Software Testing

Claire Lohr

Software testing is defined in IEEE Standard 610.12 as:

(1) An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.
(2) To conduct an activity as in (1).
(3) A set of one or more test cases.
(4) A set of one or more test procedures.
(5) A set of one or more test cases and procedures.

The single word "test" can mean all of the activity, part of the activity, all of the product of the activity, or a part of the product of the activity. As this can be confusing, it is recommended at all times to accompany "test" with more information in order to provide the particular context for a given discussion.

TYPES OF TESTS (PARTS OF THE ACTIVITY)

The SWEBOK (Chapter 5, Section 3) provides a robust list of possible *types* of tests. It has divided them into categories based on the following attributes:

- The software engineer's intuition and experience
- Specifications
- Code
- Dataflow
- Fault
- Usage
- The nature of the application

More information is provided here for some of the most commonly used types of tests:

- Equivalence class partitioning
- Boundary value
- Decision table
- Exploratory
- Operational profile

Equivalence class partitioning examines the defined acceptable range for each input to determine the following classes for each input:

- "Valid(s)": continuous range or list of values that should be legal and processed successfully by the code.
- "Invalid(s)": continuous range or list of values that should be illegal and not accepted by the software, but not cause an unacceptable result, either.

An example would be the age of an individual. The valid range could be 0 to 120 (from just born to the oldest possible age likely to be needed). This would lead to the following equivalence classes:

- Valid: {0–120}
- Invalid: {<0} and {>120}

Another example with different characteristics is the month of the year:

- Valid: {January, February, March, . . . , December}
- Invalid: {anything else}

The steps of equivalence class partitioning are:

1. Define the valid and invalid classes for each possible input.
2. Test as many of the valid classes together as possible. Any possible value in the valid class can be used.
3. Do one test for each invalid class. Any possible value in the invalid class may be used. Invalid classes may not be combined since software rarely continues processing after the first invalid it finds, so the subsequent invalids would not experience any testing. An exception is that the entry of data in a form on a Web page is often all validated at one time (for usability).

Boundary value testing mandates the testing of four values for each input:

1. The legally defined minimum
2. The legally defined maximum
3. The first possible value below the legally defined minimum
4. The first possible value above the legally defined maximum

For numeric ranges, the implementation of this technique is clear. The age of an individual defined above would result in the following test values:

1. 0
2. 120
3. –1
4. 121

It can be applied for the length of the field for nonnumeric inputs. The example below is for a person's last name, where the length of the field can be 1 to15 characters. The test values could be:

1. A
2. Abcdefghijklmno
3. {null entry}
4. Abcdefghijklmnop

A *decision table* lists the all conditions (the inputs) and all possible resultant actions (outputs; including error notifications) in the first column of a table. Then there is a column or "rule" for every possible combination of input conditions. The presence or absence of a particular input condition is marked with a Y for yes, an N for no, or an I for immaterial (meaning both yes and no). Each action is also marked with a Y for yes, if it is produced, or N for no, if it is not produced for that individual rule. The purpose of this technique is to cover all possible combinations of the input conditions in the tests. It must be used for a selected segment of the software, or it will become much too large to be useful.

An example of a decision table for computing U.S. payroll tax withholding is shown in Table 1. If there were no immaterial condition entries, there would have been eight rules: NNN, NNY, NYN, NYY, YNN, YNY, YYN, and YYY to cover all possible input condition combinations.

Exploratory testing does not require preplanning of exact data values. It instead emphasizes planning the focus of the testing process. For example, choosing to test compatibility with former releases or other products in a product line, or end-to-end consistency within one product. A plan is made before beginning (which may or may not be written), and then the testing commences. As the testing progresses, it is not mandatory to stick to the plan. The tester reacts to the results being produced and

Table 1

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Wages earned	N	Y	Y	Y
End of pay period	I	N	Y	Y
FICA limit exceeded	I	I	N	Y
Actions				
Withhold FICA tax	N	N	Y	N
Withhold Medicare tax	N	N	Y	Y
Withhold payroll tax	N	N	Y	Y

modifies the plan accordingly. For example, if many problems are being discovered with a particular aspect of the system, more tests are immediately created for that area.

An *operational profile* technique recommends that the number of tests run for each system feature follow the model of how much it is used during operation. Actual usage may be measured (or estimated for systems not yet in use). This is particularly useful for a regression test. The goal is to test more (and as a result achieve more robustness of the software) for the features that are used more. Several organizations have found that they can run fewer tests and achieve higher reliability using this technique.

TEST LEVELS (PARTS OF THE ACTIVITY)

Testing is done at more than one level as a software product is being developed or maintained. Test levels vary as to the scope of software being tested, the test objectives, the test techniques, and often the test environment. Most test level schemes start with the smallest scope to be tested (e.g., a single unit) and increase in scope until all of the systems are included (e.g., all integrated system products). Some organizations also have multiple levels of test defined for the entire system product (e.g., regression, security, performance, accessibility, usability, alpha, beta). Any one of these levels may have multiple levels within it; for example, performance testing may encompass single shot, smoke, normal, and stress levels.

The number and types of test levels vary greatly from organization to organization and even with projects within one organization. Some of the main factors that effect this decision are:

- System size
- Complexity
- Safety criticality
- The experience of the testing staff and management
- The degree of desire for certification (e.g., CMMI or ISO 9000)

Each organization makes its own decision about how many levels of test to have, and then improves it over time if the goals of the organization are not being met.

The most thorough definition of possible test levels is in IEEE/EIA 12207.0. It recognizes that different processes (e.g., acquisition, development, operation, maintenance, supporting) have different testing level needs. The acquisition process requires an acceptance test level at which the acquirer checks that all of the predefined acceptance conditions are met. The levels of test defined for development are:

- Each software unit and database
- Integrated units and components
- Tests for each software requirement
- Software qualification testing for all requirements
- System integration. Aggregates of other software configuration items, hardware, manual operations, and other systems. It is not unusual for large systems to have multiple levels of integration testing.
- System qualification testing for system requirements

The operation process defines one test level: operational testing.

The maintenance process defines the following test levels:

- All development levels for improvements and adaptation
- Test the modified parts of the system
- Test the unmodified parts
- Migration verification
- Possibly also parallel testing

The supporting processes define the following test levels (possibly performed independently):

- Verification. The software products of a specific activity successfully implements the requirements of the immediately prior activity (code and integration verification test levels)
- Validation. The final, as-built software or system product meets all of the requirements

The SWEBOK (Chapter 25, Section 2) provides thorough definitions of the unit (Section 2.2.1), integration (Section 2.1.2), and system (Section 2.1.3) test levels. All of the above test levels are defined in IEEE/EIA 12207.0.

TESTING STRATEGIES (PARTS OF THE ACTIVITY)

Since testing of all possible data values in all possible combinations in all possible sequences is inherently impossible, all organizations employ some kind of test strategy. The strategy reflects the priorities of the organization and leads to the emphasis and coverage goals for the test cases themselves. Strategies are usually focused on either "macro" or "micro" test issues. The three traditional *"macro" test strategy* issues are

- Time-to-market (speed of test development and execution)
- Amount of functionality to be delivered (and as a result, tested)
- Quality of the product (thoroughness of the testing)

These three issues are trade-offs. The product (with constant functionality) can be delivered in less time (tested less) if quality can be reduced. The product can be reduced in size and delivered on time with enough quality (as demonstrated by testing), and so on. Unfortunately, most organizations want all three. Therefore, testing requires a strategy that creates priorities for the selection and management of all test activities. Another "macro" test strategy issue is the desire for the cost of all development and maintenance activities (including testing) to meet expectations. These "macro" strategies are testing's role in supporting the organization's business goals. They are met by employing relatively "micro" test strategies focused on test processes and products.

Below are some examples of "micro" strategies that would support the above "macro" strategies.

For reduction of time to market:

- More automation to speed test execution.
- Fewer turnovers in testing staff to minimize time for learning curves.
- Better selection of the test cases that are executed, to find the more serious problems earlier.

For functionality changes (testing organizations rarely control this, but they have to adapt quickly when changes are made, both reduction in functionality and last minute addition of new functions):

- Tracing from requirements to test cases to enable finding the cases affected by the changes quickly.
- Smaller, more modular test scripts to maximize reuse during changes in functionality.

For quality of the product:

- Better tools to measure current test coverage.
- Better unit test tools for developers.
- More variety in the test types, for example, add exploratory testing to traditional domain testing.

To control test effort costs:

- Use a project management tool to estimate test activities and to track the actual expenditures.
- Add root cause analysis (determine the source of problems encountered and change the processes so that they do not occur again) to test readiness review meetings.

TEST DESIGN (CREATION OF PART OF THE PRODUCT)

Test design is the combined art and science of picking the most appropriate test techniques to use at each test level. It would consume too many resources to try to use all of them, and would lead to redundant testing and diminished results. Given most organizations' test strategies, the most common goal of test design is to get the most return with the least effort.

A good test design includes both *structured* and *unstructured* techniques. Some examples of *unstructured* techniques are:

- Random
- Ad hoc
- Exploratory

Some examples of *structured* techniques are:

- Equivalence class partitioning
- Boundary value
- Decision table

The advantage of structured techniques is that they provide linear coverage—all attributes are tested to the same degree in exactly the same manner. The advantage of unstructured techniques is that they find lots of problems, often quite serious. A robust test design will include both of these, as well as what is "normal" in operations (usage-based techniques).

Software with high criticality will require the inclusion of more techniques, and more test cases within each technique's options. Of particular importance will be the inclusion of software reliability engineered testing.

TEST COVERAGE OF CODE (CREATION OF PART OF THE PRODUCT)

The *goal for coverage of code* during testing will vary with the level of test. For a system test, coverage may be measured, but the goal is rarely anywhere near 100%, as it is too hard to set up the tests to reach all of the code for a whole system. For a new unit, the commonly recognized coverage goal is 100%. The goal for modifications to existing code is 100% of the modifications.

There are multiple ways to define if a line of code or statement was "covered." One method is to count it as covered if it was executed in any part. Definitions of what constitutes a line of code or statement vary, too. Some count physical lines of code, some count logical line of code (with a supporting definition based on the constructs of the programming language). Since coverage is most often measured by automated test tools, the definitions used by the tool developers will need to be understood. The most robust definition is to use logical lines of code with 100% coverage. Another way to describe this same definition is to call out a need for execution of every condition/branch. There is also a definition that calls for 100% of all possible paths for execution. This is rarely used, becasue when there are loops involved with some imbedded decisions, the number of possible path combinations quickly becomes impossibly huge.

A universally recognized technique for achieving 100% coverage of all statements is to use Tom McCabe's Basis Path Testing technique. This technique has the following steps:

1. Draw a flowgraph, a picture in which each logical statement (or series of statements with no embedded decisions) is represented by a circle (called a node), and arrows (called edges; some representations of this technique have lines without the directional arrows) show the transfer of control as a result of decisions.
2. Compute the metric *cyclomatic complexity.* Count the number of regions—the separately divided areas of the flowgraph (counting the outside of the flowgraph, too).
3. Choose the paths (cyclomatic complexity tells how many are needed). Cover each node and edge, and each path varying from the others by at least one edge.

Sometimes it is possible to test all of the nodes and edges with fewer than the cyclomatic complexity number of paths; McCabe defines this as "actual complexity." A simple example is shown in Table 2.

Note that there are other possible paths that were not included in Table 2. The goal of McCabe's technique is 100% branch/condition coverage, but not every possible path.

An alternative method for computing Cyclomatic Complexity is:

$$\text{number of edges} - \text{number of nodes} + 2$$

In the example above, this would be $11 - 9 + 2 = 4$.

Additional examples and explanation of cyclomatic complexity can be found in Meyers (1979). Cyclomatic complexity is frequently included on certification exams in one form or another, as it is one of the few aspects of software engineering that produces a numerical answer and, therefore, provides an easily deterministic exam question and answer.

TEST COVERAGE OF SPECIFICATIONS (CREATION OF PART OF THE PRODUCT)

The source documents for development (requirements and design) are often combined under the umbrella term "specifications." The goals of some of the levels of test are to cover all of one or more specifications. This can only be measured if the contents of a specification have been inventoried in some manner, such as putting each testable requirement in a table or database and assigning it a unique identification number (a common activity in configuration management). After the inventory has been completed, each requirement can then be traced to the test cases where it is exercised by filling in a column in the table identifying these cases. A simple example of a *traceability matrix* is shown in Table 3.

Once the tracing has been completed, it is possible to compute the *% requirements tested* or a similar metric. In the event that more tests are planned that than get executed, it is also possible with this matrix to measure the *% requirement tests executed.*

TEST EXECUTION (PART OF THE ACTIVITY)

The needs for the test execution are specified in the Test Plan. It fully specifies every component of the test environment, including:

Table 2

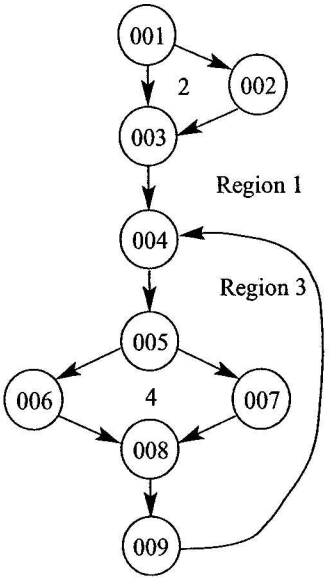
Code statements (in simplistic pseudocode)	Corresponding flowgraph
001 If A 002 then B 003 End if 004 For 1 to n, do 005 If more data, 006 Then add to total C 007 Else send error message D 008 End if 009 End for The Cyclomatic Complexity is equal to the number of regions, which is 4. Four possible test cases would follow these paths: 001,003,004,005,006,008,009 001,002,003,004,005,008,009 001,003,004,005,007,008,009 001,003,004,005,006,008,009,004,005,007,008,009 In this case, Cyclomatic Complexity could be reduced to Actual Complexity of 2, with the following paths: 001,003,004,005,006,008,009 001,002,003,004,005,007,008,009,004,005,006,008,009	

Table 3

ID #	Requirement summary	Test scenario/cases
001	Selection of items for purchase.	PUR002/35-40, 66-85 PUR003/1-57
002	Entry and validation of credit card information	CRC001/1-36 CRC002/5-89 CRC003/5-34, 75-115
003	Limit the maximum items purchased at one time as a fraud countermeasure	PUR001/56-77 PUR002/41-65

- Hardware
- Software
- Automated tools
- Data
- Personnel

All of the actual test input and procedures are specified in the test cases and test procedures. They need to be documented with enough clarity so that another individual can replicate the results. The test results are logged during execution following the test plan, including an evaluation as to the success or failure of each test case. Incident reports are recorded during test execution with enough detail to allow both the developer and subsequent tester of the repaired code to reproduce the original problem.

Testing may not go according to plan. Some common challenges are that some or all of the software to be tested is late, the tools have unexpected shortfalls in ease of use or functionality, some part of the software cannot be tested at all, and/or access to or existence of hardware is delayed. Test execution is mostly about adapting to unplanned changes.

Many organizations designate an individual as a "Test Director" or "Test Coordinator" to adapt to the changes from the plan and to redirect the total test execution effort. It is not unusual to need replanning on a daily basis. New tests are created based on the prior results, existing tests cannot be run due to functions not working at all in the software, and time starts to run out. As the life cycle steps prior to testing slip their schedules, the tendency is to still expect testing to conclude at the same time even though it starts late. The Test Director can make the best possible use of the available resources. When the test execution is completed, the results are documented in the test summary report.

TEST DOCUMENTATION (ALL OF THE PRODUCT)

Test documentation is recorded in multiple *media*:

- Traditional word processing documents
- Spreadsheets
- Databases
- Within automated test tools

The media selected for documentation will vary based on the level of detail of the test documents, the experience of the testers preparing and using the documentation, and the availability of automated test tools.

The *test documents* are usually all or part of those prescribed by IEEE Standard 829-1998, *Standard for Software Test Documentation:*

- Test Plan. The overall resources, test environment, scope of what is to be tested/not tested, and methods.
- Test Design. A more detailed level of methodology information for an identified subset of the overall scope.
- Test Cases. The actual data needed to run the tests, for example, inputs and outputs.
- Test Procedures. The steps for the pretest setup, test execution, and posttest activities.
- Test Logs. The actual test results.
- Incident Reports. Descriptions of test results that do not match expectations (often called problem, defect, issue, or anomaly reports instead).

- Test Summary Report. The pass or fail decision for the test, the rationale for that decision, a summary of all test results, and the detailed test results.
- Test Item Transmittal Report (optional). An inventory of all test documents and data being delivered as a result of a test.

The above listed documentation requirements are usually tailored by individual organizations to better meet their needs and abilities. It is common for organizations to combine one or more of the above listed documents, for example the test cases, test procedures, and test log. Incident reports are virtually always tracked with a database, which may or may not be integrated with an automated test tool. Most organizations do either a test plan or a test design, as the information within them overlaps. The test plan includes management planning information (staffing needs, training needs, schedule) that is often covered in an overall project plan. Skipping the test plan also works well when the test processes and environments are stable from one software release to the next. The test summary report is not always written; it may be a decision-making meeting at which all effected parties are represented. The trade-offs of time-to-market versus quality versus completeness of the product are discussed, and a decision is made as to what software can go into production when. The test item transmittal report is almost never used; it is most applicable when an outside organization is delivering "tested" software to a customer.

Detailed definitions of all of these test documents are contained in IEEE Standard 829-1998.

TEST MANAGEMENT (PART OF THE ACTIVITY)

Test management includes all of the normal project management activities for the test aspects of a project, such as:

- Estimating schedules
- Planning for staffing and training
- Identifying and planning tasks
- Monitoring the execution of the plans and replanning based on the results

A standard management technique that is particularly needed in testing is the concept of *management reserve.* This is where a manager has resources that are available, but held back and not allocated until something goes wrong with the existing plans. Metrics and measurement programs are a useful tool for test management. They can be used to evaluate the status and effectiveness of both the process and the product being tested.

REFERENCES

Myers, Glenford, *The Art of Software Testing,* 1979.

IEEE Std 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* IEEE, 1998.

IEEE Std 829-1998, *Standard for Software Test Documentation,* IEEE, 1998.

IEEE/EIA Std 12207.0-1996, Industry Implementation of Int. Std. ISO/IEC 12207:95, *Standard for Information Technology-Software Life Cycle Processes,* IEEE, 1996.

Software Engineering Body of Knowledge (SWEBOK), IEEE, 2004.