|  |  |  |  |
| --- | --- | --- | --- |
| Title | Software Verification & Validation Strategy | | |
| **Distribution** | Software Archive | | |
| **Author** | Paul Li; | **Date** | 2017-5-2 |
| **Review** |  | **Date** | See review chapters |
| **Approved** |  | **Date** |  |
| **Remarks** |  | | |

Contents

[1 Introduction 4](#_Toc481678627)

[1.1 Scope 4](#_Toc481678628)

[1.2 Objectives 4](#_Toc481678629)

[2 Roles and responsibilities 4](#_Toc481678630)

[3 Defining test specifications 5](#_Toc481678631)

[4 V & V Phases and Methods 6](#_Toc481678632)

[4.1 Static analysis 7](#_Toc481678633)

[4.2 Unit-Test 7](#_Toc481678634)

[4.2.1 Requirements 7](#_Toc481678635)

[4.2.2 Methods 7](#_Toc481678636)

[4.2.3 Strategy & Planning 9](#_Toc481678637)

[4.2.4 When to write test code 10](#_Toc481678638)

[4.2.5 Artefacts & Delivery 10](#_Toc481678639)

[4.2.6 Reporting & metrics 10](#_Toc481678640)

[4.3 Subsystem Integration Test 11](#_Toc481678641)

[4.3.1 Requirements 11](#_Toc481678642)

[4.3.2 Methods 11](#_Toc481678643)

[4.3.3 Strategy & Planning 12](#_Toc481678644)

[4.3.4 When to write test code 12](#_Toc481678645)

[4.3.5 Artefacts & Delivery 12](#_Toc481678646)

[4.3.6 Reporting & metrics 12](#_Toc481678647)

[4.4 Code review 13](#_Toc481678648)

[4.4.1 Requirements 13](#_Toc481678649)

[4.4.2 Methods – techniques 13](#_Toc481678650)

[4.4.3 Check List 14](#_Toc481678651)

[4.4.4 Artefacts & Delivery 15](#_Toc481678652)

[4.5 System Integration Test 15](#_Toc481678653)

[4.5.1 Requirements 15](#_Toc481678654)

[4.5.2 Methods 15](#_Toc481678655)

[4.5.3 Strategy & Planning 15](#_Toc481678656)

[4.5.4 Artefacts & Delivery 16](#_Toc481678657)

[4.5.5 Reporting & metrics 16](#_Toc481678658)

[4.6 System Validation Test 16](#_Toc481678659)

[4.6.1 Requirements 16](#_Toc481678660)

[4.6.2 Methods 16](#_Toc481678661)

[4.6.3 Strategy & Planning 16](#_Toc481678662)

[4.6.4 Artefacts & Delivery 16](#_Toc481678663)

[4.6.5 Reporting & metrics 16](#_Toc481678664)

[5 Test Process 16](#_Toc481678665)

[5.1 Design Review 17](#_Toc481678666)

[5.2 Unit-test 18](#_Toc481678667)

[5.3 Subsystem integration test 18](#_Toc481678668)

[5.4 Static analysis 20](#_Toc481678669)

[5.5 Code review 20](#_Toc481678670)

[5.6 System Integration test 21](#_Toc481678671)

[5.7 System validation 22](#_Toc481678672)

[Revision History 23](#_Toc481678673)

[6 Review 23](#_Toc481678674)

[References 24](#_Toc481678675)

# Introduction

## Scope

This document provides a guideline for testing of SW projects. For verification is intended the test activity that takes place at the end of interim development stages to ensure that progress is correct in relation to the intended design. For validation is intended the test activity that takes place when a deliverable has been implemented to ensure that it meets its approved requirements.

Both verification and validation aim to ensure that the deliverables are of a satisfactory a high level of quality.

Verification and validation activities are commonly referred to as tests.

## Objectives

These documents gives an overview of the steps and deliverables for the all design process, but does not specifically define how to verify and validate the products itself. This document gives a line to follow for validation and verification of software parts.

# Roles and responsibilities

This strategy defines a number of roles for development engineers working on the project as follows. Within a project, a development engineer may have more than one role where they are involved with several components / subsystems.

Responsibilities and requirements applicable to general verification and validation aspects are listed with each role. In addition to these, specific responsibilities are defined in the different test phases.

|  |  |
| --- | --- |
| **Role** | **General Responsibilities / Requirements** |
| Project Leader | Responsibilities: Overall responsibility for the activity scope. Accepting or rejecting deliverables. Reporting key development metrics.  Requirements: Technical leadership and the authority to accept or reject deliverables.  Good knowledge of the development process and quality procedures. Good knowledge of the requirements. |
| Developer | Responsibilities: Development of the work package from requirements analysis to design, implementation and testing. (Where there is more than one developer assigned to a work package, the first named person has responsibility for completion of these tasks.) Development of tests Informing the Approver when activities are ready for review or approval. Reaching agreement with Approver for scope of any rework.  Requirements: Professional competence in hardware engineering (design and implementation) for the specified domain.  Good knowledge of the development process and quality procedures. Good knowledge of verification and validation techniques. Good knowledge of the work package requirements. |
| Approver | Responsibilities: Verification and validation of work package artifacts at each phase. Checking adequacy of test coverage. Collecting or producing key verification and validation metrics. Allowing development process to continue, else recommending rework where appropriate. Reaching agreement with Developer for scope of any rework.  Requirements: Professional competence in hardware engineering (design and implementation) for the specified domain.  Good knowledge of the development process and quality procedures. Good knowledge of verification and validation techniques. Good knowledge of the work package requirements. |
| QA Supporter | Responsibilities: Ensure work package team understands the necessary development process and their own QA responsibilities. Providing guidance on development and quality processes. Feeding back suggestions for process improvements. Arbitrate in discussions regarding execution of process activities.  Requirements: Professional competence in hardware engineering (design and implementation) for the specified domain.  Good knowledge of the development process and quality procedures. Good knowledge of verification and validation techniques. |

# Defining test specifications

The test specifications must include the following information:

|  |  |
| --- | --- |
| Goal | * The aim of the test. This will define or reference a particular requirement or normative, where the aim of the test is to check that this has been achieved. * For simplicity in defining test specifications, each test should aim for a single goal. * If this appears that a test cannot be described by a single goal then it probably means the test can actually be decomposed further into additional lower level test branches. |
| Acceptance criteria | * A set of expected results or conditions that define when a goal is successfully achieved. |
| Test cases | * The particular use that a test exercises. There should be at least one, but in many cases there will be several. * For a functional goal, this is all the possible use cases that a particular functionality offers a user. |
| Test scenarios | * The different conditions for each test case. There should be at least one per test case, but in practice each test case is likely to have several different scenarios to sufficiently cover the possibilities of valid and invalid use. * Test scenarios should detail the steps required to achieve the test. * The number of steps and the level of detail they describe should be enough to enable the test to be understood and implemented, rather than be a complete programmatic definition |

# V & V Phases and Methods

Due to different products has different scope and situations. This document only provide the common idea for each verification phases.

* Unit-test
* Integration-test
* Validation-test

The reviews of the following document are parts of verification procedures:

* Requirement specification
* Software architecture design description
* Detailed design if available
* Unit test report
* Code review report

The development of SW test harness functionality should be done at those level with the actual Work Package activities as shown in Figure 1.



Figure 1: Implementation and test harness development activities

Execution of the required test phase can begin once both the implementation activity and corresponding test harness development are complete.

Developing the test harness in this manner offers a pragmatic way to increment functionality in steps relevant to the sequence of test phases:

* Satisfying external dependencies for compilation and static analysis.
* Providing a platform to execute subsystem unit test cases.
* Providing a platform to execute subsystem integration test (functional test) cases.
* Test system integration – mainly for ME & HW & SW integration test.
* Test system validation

## Static analysis

SW Projects shall use external tools like PC-Lint to do static analysis. Which shall also follow internal policy.

The rquirements could be as below:

* All of the error message generated by static analysis tools shall be fixed
* All of the warning message shall be analyzed, and if not generated by tool, shall take actions to fix those warnings, if could not, shall add comments in code to notify the message, and shall review through the comments when do code review.
* All the messgae generated by analyze tools also need to be cared or fixed, and shall have it added in code comments if could not be fixed as the review source.
* When newly changed code, need to do static analyzation again.

## Unit-Test

It is project specific to define which part of the source code shall have unit-test and also to reach which level; this chapter only mentions the guide line for unit-test.

### Requirements

Normally, for all Non-HW-Related functions and methods, shall have 100% coverage that means for each function/method, we could use the follow aspects to check whether the unit-test has the 100% coverage:

* Control flow check – detect control flow faults, general logic or sequencing errors
  + Statement coverage – each statement in the code is executed at least once
  + Branch coverage – achieve 100% statement and path coverage
  + Interface coverage – detect faults during both unit testing and component integration testing
* Data flow check – test the implementation based on the use of its data: how it is defined, how it is used; how it is destroyed and whether this is done in the right order.
* Positive check – with validate input, it will output right data
* Negative check – with invalidate input, it will output error data
* Fault-based check – demonstrate that faults are not in a program

For some test functions, it will not cover all the aspects mentioned above, and it is the unit-tester’s responsibility to decide which aspect shall be taken into consideration, and the related technology shall be implemented.

Some functions in project may not be able to be tested; it shall be mentioned in test plan:

* Purely hardware functions like AD/Timer/IIC, may need review to check the realization of code, and also need to be tested in integration test or validation test later. This shall be expressed in test plan.

For all those functions that shall be tested, shall also obey those rules below:

* For the function in other subsystem was called, need to create sub function to realize the branch coverage and decision coverage. May one simple subsystem or just one simple function needs to be created to help the test.
* For some functions that were tested when called in other functions, and also fulfil the coverage requirements need not to be tested again. But shall be mentioned in Test Plan.
* The unit-test shall be taken in a simple project that means it shall have some simple subsystem to help the test.

### Methods

It is important not to confuse structural (white-box) testing with debugging. The techniques appear similar because they both involve dealing with software defects and looking at the code, but they’re very different in their goals. The goal of structural testing is to find defects. The goal of debugging is to fix them. They do overlap, however, in the area of isolating where and why the defect occurs.



A white box provides the information necessary to test all the possible pathways. This includes not only correct inputs, but incorrect inputs, so that error handlers can be verified as well. This provides several advantages:

* you know how the box handles errors
* you can usually write tests that verify all code pathways
* the unit test, being more complete, is a kind of documentation guideline that the implementer can use when actually writing the code in the box
* resource dependencies are known
* Internal treatments can be inspected.

#### Nominal Test

The purpose of nominal unit testing is to make a general verification of the behaviour by executing the nominal cases. Nominal cases are those that correspond to the nominal behaviour of the component or the internal function under test and do not include exceptional behaviours and error conditions. These tests are performed on any component when created.

#### Structural Test

Structural test is another name for dynamic white-box testing.

Nominal test is a test with “normal” input values (they are selected in the range of allowed values)

Structural tests: the goal is to exercise all the branches of the code. Developer needs to analyze the code in order to find the “interesting values” which lead to passing through in branches or decisions.

The steps to perform white-box unit testing of a component are:

1. for every component, a suite of test cases is predefined including
   * initial conditions
   * inputs/steps
   * expected result
2. the unit tests are performed
3. A comparison between expected and actual results is done. A discrepancy results in the tester reconsidering the scenario, the input data, the conditions or the expected result. This step is repeated until all discrepancies are resolved.
4. the result is recorded
5. unit testing is complete when the target level of coverage has been reached

Selecting test cases is an important task in unit test and **equivalence classing** is the means to do it. Equivalence classes are the ranges of input values for which the behavior of the code is equivalent. Equivalence classing is the process of reducing the infinite set of possible test cases into a much smaller, but still equally effective, set. Instead of testing every possible input value, it is sufficient to run the test with only a subset of values from equivalence classes. Tests are run with input values inside the equivalence class range, taking each boundary value and one value between the boundaries.

When looking for equivalence classes, think about ways to group similar inputs, similar outputs and similar operations of the software. These groups are your equivalence classes.

#### Boundary tests

Boundary tests deal with boundary values which are bounds of any defined range of data. If the software can operate on the edge of its capabilities, it will almost certainly operate well under normal conditions. The unit tests should check the behavior of the code for values right before, on and right after the boundary values. These unit tests may also check the behavior of the code with numerical data at their maximum physical capabilities (useful when dealing with machines portability/computability constraints).

Examples:

* Maximum negative, maximum positive, and 0 inputs or outputs;
* Input or output strings at size limits, 1 character beyond size limits, the empty string, and strings of 1 character;
* Input or output numeric values at size limits and 1 beyond size limits;
* Empty input files and files with one character in them;
* For a range of values bounded by a and b, test (a-1), a, (a+1), (b-1), b, (b+1);
* If internal program data structures have boundaries (e.g., buffer size, table limits), use input data to exercise structures on boundaries.

#### Structural Coverage

Structural coverage includes statement coverage, branch coverage and decision coverage.

The goal of **statement coverage** is to execute every statement in the code at least once. Statement coverage tells you if every statement is executed but it does not tell you if you’ve taken all the paths through the code.

**Branch coverage (**path testing) attempts to cover all the paths in the code. Feeding the code with different parameter values activates the branches.

**Decision coverage** consists of covering all the decision alternatives (each conditional statement executed at least once each way). Decision coverage testing takes the extra conditions on the branch statements into account. If you test for all possible conditions, you will achieve branch coverage and therefore achieve statement coverage.

Structural coverage is performed after structural unit testing. The steps to perform structural coverage of a component are

1. at the end of structural testing a measure of the structural coverage is taken
2. for the parts not covered according to the assigned test class the tester determines why and either
   * completes the test plan
   * completes the testing (forgot to run some test cases)
   * Modifies the source code because it is not desirable or not specified.

### Strategy & Planning

Software unit testing is an integral part of an efficient and effective strategy for testing software. Unit Tests are performed during the “Code and Unit Tests” activity during the Construction iteration. There are some basic requirements which should always be met. In particular unit tests activities must be **reviewable**, **repeatable**, and **achievable**. Commonly, unit testing will be recorded in Unit Test Report, with guidance provided by a Unit Test Plan that identifies the required unit testing activities.

Some issues which should be considered when evaluating a unit testing strategy are:

* Has a policy been defined in the project regarding the formalism, the documentation, and the coverage?
* Does it relate to other levels of testing to give an efficient and effective overall strategy?
* Have the needs of units which are particularly critical, complex, or risky been considered?
* Will the documentation be reviewable, achievable, and are the unit tests repeatable?

Some other features must have been covered early in the project:

* Tools, environments, and extra resources to be used for unit tests.
* Criteria for considering automatic/manual/embedded testing deployment in regard to delays/costs/criticism constraints and product life cycle
* Criteria defining a typical test strategy according to the kind of work in progress: new functions, heavy changes on existing code, critical functions, real-time, multithreading …
* Criteria for unit tests acceptance.

Furthermore, a Unit Test Plan may be required according to:

* The diversity of the components and expected test requirements
* The size of the development team (when several developers work on the same project, the Unit Test Plan will ease coordination between them and will prevent integration problems)
* Department level rules.

The unit test plan is a document that describes the **objectives, scope, approach,** and **focus** of a software testing effort. The process of preparing the unit test plan is a useful way to estimate the efforts needed for the unit tests.

### When to write test code

**Developer must code the unit tests before****or even during the phase of coding; it is part of the development.**

The advantages of this are the following:

* Decreases the over cost of coding the tests
* Take advantage of the reflex ion phase = to establish the tests cases;
* Immediate detection of bugs;
* Developer never forgets the objectives;
* Code is simpler and clear, it is naturally testable.

Basically, TDD (Test Driven Development – TDD) isa low level design technique. TDD expects the low level design is made thanks to unit tests writing.

**Unit test and code always come together.**

You must maintain and increase the code by:

* Modification of code: Change the corresponding unit tests;
* Correction of a bug: First write a test that highlights the bug, then fix the bug = regression test set improvement.

### Artefacts & Delivery

Unit test artifacts consists of

* Unit Test Plan (optional but recommended)
* Unit Test code
* if needed, test harnesses and test stubs
* Unit Test Reports.

The entire set of unit test documentation is stored and archived in the configuration management tool alongside the source code. Unit test artifacts should be considered a “living” work product and changes to it over time and experience are expected.

**When a developer delivers a component which underwent Unit Tests, he also delivers the corresponding Unit Tests.**

It is also a good thing that the code and the Unit test come together in the configuration management tool, so:

**Before a delivery of a component, the developer has to run every Unit Test of this component and check that the results are the ones expected (“succeed”).**

### Reporting & metrics

For the project follow-up, UT gives a clear view on how the development progresses (mostly during a new development). Developers write unit tests before coding a function; if a unit test is NOK, then the function is not completely implemented. Code coverage results give several information:

* Between 2 versions of a product, the ratio of general coverage should increase. If it is not the case, new source code has not been tested. On an existing project, this could be justifiable but it is to check.
* An important coverage ratio shows that the application is based on a solid architecture. It is difficult to define an acceptable coverage ratio, as it depends mostly on the project, its size, its complexity, its GUI and the investment (budget-wise and planning-wise). For a project starting from scratch, one may be looking for a minimum of 60%.

Code coverage is the percentage of code line exercised by the unit tests.

**Code coverage is not a measure of quality of the software; it’s just the quantity of code checked by the unit tests.**

For the project follow-up and for capitalization as well, the following indicators can be measured:

* The code coverage (CC) by unit tests
* The code coverage by automatic unit tests ( ≤ CC)
* Time/effort spent for unit tests (this does not include time for debug and corrections).

## Subsystem Integration Test

When all subsystem is finished, it shall be integrated into Project. And the test based on the integration shall be implemented. According to V&V modle, the integration test plan shall be documented before the implementation of code. And also the integration test shall be implemented when all the subsystem code is finished and integrated into one project.

### Requirements

Subsystem integration test is grey-box testing for projects.

Base on V&V modle, subsystem is implemented as system architechure design. And subsystems also has the detail requirements and public interfaces. Integration test is just required to care about the following things:

1. Is the requirements of the subsystem validated.
2. Did the subsystem has the required interface base on architechure design, and all the interfaces work OK.
3. Do we have some unconspicuous requirements(not specific in docs) also need to test.

So integration test mainly care about the requirements and interfaces. The test first needs to check whether this all the requirements is realized. And then mainly test the public interfaces will be called by other subsystems.

If there is any changed/new requirements/code implemented in current project, subsystem integration needs to be done again to make sure this integration test pass well. And all the integration test shall be down by a different developer to

### Methods

#### Positive, valid cases

Possibly the most intuitive method, positive testing requires knowledge of valid inputs for the unit under test, usually taken from the requirements document. Test cases are then generated and run using these values and should all pass.

#### Negative, invalid cases

Negative testing requires knowledge of invalid inputs to the unit under test, which are often not specified in the requirements. Test cases are generated and run using invalid inputs and the results are examined based on how the test case failed, *i.e.*, appropriate error messages are generated, exceptions are properly handled, and the invalid input is rejected.

#### Fault‐Based Testing Methods

Fault‐based testing methods attempt to demonstrate that faults are not in a program. While it is not possible to show an implementation is 100% error free, methods such as risk assessment, failure analysis, attack patterns, fault injection, and mutation testing are used to find specific types of faults in the implementation and aim to show if those specific faults were to exist, they would have been found.

#### Grey‐box Testing Methods

Grey box testing is a combination of black box methods with white box techniques. As in white box, testers use some knowledge of the underlying code to analyze test results from block box testing methods. These methods are especially useful in testing interfaces and involve creating tests based on communication behavior and sequence diagrams.

#### Regression Testing

Regression testing attempts to show that changes made to existing software have not introduced bugs. Knowledge of the changes to the implementation as well as system interactions is utilized in regression testing. The methods include retest all, retest changed parts, and retest based on risk.

### Strategy & Planning

Software integration testing is an integral part of an efficient and effective strategy for testing software. There are some basic requirements which should always be met. In particular unit tests activities must be **reviewable**, **repeatable**, and **achievable**. Commonly, integration testing will be recorded in Test Report, with guidance provided by a Test Plan that identifies the required testing activities.

Some issues which should be considered when evaluating an integration testing strategy are:

* Has a policy been defined in the project regarding the formalism, the documentation, and the coverage?
* Does it relate to other levels of testing to give an efficient and effective overall strategy?
* Will the documentation be reviewable, achievable, and are the tests repeatable?
* Is there some function/method tested in unit- test and need not to be tested in integration test.

Some other features must have been covered early in the project:

* Tools, environments, and extra resources to be used for tests
* Criteria for considering automatic/manual/embedded testing deployment in regard to delays/costs/criticism constraints and product life cycle
* Criteria defining a typical test strategy according to the kind of work in progress: new functions, heavy changes on existing code, critical functions, real-time, multithreading …

The test plan is a document that describes the **objectives, scope, approach,** and **focus** of a software testing effort. The process of preparing the test plan is a useful way to estimate the efforts needed for the tests.

### When to write test code

**Another developer (not subsystem developer) must define the tests before the phase of coding after the architecture design. Test plan shall be finished when architecture is designed and all tests shall be realized when subsystem is finished and unit-test passed.**

You must maintain and increase the code by:

* Modification of requirements/interface: Change the corresponding tests;
* Correction of a bug: First write a test that highlights the bug, then fix the bug = regression test set improvement.

### Artefacts & Delivery

Integration test artifacts consists of

* Integration Test Plan (optional but recommended)
* Integration Test code
* if needed, test harnesses and test stubs
* Integration Test Reports.

The entire set of test documentation is stored and archived in the configuration management tool alongside the source code. Integration test artifacts should be considered a “living” work product and changes to it over time and experience are expected.

**When a developer delivers a component which underwent Integration Tests, he also delivers the corresponding Integration Tests.**

It is also a good thing that the code and the Integration test come together in the configuration management tool, so:

**Before a delivery of a component, the developer has to run every Integration Test of this component and check that the results are the ones expected (“succeed”).**

### Reporting & metrics

Integration test will give the following results:

* All the requirements (include unconspicuous requirements) are fulfilled or not, is there any tests need to be done in next step.
* All interfaces are tested, and could be used by other subsystem to implement.
* Time/effort spent for unit tests (this does not include time for debug and corrections).

## Code review

### Requirements

Code reviews are a simple idea. People are much better than machines at finding defects, especially for defects of omission (missing conditions, missing checks, etc.). Unlike machines, people make assumptions and have bias, which is what makes code reviews challenging. If code reviews are approached with the belief that everything works, they will not be successful.

This chapter presents a few simple techniques that can help guide a review to remove as much bias as possible.

### Methods – techniques

#### Code Reviews for Bug Fixes

The majority of defects injected while fixing bugs occur in areas far removed from the change. It is the impact of the change, not the change itself, which is responsible for the defect. Due to this, only focusing the code review on change itself is not enough. The impact of the change must be determined and the change must be reviewed in the context of that overall impact. The major types of impact to check include control flow and data flow. Control flow involves the statement level execution of the software around the change (such as decisions, function calls, etc.). Data flow involves how variables, messages, and values are passed through the program. Each code review for a defect should check the control flow and data flow paths that involve the change.

This is done by the following steps:

1. Identify all other functions that call the changed function, as well as all functions that this function calls.
2. Verify that the control logic is not negatively impacted by the change. For example, a routine used to be called in a later function but, due to the change, it now is skipped. This behavior change could be unintended and result in a new defect.
3. Identify all values, messages, global variables, etc. that are impacted by the change.
4. Verify that all values passed in, used, modified and created are not negatively impacted by the change. This includes return values, global state, messages, etc. For example, a state value could be changed to fix a defect. Changing this state variable could cause other code to process other, unintended code.

When reviewing new code, it is important to have two perspectives. The first is a narrow perspective, focusing on the current, specific uses of the code in the system. This can be accomplished by a top down approach, starting with the major “driver” routines and verifying it and all the functions it calls are correct. This technique is further described in the Top Down section below. The second perspective is a broad perspective, looking at each of the new methods with respect to all of its callers and its overall requirements. This technique starts at the bottom, with each routine individually checked against all callers and potential future uses that might arise. This is further described in the Bottom Up section below.

#### Top Down

This technique involves identifying each of the *driver* routines. In procedural design these routines are often the “main” routines, while in Object‐Oriented design these are often the “action” routines. In both cases, these are frequently the routines that call the most functions or are the largest overall routines.

Each of these routines is then reviewed by following each path through the code. This includes going into the called functions, checking parameters and return values. Since these routines are large and call many other routines, this will take time. But, on average, the most complex routines contain the largest number of defects, so focusing on these is the largest payoff.

In addition, key data flow paths should be checked. Key external data flow can focus on state values, global variables, databases, messages and other forms of persistent storage. Key internal data flow mostly revolves around parameters and object variables.

#### Bottom Up

This technique involves looking at each function in reference to its overall usage in the system and any potential future uses that it might have. The first step involves determining all functions that call this specific function. Once that list is created, each call and return is checked to determine if anything is incorrect. A common source of errors involves improperly propagating return values back to the caller, especially in deeply nested functions. In addition, each function should be checked against its intended use. For example, a function that parses strings may be required to handle Unicode strings. The implementation, for example, may range check for certain values only in English. This defect can easily make it to system testing, as many development teams do not test different languages themselves.

### Check List

This chapter mainly lists the common check point for code review. Some check points may directly give you idea what/how to check the codes. For projects, code review may involve many things that not listed below, and that may be different for different project, some of them may not be required for different project also.

Code Purpose

* Verify that the purpose of every program part is explained.
* Verify that every program part originate from a requirement and/or a specification
* Verify that no “dead code” exists.
  + Inaccessible or commented functions
  + Large chunks of commented code

Complete

* Ensure that the requirements, specifications, and design are completely covered by the code:
  + All needed inputs are included.
  + All specified outputs are produced.

Initialize

* Verify that all variables are initialized.
* Verify that objects are created before pointers to them are used.

Logic

* Verify that program sequencing is proper.
* Verify that all loops are properly initiated, incremented, and terminated.
* Check the start and end values to avoid one-off errors.
* Verify that type casts are correct and necessary.
* Verify that all switch statement has an break call and that there is an default statement

Functional use

* Verify that all functions, procedures, or objects are fully understood and properly used.
* Verify that all externally referenced abstractions are precisely defined.
* Check that parameter passing by reference is correctly used.

Assertions

* If applicable: Verify that erroneous data inputs are covered.

Memory

* Verify that memory is correctly allocated, i.e. malloc() only used at startup and not in code executing at runtime
* Verify that memory is correctly deallocated, i.e. free() is not used in code executing at runtime
* Verify that the Big Three (destructor, copy constructor, and assignment operator) always are explicitly declared.
* Check that the owner of an object is clear and that the owner both creates and destroys an object correctly.
* Make sure that objects are destroyed at the correct time, so no other objects are still using the destroyed object.

Static analysis

* Verify the PC\_Lint follow the MISRA 2012 rule. All the warning and error messages are well handled.

Compile

* Verify that the code compiles without errors and warnings for all platforms (GCC, VC)

Comments

* Verify that the code is commented in an appropriate way.
* Verify that the file headers are updated
* Verify that the method headers are complete and understandable
* All comments are translated to English

Project has its own checkpoints most of them listed above and also some specific checkpoints. This will be documented as a template doc.

### Artefacts & Delivery

For code review, we need to cover some aspects, and add all comments/issuses in the review checklist documents, when all this is done. It shall consists of

* Code review check list
* Code review comments

When code review is done, may need to modify some codes and comments. If no critical/big change, could have no regression test (this shall be decided after review). If not, the regression test may need to be done immediately.

## System Integration Test

### Requirements

Integration test of subsystem is done before the test. This document will guide how to do integration test, and the detail cases. This step only checks whether the ME, HW & SW could work together well or not. The test mainly cover the communication and in/output.

### Methods

#### Black-box testing

Testing based on knowledge of the entity being tested from its expected behaviour and as observed from its boundaries. The aim is to test the entity against its specifications for requirements, interfaces and data.

Applicable techniques:

* Requirements based testing.
* Boundary-value testing.
* Positive & negative test cases.
* Equivalence class testing.
* Boundary-value testing.
* State based testing.
* Use case / scenario testing.
* Operational profile testing.

### Strategy & Planning

The test needs to be hold as black-box testing. It shall be done after all subsystem integrated.

### Artefacts & Delivery

The test plan documents, mainly describe the test cases, actions, and expect result.

The test report need to have test results and check whether it is ok or not.

### Reporting & metrics

Integration test will give the following results:

* Does the ME, HW & SW work together well or not?

## System Validation Test

### Requirements

Validation test needs to be done when system is finished and integration test is done according project plan. The validation has to be started after all design reviews and module tests are finished. This document will guide how to do validation test, and the detail cases. This step only check whether the system has fulfilled the requirements includes safety requirements and functional requirements.

### Methods

#### Black-box testing

Testing based on knowledge of the entity being tested from its expected behaviour and as observed from its boundaries. The aim is to test the entity against its specifications for requirements, interfaces and data.

Applicable techniques:

* Requirements based testing.
* Boundary-value testing.
* Positive & negative test cases.
* Equivalence class testing.
* Boundary-value testing.
* State based testing.
* Use case / scenario testing.
* Operational profile testing.

### Strategy & Planning

The test needs to be hold as black-box testing. It shall be done after all system is finished. The test plan normally shall be prepared when requirements are ready.

### Artefacts & Delivery

The test plan documents, mainly describe the test cases, actions, and expect result.

The test report need to have test results and check whether it is ok or not.

### Reporting & metrics

Validation test will give the following results:

* All the requirements (include unconspicuous requirements) are fulfilled or not, is there any tests need to be done in next step. Or done on other steps.

# Test Process

The procedure for verification and validation is split a number of test phases that are executed at the different stages of the development cycle.

The overall objective of each phase is to allow the detection and removal of defects. It is important to note that testing cannot show the absence of defects.

Each phase is described in terms of the following definitions:

|  |  |
| --- | --- |
| Phase | Name of the phase that the test process is applicable to. |
| Objective | Objective that the test phase addresses. |
| Purpose | More detailed description of what the phase is for. |
| Basis | Artifacts or other entities that are used or referred to by the test phase. |
| Scope | Entities that the phase needs to be run on. |
| Entry criteria | Prerequisite artifacts or conditions that must be available or completed before the phase is started. |
| Exit criteria | Artifacts (documents, decisions, metrics etc) or conditions that must be available and complete before the phase ends. |
| Test methods | Description of the test methods that are used in the phase. |
| Regression | Applicable regression test strategy. |
| Defect responsibility | The types of defects that are specifically targeted for detection by the phase. |
| Tools | Description of tools (where applicable) used during the phase. |
| Accountable | The person ensuring that the phase was run correctly. |
| Responsible | The person / people tasked with running the phase. (Refer to role definitions for the activity responsibilities that are applicable to all test processes.) |

## Design Review

|  |  |
| --- | --- |
| Phase | Design Review |
| Objective | Checking all design document |
| Purpose | Manually detect and remove design defect |
| Basis | Design document  Requirement specification |
| Scope | All design |
| Entry criteria | Requirement specification  Design description |
| Exit criteria | Design reviewed |
| Test methods | Na |
| Regression | Review changes |
| Defect responsibility | Error during design phase |
| Tools | None |
| Accountable | Approver |
| Responsible | Developer  Approver |

## Unit-test

|  |  |
| --- | --- |
| Phase | Component integration testing/ functional testing |
| Objective | Mainly goal to get the code coverage.  Mainly to check the function of each function. |
| Purpose | Manually detect and remove design defect |
| Basis | Source code. |
| Scope | All separate function in that subsystem |
| Entry criteria | All new and changed code. |
| Exit criteria | Test report. |
| Test methods | See test chapter.2 |
| Regression | Retest the related functions |
| Defect responsibility | Developer |
| Tools | Debug tools |
| Accountable | Project Leader |
| Responsible | Developer  Approver |

## Subsystem integration test

|  |  |
| --- | --- |
| Phase | Subsystem Integration test |
| Objective | Checking subsystem requirements are met.  Checking subsystem functional requirements  Checking subsystem’s interface |
| Purpose | Validate the subsystem against its requirements. |
| Basis | Source code.  Subsystem requirement specification |
| Scope | Subsystem code package. |
| Entry criteria | Module test is done |
| Exit criteria | Test report |
| Test methods | See integration test chapter 4.3 |
| Regression | Retest the related functions and module test. |
| Defect responsibility | Developer |
| Tools | Laboratory instrument  Debug tools |
| Accountable | Approver/Tester |
| Responsible | Developer  Approver |

## Static analysis

|  |  |
| --- | --- |
| Phase | Static analysis. |
| Objective | Checking logical construction. |
| Purpose | Automatically detect and remove defects from source code.  Protect against overly complex constructs. |
| Basis | Source code.  Static analysis rules (MISRA 2004, detail is described in doc [14]) |
| Scope | All new and changed code. |
| Entry criteria | Modules containing area being analysed are locally compiled and built successfully with test harness.  Static analysis is tool set up for area being analysed. |
| Exit criteria | Static analysis report and metrics collected.  Test phase report.  Decision to rework or proceed. |
| Test methods | Static analysis (including cyclomatic complexity).  See chapter 4.1 |
| Regression | Retest all. |
| Defect responsibility | All implementation defects or unsafe constructs guarded against by static analysis rules. |
| Tools | Static analysis tool. |
| Accountable | Approver |
| Responsible | Developer  Approver |

## Code review

|  |  |
| --- | --- |
| Phase | Code Review. |
| Objective | Checking logical construction. |
| Purpose | Manually detect and remove defects from source code.  Ensure code is maintainable.  Ensure the static analysis is right handled. |
| Basis | Source code.  Code Review guideline. |
| Scope | All new and changed code. |
| Entry criteria | Module being reviewed is locally compiled and built successfully with test harness.  Design descriptions are available.  Static analysis of area being reviewed complete.  Unit tests of area being reviewed complete. |
| Exit criteria | Code review report.  Decision to rework or proceed. |
| Test methods | Review against coding guidelines.  See chapter 4.4 |
| Regression | Review changes. |
| Defect responsibility | Insufficient unit test coverage.  Non-conformances to coding guidelines. Implementation defects that exist inside a method or object. |
| Tools | None |
| Accountable | Approver |
| Responsible | Developer  Approver |

## System Integration test

|  |  |
| --- | --- |
| Phase | Integration testing  (also as HW & SW integration test) |
| Objective | Checking all SW function requirements related with HW; |
| Purpose | Detect implementation defects by covering functions need HW & SW work together. |
| Basis | Source code with related HW.  Test harness (Integration test cases) |
| Scope | System source code |
| Entry criteria | Subsystem integration and module-test passed.  Static analysis of area being tested passed.  Code review done. |
| Exit criteria | Test phase report.  Decision to rework or proceed. |
| Test methods | See integration test chapter 4.5 |
| Regression | Retest all. |
| Defect responsibility | All implementation defects not found in a module from static analysis. |
| Tools | Unit Test creator (if available) |
| Accountable | Approver |
| Responsible | Developer  Approver |

## System validation

|  |  |
| --- | --- |
| Phase | Product validation |
| Objective | Checking requirement are met |
| Purpose | Validate the system against its requirement |
| Basis | Requirement specification |
| Scope | Whole source code with related HW. |
| Entry criteria | System integration test complete |
| Exit criteria | Test report  Decision to rework or proceed |
| Test methods | See validation test chapter 4.6 |
| Regression | Retest the new changed and related functions. (retest all) |
| Defect responsibility | All defects resulting from missing or incorrect functionality in the product |
| Tools | Test equipment |
| Accountable | Project Leader |
| Responsible | Developer  Approver |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev.** | **Description of Version/Changes** | **Primary Author(s)** | **Date** |
| 00 | Initial revision. | Paul Li |  |

# Review

**Check list:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Yes | No | N.A. |
| 1. | Are all used abbreviations and technical expressions explained? | **X** |  |  |
| 2. | Are all referenced documents in the reference list? | **X** |  |  |
| 3. | Is the document clear, precise, unambiguous, maintainable and understandable? | **X** |  |  |
| 4. | Are all open issues transferred to the defects table? | **X** |  |  |
| 5. | Are all functional test declared and well descripted? | **X** |  |  |
| 6. | Is validation strategy declared and well descripted? | **X** |  |  |
| 7. | Is verification strategy declared and well descripted? | **X** |  |  |
| 8. | Are specific standard tests considered? | **X** |  |  |
| 9. | Are roles and responsible well defined? | **X** |  |  |
| 10. | Are all dependencies to other documents considered? | **X** |  |  |
| 11. | Are templates used in a correct manner? | **X** |  |  |

**Remarks:**

**Defects**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No... | Checkpoint | Description | Major Defect | done  Date |
|  |  |  |  |  |

# References

|  |  |
| --- | --- |
| Ref. | Document |
|  |  |