## Software Reliability

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the corresponding instruction is executed.

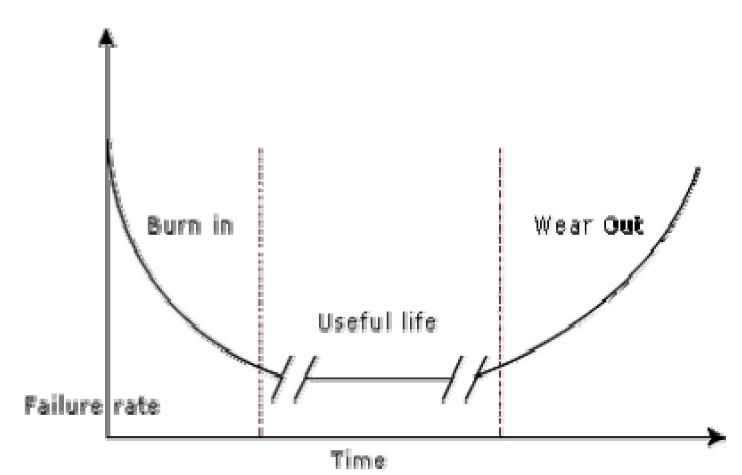
Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the “correctly” implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

**Reasons for software reliability being difficult to measure**

The reasons why software reliability is difficult to measure can be summarized as follows:

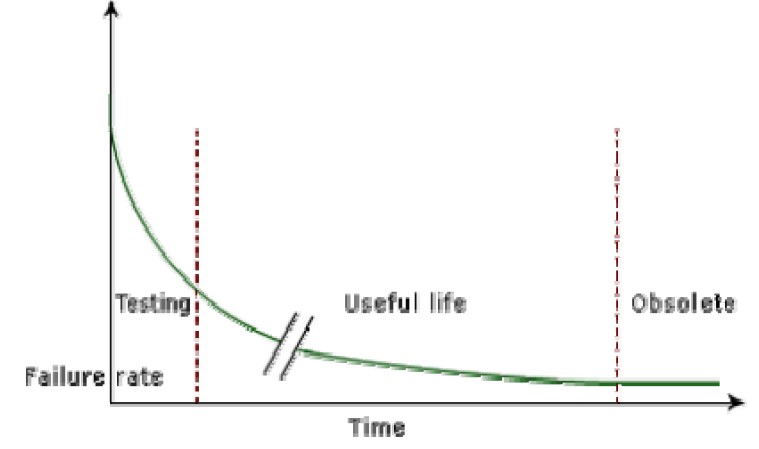
* The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
* The perceived reliability of a software product is highly observer-dependent.
* The reliability of a product keeps changing as errors are detected and fixed.
* Hardware reliability vs. software reliability differs.

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 26.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.



**Hardware**

**Software**



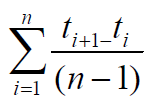
## Reliability Metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

* **Rate of occurrence of failure (ROCOF)-** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
* **Mean Time To Failure (MTTF) -** MTTF is the average time between two successive

failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants *t* , *t* , …, *t* . Then,

*1 2 n*

MTTF can be calculated as

It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.

* **Mean Time To Repair (MTTR) -** Once failure occurs, sometime is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
* **Mean Time Between Failure (MTBR) -** MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.
* **Probability of Failure on Demand (POFOD) -** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.
* **Availability-** Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

**Classification of software failures**

A possible classification of failures of software products into five different types is as follows:

* **Transient-** Transient failures occur only for certain input values while invoking a function of the system.
* **Permanent-** Permanent failures occur for all input values while invoking a function of the system.
* **Recoverable-** When recoverable failures occur, the system recovers with or without operator intervention.
* **Unrecoverable-** In unrecoverable failures, the system may need to be restarted.
* **Cosmetic-** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

# SOFTWARE QUALITY

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.

The modern view of a quality associates with a software product several quality factors such as the following:

* **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.
* **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.
* **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
* **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
* **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

## Software Quality Management System

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality.

**Managerial Structure and Individual Responsibilities-** A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

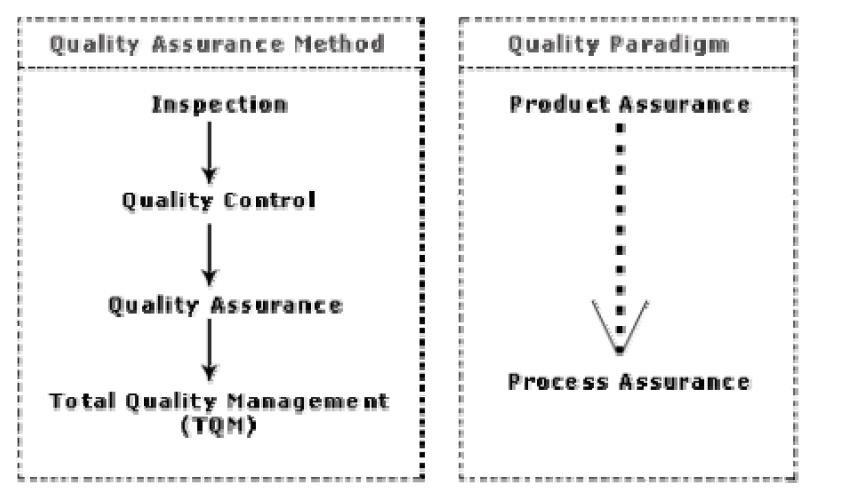
**Quality System Activities-** The quality system activities encompass the following:

* auditing of projects
* review of the quality system
* development of standards, procedures, and guidelines, etc.
* production of reports for the top management summarizing the effectiveness of the quality system in the organization.

## Evolution of Quality Management System

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. 28.1. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization’s processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance .



## ISO 9000 certification

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987. ISO certification serves as a reference for contract between independent parties. The ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organization applies to all activities related to its product or service. The ISO standard mainly addresses operational aspects and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 specifies a set of guidelines for repeatable and high quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.

## Types of ISO 9000 quality standards

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003. The ISO 9000 series of standards is based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically. The types of industries to which the different ISO standards apply are as follows.

ISO 9001 applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.

ISO 9002 applies to those organizations which do not design products but are only involved in production. Examples of these category industries include steel and car manufacturing industries that buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organizations. ISO 9003 applies to organizations that are involved only in installation and testing of the products.

## Software products vs. other products

There are mainly two differences between software products and any other type of products.

* Software is intangible in nature and therefore difficult to control. It is very difficult to control and manage anything that is not seen. In contrast, any other industries such as car manufacturing industries where one can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it is easy to accurately determine how much work has been completed and to estimate how much more time will it take.
* During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product.

## Need for obtaining ISO 9000 certification

There is a mad scramble among software development organizations for obtaining ISO certification due to the benefits it offers. Some benefits that can be acquired to organizations by obtaining ISO certification are as follows:

* Confidence of customers in an organization increases when organization qualifies for ISO certification. This is especially true in the international market. In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification. For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.
* ISO 9000 requires a well-documented software production process to be in place. A well- documented software production process contributes to repeatable and higher quality of the developed software.
* ISO 9000 makes the development process focused, efficient, and cost-effective.
* ISO 9000 certification points out the weak points of an organization and recommends remedial action.
* ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

A summary of the main requirements of ISO 9001 as they relate of software development is as follows. Section numbers in brackets correspond to those in the standard itself:

## Management Responsibility (4.1)

* The management must have an effective quality policy.
* The responsibility and authority of all those whose work affects quality must be defined and documented.
* A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
* The effectiveness of the quality system must be periodically reviewed by audits.

## Quality System (4.2)

A quality system must be maintained and documented.

## Contract Reviews (4.3)

Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

## Design Control (4.4)

* The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
* Design inputs must be verified as adequate.
* Design must be verified.
* Design output must be of required quality.
* Design changes must be controlled.

## Document Control (4.5)

* There must be proper procedures for document approval, issue and removal.
* Document changes must be controlled. Thus, use of some configuration management tools is necessary.

## Purchasing (4.6)

Purchasing material, including bought-in software must be checked for conforming to requirements.

## Purchaser Supplied Product (4.7)

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

**Product Identification (4.8)**

The product must be identifiable at all stages of the process. In software terms this means configuration management.

## Process Control (4.9)

* The development must be properly managed.
* Quality requirement must be identified in a quality plan.

## Inspection and Testing (4.10)

In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

## Inspection, Measuring and Test Equipment (4.11)

If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

## Inspection and Test Status (4.12)

The status of an item must be identified. In software terms this implies configuration management and release control.

## Control of Nonconforming Product (4.13)

In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

## Corrective Action (4.14)

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

## Handling, (4.15)

This clause deals with the storage, packing, and delivery of the software product.

## Quality records (4.16)

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

## Quality Audits (4.17)

Audits of the quality system must be carried out to ensure that it is effective.

## Training (4.18)

Training needs must be identified and met.

## Salient features of ISO 9001 certification

The salient features of ISO 9001 are as follows:

* All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
* Proper plans should be prepared and then progress against these plans should be monitored.
* Important documents should be independently checked and reviewed for effectiveness and correctness.
* The product should be tested against specification.
* Several organizational aspects should be addressed e.g., management reporting of the quality team.

## Shortcomings of ISO 9000 certification

Even though ISO 9000 aims at setting up an effective quality system in an organization, it suffers from several shortcomings. Some of these shortcomings of the ISO 9000 certification process are the following:

* ISO 9000 requires a software production process to be adhered to but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
* ISO 9000 certification process is not fool-proof and no international accreditation agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
* Organizations getting ISO 9000 certification often tend to downplay domain expertise. These organizations start to believe that since a good process is in place, any engineer is as effective as any other engineer in doing any particular activity relating to software development. However, many areas of software development are so specialized that special expertise and experience in these areas (domain expertise) is required. In manufacturing industry there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. In contrast, software development is a creative process and individual skills and experience are important.
* ISO 9000 does not automatically lead to continuous process improvement, i.e. does not automatically lead to TQM.

# COMPUTER AIDED SOFTWARE ENGINEERING

## CASE tool and its scope

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project management and configuration management.

Reasons for using CASE tools

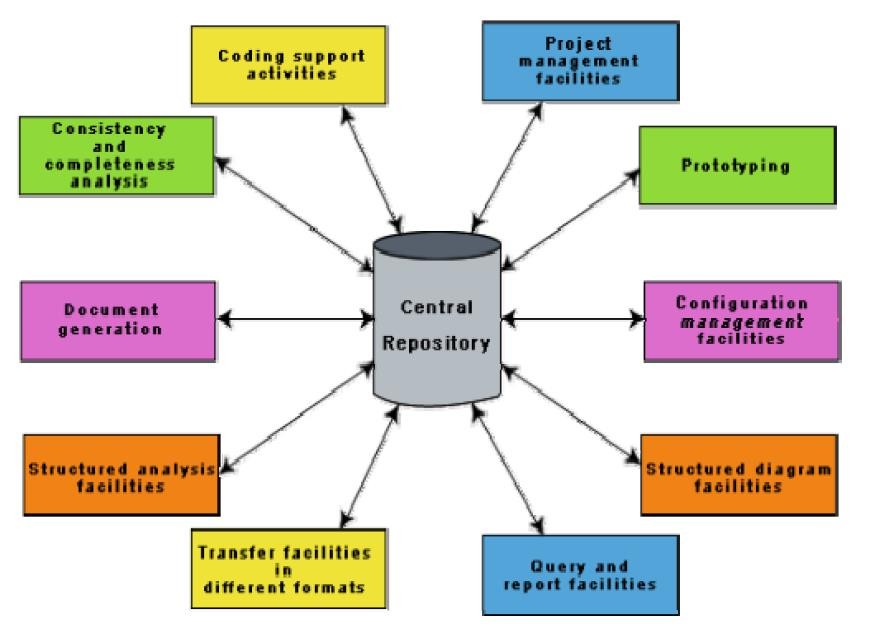
The primary reasons for using a CASE tool are:

* To increase productivity
* To help produce better quality software at lower cost

## CASE environment

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts. This central repository is usually a data dictionary containing the definition of all composite and elementary

data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilitates the automation of the step-by-step methodologies for software development. A schematic representation of a CASE environment is shown in figure below.



## CASE environment vs programming environment

A CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development.

## Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

* A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%.
* Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.

## CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced

* CASE tools take out most of the drudgery in a software engineer’s work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.
* CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
* Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

## Requirements of a prototyping CASE tool

Prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The important features of a prototyping CASE tool are as follows:

* + Define user interaction
  + Define the system control flow
  + Store and retrieve data required by the system
  + Incorporate some processing logic

## Features of a good prototyping CASE tool

There are several stand-alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype. A good prototyping tool should support the following features:

* Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
* It should integrate with the data dictionary of a CASE environment.
* If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.
* The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.

## The run time system of prototype should support mock runs of the actual system and management of the input and output data.

## Structured analysis and design with CASE tools

Several diagramming techniques are used for structured analysis and structured design. The following supports might be available from CASE tools.

* A CASE tool should support one or more of the structured analysis and design techniques.
* It should support effortlessly drawing analysis and design diagrams.
* It should support drawing for fairly complex diagrams, preferably through a hierarchy of levels.
* The CASE tool should provide easy navigation through the different levels and through the design and analysis.
* The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Whenever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there arises heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

## Code generation and CASE tools

As far as code generation is concerned, the general expectation of a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic supports expected from a CASE tool during code generation phase are the following:

* The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
* The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular languages.
* It should generate database tables for relational database management systems.
* The tool should generate code for user interface from prototype definition for X window and MS window based applications.

## Test case generation CASE tool

The CASE tool for test case generation should have the following features:

* It should support both design and requirement testing.
* It should generate test set reports in ASCII format which can be directly imported into the test plan document.