Software Testing

Part 1 Fundamentals Of Testing

1.1) Why Test?

Software is an integral part of life from Business applications to consumer products. Most people have experienced software that did not work as expected.

Software that does not work correctly leads to many problems among which are:

- loss of money
- loss of time
- damage to reputation
- in some cases can cause injury or even death

The main problems that software failure causes to the Blaise team is **damage to our reputation** and the **loss of time** repairing and issuing patches...

Rigorous testing of systems and documentation can help to **reduce the risk** of problems occurring during operation and contribute to the quality of the software system, if the defects are found and corrected before the system is released.

Causes of defects:

A human being can make an **error** (mistake), which produces a **defect** (fault,bug) in the program, or in a document. If a defect in code is executed, the system may fail to do what it is supposed to (or do something it shouldn't) causing a **failure**.

Factors which cause defects:

Humans are fallible
Time pressure
complex code
complexity of infrastructure
changing technologies
system interactions
environmental conditions

Error ---> Defect ---> Failure

An **error** is a human action producing an incorrect result

When a programmer makes an error they introduce a fault to their program

Errors are inevitable in any complex activity

A **defect** is a fault or bug [a manifestation of human error] in the software

Defects can be cause by requirements, design or coding errors

Defects are discovered either by inspecting code or by inferring their existence from software failures

A **Failure** is a deviation of the software from its expected service [or delivery]

Failure occurs when the software does the wrong thing

Failure usually happens when a program executes with a set of inputs that cause a defect

How does a developer mitigate errors ?

- Concentrate on the task
- Developer testing [as you go]
- Understand requirements fully [seek clarification otherwise\ never assume]
- Code reviews
- Test early and often on code being developed

How much Testing is enough ?

There are an infinite number of tests we could apply. Software is never perfect.

Objective coverage measures can be used:

A degree of Risk Management should be employed [Appendix 1]
Test design techniques give an objective target [Part 2]
Testing standards - can impose a high level of testing [Part 3]

In most System development activity **time** is the limiting factor. **Minimizing the risks** inherent is the key to software testing.

Risk = (Impact * probability)

Experience tells us:

bugs are sociable [they tend to cluster]
some parts of the system will be relatively bug free
bug fixing and maintenance are also error prone [changes often can cause other faults]

What about bugs we don't find ?

If they are not in Business critical parts of the system and do not impact on the customer - would they care? If they are not in system critical parts of the

system - they should be very low impact. If they are in system critical parts they should be very obscure...

1.2) What is Testing

7 Principles of testing have been suggested over the past 40 years and offer general guidelines common for all testing:

1) Testing shows the presence of defects

Testing can show defects that are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but it is not proof of 100% correctness

2) Exhaustive testing is impossible

Testing everything [all inputs and preconditions] is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts

3) Early testing

To minimize defects later on - testing activities should be started as early as possible and should be performed throughout the development life cycle - It should be focused on defined objects

4) Defect clustering

Testing efforts should be focused proportionally to the expected and later observed defect density of modules.

5) Pesticide paradox

The principle that bugs become immune to pesticide - test cases need to be reviews and revised regularly, new and different tests need to be devised to exercise different parts of the software.

6) Testing is context dependent

Testing should be done differently in different contexts. Planning, testing, documentation should be tailored for what is being tested.

Testing should prove [in documentation] that an action has worked. Input data, action performed and outcome shows be referenced

7) Absence of error fallacy

Declaring that a test has unearthed no errors is not the same as declaring the software is 'error free'. In order to ensure that adequate software testing procedure are carried out in every situation, testers should assume that all software contains some defects.

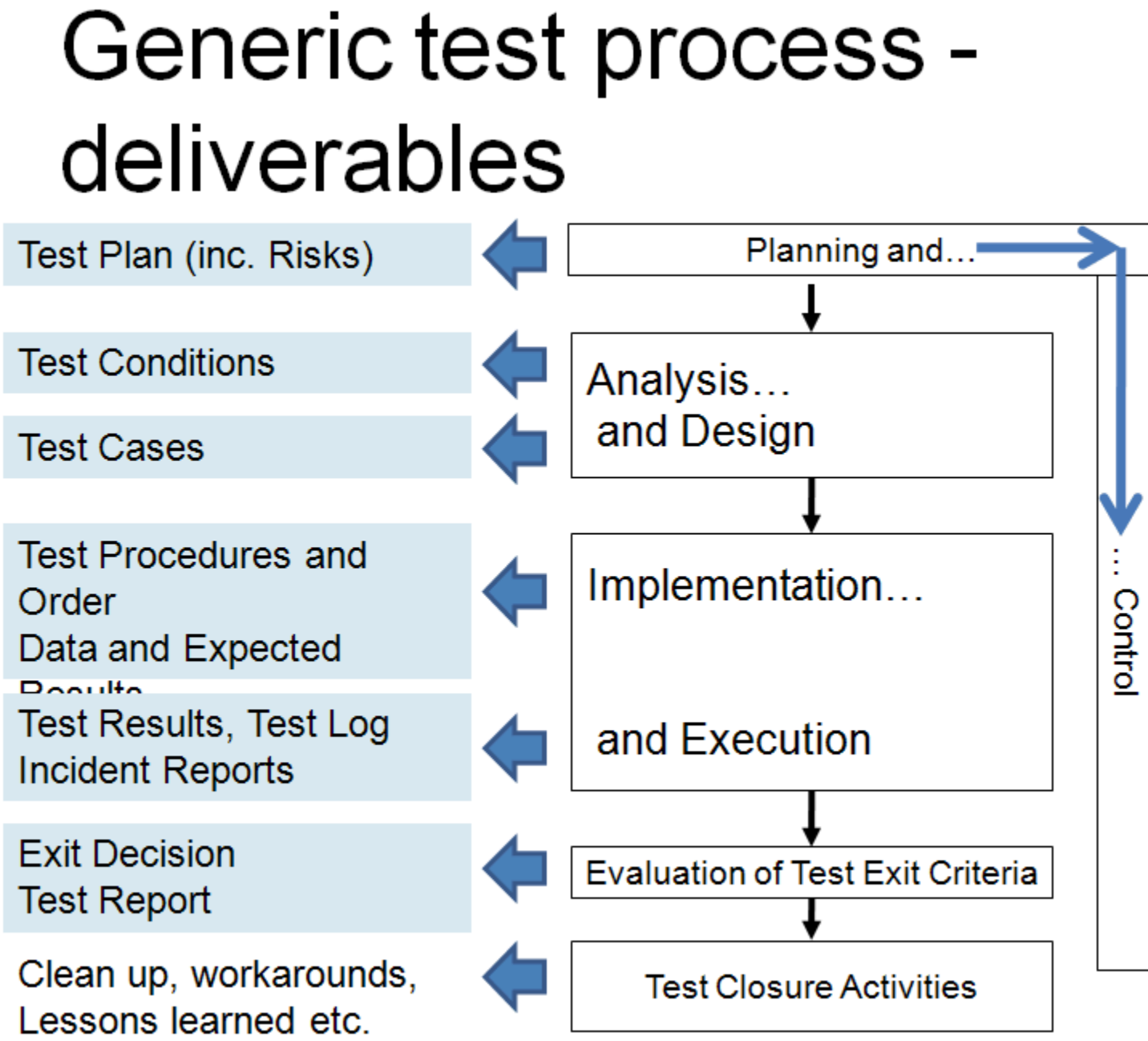
Summary

Software testing good practice is an essential part of ensuring the quality of IT products. While software testing cannot guarantee that the software contains no errors, it does contribute significantly to the identification and reduction of faults, improving the likelihood that the software implementation will succeed.

Testing is not Debugging & **Testing is not** just running tests

- Testing shows failures caused by defects, Debugging is a development [only] activity that identifies the cause of a defect, repairs the code and checks that the defect has been fixed correctly

1.3) Test Management



Planning and Control:

Deliverable - High level Test Plan

Who: Test leader

Test planning identifies, at a high level, the scope, approach and dependencies:

the components to be tested
additional infrastructure needed to test the component
approach required for the test design
completion and exit criteria

Test planning should perform the following major tasks:

Determine the scope and risks, identify the objectives
Determine the test approach [tailored to system: techniques, items, coverage, testware]
Determine resources
Implement test policy/strategy
Schedule test analysis and design
Schedule test implementation, execution and evaluation
Determine exit criteria

Test Control means measuring, monitoring and analyzing results. Initiating corrective actions if needed. It means monitoring progress
and coverage throughout the Testing process

Test Analysis & Design:

Deliverables: Analysis - Test Conditions, Design - Test Cases

Who: Test leader

Establish test basis (baselines such as requirements, architecture, design, interfaces)

Evaluate the testability of the requirements

If baselines are testable - test conditions are defined [based on analysis of test items, specification, behaviour and structure]

Prepare test inventory [prioritizing by risks and test conditions]

Specify test cases from the test conditions

Design [or specify] test environment

Establish traceability between test baseline and test cases

Test Implementation:

Deliverables: Test Procedures, Data, Expected Results, Order of tests

Who: Test leader in collaboration with Tester

Create all materials required to execute tests. Environment should be defined here also. Group tests into clusters or cycles for efficient testing

Test Execution:

Deliverables: Test logs, Incident Reports

Who: Tester

Check all is ready.

Tests follow scripts [as defined]

Verify actual meets expected and update incident report if not

Log progress as we go through tests

Raising Incidents:

Report test failures or any discrepancies as incidents

Cause should be established

Fixes made

Second iteration of testing can begin

Re-Tests are performed on failures to confirm the defect has been successfully removed

Regression testing need not be performed at this point as there should be at least 3 iterations through the test plan
to ensure maximum coverage.

Evaluation of Test exit criteria and Closure:

Deliverables: Exit Decision, Test Report, Clean ups, workarounds, Lesson learned

Who: Test leader

Test execution should be examined against defined objectives [at planning] to establish exit criteria

Exit criteria may involve time pressures [some faults may be acceptable for this release ?]

Test logs checked against exit criteria

Assess if more testing is needed

Test Summary report should be written for all stakeholders

Summary report should provide evidence to make a decision on release

Closure activities should collect all data to consolidate experience, problems & lessons learned for future tests

Part 2 Testing in the Software LifeCycle

2.1) Test Levels

For each of the test levels the following can be identified:

- Generic Objectives
- Work products for deriving test cases (ie the test basis),
- Test object (what is being tested)
- Typical defects and failures (historical)
- Test harness and tool support

2.1.1 Component Testing

Test Basis: Component Requirements, Detailed design, Code

Test Objects: Components, Programs, Data conversion/migration programmes, Database modules

Component testing (also known as Unit, module or program testing) searches for defects, and verifies the functioning of software modules, programs, Objects, classes etc that are separately testable.

Typically, component testing occurs with access to the code under test in the developer environment using a debugger. Defects are normally fixed as they are found without formally managing the defects.

One approach to component testing is to prepare and automate test cases before coding. This approach is highly iterative and is based on cycles of developing test cases, then building and integrating small pieces of code and executing the component tests correcting any issues and iterating until they pass.

This is the only level of testing that Code is used as a test basis [excepting Static Tests and reviews].

2.1.2 Integration Testing

Test Basis: System design, Architecture, Workflows, Use cases

Test Objects: SubSystems, Database implementation, Interfaces, Infrastructure

Integration testing tests interfaces between components, interactions with different parts of a system.

There is more than one level of Integration testing and it may be carried out on test objects of varying size:

- Component integration testing tests the interaction between software components and is done after component testing
- System integration testing tests the interactions between the different systems of between the hardware and software - System Integration tests are usually done after System Tests.

The greater the scope of Integration, the more difficult it becomes to isolate defects to a specific component or systems. In order to ease fault isolation and detect defects early, Integration should be incremental rather than "big bang".

Testers should be concentrating solely on the Integration itself. For example if they are integration module a with module b they should be interested

in testing the communication between the modules, not the functionality of each module as that should be completed in Component testing.

2.1.3 System Testing

Test Basis: System and Software requirements, Use cases, Functional specification, Risk analysis report

Typical Test Objects: System & user manuals, System configuration and configuration data

System Tests are concerned with the behaviour of the whole System/product. Scope of these tests should be clearly stated in Master test plan.

Testing environment should the same as or as close as possible to the final production environment. This will minimize the risk of environment-specific failures.

System tests include tests on:

- Established and measured risks
- Requirement specifications
- Business process
- Use cases
- System resources
- System behaviour
- Interaction with operating system

System tests should investigate functional and non-functional requirements of the system and data Quality characteristics.

2.1.4 Acceptance Testing

Test Basis: User requirements, Use cases, Business Requirements , Risk analysis report

Typical Test Objects: Forms, reports, configuration data, User processes and procedures

Acceptance testing is often the responsibility of the customer [user of the system]. The goal of Acceptance testing is to establish confidence in the system.

Finding defects is not the main focus of this type of testing - It is more of an assessment of the systems readiness for deployment...

Acceptance can occur at various times in the Development life-cycle: Acceptance testing on the usability of a component or testing of a new functional enhancement which could come before the System tests.

2.2) Test Types

2.2.1 Functional (Black box) Testing

The functions are 'what' the system does.

Functional tests are based on the functions and features and their interoperability with specific systems and may be performed at all test levels.

Useful models may be used for performing functional tests: Process flows; state transition models; specification testing

Security testing is an example of functional testing - investigating how the systems handles firewalls and virus checkers etc.

Treats software under test as a Black-box ie no knowledge should be required of internals of the system or component is required.

2.2.2 Non-Functional Testing

Non-functional tests are 'how' the system works.

It is where the behavioural aspects of the system are tested and in most cases uses black-box design techniques to accomplish that. Non-functional tests may

be performed at all levels.

Non functional testing includes (but is not limited to):

- Performance tests - response times at producing outputs or performing actions etc
- Load tests
- Stress tests
- Usability tests
- Maintainability tests
- Portability tests

These tests can be referenced to a quality model such as the one defined in 'Software Engineering - Software Product Quality (ISO 9126)

Non-Functional testing may highlight requirement difficulties that need to be addressed - eg to clarify assumptions or highlight necessary requirements not not stated originally.

2.2.3 Structural (White box) Testing

Structural (White box) testing may be performed at all levels. These tests are best performed using Specification based test techniques (Boundary test, equivalence partitioning, Decision tables, use cases etc). The tests are generally performed on the code but also tests the architecture of the system or integration between components (such as a call hierarchy).. Baseline documents are used to determine expected results.

2.2.4 Re-Testing and Regression Testing

After a defect is detected and fixed, the software should be re-tested to confirm that the original defect has been removed.

Regression testing is the repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of the change(s).

These defects may be in the software being tested, or in another related or unrelated software component. Regression testing is performed when the software or its environment is changed.

Regression testing may be performed at all test levels, and included functional, non-functional and structural testing.

2.3) Maintenance Testing

Once deployed a software system is often in service for years or decades. During this time the system, its data or its environment are often corrected, changed or extended. The planning of releases in advance is crucial for successful maintenance testing. A distinction must be made between 'hot fixes' and planned releases. Maintenance testing is done on an existing system and is triggered by modifications, migration or retirement of the software or system.

Modifications include enhancements (release based), corrective or emergency changes, changes of environment, such as a new operating system or patches to correct exposed or discovered vulnerabilities of the system.

In addition to testing what has been changed, maintenance testing includes regression testing to parts of the system that have not been changed. The scope of maintenance testing is related to the risk of the change, the size of the existing system and to the size of the change. Depending on the changes, maintenance testing may be done at all levels of the system and for all test types. Determining how the existing system may be affected by the changes is called impact analysis, and it is used to decide how much regression testing to do.

Maintenance testing can be difficult if specifications are out of date or missing, or testers with domain knowledge are not available.

2.4) Static Testing

Unlike Dynamic testing which requires the execution of software, static testing relies on the manual examination and the analysis of the code or project documentation without the execution of the code.

These reviews are a way of testing software work products before dynamic testing begins. Defects detected during reviews early in the lifecycle (eg defects found in requirements) are much cheaper to remove that those found by running tests on executed code.

The main activity is to examine a work product and make comments about it.

Any software work product can be reviewed:

- Requirement specifications
- Design specifications
- Code
- Test plans
- Test specifications, test cases & test scripts
- User guides, help guides or web pages

Benefits of properly planned reviews include:

- early detection and correction
- development productivity improvements
- reduced development time consumption
- reduced testing cost and time
- Improved team communication

Reviews, Static analysis and dynamic testing have the same objective - identifying defects. They are complementary. Compared to dynamic testing, static

techniques find the causes of failures (defects) rather than the failures themselves.

Defects that are easier to find in reviews than in dynamic testing are:

- deviations from standards
- requirement defects
- design defects
- Code efficiency [or redundancy]

Types of review performed in Static Testing:

1) Informal review

No formal prcoess. Technical leader and coder(s) review code or code designs. Relies on skills/knowledge of reviewers however it is an inexpensive way to get some benefits

2) Walkthrough

Meeting led by author. peer group participation. Can go for informal to very formal. Author circulates document to reviewers prior to meeting. Optional scribe [who is not the author] to document comments to be given to the author. Main purpose is learning, giving the team more understanding and also finding defects

3) Technical Review

Can be performed as a peer review without management input but must include technial experts. Ideally led by a moderator (not the author). Pre-meeting preparation by reviewers. can use checklists.Output is a report on findings with a verdict on whether the product meets it requirements and recommendations related to findings. Purpose is to discuss product make decisions, evaluate alternatives, find defects and solve technical problems, check conformance to specifications.

4) Inspection

Led by a trained moderator who is not the author. It is a formal process based on checklists and includes metrics gathering. Pre-meeting preparation is essential, this included reading all source documentation. An inspection report is produced which contains a list of defects found and metrics that can be used to aid process improvement

A typical formal review should include the following activities:

1. Planning
   - Define review criteria
   - Select the personnel
   - Allocate roles
   - Establish what parts of document(s) are up for review
2. Kick-off
   - Distribute documents
   - Explain objectives, process and documents to the participants
3. Individual preparation
   - read documents
   - Note potential defects, prepare questions and comments
4. Have meeting [Examination/evaluation/recording of results]
   - make decisions about found defects
   - document results or minutes
5. Rework
   - fix found defects
   - record updated status of defects
6. Follow-up
   - Checking that defects have been addressed
   - Gather metrics

Roles and Responsibilities

Manager: Decides on the execution of a review, allocates time in project schedules and after, determines if review objectives have been met

Moderator: Person who leads the review of the documents. chairs the meeting and follows up on defects and gathers metrics

Author: writer or person with chief responsibility for the document

Reviewers: Individuals with technical skills who after the necessary preparation can identify and describe defects in the product under review.

Scribe: minute taker and person who records all issues and problem points identified during the meeting

Success Factors for a Review:

- Review has clearly defined objectives

- The correct people are involved

- Defects found are welcomed and expressed objectively [the focus should be on high quality results]

- Review must be conducted in an atmosphere of trust

- Emphasis on learning and process improvement

Part 3 Categories of Test Design Techniques

3.1) Specification or Black box techniques

3.1.1 Equivalence Partitioning

Example:

Test condition: valid inputs are integers in the range 100 to 999 inclusive

These are the test cases for Equivalence partition tests:

- Valid partition 100 to 999 inclusive eg 450
- Non valid partitions: Integer less than 100 eg 55
  Integer greater than 999 eg 1055
  Decimal numbers eg 1.5
  Non-numeric characters eg. woe

While we cannot test for every valid partition (Keying every integer between 100 - 999 inclusive) we should do at least one test with a valid value.

So there are 5 test cases (**+**) derived from Equivalence partition tests on the above test condition

Example of Equivalence tests to be performed:

Decimal/Alpha 55 100 450 999 1055

Invalid partition Valid Partition Invalid Partition

3.1.2 Boundary value Analysis

Example:

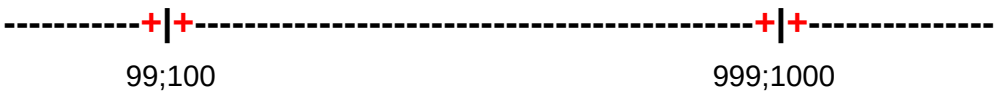
This is the test condition: valid input: integers in the range 100 to 999 inclusive

Valid Boundary - 100 and 999

Invalid Boundary - 99 and 1000

So there are 4 test cases (+) derived from the test condition in the boundary test

Example of Boundary tests:



3.1.3 Decision Tables

Specifications define the conditions under which a function operates. The conditions can get quite complex so its important to assure ourselves that every combination of these conditions has been tested. So, we try to document all the decisions in order to explore all possible combinations.

A decision table lists all the input conditions and all the actions that arise from them.The conditions are structured into tables as rows and below the conditions are the resulting actions arise from the combination of true/false conditions in the top part of the table.

Example:

The Personal module is only to be asked of People aged 18 (Condition 1) or over, who are Students (Condition 2) or working (Condition 3) and Wave 5 cases only (Condition 4)

It might be coded as follows:

IF (Age >= 18) and (Wave = 5) and ((Job = Student) or (Job = Working)) Then

Ask Module

Else

Goto End

Endif

In order to achieve the greatest coverage of all possible conditions, the tester should draft and refine a table something like the following:

	Test cases for Entry into Personal module									
	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**	**Rule 8**	**Rule 9**	**Rule 10**
Conditions										
Aged >= 18	Yes	Yes	Yes	Yes	No	No	No	No	Yes	Yes
Students	No	Yes	Yes	Yes	Yes	No	No	No	No	Yes
Working	Yes	No	Yes	Yes	Yes	Yes	No	No	No	No
Wave 5	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No
Actions										
Ask Module	Yes	Yes	-	Yes	-	-	-	-	-	-
End Module	-	-	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes

Second reduction of table will remove duplicate tests (cases where respondent is not 18 or over)

	Test cases for Entry into Personal module						
	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**	**Rule 6**	**Rule 7**
Conditions							
Aged >= 18	Yes	Yes	Yes	Yes	No	Yes	Yes
Students	No	Yes	Yes	Yes	Yes	No	Yes
Working	Yes	No	Yes	Yes	Yes	No	No
Wave 5	Yes	Yes	No	Yes	Yes	No	No
Actions							
Ask Module	Yes	Yes	-	Yes	-	-	-
End Module	-	-	Yes	-	Yes	Yes	Yes

Third reduction removes all duplicate tests for Wave not = 5

	Test cases for Entry into Personal module				
	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	**Rule 5**
Conditions					
Aged >= 18	Yes	Yes	Yes	Yes	No
Students	No	Yes	Yes	Yes	Yes
Working	Yes	No	Yes	Yes	Yes
Wave 5	Yes	Yes	No	Yes	Yes
Actions					
Ask Module	Yes	Yes	-	Yes	-
End Module	-	-	Yes	-	Yes

So, Rules 1 - 5 in decision table above are test cases which should be performed by the tester to provide maximum coverage for all possible combinations of conditions.The Expected results of each test are the actions of each rule.

This technique is particularly useful in systems where combinations of input conditions produce various actions.

3.1.4 State transition Testing

This technique is more concerned with systems in which outputs are triggered by changes to the input conditions. It charts a change of 'state' and the transition

that triggered the change.

Example:

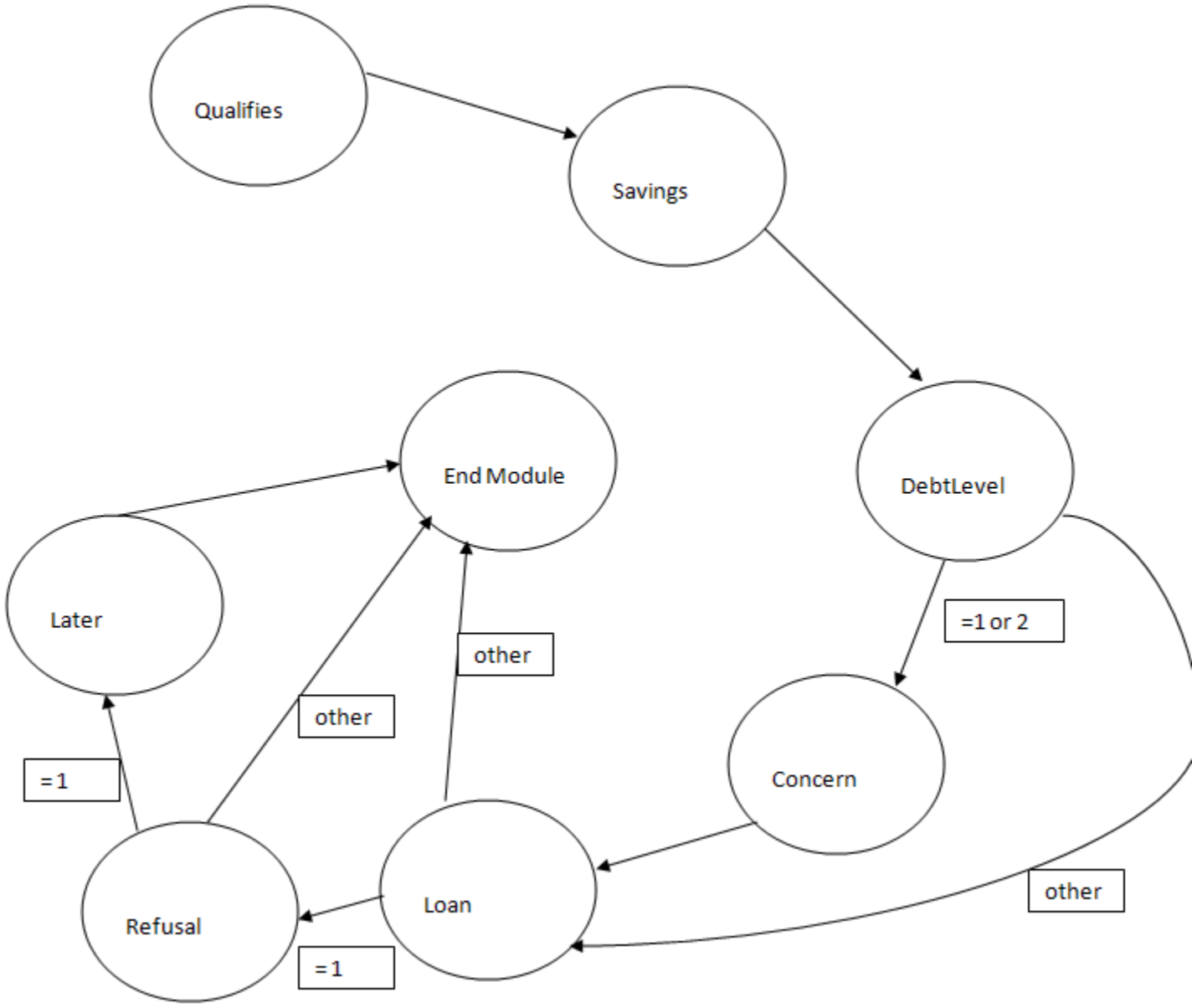
Requirements:

The Personal module is only to be asked of People aged 18 or over who are Students or working and at Wave 5. Once they **qualify** for the questionnaire they will
be asked a question on **Savings** then a question on their **DebtLevel** If they answer 1 or 2 to Debtlevel they will be asked the **Concern** question otherwise they should skip
to the **Loan** Question. If Loan = 1 they should go to **Refusal** question otherwise skip to End. If Refusal is 1 they should be asked **Later** otherwise skip to end. After
Later end module.

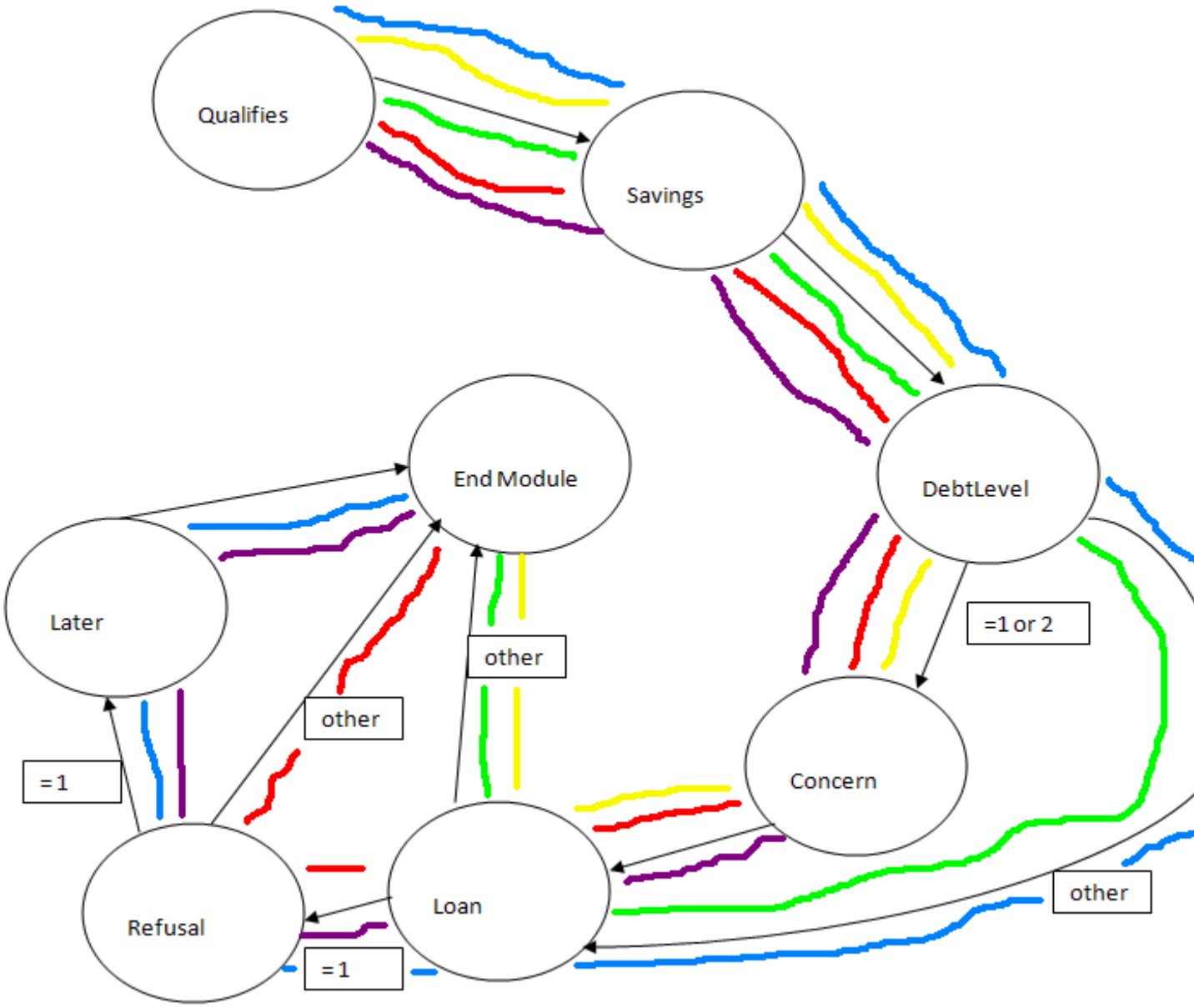
PseudoCode [based on Requirements]:

```
If (age is 18 or over) and (students or working = yes) and (wave = 5)
  Savings
  Debtlevel
  If Debtlevel = 1 or 2 then
    Concern
  endif
  Loan
  IF Loan = 1 then
     Refusal
     If Refusal = 1 then
       Later
     EndIf
  Endif
```

Can be Illustrated in a Transition diagram as follows:


Qualifies
Savings
End Module
DebtLevel
Later
=1 or 2
other
other
Concern
=1
Refusal
Loan
other
=1

The coloured lines represent all paths through the module which should be tested. Each colour is a test case:



From the routes through the Module we should be able to establish the Test cases to test for 100% coverage of the module

```
If (age is 18 or over) and (students or working = yes) and (wave = 5)
   Savings
   Debtlevel
   If Debtlevel = 1 or 2 then
      Concern
   endif
   Loan
   IF Loan = 1 then
      Refusal
      If Refusal = 1 then
         Later
      EndIf
   Endif
```

Test Cases:

```
Debtlevel = 3
loan = 1
Refusal = 1
Later
End

Debtlevel = 3
Loan = 1
Refusal = 2
end

DebtLevel = 1
Concern
Loan=1
Refusal=2
End
```

3.1.5 Use Case Testing

Use Cases specify functionality as business scenarios and process flows.They capture the individual interactions between 'actors' who represent the users and the system.

4.2) Structure based or White Box techniques:

This testing can be performed at the following levels

Component level: structure of a software component, ie statements, decisions, branches and even distinct paths

Integration level : Testing the paths of different components or modules

System level : Menu testing or a Business process.

It can utilize flow charts, control flow diagrams to visualize the alternatives for each decision.

4.2.1 Statement testing and coverage

Statement testing is aimed at exercising programming statements - ie executable statements that 'do' something. Statement coverage

requires you to exercise each executable statement at least ONCE.

```
DELCAR    RULES


YES       Wh_Ans_HS

YES       StrTimeHS.KEEP {KEEP statements put before IF/EMPTY condition & the question,         }
YES       StrDateHS.KEEP {then times are not reset to empty when/if Wh_Ans_HS values are changed}

YES       IF (Wh_Ans_HS <> EMPTY) AND (Wh_Ans_HS <> DK) AND (Wh_Ans_HS <> RF) THEN
YES           IF (StrTimeHS = EMPTY) AND (StrDateHS = EMPTY) THEN
YES               StrTimeHS := SYSTIME
YES               StrDateHS := SYSDATE
PUNCT           ENDIF
PUNCT       ENDIF

YES       IF (Wh_Ans_HS=No)  THEN
YES          No_Ans_H
PUNCT       ENDIF

DECLAR       NEWPAGE

DECLAR        CHECK
YES          (No_Ans_H <= HouseDetails.No_Pers) OR (No_Ans_H = 21) INVOLVING (No_Ans_H)
TEXT          "@RError:@R
           @/@sThere is no person on this line number.@s"

YES          (No_Ans_H <> QHRP.DVHRPNum) INVOLVING (No_Ans_H)
TEXT            "@RError:@R
           @/@sYou have enter the HRP's person number, if the HRP is available to answer the housing que

YES          IF (QthComp.PersNo[No_Ans_H].PersonBasic.AgeCalc <> EMPTY) THEN
DECLAR             SIGNAL
YES             (QthComp.PersNo[No_Ans_H].PersonBasic.AgeCalc  >15) INVOLVING (No_Ans_H)
TEXT             "@RWarning:@R
              @/@SThis person is less than 16 years of age.@S"
YES          ELSEIF (PersonDetailsTable.PersonID[No_Ans_H].Person.AgeCalc = EMPTY) THEN
DECLAR             SIGNAL
YES             (QthComp.PersNo[No_Ans_H].PersonBasic.AgeCalc >0) INVOLVING  (No_Ans_H)
TEXT             "@RWarning:@R
              @/@SThis person has not answered age and may be less than 16 years of age.@S"
PUNCT          ENDIF
```

Any code marked 'Yes' in the above example is a statement and needs to be tested. Declarations and punctuations can generally be ignored.

You achieve 100 percent statement coverage if your tests execute every statement (or line of code) in the program.

Procedure for Statement testing:

1. Identify all executable statements
2. Trace execution from first statement
3. For each decision encountered, choose the true outcome first; write down values for the variables that make the statement true
4. Are all statements covered? If not select values to reach the next uncovered statement in the code and continue covering statements as you proceed
5. Repeat 4 until all statements covered by at least one test case
6. For all test cases record the variable values required to force the execution path you take

4.2.2 Decision testing and coverage

Decision testing requires you to exercise all outcomes of the decision in the code.

Procedure for Decision testing:

Follow the procedure for statement testing and create a covering set of test cases

Identify all the decisions and mark the listing with a 'True' or 'False' depending on the outcome

For each test case - mark all outcomes

Select additional test values to reach next outcome and create a test case for each possible outcome

Some tasks of the Tester:

Review and contribute to Test Planning

Analyse, review and assess user requirements, specifications and models for testability

Set up the test environment [should mimic live environment]

Prepare and acquire test data

Implement the tests, execute and log the tests, evaluate the results and document the deviations from expected results

Measure performance of components and/or systems

Review tests performed by other team members

Appendix 1: Risk and Testing

Risk is a big factor in Test Management. If there were no risk of adverse future events in Software or Hardware development then there would be no need for testing.

Risk can be defined as the chance of an event, hazard, threat or situation occurring and its undesirable consequences.

In a project a test leader will use risk in two different ways: **project risks** and **product risks**. In both instances the calculation of the Risk is:

Level of Risk = Probability of the risk occurring * the impact if it did happen

Project Risks [This need not concern us too much as our focus is on the product]

Project Risks are identified and documented in all our Blaise Development plans. Risks in Development plans should always be monitored and

updated as required.

Project Risks always include:

Supplier issues:

- Failure of a third party to deliver
- Contractual or business issues

Organisational factors:

- Skills, training, staffing issues
- Communication issues
- Developmental and testing practices - [eg failure to follow up on issues]

Technical issues:

- Problems with requirements
- Environmental issues
- Quality of design, code, configuration data, test data and testing

Project risks should be identified during test planning and should be documented in something like the IEEE 829 test plan. A risk register should also be maintained

by the test leader.

Product Risks

Potential failure areas in software are known as product risks, as they are a risk to the quality of the product.

Examples of Product risk are:

- Failure-prone software delivered
- The potential that a defect could cause harm to an individual
- Poor software characteristics (functionality, security, reliability, usability, performance)
- Poor Data integrity or quality(data migration, data conversion problems etc)
- Software not performing as intended

The specific product risks in our Blaise Survey Instrument are:

- Routing [Logical errors, misinterpretation of specs, incorrect conditions]
- Text errors [Spelling errors, Wrong text or not matching Specs]
- Assignments, Calculations and Derived variables
- Variables [Types, Ranges]

Risks should be used to decide where to start testing in the software development lifecycle. For example: the risk of poor requirements could be mitigated by the use of formal

reviews as soon as the requirements have been received.

Product risks should provide information enabling the tester to make decision on how much testing should be carried out on specific components or systems. The more risk there

is, the more detailed and comprehensive the testing may be.

Mitigating product risks can sometimes involve non-test activites - re-assigning programmers, requirements rewrite etc.

In a risk based approach to development and testing the risks identified will determine the level of test coverage required, it should also prioritise testing in an attempt to find

critical defects as early as possible.

To ensure that the chance of a product failure is minimized, risk management activities provide a disciplined approach to:

- Assess (and reassess on a regular basis) what can go wrong (risks)
- Determine what risks are important to deal with
- Implement action to deal with those risks

In addition, testing may support the identification of new risks, may help determine what risks should be reduced and may lower uncertainty about risks.

Risk impacts can vary on our Blaise Systems:

High impact is anything error that would cause us to issue a patch or re-release.

Show stoppers such as:

- Routing into modules
- Menu problems

or anything that prevents the Interviewer carrying out their job correctly

Lower Impact might be text problems or other minor issues that it would be possible to provide a workaround for.

Appendix 2: Blaise Testing Standards

No.	Test level:	Who:	Documents	Test Techniques:	Comments
1.	**Requirements Testing**	Project Manager in collaboration with programmer and author of requirements	Risk Analysis (as part of Development Plan) Requirements Query Log	Static testing: Reviews, Walk-through	
2.	**Component Testing**	Programmer	Requirements Query log Test Log	Functional Testing (white box)	Test of code based on specifications.
3.	**Independent component Testing**	Developer other than programmer	Test Log	Functional Testing (white box) Structural testing: Boundary Testing, Equivalence Partitioning, Statement Testing, Decision tables	Blaise colleague(s) should test the component based on the specification
4.	**Integration testing**	Developers - perhaps overseen and documented by one person	Test Log	Non-Functional testing (Black box) Interaction between components of the system.	Questionnaire/Manipula scripts/User Interface
5.	**System Testing**	Performed and documented by a development team member	Test Log	Structural (Black box) Performance testing, load testing, stress testing, usability, Inputs/Outputs testing the behaviour of the system	Test environment must correspond to the live in order for tests to be meaningful. Use case or scenario testing methods are ideal here
6.	**Acceptance testing**	Business area or Interviewers		Non-Functional (Black box) Verification of delivery on specifications	Acceptance testing can be performed at any stage in the development life-cycle.