GUIDELINE

Software Testing

Code	07e-HD/PM/HDCV/FSOFT
Issue/revision	2/0
Effective date	05/08/2008

TABLE OF CONTENT

1 INTRODUCTION 4

1.1 Purpose 4

1.2 Application scope 4

1.3 Related documents 4

1.4 Definitions 4

2 TEST LIFECYCLE 6

2.1 The Development V-Model 7

2.2 Criteria to stop testing 8

3 TEST STAGES 8

3.1 Unit test 8

3.1.1 Objectives 8

3.1.2 Unit Test Process 8

3.1.3 Unit Test Plan 9

3.1.4 Unit Test design 10

3.1.5 Unit Test Case 10

3.1.6 Test Environment 10

3.1.7 Unit test tools 11

3.1.8 Unit test Measurement 11

3.2 Integration Test 12

3.2.1 Objectives 12

3.2.2 Integration test planning 12

3.2.3 Integration test environment 12

3.2.4 Integration test design 13

3.3 System test 13

3.3.1 Objectives 14

3.3.2 Test process 14

3.3.3 System Test planning 14

3.4 Acceptance test 15

4 TEST TYPE 16

4.1 Functional Test 16

4.2 User Interface Test 16

4.3 Data and Database Integrity Test 16

4.4 Performance Test 16
4.5 Security and Access Control Testing 16
4.6 Regression Test 17

5 TEST TECHNIQUES 18

5.1 Black Box Testing 18
5.1.1 Black box testing types 18
5.1.2 Black box testing techniques (for functional test) 18

5.2 White Box Testing 19
5.2.1 Basis Path Testing 19

6 TEST TOOLS 22

6.1 DMS for Defect management 22
6.2 Virtual Ruler and Eye Dropper 22
6.3 Rational Robot Functional test tool 22
6.4 Rational Performance test tool 22

7 TEST EVALUATION 23

7.1 Make "as-run" test procedure log during test execution 23
7.2 Analyse “as-run” test procedure log 23
7.3 Evaluate Requirements-based Test Coverage 24
7.4 Evaluate Code-based Test Coverage 25
7.5 Analyze Defects 27

8 MEASUREMENTS 28

1 INTRODUCTION

1.1 Purpose

The goal of this document is to:

- define typical stages, types and techniques of testing;
- guide project team in planning, monitoring, performing and evaluation test activities;
- Introduce some test tools

Test activities in software projects are typically performed by:

- Development team for Unit test;
- Test team for Integration and System test;
- Customer for Acceptance test.

1.2 Application scope

This document is applied for FSoft software projects

1.3 Related documents

No	Code	Name of documents
1.	05e-QT/PM/HDCV/FSOFT	Test process
2.	02-BM/PM/HDCV/FSOFT	Test plan template
3.	02ce-BM/PM/HDCV/FSOFT	Test case template

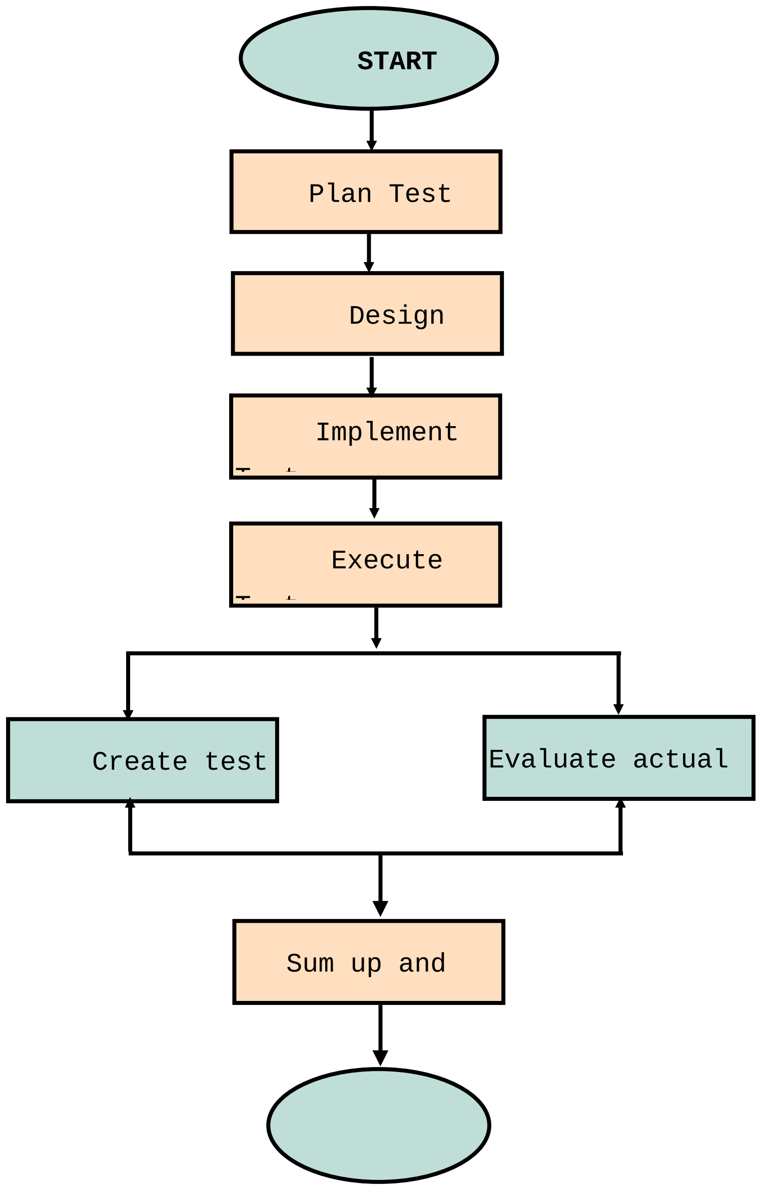
1.4 Definitions

Terms	Explanation
DMS	Defect Management System
PM	Project Manager
PTL	Project Technical Leader

Terms	Explanation
PP	Project Plan
QA	Quality Assurance Officer
QC	Quality Control
SEPG	Software engineering process group
SRS	Software Requirement Specification
TP	Test Plan
TC	Test Case
URS	User Requirements Specification document
UCL	Upper Control Limit
VP OPS	Vice President for Operation
WO	Work Order

2 TEST LIFECYCLE

Test lifecycle is presented in below flow:



Test model is the set of all test plans, test cases and test reports.

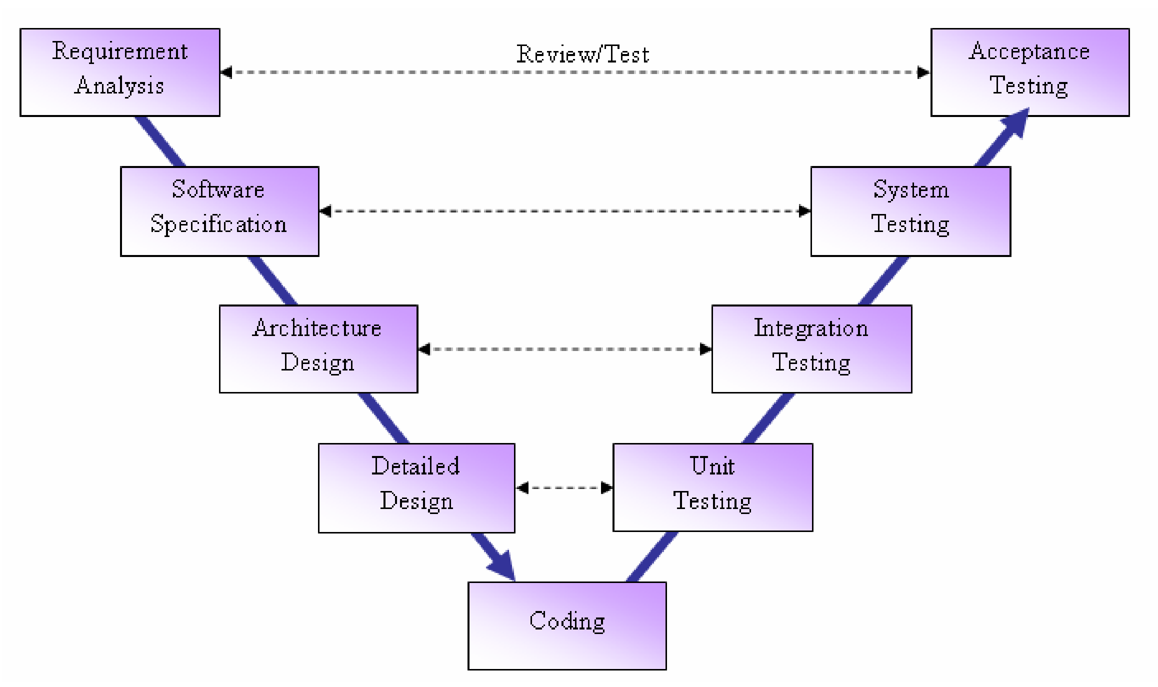
It is applied for each test stage.

Detailed steps and activities of testing are described in Process_Test (section 4).

In maintenance/enhancement projects, regression test shall be performed periodically. Moreover, regression test will be performed before the latest release. The regression test shall be planned in test plan or project plan.

2.1 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 Software Development V-Model

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 activities being conducted. This way as we come up the right side of the V the testing is ready to commence.

System test plan needs to be generated much earlier before the system test phase:

- System Test Plan is completed with Software Specification.
- Test case will be done when the Detail Design is finished.
- System Test execution starts after coding.

2.2 Criteria to stop testing

- Time runs out
- A certain number of defects found
- Require a certain test coverage
- Stop when testing becomes unproductive.

3 TEST STAGES

This section depicts in detail what should be done in different test stages:

- Unit test
- Integration test
- System test
- Acceptance test

3.1 Unit test

3.1.1 Objectives

Unit testing is applied at the level of each module of the system under test. The developers themselves usually do it, before the module is handed over for integration with other modules.

Unit test represents the lowest level component available for testing. It is very common at unit testing level to adopt the white box approach. The result of unit test is defects in DMS. Normally, defects of unit test should be more than 25% of total defects of a project.

For more detailed guideline, refer to Guideline_Unit Test.

3.1.2 Unit Test Process

Apart from the standard steps defined in Test process document and detailed in section "2. Test lifecycle", there are some tasks specific to unit testing:

1. Unit Test Planning
   - It may be planned with other test activities (such as system test, integration test) in test plan or included in project schedule
   - Requirement for test: Identify the features/methods to be tested

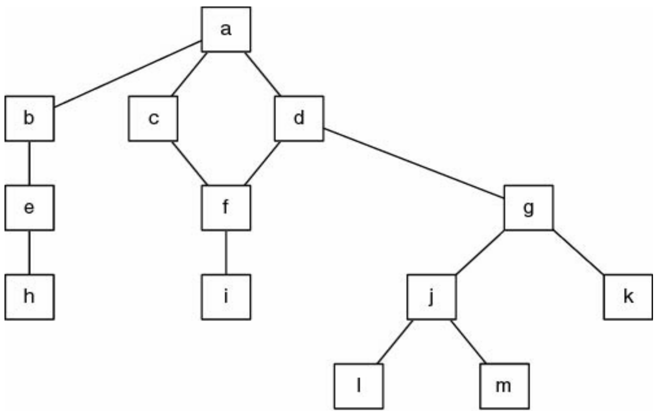
- Design approach to do unit test
    - Method of analyzing test (visual or comparator program)
    - Technique used to test (Black box test/ White box test)
    - Tools to use by Unit test

2. Unit test implementation

   In order to test a code unit (a module, class or function), we typically have to create stubs or drivers.

   - Test stub is a simple dummy code unit, which is developed instead of the **code unit being called** by the code-under-test
   - Test driver is a simple dummy code unit, which is developed instead of **the code unit calling** the code-under-test

   Hereunder is an example of a software package that consists of 13 modules



To test module a, stubs should be developed instead of modules b, c, d

To test module h, it requires a driver for replacing module e

Testing module d requires a driver (for a) and two stubs (for f and g)

3.1.3 Unit Test Plan

To execute the unit test effectively, it is necessary to generate the Unit Test Plan (UTP). It should be completed with Detail Design.

The Unit Test Plan entails the development of a document, which instructs what tests to perform on each module in the system. The objective is to ensure that the particular module under test works properly and performs all the desired functions.

Unit Test Plan provides a test with a list of desired inputs into the module and a corresponding list of expected outputs from the module. The module is passed when all the inputs are entered and all the expected outputs are generated. Any deviation from the expected outputs will be noted. The list of inputs can, at least at first, be derived from the requirements. UTP helps the developer to make sure that every line of code and every possible conditional statement are executed at least once.

3.1.4 Unit Test design

Unit test should:

- Verify operation at normal parameter values.
- Verify operation at limit parameter values.
- Verify operation outside parameter values.
- Check the normal termination of all loops.
- Check the abnormal termination of all loops.
- Check the normal termination of all recursions.
- Check the abnormal termination of all recursions.
- Verify the handling of all data structures accessed by that function.
- Verify the handling of all files accessed by that member function.
- Verify the handling of all error conditions.
- Check the performance test if necessary.
- Ensure that every statement executes.
- Ensure that every decision executes at all sides of all branches.

3.1.5 Unit Test Case

Development teams have to create Unit test case to make sure that each statement, decision branch, or path is tested with at least one test case.

3.1.6 Test Environment

Unit tests are typically conducted in an isolated or special test environment. Since a module is not a stand-alone program, we may have to create driver and/or stub software for each unit test. *Drivers and stubs are not delivered with software. They are used only for testing*.

In most applications a driver is nothing more than a *main program* that accepts test case data, passes such data to the module (to be tested) and prints the relevant results. Stubs serve to replace modules that are subordinate (called by) to the module to be tested. A stub or *dummy subprogram* use the subordinate modules interface may do minimal data manipulation, prints verification of entry and returns.

The driver and stub represent overhead, that is, both are software that must be written but are not delivered with the final software product. If the driver and stub programs are kept simple, the actual overhead is relatively low. Many modules cannot be unit-tested with simple drivers and stubs. In such cases, complete testing can be postponed till integration testing (where drivers and stubs are used).

3.1.7 Unit test tools

A Unit Test tool is a bunch of source code available for:

- Create Test suite
- Create Test case
- Run test
- Get the test report

There are unit test tools for many languages available:

- JUNIT for Java
- CPPUnit for C++
- pyUnit for Python
- PerlUnit for Perl

There are some tools available to measure, analyze and track the coverage of unit test, such as:

- Cobertura
- DJUnit

3.1.8 Unit test Measurement

Several different unit test coverage criteria exist. Each gives users a set of test conditions.

- Statement coverage
- Branch coverage
- Path coverage: a complete path is a sequence of nodes in the program flow graph from program start to program end.
- Condition coverage: to measure how many possible outcomes of conditions are tested.

3.2 Integration Test

3.2.1 Objectives

Integration testing is the next level of Unit testing, it is implemented by test team and focuses on testing the integration of "units of code" or components. Before Integration Testing, it is important that all the components have been successfully unit tested.

These "integrated" components are tested to weed out errors and bugs caused due to the integration. This is a very important step in the Software Development Life Cycle.

Integration testing checks whether the internal and external interfaces of the system-under-test work properly.

3.2.2 Integration test planning

Integration test plan may be included in test plan or integration plan depending on that will do that: test team or development team.

For small projects and simple systems (refer to Product Integration guideline for detail), internal/external interfaces can be verified in unit test stage.

For the big projects and complex systems, separate integration test may be planned and conducted by development team in assembly project software product from its modules. Integration test technique is based on the product integration sequences selected by project team. In that case, integration test is a part of product integration activities. Integration test cases can be developed separately or merged to product integration plan.

Project interfaces to external systems can be verified in system test conducted by test team. In that case the external interfaces are requested explicitly in URD or SRS. Integration test cases may be included in project system test case.

3.2.3 Integration test environment

Integration test environment should be independent from the development environment. In case of simple and small system (as defined in Integration guideline), they can be the same.

To test system interfaces, stub and driver programs should be created and used.

Stubs stand-in for finished sub routines or sub-systems and often consists of nothing more than a function header with no body. If a stub needs to return values for testing purposes, it may read and return test data from a file, return hard-coded values, or obtain data from a user (the tester) and return it.

3.2.4 Integration test design

Integration test design is created to verify system interfaces

Its objectives are to detect faults due to interface errors or invalid assumptions about interfaces

It is particularly important for object-oriented development as objects are defined by their interfaces

3.2.4.1 Interfaces types

- Parameter interfaces: Data passed from one procedure to another
- Shared memory interfaces: Block of memory is shared between procedures
- Procedural interfaces: Sub-system encapsulates a set of procedures to be called by other sub-systems
- Message passing interfaces: Sub-systems request services from other sub-systems

3.2.4.2 Interface errors

- Interface misuse: A calling component calls another component and makes an error in its use of its interface (e.g. parameters in the wrong order)
- Interface misunderstanding: A calling component embeds assumptions about the behavior of the called component which are incorrect
- Timing errors: The called and the calling component operate at different speeds and out-of-date information is accessed

3.2.4.3 Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated

3.2.5 Integration Test case

The Integration test cases specifically focus on the flow of data/information/control from one component to the other. So the Integration Test cases should typically focus on scenarios where one component is being called from another. Also the overall application functionality should be tested to make sure the app works when the different components are brought together.

Therefore the Integration Test Cases should be as detailed as possible

3.3 System test

3.3.1 Objectives

System test is to verify if the whole system functions in correspondence to the system requirement.

System testing is usually performed after unit and integration testing have been successfully applied for all modules.

System test technique is normally black box test and implemented by Test team.

3.3.2 Test process

Follow the process stated in section 2 of Test lifecycle strictly. Especially the test environment must be independent from the development one.

3.3.3 System Test planning

System test plan is normally included in test plan.

Test plan is to define overall project testing structure, including the organizations involved, their relationship to one another and the authority and responsibility of each organization shall be provided.

Test plan should identify what to be tested, activities, resources, and planning processes deemed to be necessary. An overview of the facilities to be used, products and milestones of testing are described in the plan.

Refer to Template_Test Plan for more information.

3.3.4 System Test Case

System test cases are created to verify that the application meets the requirements defined in the specification documents.

System test cases should provide specific detail of the behavior or function being tested. Actual input data and expected output data are provided. The expected output is precise and detailed enough to be implemented. The data needs to run on the actual software being released.

The test cases should be written in enough detail that they could be given to a new team member who would be able to quickly start the tests and find defects.
The preparation of the test cases and scenarios need time and effort. To maximize efficiency this process, testers should:

- Keep information in one location only, to maximize re-use and minimize the effect of changes.
- Complete steps one time to avoid redundancy and extra work.
- Complete as many steps as possible, as early as possible.

Following are steps to create System test cases:

- Identify the business scenario types to be tested
- Identify test cases for each requirement
- Create test scenarios that incorporate all of the test cases
- Document the test script that includes the system entry information necessary to test the scenario and the expected result in terms of the system

After each step, review the information is necessary to ensure accuracy and completeness.

3.4 Acceptance test

Acceptance Test demonstrates to the customer that predefined acceptance criteria have been met by the system.

Typically it is used as a mechanism to handover the system.

4 TEST TYPE

There are many types of test. This section introduces most common test types.

4.1 Functional Test

Function testing should focus on any requirements for test that can be traced directly to use cases or business functions and business rules. The goals of these tests are to verify proper data acceptance, processing, and retrieval, and the appropriate implementation of the business rules.

4.2 User Interface Test

The goal of User Interface test is to ensure that the User Interface provides the user with the appropriate access and navigation through the functions of the target-of-test.

4.3 Data and Database Integrity Test

Data and Database Integrity Test should focus on interface of data, and ensure that database access methods and processes function properly and without data corruption.

4.4 Performance Test

Performance test is to measure and evaluate response times, transaction rates, and other time-sensitive requirements.

It includes Load test, Stress test, Volume test, …

4.5 Security and Access Control Testing

Security and Access Control Testing focus on two key areas of security:

- Application-level security, including access to the Data or Business Functions
- System-level Security, including logging into or remote access from the system.

Application-level security ensures that, based upon the desired security, actors are restricted to specific functions or use cases, or are limited in the data that is available to them. For example, everyone may be permitted to enter data and create new accounts, but only managers can delete them. If there is security at the data level, testing ensures that" user type one" can see all customer information, including financial data, however," user two" only sees the demographic data for the same client.

System-level security ensures that only those users granted access to the system are capable of accessing the applications and only through the appropriate gateways

4.6 Regression Test

Regression test is a type of software testing which seeks to reduce this risk. Regression testing should be conducted during all stages of testing after a functional change, reduction, improvement, or repair has been made. This technique assures that the change will not cause adverse effects on parts of the application or system that were not supposed to change. Regression testing can be a very expensive undertaking, both in terms of time and money.

Regression test must be performed when:

- Total number of change requests arisen since the latest baseline with regression test is 10% bigger than the number of requirements in that baseline.
- The rate of total number of defects detected after the acceptance test or in operation divides total number of man-months of the project is bigger than 1.

For maintenance projects, trigger for regression test must be defined in Test plan. Test leader has to identify when the team must conduct regression test and scope of regression test. The schedule of regression test must be defined in project schedule.

Fsoft functional test tool should be employed in regression test if the related project satisfies the condition defined in section 8. Test tool

There are some types of regression test:

- Unit Regression Testing: This retests a single program or component after a change has been made. At a minimum, the developer should always execute unit regression testing when a change is made.
- Regional Regression Testing: This retests modules connected to the program or component that have been changed. This is a significant timesaving over executing a full regression test, and still helps assure the project team and users that no new defects were introduced.
- Full Regression Testing: This retests the entire application after a change has been made. A full regression test is usually executed when multiple changes have been made to critical components of the application.

5 TEST TECHNIQUES

There are two classes of test techniques:

- Without regard to the internal structure of the system, called *testing to specifications*, *requirement based testing, data-driven testing, input/output-driven testing, functional testing*, or *black-box testing*
- Based on the internal structure of the system, called *testing to code, path-oriented testing, logic-driven testing, structural testing, glass-box testing,* or *white-box testing*

5.1 Black Box Testing

5.1.1 Black box testing types

- *Functional tests* exercise code with valid or invalid input for which the expected output is known
- *Performance tests* evaluate response time, memory usage, throughput, device utilization, and execution time

5.1.2 Black box testing techniques (for functional test)

There are many black box testing techniques. The followings are most popular.

5.1.2.1 Equivalence Partitioning

Because exhaustive black-box testing (test all combinations of input values) is infeasible, black-box test cases must be chosen to find different faults if possible.

Assume that similar inputs will evoke similar responses; we group inputs into *equivalence classes*.

In *equivalence partitioning*, only one or a few test cases are chosen to represent an entire equivalence class

Input data should be divided into equivalence classes based on consideration of:

- Valid vs. invalid input values
- Valid vs. invalid output values
- Similarity and difference of input values

- Similarity and difference of output values
- Similarity and difference of expected processing

5.1.2.2 Boundary Value Analysis

Black-box test cases can be chosen to maximize the chance of finding faults.

Experience has shown that values at or near equivalence class boundaries are more likely to detect faults.

Boundary value analysis is choosing test cases at or near the boundaries of equivalence classes.

5.2 White Box Testing

The effectiveness or thoroughness of white-box testing is commonly expressed in terms of *test or code coverage metrics*, which measure the fraction of code exercised by test cases.

Test cases are selected to achieve target test coverage levels. (See 7.4 Evaluate Code-based Test Coverage)

White box test cases should be selected to:

- Guarantee that all independent paths within a module have been exercised at least once
- Exercise all logical decisions on their true and false conditions
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

Basis Path is the most popular white box testing technique.

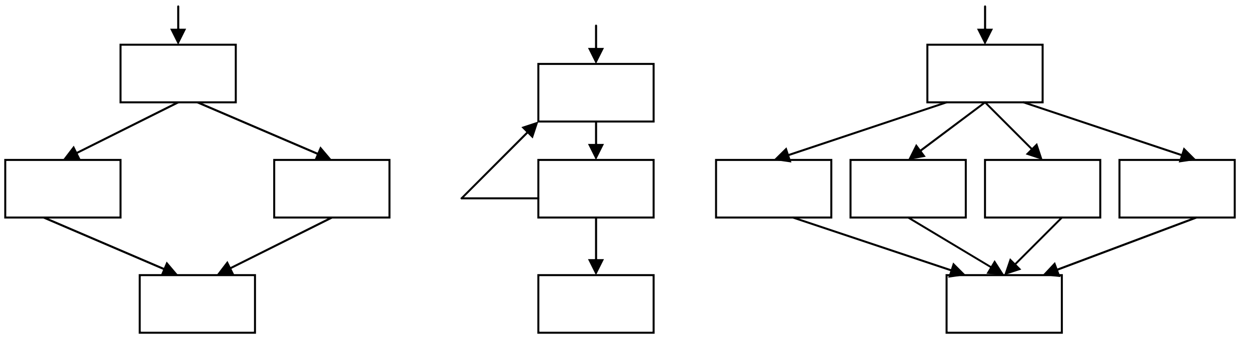
5.2.1 Basis Path Testing

Test cases are derived from the code logic and are independent of the software requirements specification. A set of test cases produced by the method is said to be a basis test set. The name comes from the fact that the paths taken by the test cases in the basis test set form a basis for the set of all possible paths through the program. That is, every path (regardless of how many may exist) may be built up as a linear combination of the path segments traveled by the basis cases.

Basis Path Testing Summary

5.2.1.1 Step 1

Develop the program flow graph for a selected code segment by straightforward substitution of the following flow of control sub-graphs.



If -then - else **loop - while** **case - of**

- Proceed statement by statement
- Ignore sequential statements
- Add a node for any branching or decision statement
- Expand the node by substituting the appropriate sub-graph representing it.

5.2.1.2 Step 2

Compute complexity (c) from the flow graph using the formulas

C = # Edges - # Nodes + 1

5.2.1.3 Step 3

Find C independent test cases

- Select an arbitrary test case to start
- Create the next case by following a similar path but modified as to exercise at least one new edge
- Continue until 'C' test cases are derived

5.2.1.4 Step 4

Obtain Expected Results for each of the C Cases

- Use program specifications and determine what data should do (best if someone else, the Analyst for example, does this)

5.2.1.5 Step 5

Confirm that actual results match expected results

- Execute or walk through each basis case

Effectiveness of Basis Path Testing

Effectiveness

- Much stronger than complete coverage
- Will detect almost all errors
- Also catches a broad class of other types of errors
- Excellent vehicle for Code Review and walkthroughs
- Can be applied to high-level logic or pseudo code

Efficiency

- Well-defined procedure
- Efficient in machine resources and engineer time
- Generates simple and easy-to-execute test cases
- A good Cost Vs. Confidence Trade-off

6 TEST TOOLS

Testers should use the test tools available in Fsoft

6.1 DMS for Defect management

All defects found by test should be log into DMS for tracking. Testers and Project leaders are responsible to monitor the status of defects and report test result to QA by Final inspection

6.2 Virtual Ruler and Eye Dropper

These tools are used for Graphic test

6.3 Rational Robot Functional test tool

The automated functional test should be applied for maintenance project, which follows another FSoft project.

Test team must have both of manual and automated test skills. The team should include the manual tester of the previous project and test automator, who is trained in Rational Robot Functional test training course

6.4 Rational Performance test tool

The tool should be applied in case of explicit requirement on performance of products.

Test team must have both of manual and automated test skills. The team should include a test automate, who is trained in Rational Performance test training course

7 TEST EVALUATION

7.1 Make "as-run" test procedure log during test execution

Purpose: To record “as-run” test procedures during test execution for later evaluation.

To log “as-run” test procedures:

- During and after execution of each test case, tester logs the following, but not limit to, information of test execution:
  - Test case reference (if the related test case exists)
  - Tester name
  - Test data
  - Test execution time
  - Test environment/conditions
  - Actual test actions
  - Actual behaviors of the system-under-test
- “As-run” test procedure log should be conducted throughout the test period of the project. Normally it is implemented in the form of defect log (in defect description field of DMS). The above data should be logged if the actual result is different from the expected one. Sometimes customer may require test log for both successful and unsuccessful cases. In that case, a separated test log document (sometimes it is named as “Test report”) should be created.

7.2 Analyse “as-run” test procedure log

Purpose: To determine if test execution has been carried out properly.

Analyze test log or “as-run” method documentation to identify that bad test results are due to test design problems, method problems, defects or a test environment problem. Based on the analysis result, corrective actions will be defined to update test case, correct the code-under-test, re-configure test environment or change test execution method.

7.3 Evaluate Requirements-based Test Coverage

Purpose: To determine if the required (or appropriate) requirements-based test coverage has been achieved

To evaluate requirements-based test coverage, you need to review the test results and determine:

- The ratio between how many requirement-based tests (test cases) has been performed in this iteration and a total number of tests for the target for test.
- The ratio of successfully performed test cases.

The objective is to ensure that 100 % of the requirements-based tests targeted for this iteration have been executed successfully. If this is not possible or feasible, different test coverage criteria should be identified, based upon:

- Risk or priority
- Acceptable coverage percent

Requirements-based test coverage is measured several times during the test life cycle and provides the identification of the test coverage at a milestone in the testing life cycle (such as the planned, implemented, executed, and successful test coverage).

- Test coverage is calculated by the following equation:

  Test Coverage = T(p,i,x,s) / RfT

  where:
  T is the number of Tests (planned, implemented, executed, or successful) as expressed as test procedures or test cases.

  RfT is the total number of Requirements for Test.

- In the Plan Test activity, the test coverage is calculated to determine the planned test coverage and is calculated in the following manner:

  Test Coverage (planned) = Tp / RfT

  where:
  Tp is the number of planned Tests as expressed as test procedures or test cases.

  RfT is the total number of Requirements for Test.

- In the Implement Test activity, as test procedures are being implemented (as test scripts), test coverage is calculated using the following equation:

Test Coverage (implemented) = Ti / RfT

where:
Ti is the number of Tests implemented as expressed by the number of test procedures or test cases for which there are corresponding test scripts.

RfT is the total number of Requirements for Test.

- In the Execute Test activity, there are two test coverage measures used, one identifies the test coverage achieved by executing the tests, while the second identifies the successful test coverage (those tests that executed without failures, such as defects or unexpected results).

  These coverage measures are calculated by the following equations:

  Test Coverage (executed) = Tx / RfT

  Where:
  Tx is the number of Tests executed as expressed as test procedures or test cases.

  RfT is the total number of Requirements for Test.

  Successful Test Coverage (executed) = Ts / RfT

  where:
  Ts is the number of Tests executed as expressed as test procedures or test cases which completed successfully and without defects.

  RfT is the total number of Requirements for Test.

Turning the above ratios into percentages allows the following statement of requirements-based test coverage:

x% of test cases (T(p,i,x,s) in the above equations) have been covered with a success rate of y%

This is a meaningful statement of test coverage that can be matched against defined success criteria. If the criteria have not been met, then the statement provides a basis for predicting how much testing effort remains.

7.4 Evaluate Code-based Test Coverage

Purpose: To determine if the required (or appropriate) code-based test coverage has been achieved.

To evaluate code-based test coverage, you need to review the test results and determine:

- The ratio between the code that has been executed during test (such as lines or statements) in this iteration and the total code in the test target.

The objective is to ensure that 100 % of the code targeted for this iteration phase been executed successfully. If this is not possible or feasible, different test coverage criteria should be identified, based upon:

- Risk or priority
- Acceptable coverage percent

Code-based test coverage measures how much code has been executed during the test, compared to how much code there is left to execute. Code coverage can either be based on control flows (statement, branch, or paths) or data flows. In control-flow coverage, the aim is to test lines of code, branch conditions, paths through the code, or other elements of the software's flow of control. In data-flow coverage, the aim is to test that data states remain valid through the operation of the software, for example, that a data element is defined before it is used.

Code-based test coverage is calculated by the following equation:

Test Coverage = Ie / TIic

Where:
Ie is the number of items executed expressed as code statements, code branches, code paths, data state decision points, or data element names.

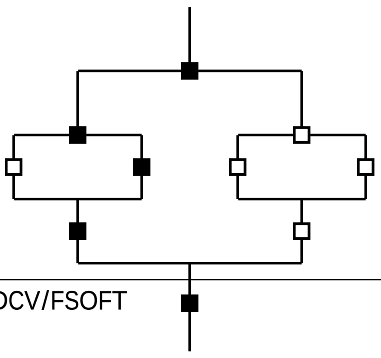
TIic is the total number of items in the code.

Turning this ratio into a percentage allows the following statement of code-based test coverage:

x% of test cases (I in the above equation) have been covered with a success rate of y%

This is a meaningful statement of test coverage that can be matched against defined success criteria. If the criteria have not been met, then the statement provides a basis for predicting how much testing effort remains.

Example:



The blackened nodes trace the execution path for a single test case. For purposes of illustration, we regard each node as a statement; hence statement coverage is the number of black nodes divided by the number of nodes.

There are two branch directions out of each test, so there are six altogether, of which three are taken.

There are 4 ways to trace a path through the graph from top to bottom.

Note that statement coverage is higher than branch coverage, which is higher than path coverage. Generally these relationships will persist given a program and an arbitrary collection of test cases. Consequently statement coverage goals are generally set higher than branch coverage goals, which are in turn set higher than path coverage goals.

In this example:

Statement Coverage: 5/10 = 50%

Branch Coverage: 2/6 = 33%

Path Coverage: 1/4 = 25%

7.5 Analyze Defects

Purpose:

- To evaluate the defects and recommend the appropriate follow-on activity
- To produce objective reports communicating the results of testing

The most common defect measures used include different measures (often displayed in the form of a graph):

- Defect Rate - Total weighted defects/ Project size (in UCP).
- Defect Trend - the defect count is shown as a function over time.
- Defect Aging - a special defect density report in which the defect counts are shown as a function of the length of time a defect remained in a given status (waiting-for-test, error, pending, tested, accepted, etc.)
- Defect Origin - indicates the actual process that caused the defect (such as Coding, Correction, Customer Support, Deployment...)

- Defect Type - used to group defects by classifying types of problem (such as Requirement misunderstanding, Feature Missing, Business Logic…)
- Defect Severity – indicate the affect of the defect to the system. There are 04 severities available for a defect: Fatal, Serious, Medium, And Cosmetic.
- Determine if Test Completion and Success Criteria Have Been Achieved.

8 MEASUREMENTS

Refer to Metrics Guidelines.

The key measures of a test include coverage and quality.

- Test coverage is the measurement of testing completeness, and is based on the coverage of testing, expressed either by the coverage of test requirements and test cases, or the coverage of executed code.
- Quality is a measure of reliability, stability, and the performance of the target of test (system or application-under-test). Quality is based upon the evaluation of test and the analyses of defects discovered during the testing.

Besides, we can evaluate functional test automation with the following measurements:

- Saved test execution effort when using functional test tool in regression test (Total of manual test effort – Total testers' effort spent for execute test scripts and test result analysis)
- The increment of test coverage when using functional test tool
- The automated test case coverage (30% or above is recommended)

Approver	Reviewer	Creator
Nguyen Lam Phuong	**Nguyen Thi Thu Ha**	**Nguyen Duy Binh**