................... 27

c
controllability .............................. 80
cost relative .................................. 15
testing ......................................... 13
coverage
black box ...................................... 74
functional ...................................... 27
risk ............................................ 20
structural ...................................... 27
customer ........................................ 29

d
defect ........................................ 11, 18
cost ........................................... 4, 13
detection ...................................... 2
impact ........................................ 2
prevention ...................................... 2
zero ............................................. 4
development lifecycle

defect prevention ......................... 25
parallel testing .......................... 24
traditional .................................. 24
V-model ....................................... 26
dynamic testing ............................ 27

e
error seeding ............................... 32

f
failure .......................................... 18
formal inspections ......................... 27
functional requirements .................. 27
functional testing .......................... 27

h
heuristic
true ........................................ 75

i
IEEE Standards
test plan ................................... 41
installation testing ......................... 30
integration
bottom-up .................................. 28
top-down ..................................... 28
integration testing ........................ 28
interface testing ............................ 28
Further Reading

L
load testing ........................................... 29

M
module testing ........................................ 28
mutation testing ....................................... 32

N
negative test cases .................................. 58

O
oracles ................................................. 75
consistent ............................................. 75
heuristic ............................................... 77
sampling ............................................... 75, 76
ttrue ..................................................... 75, 76

P
parallel testing ...................................... 31
peer review ............................................ 27
performance testing ................................. 29
postcondition ......................................... 56
precondition .......................................... 56
predictability ........................................ 80

Q
quality assurance ................................... 7
quality control ....................................... 7

R
regression testing ................................... 31
reliability testing .................................... 30
repeatability ......................................... 81
risk ...................................................... 18
effort allocation ...................................... 20
matrix ................................................... 18
prioritise ............................................. 18
project ................................................ 12
risk-based testing ................................... 18

S
security testing ...................................... 30
simulator ............................................. 81
static testing ......................................... 27
stress testing .......................................... 29
structural testing ..................................... 27
syntax checkers ....................................... 27
system testing ........................................ 29

T
test
activities ............................................... 39
driver .................................................... 81
environments .......................................... 80
factors ................................................... 34, 35
harness ............................................... 81
oracles ................................................ 75, 82
stub ...................................................... 81
test plan ............................................... 40–41, 40
objectives ............................................. 40
test plan
IEEE Standards ..................................... 41
test strategy cube .................................... 33
test teams
independent .......................................... 4
testability ............................................. 80
testers
career .................................................. 21
personality ........................................... 21
skills ................................................... 21
testing
acceptance ........................................... 29
alpha ................................................... 30
backup/recovery ..................................... 30
beta ..................................................... 30
black-box ............................................. 27
dynamic ............................................... 27
exhaustive ............................................ 8
functional ............................................ 27
installation ........................................... 30
integration ............................................ 28
interface ............................................... 28
load ...................................................... 29
module ................................................. 28
mutation ............................................. 32
parallel ............................................... 31
performance ......................................... 29
purpose ............................................... 7
regression ............................................ 31
reliability ............................................ 30
security ............................................... 30
static ................................................... 27
strategy ............................................... 33
stress ................................................... 29
structural ............................................. 27
Subsystem ............................................ 28
system ................................................ 29
test plan
transaction flow ..................................... 59
unit ..................................................... 28
usability ............................................. 29
white-box ............................................ 27
trace matrix .......................................... 27
transaction flow ...................................... 59
transaction flow testing ......................... 59
type checkers ........................................ 27

U
unit testing .......................................... 28
usability testing ..................................... 29

W
walkthroughs ......................................... 27
white-box testing ..................................... 27
Check "relative cost per development stage" [Mar83a]

Slides to be created

Indicate that test strategy is documented in a variety of documents, e.g. test strategy, master test plan, v & v plan. Provide examples, provide outline of structures. Identify what things need to be identified, see scope from Compaq work. Provide template from Compaq work.

Also check out US DoD 2167A or 498

Update slides

Include section on risk based testing
Here’s my hot topic of the month. I am perplexed by the lack of useful information on what we might call ‘Risk-Based Testing’. I've seen many references to this so-called discipline, but am disappointed by what I've found in the books on my shelf that give some space to risk. None of the old faithful testing books help me at all. The books below give risk a pretty good mention. Pretty good books, but even these don't help me as a tester.

These books are:

1. Managing Risk, Elaine Hall, Addison Wesley ISBN 0 201 25592 8
5. Assessment and Control of Software Risks, Capers Jones, Yourdon Press, ISBN 0 13 741406 4
6. Against the Gods: the Remarkable Story of Risk, Peter L Bernstein, John Wiley

I found the last book to be the most interesting, but its not about software - never mind - I learned something about probability and the stock market...

Perry's book comes closest to describing a method for using risk to determine which tests to perform.

I don't think anyone has adequately described the relationship between risk and testing, or offered a sensible methodology of using risk to assist in test strategy, specification or analysis.

Fact is, all that has been said by the most advanced practitioners is:

a. risks are bad.
b. testing reduces risk (allegedly)
c. we should test more (whatever that means) where the risk of failure (whatever that means) is highest (whatever that means) and vice versa (whatever that means).

Personally, I don't think this is a very difficult concept. It takes as long to learn it as say it.

BUT IS THAT ALL THERE IS TO IT? DOES THIS HELP A PRACTITIONER DO THEIR JOB?

What is the problem?
---------------------

The first problem, is that INTUITIVELY, all testers do RBT (risk based testing) already. Given this outline of RBT, any fool can dream up things that could go wrong with software and brainstorm a list of 'hazards' and then we translate them
To define index

To update slides

Include in powerpoint

To define index

Provide sample of stubs and harnesses, etc.

Insert into slide

To define index

Review explanation of graphs

Review section for repetition

To define index

Include info on load testing tools

Refine list to include those covered by marick web pages

To define index

What are the number of testers to developers

Include sample estimates of effort involved. Provide sample breakdown, e.g. provide test plan. Indicate how times can vary depending on case

Expand explanation of completion criteria. Define release criteria.

Update index