5

# Software Verification

Software verification has proved its effectiveness in the software world and its usage is increasing day by day. The most important aspect of software verification is its implementation in the early phases of the software development life cycle. There was a time when people used to say that "testing is a post-mortem activity where testers are only finding the damages already been done and making changes in the program to get rid of these damages." Testing primarily used to be validation oriented where the program was required for execution and was available only in the later phases of software development. Any testing activity which requires program execution comes under the 'validation' category. In short, whenever we execute the program with its input(s) and get output(s), that type of testing is known as software validation.

What is software verification? How can we apply this in the early phases of software development? If we review any document for the purpose of finding faults, it is called verification. Reviewing a document is possible from the first phase of software development i.e. software requirement and analysis phase where the end product is the SRS document.

Verification is the process of manually examining / reviewing a document. The document may be SRS, SDD, the program itself or any document prepared during any phase of software development. We may call this as static testing because the execution of the program is not required. We evaluate, review and inspect documents which are generated after the completion of every phase of software development. As per IEEE, "verification is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" [IEEE01]. Testing includes both verification and validation activities and they are complementary to each other. If effective verification is carried out, we may detect a number of faults in the early phases of the software development life cycle and ultimately may be able to produce a quality product within time and budget.

## 5.1 VERIFICATION METHODS

The objective of any verification method is to review the documents with the purpose of finding faults. Many methods are commonly used in practice like peer reviews, walkthroughs, inspections, etc. Verification helps in prevention of potential faults, which may lead to failure of software.

After the completion of the implementation phase, we start testing the program by executing it. We may carry out verification also by reviewing the program manually and examining the critical areas carefully. Verification and validation activities may be performed after the implementation phase. However, only verification is possible in the phases prior to implementation like the requirement phase, the design phase and even most of the implementation phase.

### 5.1.1 Peer Reviews

Any type of testing (verification or validation), even adhoc and undisciplined, is better than no testing if it is carried out by person(s) other than the developers / writers of the document with the purpose of finding faults. This is the simplest way of reviewing the documents / programs to find out faults during verification. We give the document(s) / program(s) to someone else and ask to review the document(s) / program(s). We expect views about the quality of the document(s) and also expect to find faults. This type of informal activity may give very good results without spending any significant resources. Many studies have shown the importance of peer review due to its efficiency and significance. Our thrust should be to find faults in the document(s) / program(s) and not in the persons who have developed them. The activities involved may be SRS document verification, SDD verification and program verification. The reviewer may prepare a report of observations and findings or may inform verbally during discussions. This is an informal activity to be carried out by peers and may be very effective if reviewers have domain knowledge, good programming skills and proper involvement.

### 5.1.2 Walkthroughs

Walkthroughs are more formal and systematic than peer reviews. In a walkthrough, the author of the document presents the document to a small group of two to seven persons. Participants are not expected to prepare anything. Only the presenter, who is the author, prepares for the meeting. The document(s) is / are distributed to all participants. During the meeting, the author introduces the material in order to make them familiar with it. All participants are free to ask questions. All participants may write their observations on any display mechanism like boards, sheets, projection systems, etc. so that every one may see and give views. After the review, the author writes a report about findings and any faults pointed out in the meeting.

The disadvantages of this system are the non-preparation of participants and incompleteness of the document(s) presented by the author(s). The author may hide some critical areas and unnecessarily emphasize on some specific areas of his / her interest. The participants may not be able to ask many penetrating questions. Walkthroughs may help us to find potential faults and may also be used for sharing the documents with others.

### 5.1.3 Inspections

Many names are used for this verification method like formal reviews, technical reviews, inspections, formal technical reviews, etc. This is the most structured and most formal type of verification method and is commonly known as inspections. These are different from peer reviews and walkthroughs. The presenter is not the author but some other person who prepares and understands the document being presented. This forces that person to learn and review that document prior to the meeting. The document(s) is / are distributed to all participants in advance in order to give them sufficient time for preparation. Rules for such meetings are fixed and communicated to all participants. A team of three to six participants are constituted which is led by an impartial moderator. A presenter and a recorder are also added to this team to assure that the rules are followed and views are documented properly.

Every person in the group participates openly, actively and follows the rules about how such a review is to be conducted. Everyone may get time to express their views, potential faults and critical areas. Important points are displayed by some display mechanism so that everyone can see them. The moderator, preferably a senior person, conducts such meetings and respects everyone's views. The idea is not to criticize anyone but to understand their views in order to improve the quality of the document being presented. Sometimes a checklist is also used to review the document.

After the meeting, a report is prepared by the moderator and circulated to all participants. They may give their views again, if any, or discuss with the moderator. A final report is prepared after incorporating necessary suggestions by the moderator. Inspections are very effective to find potential faults and problems in the document like SRS, SDD, source code, etc. Critical inspections always help find many faults and improve these documents, and prevent the propagation of a fault from one phase to another phase of the software development life cycle.

### 5.1.4 Applications

All three verification methods are popular and have their own strengths and weaknesses. These methods are compared on specific issues and this comparison is given in Table 5.1.

**Table 5.1.** Comparison of verification methods

| S. No. | Method | Presenter | Number of Participants | Prior preparation | Report | Strengths | Weaknesses |
|---|---|---|---|---|---|---|---|
| 1. | Peer reviews | No one | 1 or 2 | Not required | Optional | Inexpensive but find some faults | Output is dependent on the ability of the reviewer |
| 2. | Walkthrough | Author | 2 to 7 participants | Only presenter is required to be prepared | Prepared by presenter | Knowledge sharing | Find few faults and not very expensive |
| 3. | Inspections | Someone other than author | 3 to 6 participants | All participants are required to be prepared | Prepared by moderator | Effective and find many faults | Expensive and requires very skilled participants |

The SRS verification offers the biggest potential saving to the software development effort. Inspections must be carried out at this level. For any reasonably sized project, the SRS document becomes critical and the source of many faults. Inspections shall improve this document and faults are removed at this stage itself without much impact and cost. For small sized projects, peer reviews may be useful but results are heavily dependent on the ability and involvement of the reviewer. Walkthroughs are normally used to sensitize participants about the new initiative of the organization. Their views may add new functionality or may identify weak areas of the project.

Verification is always more effective than validation. It may find faults that are nearly impossible to detect during validation. Most importantly, it allows us to find faults at the earliest possible time and in the early phases of software development. However, in most organizations the distribution of verification / validation is 20/80, or even less for verification.

specific tailoring occurs in section 3 entitled 'specific requirements'. The general organization of the SRS document is given in Table 5.2.

**Table 5.2.** Organization of the SRS [IEEE98a]

| | |
|---|---|
| 1. | Introduction |
| 1.1 | Purpose0 |
| 1.2 | Scope |
| 1.3 | Definitions, Acronyms and Abbreviations |
| 1.4 | References |
| 1.5 | Overview |
| 2. | The Overall Description |
| 2.1 | Product Perspective |
| 2.1.1 | System Interfaces |
| 2.1.2 | Interfaces |
| 2.1.3 | Hardware Interfaces |
| 2.1.4 | Software Interfaces |
| 2.1.5 | Communications interfaces |
| 2.1.6 | Memory Constraints |
| 2.1.7 | Operations |
| 2.1.8 | Site Adaptation Requirements |
| 2.2 | Product Functions |
| 2.3 | User Characteristics |
| 2.4 | Constraints |
| 2.5 | Assumptions and Dependencies |
| 2.6 | Apportioning of Requirements |
| 3. | Specific Requirements |
| 3.1 | External interfaces |
| 3.2 | Functions |
| 3.3 | Performance Requirements |
| 3.4 | Logical Database Requirements |
| 3.5 | Design Constraints |
| 3.5.1 | Standards Compliance |
| 3.6 | Software System Attributes |
| 3.6.1 | Reliability |
| 3.6.2 | Availability |
| 3.6.3 | Security |
| 3.6.4 | Maintainability |
| 3.6.5 | Portability |
| 3.7 | Organizing the Specific Requirements |
| 3.7.1 | System Mode |
| 3.7.2 | User Class |
| 3.7.3 | Objects |
| 3.7.4 | Feature |
| 3.7.5 | Stimulus |
| 3.7.6 | Response |
| 3.7.7 | Functional Hierarchy |
| 3.8 | Additional Comments |
| 4. | Change Management Process |
| 5. | Document Approvals |
| 6. | Supporting Information |

IEEE Std 830–1998 Recommended Practice for Software Requirements Specifications – reprinted with permission from IEEE, 3 Park Avenue, New York, NY 10016 – 5997 USA, Copyright 1998, by IEEE.

### 5.2.3 SRS Document Checklist

The SRS document is reviewed by the testing person(s) by using any verification method (like peer reviews, walkthroughs, inspections, etc.). We may use inspections due to their effectiveness and capability to produce good results. We may conduct reviews twice or even more often. Every review will improve the quality of the document but may consume resources and increase the cost of the software development**.**

A checklist is a popular verification tool which consists of a list of critical information content that a deliverable should contain. A checklist may also look for duplicate information, missing information, unclear information, wrong information, etc. Checklists are used during reviewing and may make reviews more structured and effective. An SRS document checklist should address the following issues:

**(i) Correctness**

Every requirement stated in the SRS should correctly represent an expectation from the proposed software. We do not have standards, guidelines or tools to ensure the correctness of the software. If the expectation is that the software should respond to all button presses within 2 seconds, but the SRS states that 'the software shall respond to all buttons presses within 20 seconds', then that requirement is incorrectly documented.

**(ii) Ambiguity**

There may be an ambiguity in a stated requirement. If a requirement conveys more than one meaning, it is a serious problem. Every requirement must have a single interpretation only. We give a portion of the SRS document (having one or two requirements) to 10 persons and ask their interpretations. If we get more than one interpretation, then there may be an ambiguity in the requirement(s). Hence, requirement statement should be short, explicit, precise and clear. However, it is difficult to achieve this due to the usage of natural languages (like English), which are inherently ambiguous. A checklist should focus on ambiguous words and should have potential ambiguity indicators.

**(iii) Completeness**

The SRS document should contain all significant functional requirements and non-functional requirements. It should also have forms (external interfaces) with validity checks, constraints, attributes and full labels and references of all figures, tables, diagrams, etc. The completeness of the SRS document must be checked thoroughly by a checklist.

**(iv) Consistency**

Consistency of the document may be maintained if the stated requirements do not differ with other stated requirements within the SRS document. For example, in the overall description of the SRS document, it may be stated that the passing percentage is 50 in 'result management software' and elsewhere, the passing percentage is mentioned as 40. In one section, it is written that the semester mark sheet will be issued to colleges and elsewhere it is mentioned that the semester mark sheet will be issued directly to students. These are examples of inconsistencies and should be avoided. The checklist should highlight such issues and should be designed to find inconsistencies.

**(v) Verifiability**

The SRS document is said to be verifiable, if and only if, every requirement stated therein is verifiable. Non-verifiable requirements include statements like 'good interfaces', 'excellent response time', 'usually', 'well', etc. These statements should not be used.

An example of a verifiable statement is 'semester mark sheet shall be displayed on the screen within 10 seconds'. We should use measurable terms and avoid vague terms. The checklist should check the non-verifiable requirements.

**(vi) Modifiability**

The SRS document should incorporate modifications without disturbing its structure and style. Thus, changes may be made easily, completely and consistently while retaining the framework. Modifiability is a very important characteristic due to frequent changes in the requirements. What is constant in life? It is change and if we can handle it properly, then it may have a very positive impact on the quality of the SRS document.

**(vii) Traceability**

The SRS document is traceable if the origin of each requirement is clear and may also help for future development. Traceability may help to structure the document and should find place in the design of the checklist.

**(viii) Feasibility**

Some of the requirements may not be feasible to implement due to technical reasons or lack of resources. Such requirements should be identified and accordingly removed from the SRS document. A checklist may also help to find non-feasible requirements.

The SRS document is the source of all future problems. It must be reviewed effectively to improve its quality. Every review process will add to the improvement of its quality which is dependent on the characteristics discussed above. A checklist should be designed to address the above-mentioned issues. A well-designed and meaningful checklist may help the objective of producing good quality maintainable software, delivered on time and within budget. There may be many ways to design a checklist. A good checklist must address the above-mentioned characteristics. A generic checklist is given in Table 5.3, which may be tailored as per the need of a project.

**Section - I**

| | |
|---|---|
| Name of reviewer<br>Organization<br>Group Number<br>Date of review<br>Project Title | |

**Section – II**

**Table 5.3.** Checklist for the SRS document

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| | **Introduction** | | |
| 1. | Is the purpose of the project clearly defined? | | |
| 2. | Is the scope clearly defined? | | |
| 3. | Is document format as per standard / guidelines (e.g. IEEE 830-1998) | | |
| 4. | Is the project formally approved by the customer? | | |
| 5. | Are all requirements, interfaces, constraints, definitions, etc. listed in the appropriate sections? | | |

(*Contd.*)

(*Contd.*)

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| 6. | Is the expected response time from the user's point of view, specified for all operations? | | |
| 7. | Do all stated requirements express the expectations of the customer? | | |
| 8. | Are there areas not addressed in the SRS document that need to be? | | |
| 9. | Are non-functional requirements stated? | | |
| 10. | Are validity checks properly defined for every input condition? | | |
| | **Ambiguity** | | |
| 11 | Are functional requirements separated from non-functional requirements? | | |
| 12. | Is any requirement conveying more than one interpretation? | | |
| 13. | Are all requirements clearly understandable? | | |
| 14. | Does any requirement conflict with or duplicate with other requirements? | | |
| 15. | Are there ambiguous or implied requirements? | | |
| | **Completeness** | | |
| 16. | Are all functional and non-functional requirements stated? | | |
| 17. | Are forms available with validity checks? | | |
| 18. | Are all reports available in the specified format? | | |
| 19. | Are all references, constraints, assumptions, terms and unit of measures clearly stated? | | |
| 20. | Has analysis been performed to identify missing requirements? | | |
| | **Consistency** | | |
| 21. | Are the requirements specified at a consistent level of detail? | | |
| 22. | Should any requirement be specified in more detail? | | |
| 23. | Should any requirement be specified in less detail? | | |
| 24. | Are the requirements consistent with other documents of the project? | | |
| 25. | Is there any difference in the stated requirement at two places? | | |
| | **Verifiability** | | |
| 26. | Are all stated requirements verifiable? | | |
| 27. | Are requirements written in a language and vocabulary that the stakeholders can understand? | | |
| 28. | Are there any non-verifiable words? | | |
| 29. | Are all paths of a use case verifiable? | | |
| 30. | Is each requirement testable? | | |
| | **Modifiability** | | |
| 31. | Are all stated requirements modifiable? | | |
| 32. | Have redundant requirements been consolidated? | | |
| 33. | Has the document been designed to incorporate changes? | | |

(*Contd.*)

(*Contd.*)

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| 34. | Is the format structure and style of the document standard? | | |
| 35. | Is there any procedure to document a change? | | |
| | **Traceability** | | |
| 36. | Can any requirement be traced back to its origin or source? | | |
| 37. | Is every requirement uniquely identifiable? | | |
| 38. | Are all requirements clearly understandable for implementation? | | |
| 39. | Has each requirement been cross referenced to requirements in previous project documents that are relevant? | | |
| 40. | Is each requirement identified such that it facilitates referencing of each requirement in future development and enhancement efforts? | | |
| | **Feasibility** | | |
| 41. | Is every stated requirement feasible? | | |
| 42. | Is any requirement non-feasible due to technical reasons? | | |
| 43. | Is any requirement non-feasible due to lack of resources? | | |
| 44. | Is any requirement feasible but very difficult to implement? | | |
| 45. | Is any requirement very complex? | | |
| | **General** | | |
| 46. | Is the document concise and easy to follow? | | |
| 47. | Are requirements stated clearly and consistently without contradicting themselves or other requirements? | | |
| 48. | Are all forms, figures and tables uniquely numbered? | | |
| 49. | Are hardware and other communication requirements stated clearly? | | |
| 50. | Are all stated requirements necessary? | | |

## 5.3 SOFTWARE DESIGN DESCRIPTION (SDD) DOCUMENT VERIFICATION

We prepare the SDD document from the SRS document. Every requirement stated therein is translated into design information required for planning and implementation of a software system. It represents the system as a combination of design entities and also describes the important properties and relationship among those entities. A design entity is an element (unit) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced. Our objective is to partition the system into separate units that can be considered, implemented and changed independently.

The design entities may have different nature, but may also have common characteristics. Each entity shall have a purpose, function and a name. There is a common relationship among entities such as interfaces or shared data. The common characteristics are described by attributes of entities. Attributes are the questions about entities. The answer to these questions is the values of the attributes. The collection of answers provides a complete description of an entity. The SDD should address all design entities along with their attributes.

### 5.3.1 Organization of the SDD Document

We may have different views about the essential aspects of software design. However, we have the IEEE recommended practice for software design description (IEEE STD 1016-1998), which is a popular way to organize an SDD document [IEEE98b]. The organization of SDD is given in as per IEEE STD 1016-1998. The entity / attribute information may be organized in several ways to reveal all the essential aspects of a design. Hence, there may be a number of ways to view the design. Each design view gives a separate concern about the system. These views provide a comprehensive description of the design in a concise and usable form that simplifies information access and assimilation. Two popular design techniques are function oriented design and object oriented design. We may use any approach depending on the nature and complexity of the project. Our purpose is to prepare a quality document that translates all requirements into design entities along with its attributes. The verification process may be carried out many times in order to improve the quality of the SDD. The SDD provides a bridge between software requirements and implementation. Hence, strength of the bridge is the strength of the final software system.

### 5.3.2 The SDD Document Checklist

The SDD document verification checklist may provide opportunities to reviewers for focusing on important areas of the design. The software design starts as a process for translating requirements stated in the SRS document in a user-oriented functional design. The system developers, customers and project team may finalise this design and use it as a basis for a more technical system design. A checklist may help to structure the design review process. There are many ways to design a checklist which may vary with the nature, scope, size and complexity of the project. One form of checklist is given in Table 5.4. However, organizations may modify this checklist depending on software engineering practices and type of the project.

**Section – I**

| | |
|---|---|
| Name of reviewer | |
| Organization | |
| Group Number | |
| Date of review | |
| Project title | |

**Section – II**

**Table 5.4.** Checklist for the SDD Document

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| | **General Issues** | | |
| 1. | Is the document easy to read? | | |
| 2. | Is the document easy to understand? | | |

(*Contd.*)

(*Contd.*)

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| 3. | Is the document format as per IEEE std. 1016-1998? | | |
| 4. | Does the document look professional? | | |
| 5. | Is system architecture (including hardware, software, database and data communication structures) specified? | | |
| | **System Architecture** | | |
| 6. | Is the architecture understandable? | | |
| 7. | Are figures used to show the architecture of the system? | | |
| 8. | Are all essentials described clearly and consistently? (Essentials may be software component(s), networks, hardware, databases, operating system, etc). | | |
| 9. | Is the software architecture consistent with existing policies, guidelines and standards? | | |
| 10. | Is the architecture complete with essential details? | | |
| | **Software Design** | | |
| 11. | Is the design as per standards? | | |
| 12. | Are all design entities described? | | |
| 13. | Are all attributes defined clearly? | | |
| 14. | Are all interfaces shown amongst the design entities? | | |
| 15. | Are all stated objectives addressed? | | |
| 16. | Is the data dictionary specified in tabular form? | | |
| | **Data Design** | | |
| 17. | Are all definitions of data elements included in the data dictionary? | | |
| 18. | Are all appropriate attributes that describe each data element included in the data dictionary? | | |
| 19. | Is interface data design described? | | |
| 20. | Is data design consistent with existing policies, procedures, guidelines, standards and technological directives? | | |
| | **Interface Design** | | |
| 21. | Is the user interface for every application described? | | |
| 22. | Are all fields available on every screen? | | |
| 23. | Is the quality of screen acceptable? | | |
| 24. | Are all major functions supporting each interface addressed? | | |
| 25. | Are all validity checks for every field specified? | | |
| | **Traceability** | | |
| 26. | Is every requirement stated in the SRS addressed in design? | | |
| 27. | Does every design entity address at least one requirement? | | |
| 28. | Is there any missing requirement? | | |
| 29. | Is the Requirement Traceability Matrix (RTM) prepared? | | |
| 30. | Does the RTM indicate that every requirement has been addressed clearly? | | |

## 5.4 SOURCE CODE REVIEWS

A source code review involves one or more reviewers examining the source code and providing feedback to the developers, both positive and negative. Reviewers should not be from the development team. Robert Bogue [BOGU09] has given his views about source code reviews as:

> "Code reviews in most organizations are a painful experience for everyone involved. The developer often feels like it's a bashing session designed to beat out their will. The development leads are often confused as to what is important to point out and what is not. And other developers that may be involved often use this as a chance to show how much better they can be by pointing out possible issues in someone else's code."

We may review the source code for syntax, standards defined, readability and maintainability. Typically, reviews will have a standard checklist as a guide for finding common mistakes and to validate the source code against established coding standards. The source code reviews always improve the quality and find all types of faults. The faults may be due to poor structure, violation of business rules, simple omissions, etc. Reviewing the source code has proved to be an effective way to find faults and is considered as a good practice for software development.

### 5.4.1 Issues Related to Source Code Reviews

We should follow good software engineering practices to produce good quality maintainable software within time at a reasonable cost. Source code reviews help us to achieve this objective. Some of the recommended software engineering practices are given as:

1. Always use meaningful variables.
2. Avoid confusing words in names. Do not abbreviate 'Number' to 'No'; 'Num' is a better choice.
3. Declare local variables and avoid global variables to the extent possible. Thus, minimize the scope of variables.
4. Minimize the visibility of variables.
5. Do not overload variables with multiple meanings.
6. Define all variables with meaningful, consistent and clear names.
7. Do not unnecessarily declare variables.
8. Use comments to increase the readability of the source code.
9. Generally, comments should describe what the source code does and not how the source code works.
10. Always update comments while changing the source code.
11. Use spaces and not TABS.
12. All divisors should be tested for zero or garbage value.
13. Always remove unused lines of the source code.
14. Minimize the module coupling and maximize the module strength.
15. File names should only contain A-Z, a-z, 0-9, '_' and '.'.
16. The source code file names should be all lower case.
17. All loops, branches and logic constructs should be complete, correct and properly nested and also avoid deep nesting.

18. Complex algorithms should be thoroughly explained.
19. The reasons for declaring static variables should be given.
20. Always ensure that loops iterate the correct number of times.
21. When memory is not required, it is essential to make it free.
22. Release all allocated memory and resources after the usage.
23. Stack space should be available for running a recursive function. Generally, it is better to write iterative functions.
24. Do not reinvent the wheel. Use existing source code as much as possible. However, do not over-rely on this source code during testing. This portion should also be tested thoroughly.

We may add many such issues which are to be addressed during reviewing. A good checklist may help the reviewers to organize and structure the review process of the source code.

### 5.4.2 Checklist of Source Code Reviews

A checklist should at least address the above-mentioned issues. However, other issues may also be added depending on the nature and complexity of the project. A generic checklist is given in Table 5.5. We may also prepare a programming language specific checklist which may also consider the specific language issues.

**Section – I**

| | |
|---|---|
| Name of reviewer | |
| Organization | |
| Group Number | |
| Date of review | |
| Project title | |

**Section – II**

**Table 5.5.** Source code reviews checklist

| S. No. | Description | Yes/No/NA | Remarks |
|---|---|---|---|
| | **Structure** | | |
| 1. | Does the source code correctly and completely implement the design? | | |
| 2. | Is there any coding standard being followed? | | |
| 3. | Has the developer tested the source code? | | |
| 4. | Does the source code execute as expected? | | |
| 5. | Is the source code clear and easy to understand? | | |
| 6. | Are all functions in the design coded? | | |
| 7. | Is the source code properly structured? | | |
| 8. | Are there any blocks of repeated source code that can be combined to form a single module? | | |
| 9. | Is any module very complex and should be decomposed into two or more modules? | | |
| 10. | Is the source code fault tolerant? | | |

*(Contd.)*