SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT 1- SOFTWARE QUALITY

Definition of Software Quality, Quality Planning, Quality system – Quality Control Vs Quality Assurance – Product life cycle – Project life cycle models. The Software Quality Challenge - Software Quality Factors - Components of the Software Quality Assurance System. Pre-Project Software Quality Components - Contract Review - Development and Quality Plans.

SCSA3002 -QUALITY ENGINEERING

INTRODUCTION

Software Quality

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally develop software

Three (3) important points to remember on software quality

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for good maintainability). If software conforms to its explicit requirements, but fails to meet implicit requirements.

Quality Characteristics

Is any property or element that can be used to define the nature of a product. Each characteristic can be physical or chemical properties such as size, weight, volume, color or composition.

Software Quality

1. Is achieved through a disciplined approach - called software engineering SE
2. Can be defined, described, and measured
3. Can be assessed before any code has been written4

Cannot be tested into a product

Software quality challenges

1. Defining it
2. Describing it (qualitatively)
3. Measuring it (quantitatively)
4. Achieving it (technically)

Designing software is a *creative* task, and like most such tasks, success is more likely if the designer follows what might be termed a set of *rules of form.*The rules of form also provide some way of assessing the *quality* of the eventual product, and possibly of the processes that led to it.

REALISING QUALITY

A set of abstract quality factors (‘the *ilities*’) has been defined. These cannot be measured directly but do relate to the ultimate goal.

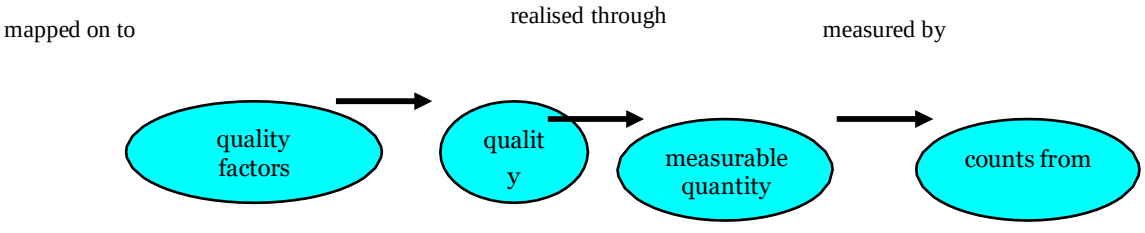


Figure1. Quality factors

Software Quality Factors (by McCall)

1) Product Revision (changing it)

- *Flexibility (can I change it?)*
  The effort required to modify an operational program. Change and enhancement of the system should be easily implementable.

- *Maintainability (can I fix it?)*
  The effort required to locate and fix an error in a program. The system should be easy to keep up for its intended use. Changes for improving operational efficiency should be easy to implement. Failed operations should be easy to restore to satisfactory condition.

- *Testability (can I test it?)*
  The effort required to test a program to ensure that it performs its intended function . The ability of the system to produce quality product units should be easily testable. Useful messages should be generated for testing and debugging purposes.

2) Product Transition (modifying it to work in a different environment)
Interoperability (Will I be able to interface it with another system?)
The effort required to couple one system to another.

- *Portability (Will I be able to use it on another machine?)*
  The effort required to transfer the program from one hardware and/or software system environment to another. The system should be portable among people and among machines. Attainment of theother quality characteristics greatly facilitates portability.

- *Reusability (Will I be able to reuse some of the software?)*
  The extent to which a program (or part of a program) can be reused in other applications-related to the packaging and scope of the functions that the program performs.

3) Product Operations (using it)

- *Correctness (Does it do what I want?)*
  The extent to which a program satisfies its specification and fulfills the customer's mission objectives. The extent to which software is free from design defects and from coding defects.; that is fault-free.

- *Reliability (Does it do it accurately all of the time?)*
  The extent to which a program can be expected to perform its intended function with required precisions under stated conditions for a stated period of time.

- *Efficiency (Will it run on my hardware as well as it can?)*
  The extent to which a software performs its function.with a minimum consumption of computing resources. It should not use any hardware components or peripheral equipment unnecessarily.

- *Integrity (Is it secure?)*

The extent to which access to software or data by unauthorized persons can be controlled.

- *Usability (Is it designed for the use?)*

The effort required to learn, operate, prepare input, and interpret output of a program.

Software Quality Assurance (SQA)

Software quality assurance is a planned effort to ensure that a software product fulfills these criteria and has additional attributes specific to the project, e.g., portability, efficiency, reusability, and flexibility. It is the collection of activities and functions used to monitor and control a software project so that specific objectives are achieved with the desired level of confidence. It is not the sole responsibility of the software quality assurance group but is determined by the consensus of the project manager, project leader, project personnel, and users.

A formal definition of software quality assurance is that is 'the systematic activities providing evidence of the fitness for use of the total software product." Software quality assurance is achieved through the use of established guidelines.for quality control to ensure the Integrity and prolonged life of software. The relationships between quality assurance, quality control, the auditing function, and software testing are often confused.

Quality assurance is the set of support activities needed to provide adequate confidence that processes are established and continuously improved in order to products that meet specifications and are fit for use. Quality control is the process by which product quality is compared with applicable standards and the action taken when nonconformance is detected. Auditing is the inspection/ assessment activity that verifies compliance with plans, policies, and procedures.

SQAActivities

SQA is ensured through a Quality Management System (QMS), QMS is made of several components; it is a system integrated in the bigger system of software development, which comprises project, process and product management systems.

The Software Engineering Institute (SEI) recommends a set of activities, which, when implemented effectively, assures the designed quality. These activities include:

- Quality assurance planning
- Data gathering on key quality defining parameters
- Data analysis and reporting
- Quality control mechanisms

The first and foremost requirement in SQA is that it is a separate group responsible for quality in the organization. They set the goals, standards and mechanisms (systems) for SQA. The role of the SQA group is to assist the software development team in managing the quality requirements of the software. Every software has certain quality goals specified by the customer. These quality goals are

to be achieved by the development team by introducing a set of activities or ensuring the delivery of quality to the customer.

SQA activities operate on the normal activities of quality management. These activities play the role of monitoring, tracking, evaluations, auditing and reviews to ensure that the quality policy of the organization is implemented. These activities are independently carried out, and feedback is given to the development team. The responsibility of delivering the required quality to the customer rests with the development team. The development team has an obligation to implement quality policy in terms of goals, objectives, procedures, checks and controls, documentation and feedback to management. For example, the quality policy stipulates preparation of a test plan for stages for development as well as at the end of the development process. SQA has a variety of tools to implement the policy.

They are

- Auditing
- Inspection

Verify compliance with those norms and practices specified in QA policy; deviations are set right. Ensure that deviations are documented and reported and put into the QA database for guidance. Design and architecture is reviewed to ensure that standards are met and customer quality is assured. Implement change management. Collect data on various observations in the process of auditing, inspection and reviews to build QA database and to improve various standards.

Software Defects

The causes for not meeting the quality commonly are

- Imprecise requirement and software specifications
- Lack of understanding of customer requirements
- International deviations
- Violation of standards (Design, Programming)
- Erroneous data representation
- Improper interface
- Faulty logic in rules and processes
- Erroneous testing
- Incomplete and defective documentation
- H&S platforms not coping up to required standards
- Lack of domain knowledge

Statistical analysis of errors or defects helps to focus and concentrate on probable areas where SQA efforts are necessary.

SQA is also concerned with two other aspects namely, software reliability and software safety. Software reliability is defined as the probability of failure free operation of a computer program in a

specified environment for a specified time.

The nature of failure may be such that one error may require only a few repair, and other may need hours. SQA collects data on these failures and examines why these failures could not be prevented through earlier SQA activities.

A simple measure of reliability is Mean Time between Failure (MTBF). MTBF = MTTF + MTTR
Where MTTF is Mean Time to Failure and MTTR is Mean Time to Repair.

Software safety deals with identification and assessment of potential hazards of software failing, and its impact on the system or in the environment in which it operates. Software safety needs are more crucial in process control systems, health care systems, defense and so on. SQA activities concentrate on such areas of software where failure affects the customer system adversely.

In short, SQA efforts assure software quality, reliability, availability and safety.
Software Quality Assurance Components

The SQA Components that are used by the Software Quality Assurance System can be classified into six different categories; each of which is necessary to guarantee maximum quality and to ensure the compliance with the standards and procedures.

"The environment in which software development and maintenance is undertaken directly influences the SQA components. Alongside this various errors will also affect the SQA components; therefore it is usually necessary to include all of the components."

casd.csie.ncku.edu.tw/sq/ch04.ppt

The six different components are broken down into the following categories

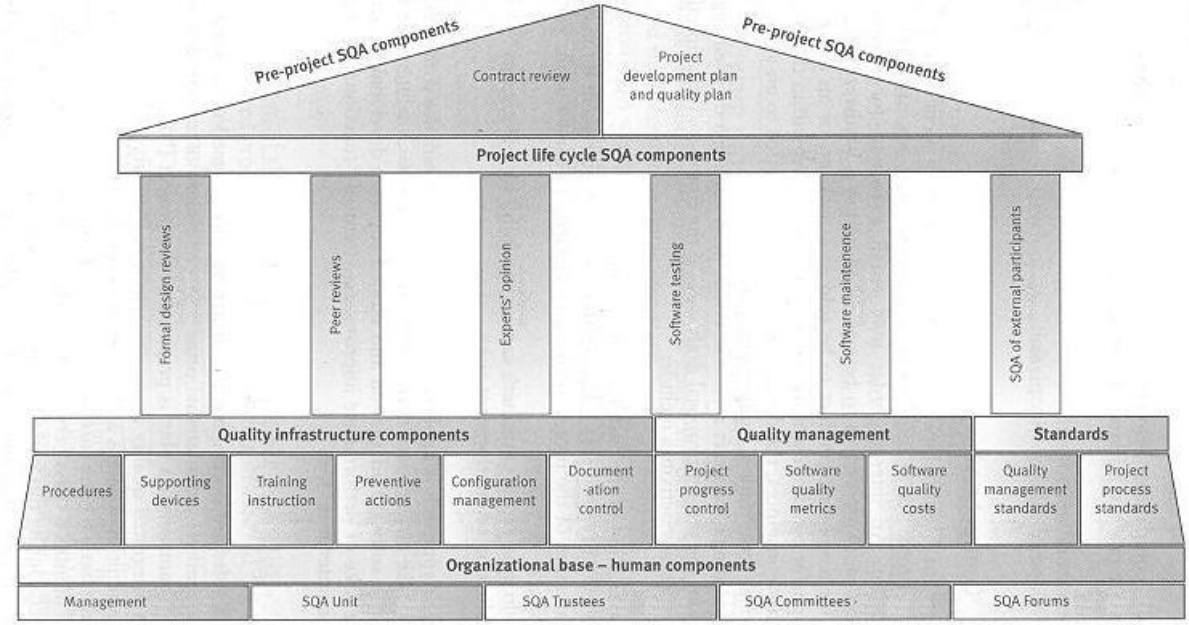


Figure 1: The SQA System Overview

1. Pre-project Components

The Pre-Project component ensures that the project has been adequately defined ensuring that the resources available, budget and so forth have not been misinterpreted by the client or the organisation. This improves the preliminary steps taken prior to initiating work on the project itself, thus leading to fewer errors later in the development phase.

There are two key concepts to this component;

Contract Review

The Contract Review begins the negotiations of a contract between the company and the client. One of the key focuses is on the areas or points that could go wrong, known as development risks.
The contract ensures that commitments have been documented including an agreement upon the functional specification, budget and the schedule.
The contract review activities must include a detailed examination of the project proposal draft and the contract draft, each of which have different objectives, to ensure that the commitments have been properly and clearly defined. A checklist is often used by reviewers to make this stage easier and it is expected that the review stage will occur more than once.
The outcome from this stage is a documented agreement between the company and client stating the projects requirements, budgets and so on ensuring that no new changes have been added to the contract draft.

Development and Quality Plans

The development and Quality Plans stage occurs once an agreement has been made and a software development contract has been signed. The time between a proposal, the contract review and the signing of the contract could be prolonged and during this time changes may occur. Consequently proposal materials need to be revised and new plans are required; a development plan and a quality plan.
The development plan focuses on eleven specific elements, some of which are; the products of the project i.e. software products and design documents specifying deliverables. Project interfaces detailing what, if any, interfaces the product will have with existing software packages. The methodology clearly states which methodology should be used at each phase of the project.
The quality plan has four main elements; however, how many of these are used is dependent upon theproject. Quality Goals refers to the goals of the product, it is always better to have more goals as it is easier to see how well the system performs. Planned review activities are a listing of all the planned reviews such as Design Reviews (DR) and Code Inspections. Planned software tests details all the software tests that are to be performed with details such as the unit, integration or complete system to be tested. Finally, planned acceptance tests for externally developed software; this is only necessary for products not developed in-house.
Approval of the two plans is necessary and is approved or rejected according to the procedures withinthe organisation.

2. Project Lifecycle Activities and Assessment

This component has two main stages;

1. The Development Lifecycle stage which aims to detect design and programming errors.

2. The Operation-Maintenance stage focuses on maintenance tasks that improve functionality.

When developing the product there are a number of activities that take place, these are reviews, such as formal design reviews and peer reviews, expert opinions, software testing, software maintenance and finally, the assurance of the quality of external participants' work which ensures that any efforts by external members meet the quality and standards of the organization.

The reviews occur during the design phase of the development process and should be conducted at any milestone. The formal Design Review (DR) is a very important document as the project cannot continue until a formal approval by the DR committee has been received. The document itself includes a list of action items (corrections) that are required. The peer review, guided by checklists,

standards and past problems, is a document that reviews short documents or sections to attempt to detect design and programming faults documenting a list of its findings.

'Expert Opinions' is the use of an external member to review in-house work; it can reinforce the internal (in-house) quality assurance activities. Although not all organizations may use this approach it is useful for small organizations which may not have sufficient professional capabilities in-house or for a replacement to the regular in-house professional for whatever reasons. The outcome from this is often a document similar to a formal design review.

"Software Testing is a formal process carried out by a specialized team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer. All associated tests are performed according to the approved test procedures on approved test cases"

D.Galin (2003)

In general, Software Testing is aimed at reviewing the running and functionality of the software in which the testing is based on a prepared list of test cases.

Software Maintenance specifies three categories in which maintenance for a system can fall into. The first is corrective maintenance in which the correction of failed software, for whatever reason, occurs.The second is adaptive maintenance in which maintenance occurs to the system it needs to be adapted; however the basic functionality of the system is left untouched. Finally, the third is functionality improvement where the system is modified to add improvements, this could significantly change the system compared to the other two points.

3. Infrastructure components for error prevention and Improvement

The main goal of this component is to reduce the number of errors in the system and improve productivity. This is achieved through the use of a number of sub-components.

1. Procedures and Work Instructions

A procedure describes how the process, a specific development activity, is performed whilst a work instruction is more specific and provides detailed directions for the use of methods. Having a detailed guideline ensures that all members know how to achieve some goal.

"Work instructions and procedures contribute to the correct and effective performance of establishedtechnologies and routines"

http://kur2003.if.itb.ac.id/file/SW%20Quality%20Components.pdf

2. Supporting Quality Devices

'Quality Devices' are used to maximize efficiency and quality. A quality device could be a template or a checklist which saves time as the document will be complete and will not have to be developed, from scratch each time.

Using Quality Devices offers an improved form of communication and provides standardization within the organization as each document will be of the same nature.

3. Staff Training, Instruction and Certification

Staff training is a vital element to avoiding errors throughout the development of the system. Having well trained professional staff enables efficient and high quality performance from each member.

New staffs have to be trained in respect to the standards and procedures of the organization and existing staff should be re-trained when assignments change.

4. Preventive and Corrective Actions

Studying existing data for similar faults and failures can enable future ones to be solved easily either by correcting it once it has occurred or by preventing it from happening.

The data should be recorded, or found, in design review reports, software test reports or customer complaints. It should be managed effectively so that if it occurs in the future the solution, if one was found, can be easily accessible.

5. Configuration Management

This deals with the dangers of version releases. With intense work focusing on new versions and new software releases dangers arise when different members carry out the same tasks. This can lead to misidentification or the versions or releases or loss of records or development activities.

Configuration management introduces procedures that are used to control the change process and monitor it.

6. Documentation Control

Document Control is necessary to ensure long term availability of

controlled documents. Controlled documents are maintained and updated. They are formally approved and contain evidence of the systems performance.

4. Software Quality Management

Quality management not only focuses on product quality but also the means in which it can be achieved.
Some of the components used to support the managerial control of software development projects arethe Project Progress Control, Software Quality Metrics and Software Quality costs.
The aim of Project Progress Control is simply to monitor the progress of the project to ensure that it does not deviate from its initial plans. It focuses particularly on monitoring resource usages, schedules (whether they are being met), risk management activities and the budget.
A Software Quality Metric is a measurement that is used to evaluate software quality in a system. Thesoftware is the input, and the output is a numeric value which represents the **degree to which the software possesses a given quality attribute.** The measures can apply to functional quality, productivity and the organization side of the project.
The Software Quality Costs are the costs that incur throughout the entire project; the total cost is calculated from the costs of control and the costs of failure. The organisations main interest is the sum of the total costs which will determine a success or failure.

5. SQA Standards, System Certification, and Assessment Components

This component focuses on using external tools to achieve the, in-house, goals of software qualityassurance. There are three main objectives, D.Galin (2003):

1. Utilization of international professional knowledge

2. Improvement of coordination with other organizations quality systems

3. Objective professional evaluation and measurement of the achievements of the organizations qualitysystems

The standards available to achieve the above objectives can be classified into two sub-classes

1. Quality Management Standards – 'what' is required

2. Project Process Standards – 'how' it is done

The outcome of this component is simply to use external tools, i.e. staff, to achieve the desired in-house software quality assurance complying with the standards of the organization.

6. Organizing for SQA – The Human Components

SQA cannot be directly applied to an organization; instead an organizational base is required. The organizational base is collectively made up of the SQA Unit, SQA trustees, committees and forums along with the continuous support of the management.

The SQA Unit focuses purely on Software Quality Assurance, thus ensuring that all standards, procedures and components are efficiently and correctly in use. This is, in part, done through the audits and quality programs that the SQA Unit is required to produce.

Management must ensure that all staff are aware of the quality policy, they are required to define sufficient resources and accurately assign an adequate number of staff to the tasks.

The SQA organizational base has three main objects

1. To aid the development and implementation of the SQA system

2. To detect and prevent deviations from organizational standards and procedures

3. To continuously suggest improvements that will benefit the SQA components

How to Design and Implement a Basic Quality Assurance Plan

A quality assurance plan should generally include two basic areas: how to address errors (quality- related events), and how to improve practice before an error occurs (continuous quality improvement). This document outlines steps to take in establishing a QA plan plan.

I. Design a means to effectively document quality-related events (QREs) and educate staff appropriately

1. Collect all relevant details of the event, identify the root cause(s), and make a plan to avoid thesame error in the future (consider the example provided on the Board of Pharmacy's website)

2. Always educate staff on documented QREs and their resulting plans. 3. Many errors reported to theBoard are due to poor customer service in resolving the issueconsider including training on how to handle an error as part of your plan

II. Identify one or two quality related parameters you would like to measure and improve. You might consider two categories of parameters:

1. Areas known to require improvement. a. These areas may be identified through a previous dispensing or procedural error, a deficiency notice from the Board, or observations of pharmacy staff. b. Monitoring will be with the intent to track successful improvement.

2. Areas expected to be satisfactory a. These areas may be identified as perceived strengths in your pharmacy. b. The intent of monitoring may be to verify that processes are done correctly and to identify unsuspected weaknesses.

III. Design a method to measure the identified areas. Here are some tips:

1. Focus on quantitative measures that can show clear results
2. Utilize your computer systems capabilities where appropriate
3. Use random samples where appropriate (e.g. you don't necessarily have to go through the entireprescription log book to quantify counseling documentation)
4. Consider a method that can be accomplished in a reasonable amount of time by appropriate staff.Keep it simple.
5. Consider a method that can be done consistently as part of normal procedures.
6. Determine how often the measurement will be repeated and make plans to ensure it is notforgotten.

IV. Set appropriate goals

1. Perfection is not always a realistic goal. Determine what is acceptable for your practice.
2. Set an attainable goal and be prepared to update the goal when it is achieved.
3. Include instructions on what the person taking the measurement should do if the goal is not met(e.g. who to contact)

V. Be prepared to make new plans when goals are not met

1. Set a deadline for when unmet goals will be addressed 2. Be prepared to change policies orprocedures in order to improve areas of deficiency

VI. Educate your staff on the Quality Assurance Plan, both at inception and at regular intervals.Include:

1. Why it is being done

2. What is being tracked?

3. How to perform measurements

4. Progress in areas being monitored, including improvements implemented as a result thereof

5. Updates on any QREs, including the plan to avoid those errors in the future

VII. Quality assurance never ends

Continue to update your plan as necessary. Over time, the entire prescription process can bemonitored and improved.

Product life cycle

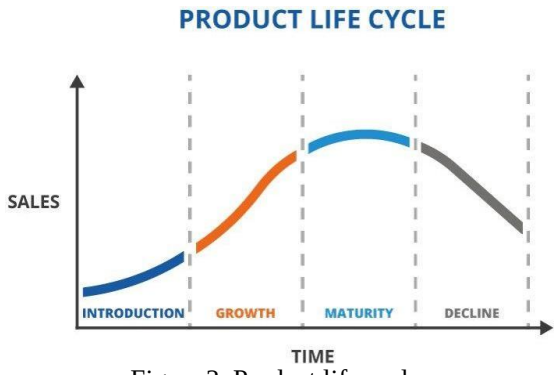


Figure 2. Product life cycle

The various stages of a product's life cycle determine how it is marketed to consumers. Successfully introducing a product to the market should see a rise in demand and popularity, pushing older products from the market. As the new product becomes established, the marketing efforts lessen and

the associated costs of marketing and production drop. As the product moves from maturity to decline, so demand wanes and the product can be removed from the market, possibly to be replaced by a newer alternative.

Managing the four stages of the life cycle can help increase profitability and maximise returns, whilea failure to do so could see a product fail to meet its potential and reduce its shelf life.

Writing in the Harvard Business Review in 1965, marketing professor Theodore Levitt declared that the innovator had the most to lose as many new products fail at the introductory stage of the product life cycle. These failures are particularly costly as they come after investment has already been made in research, development and production. Because of this, many businesses avoid genuine innovation in favour of waiting for someone else to develop a successful product before cloning it.

Stages

There are four stages of a product's life cycle, as follows:

1. Market Introduction and Development

This product life cycle stage involves developing a market strategy, usually through an investment in advertising and marketing to make consumers aware of the product and its benefits.

At this stage, sales tend to be slow as demand is created. This stage can take time to move through, depending on the complexity of the product, how new and innovative it is, how it suits customer needs and whether there is any competition in the marketplace. A new product development that is suited to customer needs is more likely to succeed, but there is plenty of evidence that products can fail at this point, meaning that stage two is never reached. For this reason, many companies prefer to follow in the footsteps of an innovative pioneer, improving an existing product and releasing their own version.

2. Market Growth

If a product successfully navigates through the market introduction it is ready to enter the growth stage of the life cycle. This should see growing demand promote an increase in production and the product becoming more widely available.

The steady growth of the market introduction and development stage now turns into a sharp upturn as the product takes off. At this point competitors may enter the market with their own versions of your product – either direct copies or with some improvements. Branding becomes important to maintain your position in the marketplace as the consumer is given a choice to go elsewhere. Product pricing and availability in the marketplace become important factors to continue driving sales in the face of increasing competition. At this point the life cycle moves to stage three; market maturity.

3. Market Maturity

At this point a product is established in the marketplace and so the cost of producing and marketing the existing product will decline. As the product life cycle reaches this mature stage there are the beginnings of market saturation. Many consumers will now have bought the product and competitors will be established, meaning that branding, price and product differentiation becomes even more

important to maintain a market share. Retailers will not seek to promote your product as they may have done in stage one, but will instead become stockists and order takers.

4. Market Decline

Eventually, as competition continues to rise, with other companies

seeking to emulate your success with additional product features or lower prices, so the life cycle will go into decline. Decline can also be caused by new innovations that supersede your existing product, such as horse-drawn carriages going out of fashion as the automobile took over.

Many companies will begin to move onto different ventures as market saturation means there is no longer any profit to be gained. Of course, some companies will survive the decline and may continue to offer the product but production is likely to be on a smaller scale and prices and profit margins may become depressed. Consumers may also turn away from a product in favour of a new alternative, although this can be reversed in some instances with styles and fashions coming back into play to revive interest in an older product.

Product Life Cycle Strategy and Management

Having a properly managed product life cycle strategy can help extend the life cycle of your product in the market.

The strategy begins right at the market introduction stage with setting of pricing. Options include 'price skimming,' where the initial price is set high and then lowered in order to 'skim' consumer groups as the market grows. Alternatively, you can opt for price penetration, setting the price low to reach as much of the market as quickly as possible before increasing the price once established.

Product advertising and packaging are equally important in order to appeal to the target market. In addition, it is important to market your product to new demographics in order to grow your revenue stream.

Products may also become redundant or need to be pivoted to meet changing demands. An example of this is Netflix, who moved from a DVD rental delivery model to subscription streaming.

Understanding the product life cycle allows you to keep reinventing and innovating with an existing product (like the iPhone) to reinvigorate demand and elongate the product's market life.

Examples

Many products or brands have gone into decline as consumer needs change or new innovations are introduced. Some industries operate in several stages of the product life cycle simultaneously, such as with televisual entertainment, where flat screen TVs are at the mature phase, on-demand programming is in the growth stage, DVDs are in decline and video cassettes are now largely redundant. Many of the most successful

products in the world stay at the mature stage for as long as possible, with small updates and redesigns along with renewed marketing to keep them in the thoughts of consumers, such as with the Apple iPhone.

Here are a few well-known examples of products that have passed or are passing through the product life cycle:

1. Typewriters

The typewriter was hugely popular following its introduction in the late 19th century due to the way it made writing easier and more efficient. Quickly moving through market growth to maturity, the typewriter began to go into decline with the advent of the electronic word processor and then computers, laptops and smartphones. While there are still typewriters available, the product is now at the end of its decline phase with few sales and little demand. Meanwhile, desktop computers, laptops, smartphones and tablets are all experiencing the growth or maturity phases of the product lifecycle.

2. Video Cassette Recorders (VCRs)

Having first appeared as a relatively expensive product, VCRs experienced large-scale product growth as prices reduced leading to market maturation when they could be found in many homes. However, the creation of DVDs and then more recently streaming services, VCRs are now effectively obsolete. Once a ground-breaking product VCRs are now deep in a decline stage from which it seems unlikely they will ever recover.

3. Electric Vehicles

Electric vehicles are experiencing a growth stage in their product life cycle as companies work to push them into the marketplace with continued design improvements. Although electric vehicles are not new, the consistent innovation in the market and the improving sales potential means that they arestill growing and not yet into the mature phase.

4. AI Products

Like electric vehicles, artificial intelligence (AI) has been in development and use for years, but dueto the continued developments in AI, there are many products that are still in the market introduction stage of the product life cycle. These include innovations that are still being developed, such as autonomous vehicles, which are yet to be adopted by consumers.

Conclusion

Understanding how a product's life cycle works allows companies to work out whether their products are meeting the needs of the target market and, thereby, when they may need to change focus or develop something new.

Examining a product in relation to market needs, competition, costs and profits allows a company to pivot their product focus to maintain longevity in the marketplace.

Knowing when a product is going into decline prevents your company from following as a result of being overly reliant on a fading market. A product life cycle strategy means that you can reinvigorate an existing product, develop a new replacement product or change direction to stay abreast of a changing marketplace.

While all products have a life cycle, many of the most successful ones are able to maintain the mature stage of the life cycle for many years before any eventual decline.

Pre-project Software Quality Components

These components help to improve the preliminary steps taken before starting a project. It includes −

- Contract Review
- Development and Quality Plans

Contract Review

Normally, a software is developed for a contract negotiated with a customer or for an internal order to develop a firmware to be embedded within a hardware product. In all these cases, the development unit is committed to an agreed-upon functional specification, budget and schedule. Hence, contract review activities must include a detailed examination of the project proposal draft and the contract drafts.

Specifically, contract review activities include −

- Clarification of the customer's requirements
- Review of the project's schedule and resource requirement estimates
- Evaluation of the professional staff's capacity to carry out the proposed project
- Evaluation of the customer's capacity to fulfil his obligations
- Evaluation of development risks

Development and Quality Plans

After signing the software development contract with an organization or an internal department of the same organization, a development plan of the project and its integrated quality assurance activities are prepared. These plans include additional details and needed revisions based on prior plans that provided the basis for the current proposal and contract.

Most of the time, it takes several months between the tender submission and the signing of the contract. During these period, resources such as staff availability, professional capabilities may get changed. The plans are then revised to reflect the changes that occurred in the interim.

The main issues treated in the project development plan are −

- Schedules
- Required manpower and hardware resources
- Risk evaluations
- Organizational issues: team members, subcontractors and partnerships
- Project methodology, development tools, etc.
- Software reuse plans

The main issues treated in the project's quality plan are −

- Quality goals, expressed in the appropriate measurable terms
- Criteria for starting and ending each project stage
- Lists of reviews, tests, and other scheduled verification and validation activities

Unit-2

SOFTWARE ENGINEERING ACTIVITIES

Estimation, Software requirements gathering, Analysis, Architecture, Design, development, Testing and Maintenance.

1. Estimation

Effective software project estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. As a whole, the software industry doesn't estimate projects well and doesn't use estimates appropriately. We suffer far more than we should as a result and we need to focus some effort on improving the situation.

Under-estimating a project leads to under-staffing it (resulting in staff burnout), under-scoping the quality assurance effort (running the risk of low quality deliverables), and setting too short a schedule (resulting in loss of credibility as deadlines are missed). For those who figure on avoiding this situation by generously padding the estimate, over-estimating a project can be just about as bad for the organization! If you give a project more resources than it really needs without sufficient scope controls it will use them. The project is then likely to cost more than it should (a negative impact on the bottom line), take longer to deliver than necessary (resulting in lost opportunities), and delay the use of your resources on the next project.

Software Project Estimation

The four basic steps in software project estimation are:

1) Estimate the size of the development product. This generally ends up in either Lines of Code (LOC) or Function Points (FP), but there are other possible units of measure. A discussion of the pros & cons of each is discussed in some of the material referenced at the end of this report.
2) Estimate the effort in person-months or person-hours.
3) Estimate the schedule in calendar months.
4) Estimate the project cost in dollars (or local currency)

Estimating size

An accurate estimate of the size of the software to be built is the first step to an effective estimate. Your source(s) of information regarding the scope of the project should, wherever possible, start with formal descriptions of the requirements - for example, a customer's requirements specification or request for proposal, a system specification, and a software requirements specification. If you are [re-]estimating a project in later phases of the project's lifecycle, design documents can be used to provide additional detail. Don't let the lack of a formal scope specification stop you from doing an initial project estimate. A verbal

description or a whiteboard outline are sometimes all you have to start with. In any case, you must communicate the level of risk and uncertainty in an estimate to all concerned and you must re-estimate the project as soon as more scope information is determined.

Two main ways you can estimate product size are:

1) By analogy. Having done a similar project in the past and knowing its size, you estimate each major piece of the new project as a percentage of the size of a similar piece of the previous project. Estimate the total size of the new project by adding up the estimated sizes of each of the pieces. An experienced estimator can produce reasonably good size estimates by analogy if accurate size values are available for the previous project and if the new project is sufficiently similar to the previous one.

2) By counting product features and using an algorithmic approach such as Function Points to convert the count into an estimate of size. Macro-level "product features" may include the number of subsystems, classes/modules, methods/functions. More detailed "product features" may include the number of screens, dialogs, files, database tables, reports, messages, and so on.

Estimating effort

Once you have an estimate of the size of your product, you can derive the effort estimate. This conversion from software size to total project effort can only be done if you have a defined software development lifecycle and development process that you follow to specify, design, develop, and test the software. A software development project involves far more than simply coding the software – in fact, coding is often the smallest part of the overall effort. Writing and reviewing documentation, implementing prototypes, designing the deliverables, and reviewing and testing the code take up the larger portion of overall project effort. The project effort estimate requires you to identify and estimate, and then sum up all the activities you must perform to build a product of the estimated size.

There are two main ways to derive effort from size:

1) The best way is to use your organization's own historical data to determine how much effort previous projects of the estimated size have taken. This, of course, assumes (a) your organization has been documenting actual results from previous projects, (b) that you have at least one past project of similar size (it is even better if you have several projects of similar size as this reinforces that you consistently need a certain level of effort to develop projects of a given size), and (c) that you will follow a similar development lifecycle, use a similar development methodology, use similar tools, and use a team with similar skills and experience for the new project.

2) If you don't have historical data from your own organization because you haven't started collecting it yet or because your new project is very different in one or more key aspects, you can use a mature and generally accepted algorithmic approach such as Barry Boehm's COCOMO model or the Putnam Methodology to convert a size estimate into an effort estimate. These models have been derived by studying a significant number of

completed projects from various organizations to see how their project sizes mapped into total project effort. These "industry data" models may not be as accurate as your own historical data, but they can give you useful ballpark effort estimates.

Estimating schedule

The third step in estimating a software development project is to determine the project schedule from the effort estimate. This generally involves estimating the number of people who will work on the project, what they will work on (the Work Breakdown Structure), when they will start working on the project and when they will finish (this is the "staffing profile"). Once you have this information, you need to lay it out into a calendar schedule. Again, historical data from your organization's past projects or industry data models can be used to predict the number of people you will need for a project of a given size and how work can be broken down into a schedule.

If you have nothing else, a schedule estimation rule of thumb [McConnell 1996] can be used to get a rough idea of the total calendar time required:

Schedule in months = 3.0 * (effort-months) $^{1/3}$

Opinions vary as to whether 2.0 or 2.5 or even 4.0 should be used in place of the 3.0 value – only by trying it out will you see what works for you.

Estimating Cost

There are many factors to consider when estimating the total cost of a project. These include labor, hardware and software purchases or rentals, travel for meeting or testing purposes, telecommunications (e.g., long- distance phone calls, video-conferences, dedicated lines for testing, etc.), training courses, office space, and so on.

Exactly how you estimate total project cost will depend on how your organization allocates costs. Some costs may not be allocated to individual projects and may be taken care of by adding an overhead value to labor rates ($ per hour). Often, a software development project manager will only estimate the labor cost and identify any additional project costs not considered "overhead" by the organization.

The simplest labor cost can be obtained by multiplying the project's effort estimate (in hours) by a general labor rate ($ per hour). A more accurate labor cost would result from using a specific labor rate for each staff position (e.g., Technical, QA, Project Management, Documentation, Support, etc.). You would have to determine what percentage of total project effort should be allocated to each position. Again, historical data or industry data models can help.

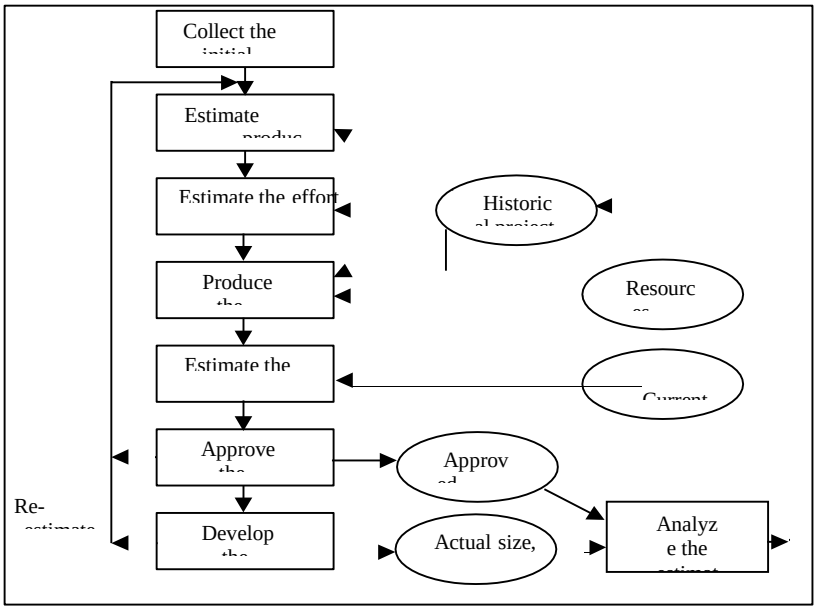


Figure 1 – The Basic Project Estimation Process

Working Backwards from Available Time

Projects often have a delivery date specified for them that isn't negotiable - "The new release has to be out in 6 months"; "The customer's new telephone switches go on-line in 12 months and our software has to be ready then". If you already know how much time you have, the only thing you can do is negotiate the set of functionality you can implement in the time available. Since there is always more to do than time available, functionality has to be prioritized and selected so that a cohesive package of software can be delivered on time.

Working backwards doesn't mean you skip any steps in the basic estimation process outlined above. You still need to size the product, although here you really do have to break it down into a number of pieces you can either select or remove from the deliverable, and you still need to estimate effort, schedule, and cost.
This is where estimation tools can be really useful. Trying to fit a set of functionality into a fixed timeframe requires a number of "what if" scenarios to be generated. To do this manually would take too much time and effort. Some tools allow you to play with various options easily and quickly.

Important factors that affect the accuracy of your estimates, such as:

- The accuracy of all the estimate's input data (the old adage, "Garbage in, Garbage out", holds true)
- The accuracy of any estimate calculations (e.g., converting between Function Points and LOC has a certain margin of error)
- How closely the historical data or industry data used to calibrate the model matches the project you are estimating
- The predictability of your organization's software development process, and whether or

not the actual project was carefully planned, monitored and controlled, and no major surprises occurred that caused unexpected delays.

2. SOFTWARE REQUIREMENTS AND ANALYSIS

Software requirements are necessary

- To introduce the concepts of user and system requirements
- To describe functional and non-functional requirements
- To explain how software requirements may be organized in a requirements document

What is a requirement?

- The requirements for the system are the description of the services provided by the system and its operational constraints
  - It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;

Both these statements may be called requirements

Requirements engineering:

- The process of finding out, analyzing documenting and checking these services and constraints is called requirement engineering.
- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

Types of requirement:

- **User requirements**
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- **System requirements**
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

REQUIREMENTS ENGINEERING PROCESSES

The **goal** of requirements engineering process is to create and maintain a system requirements document. The overall process includes four high-level requirement engineering sub-processes. These are concerned with

- ✓ Assessing whether the system is useful to the business(feasibility study)
- ✓ Discovering requirements(elicitation and analysis)
- ✓ Converting these requirements into some standard form(specification)
  - ✓ Checking that the requirements actually define the system that the customer wants

(validation) the process of managing the changes in the requirements is called **requirement**

management.

Requirements engineering:

The alternative perspective on the requirements engineering process presents the process as a **three-stage activity** where the activities are organized as an iterative process around a spiral. The amount of time and effort devoted to each activity in iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and non-functional requirements and the user requirements. Later in the process, in the outer rings of the spiral, more effort will be devoted to system requirements engineering and system modeling.

This spiral model accommodates approaches to development in which the requirements are developed to different levels of detail. The number of iterations around the spiral can vary, so the spiral can be exited after some or all of the user requirements have been elicited.

1) FEASIBILITY STUDIES

A feasibility study decides whether or not the proposed system is worthwhile. The input to the feasibility study is a set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes. The results of the feasibility study should be a report that recommends whether or not it worth carrying on with the requirements engineering and system development process.

- A short focused study that checks
- If the system contributes to organizational objectives;
- If the system can be engineered using current technology and within budget;
- If the system can be integrated with other systems that are used.

Feasibility study implementation:

- A feasibility study involves information assessment, information collection and report writing.
- Questions for people in the organization
- What if the system wasn't implemented?
- What are current process problems?
- How will the proposed system help?
- What will be the integration problems?
- Is new technology needed? What skills?
- What facilities must be supported by the proposed system?

In a feasibility study, you may consult information sources such as the managers of the departments where the system will be used, software engineers who are familiar with the type of system that is proposed, technology experts and end-users of the system. They should try to complete a feasibility study in two or three weeks.

Once you have the information, you write the feasibility study report. You should make a recommendation about whether or not the system development should continue. In the report, you may propose changes to the scope, budget and schedule of the system and suggest further high-level requirements for the system.

2) REQUIREMENT ELICITATION AND ANALYSIS:

The requirement engineering process is requirements elicitation and analysis.

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*
-

Problems of requirements analysis

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organizational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

Process activities

1. Requirements discovery

– Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

2. Requirements classification and organization

– Groups related requirements and organizes them into coherent clusters.

3. Prioritization and negotiation

– Prioritizing requirements and resolving requirements conflicts.

4. Requirements documentation

– Requirements are documented and input into the next round of the spiral.

The process cycle starts with requirements discovery and ends with requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle.

Requirements classification and organization is primarily concerned with identifying overlapping requirements from different stakeholders and grouping related requirements. The most common way of grouping requirements is to use a model of the system architecture to identify subsystems and to associate requirements with each sub-system.

Inevitably, stakeholders have different views on the importance and priority of requirements, and sometimes these view conflict. During the process, you should organize regular stakeholder negotiations so that compromises can be reached.

In the requirement documenting stage, the requirements that have been elicited are documented in such a way that they can be used to help with further requirements discovery.

2.1) REQUIREMENTS DISCOVERY:

- Requirement discovery is the process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems.
- They interact with stakeholders through interview and observation and may use scenarios and prototypes to help with the requirements discovery.
- Stakeholders range from system end-users through managers and external stakeholders such

as regulators who certify the acceptability of the system.

- For example, system stakeholder for a bank ATM include

1. Bank customers
2. Representatives of other banks
3. Bank managers
4. Counter staff
5. Database administrators
6. Security managers
7. Marketing department
8. Hardware and software maintenance engineers
9. Banking regulators

Requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints, where each viewpoints, where each viewpoint presents a sub-set of the requirements for the system.

Viewpoints:

- Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.
- This multi-perspective analysis is important as there is no single correct way to analyses system requirements.
  •

Types of viewpoint:

1. **Interactor viewpoints**

– People or other systems that interact directly with the system. These viewpoints provide detailed system requirements covering the system features and interfaces. In an ATM, the customer's and the account database are interactor VPs.

2. **Indirect viewpoints**

– Stakeholders who do not use the system themselves but who influence the requirements. These viewpoints are more likely to provide higher-level organization requirements and constraints. In an ATM, management and security staff are indirect viewpoints.

3. **Domain viewpoints**

– Domain characteristics and constraints that influence the requirements. These viewpoints normally provide domain constraints that apply to the system. In an ATM, an example would be standards for inter-bank communications.

Typically, these viewpoints provide different types of requirements.

Viewpoint identification:

- Identify viewpoints using

– Providers and receivers of system services;
– Systems that interact directly with the system being specified;
– Regulations and standards;
– Sources of business and non-functional requirements.
– Engineers who have to develop and maintain the system;
– Marketing and other business viewpoints.

3) REQUIREMENTS VALIDATION

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking:
- ***Validity***: Does the system provide the functions which best support the customer's needs?
- ***Consistency***: Are there any requirements conflicts?
- ***Completeness***: Are all functions required by the customer included?
- ***Realism***: Can the requirements be implemented given available budget and technology
- ***Verifiability***: Can the requirements be checked?

Requirements validation techniques
- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews:
- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks:
- ***Verifiability*:** Is the requirement realistically testable?
- ***Comprehensibility*:** Is the requirement properly understood?
- ***Traceability*:** Is the origin of the requirement clearly stated?
- ***Adaptability*:** Can the requirement be changed without a large impact on other requirements?

4) REQUIREMENTS MANAGEMENT
- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably incomplete and inconsistent
  - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
  - Different viewpoints have different requirements and these are often contradictory.

Requirements change
- The priority of requirements from different viewpoints changes during the development process.
- System customers may specify requirements from a business perspective that conflict with end-user requirements.

- The business and technical environment of the system changes during its development.

Requirements evolution:

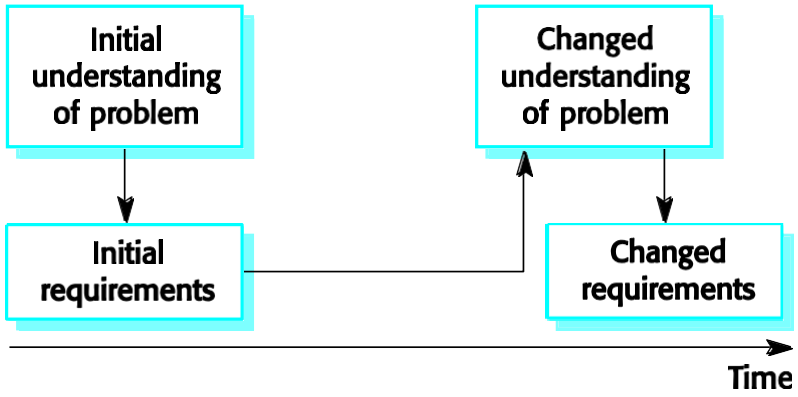


Figure 2: Requirement Evolution

4.1) Enduring and volatile requirements:

- ***Enduring requirements***: Stable requirements derived from the core activity of the customer organization. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models
- ***Volatile requirements***: Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

Requirements classification:

quirement Type	**Description**
Mutable requirements	Requirements that change because of changes to the environment in which the Organization is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops During the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organizations processes and open up new ways of working which generate new system requirements

Compatibility requirements	Requirements that depend on the particular systems or business processes within an organization. As these change, the compatibility requirements on the commissioned Or delivered system may also have to evolve.

4.2) Requirements management planning:

- During the requirements engineering process, you have to plan:
  - Requirements identification
- How requirements are individually identified;
  - A change management process
- The process followed when analyzing a requirements change;
  - Traceability policies
- The amount of information about requirements relationships that is maintained;
  - CASE tool support
- The tool support required to help manage requirements change;

Traceability:

Traceability is concerned with the relationships between requirements, their sources and the system design

- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability - Links from the requirements to the design;

CASE tool support:

- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management
  - Automated retrieval of the links between requirements.

An analysis and design workbench

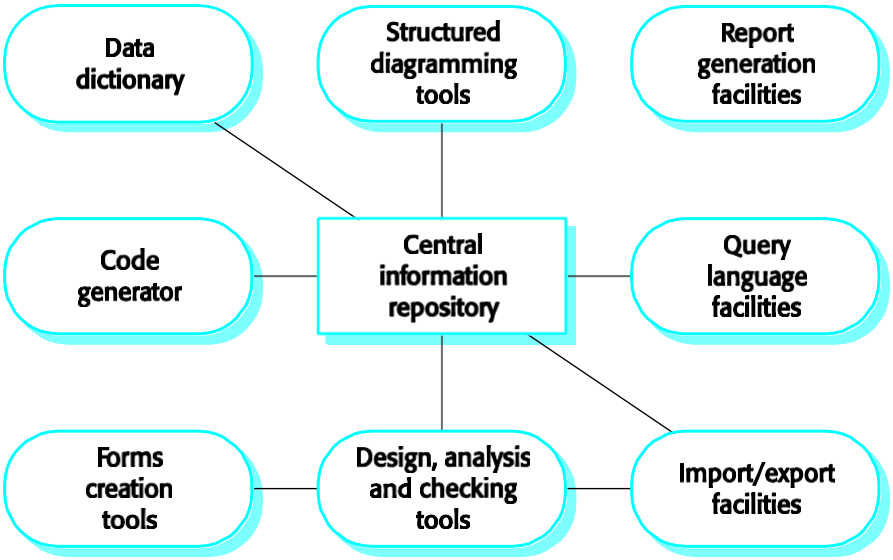


Figure 3: Analysis and Design benchmark

Analysis workbench components:

- Diagram editors
- Model analysis and checking tools
- Repository and associated query language
- Data dictionary
- Report definition and generation tools
- Forms definition tools
- Import/export translators
- Code generation tools

3. DESIGN AND DEVELOPMENT

Design engineering encompasses the **set of principles, concepts, and practices** that lead to thedevelopment of a high- quality system or product.

✓ Design principles establish an overriding philosophy that guides the designer in the work that isperformed.

✓ Design concepts must be understood before the mechanics of design practice are applied and

✓ Design practice itself leads to the creation of various representations of the software that serve as aguide for the construction activity that follows.

What is design?

Design is what virtually every engineer wants to do. It is the place where creativity rules – customer'srequirements, business needs, and technical considerations all come together in the formulation of a product or a system. Design creates a representation or model of the software, but unlike the analysis model, the design model provides detail about software data structures, architecture, interfaces, and components that are necessary to implement the system.

Why is it important?

Design allows a software engineer to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end – users becomeinvolved in large numbers. Design is the place where software quality is established.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence. Another **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

1) DESIGN PROCESS AND DESIGN QUALITY:

Software design is an iterative process through which requirements are translated into a"blueprint" for constructing the software.

Goals of design:

McLaughlin suggests three characteristics that serve as a guide for the evaluation of a good design.

- The design must implement all of the explicit requirements contained in the analysis model, and itmust accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for thosewho test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, andbehavioral domains from an implementation perspective.

Quality guidelines:

In order to evaluate the quality of a design representation we must establish technical criteria for gooddesign. These are the following guidelines:

1. A design should exhibit an architecture that
   a. has been created using recognizable architectural styles or patterns
   b. is composed of components that exhibit good design characteristics and
   c. Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements orsubsystems.
3. A design should contain distinct representation of data, architecture, interfaces and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented andare drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interface that reduce the complexity of connections between componentsand with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communication its meaning.

These design guidelines are not achieved by chance. Design engineering encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

Quality attributes:
The FURPS quality attributes represent a target for all software design:

- ***Functionality*** is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ***Usability*** is assessed by considering human factors, overall aesthetics, consistency and documentation.
- ***Reliability*** is evaluated by measuring the frequency and severity of failure, the accuracy of output results, and the mean – time –to- failure (MTTF), the ability to recover from failure, and the predictability of the program.
- ***Performance*** is measured by processing speed, response time, resource consumption, throughput,and efficiency
- ***Supportability c***ombines the ability to extend the program (extensibility), adaptability, serviceability- these three attributes represent a more common term maintainability

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security.
Another may demand performance with particular emphasis on processing speed.

A third might focus on reliability.

2) DESIGN CONCEPTS:

M.A Jackson once said: "The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide thenecessary framework for "getting it right."

I. **Abstraction:** Many levels of abstraction are there.

- ✓ At the highest level of abstraction, a solution is stated in broad terms using the language of theproblem environment.
- ✓ At lower levels of abstraction, a more detailed description of the solution is provided.

A ***procedural abstraction*** refers to a sequence of instructions that have a specific and limited function. The name of procedural abstraction implies these functions, but specific details are suppressed.
A ***data abstraction*** is a named collection of data that describes a data object.
In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing operation, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

II. Architecture:

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system". In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

One **goal** of software design is to derive an architectural rendering of a system. The rendering serves as a framework from which more detailed design activities are conducted.

The architectural design can be represented using one or more of a number of different models. ***Structured models*** represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function external events.

Process models focus on the design of the business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

III. Patterns:

Brad Appleton defines a ***design pattern*** in the following manner: "a pattern is a named nugget of inside which conveys that essence of a proven solution to a recurring problem within a certain context amidst competing concerns." Stated in another way, a design pattern describes a design structure that solves a particular design within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
The intent of each design pattern is to provide a description that enables a designer to determine

1) Whether the pattern is capable to the current work,
2) Whether the pattern can be reused,
3) Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

IV. Modularity:

Software architecture and design patterns embody ***modularity*;** software is divided into separately named and addressable components, sometimes called ***modules*** that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

The "divide and conquer" strategy- it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to modularity

and software. If we subdivide software indefinitely, the effort required to develop it will become negligibly small. The effort to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort associated with integrating the modules also grow.

Under modularity or over modularity should be avoided. We modularize a design so that development can be more easily planned; software increment can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

V. Information Hiding:

The principle of *information hiding* suggests that modules be "characterized by design decision that hides from all others."

Modules should be specified and designed so that information contained within a module is inaccessible toother modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modulesthat communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and local data structure used by module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within software.

VI. Functional Independence:

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. *Functional independence* is achieved by developing modules with "single minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific sub function of requirements and has a simple interface when viewed from other parts of the program structure.

Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified. Independent sign or code modifications are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling. *Cohesion* is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules. Cohesion is a natural extension of the information hiding.

A cohesion module performs a single task, requiring little interaction with other

components in other parts of a program. Stated simply, a cohesive module should do just one thing.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagates throughout a system.

VII. Refinement:

Stepwise refinement is a top- down design strategy originally proposed by Nicklaus wirth. A program is development by successively refining levels of procedural detail. A hierarchy is development by decomposing a macroscopic statement of function in a step wise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

VIII. Refactoring:

Refactoring is a reorganization technique that simplifies the design of a component without changing its function or behavior. Fowler defines refactoring in the following manner: "refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The designer may decide that the component should be refactored into 3 separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

IX. Design classes:

The software team must define a set of design classes that

1. Refine the analysis classes by providing design detail that will enable the classes to be implemented, and
2. Create a new set of design classes that implement a software infrastructure to support the design solution.

Five different types of design classes, each representing a different layer of the design architecture are suggested.

- **User interface classes:** define all abstractions that are necessary for human computer

interaction. In many cases, HCL occurs within the context of a *metaphor* and the design classes for the interface may be visual representations of the elements of the metaphor.

- **Business domain classes:** are often refinements of the analysis classes defined earlier. The classesidentify the attributes and services that are required to implement some element of the business domain.
- **Process classes** implement lower – level business abstractions required to fully manage the business domain classes.
- **Persistent classes** represent data stores that will persist beyond the execution of the software.
- **System classes** implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the design model evolves, the software team must develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. Each design class be reviewed to ensure that it is "well-formed." They define **four characteristics of a well- formed design class.**

Complete and sufficient: A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness: Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.

High cohesion: A cohesive design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.

Low coupling: Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of classes in other subsystems. This restriction, called the *law of Demeter,* suggests that a method should only sent messages to methods in neighboring classes.

1) SOFTWARE ARCHITECTURE: What Is Architecture?

Architectural design represents the structure of data and program components that are required tobuild a computer-based system. It considers

- the architectural style that the system will take,
- the structure and properties of the components that constitute the system, and
- The interrelationships that occur among all architectural components of a system.

The architecture is a representation that enables a software engineer to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) Consider architectural alternatives at a stage when making design changes is still relativelyeasy, (3) reducing the risks associated with the construction of the software.

The design of software architecture considers two levels of the design pyramid

- data design
- Architectural design.

✓ Data design enables us to represent the data component of the architecture.
✓ Architectural design focuses on the representation of the structure of software components, theirproperties, and interactions.

Why Is Architecture Important?

Bass and his colleagues [BAS98] identify three key reasons that software architecture is important:

- Representations of software architecture are an enabler for communication between all parties(stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system isstructured and how its components work together"

2) DATA DESIGN:

The data design activity translates data objects as part of the analysis model into data structures atthe software component level and, when necessary, a database architecture at the application level.

➢ At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

➢ At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system.

➢ At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself.

2.1) Data design at the Architectural Level:

The challenge for a business has been to extract useful information from this data environment, particularlywhen the information desired is cross functional.

To solve this challenge, the business IT community has developed ***data mining*** techniques, also called ***knowledge discovery in databases*** (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. An alternative solution, called a ***data warehouse***, adds an additional layer to the data architecture. A data warehouse is a large, independent database that encompassessome, but not all, of the data that are stored in databases that serve the set of applications required by a business.

2.2) Data design at the Component Level:

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. The following set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low-level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

3) ARCHITECTURAL STYLES AND PATTERNS:

The builder has used an ***architectural style*** as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod).

The software that is built for computer-based systems also exhibits one of many architectural

Styles.

Each style describes a system category that encompasses

(1) A set of ***components*** (e.g., a database, computational modules) that perform a function required by a system;

(2) A set of ***connectors*** that enable "communication, coordination's and cooperation" among components;

(3) ***Constraints*** that define how components can be integrated to form the system; and

(4) ***Semantic models*** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

An ***architectural pattern, like*** an architectural style, imposes a transformation the design of architecture. However, a pattern differs from a style in a number of fundamental ways:

(1) The scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety.

(2) A pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level.

(3) Architectural patterns tend to address specific behavioral issues within the context of the architectural.

USER INTERFACE DESIGN

Interface design focuses on three areas of concern:

(1) the design of interfaces between software components,

(2) the design of interfaces between the software and other nonhuman producers and

consumers ofinformation (i.e., other external entities), and
(3) The design of the interface between a human (i.e., the user) and the computer.

What is User Interface Design?

User interface design creates an effective communication medium between a human And a computer. Following a set of interface design principles, design identifies interface objects andactions and then creates a screen layout that forms the basis for a user interface prototype.

Why is User Interface Design important?

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. Because it molds a user's perception of the software, the interface has to be right.

1.1 THE GOLDEN RULES

Theo Mandel coins three "golden rules":

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide thisimportant software design activity.

Place the User in Control

Mandel [MAN97] defines a number of design principles that allow the user to maintain control:

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**. Word processor – spell checking – move to edit and back; enter and exit with little or no effort
- **Provide for flexible interaction**. Several modes of interaction – keyboard, mouse, digitizer pen orvoice recognition, but not every action is amenable to every interaction need. Difficult to draw a circle using keyboard commands.
- **Allow user interaction to be interruptible and undoable**. User stop and do something and then resume where left off. Be able to undo any action.
- **Streamline interaction as skill levels advance and allow the interaction to be customized**. Perform same actions repeatedly; have macro mechanism so user can customize interface.
- **Hide technical internals from the casual user**. Never required to use OS commands; file management functions or other arcane computing technology.
- **Design for direct interaction with objects that appear on the screen.** User has feel of control when interact directly with objects; stretch an object.

Reduce the User's Memory Load:

- ✓ The more a user has to remember, the more error-prone interaction with the system will be.
- ✓ Good interface design does not tax the user's memory
- ✓ System should remember pertinent details and assist the user with interaction scenario that assists user recall.

Mandel defines design principles that enable an interface to reduce the user's memory load:

- ➢ **Reduce demand on short-term memory**. Complex tasks can put a significant burden on short term memory. System designed to reduce the requirement to remember past actions and results; visual cues to recognize past actions, rather than recall them.
- ➢ **Establish meaningful defaults**. Initial defaults for average user; but specify individual preferences with a reset option.
- ➢ **Define shortcuts that are intuitive**. Use mnemonics like Alt-P.
- ➢ **The visual layout of the interface should be based on a real world metaphor**. Bill payment – check book and check register metaphor to guide a user through the bill paying process; user has less to memorize
- ➢ **Disclose information in a progressive fashion**. Organize hierarchically. High level of abstraction and then elaborate. Word underlining function – number of functions, but not all listed. User picks underlining then all options presented

Make the Interface Consistent

Interface present and acquire information in a consistent fashion.

1. All visual information is organized to a design standard for all screen displays
2. Input mechanisms are constrained to limited set used consistently throughout the application
3. Mechanisms for navigation from task to task are consistently defined and implementedMandel [MAN97] defines a set of design principles that help make the interface consistent:

- ➢ **Allow the user to put the current task into a meaningful context.** Because of many screens and heavy interaction, it is important to provide indicators – window tiles, graphical icons, consistent color coding so that the user knows the context of the work at hand; where came from and alternatives of where to go.
- ➢ **Maintain consistency across a family of applications**. For applications or products implementation should use the same design rules so that consistency is maintained for all interaction
- ➢ **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.** Unless a compelling reason presents itself don't change interactive sequences that have become de facto standards. (alt-S to scaling)

1.2 USER INTERFACE DESIGN

1.2.1 Interface Design Models

Four different models come into play when a user interface is to be designed.

- The software engineer creates a *design model,*
- a human engineer (or the software engineer) establishes a *user model,*

- the end-user develops a mental image that is often called the *user's model* or the *system perception*, and
- The implementers of the system create an *implementation model.*

The role of interface designer is to reconcile these differences and derive a consistent representation of theinterface.

User Model: The user model establishes the profile of end-users of the system. To build an effective userinterface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [SHN90]. In addition, users can be categorized as

- ✓ Novices.
- ✓ Knowledgeable, intermittent users.
- ✓ Knowledgeable, frequent users.

Design Model: A design model of the entire system incorporates data, architectural, interface andprocedural representations of the software.

Mental Model: The user's mental model (system perception) is the image of the system that end-userscarry in their heads.

Implementation Model: The implementation model combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system syntax and semantics.

These models enable the interface designer to satisfy a key element of the most importantprinciple of user interface design: **"Know the user, know the tasks."**

1.2.2 The User Interface Design Process: (steps in interface design)

The user interface design process encompasses four distinct framework activities:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

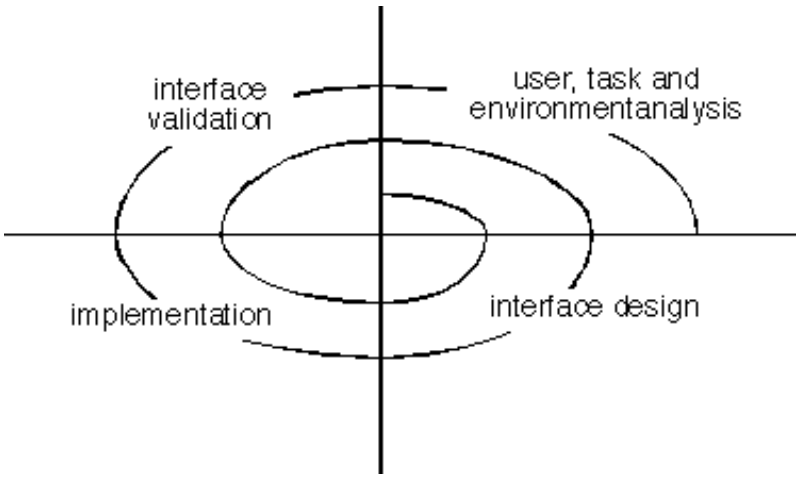


Figure 4: User Interface Design Process

(1) User Task and Environmental Analysis:

The interface analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, the software engineer attempts to understand the system perception (Section 15.2.1) for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

(2) Interface Design:

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

(3) Interface Construction (implementation)

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 15.5) may be used to complete the construction of the interface.

(4) Interface Validation:

Validation focuses on

(1) the ability of the interface to implement every user task correctly, to accommodate all taskvariations, and to achieve all general user requirements;

(2) the degree to which the interface is easy to use and easy to learn; and

(3) The users' acceptance of the interface as a useful tool in their work.

After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used e.g., questionnaires, rating sheets), the designer may extract information form these data (e.g., 80percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indicationof the amount of learning required by user of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication oninteraction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load onusers of the system.
4. Interface styles, help facilities, and error handling protocol provide a general indication of the complexityof the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect 2qualitaive data, questionnaires can be distributed to users of the prototype. Questions can be (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert scales (e.g., strongly.

Users are observed during interaction, and data-such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period-are collected and used as a guide for interface modification.

4. TESTING AND MAINTANANCE

A strategic Approach for Software testing

- Software Testing
- One of the important phases of software development
- Testing is the process of execution of a program with the intention of finding errors
- Involves 40% of total project cost
- Testing Strategy
- A road map that incorporates test planning, test case design, test execution and resultant data

collection and execution

- **Validation** refers to a different set of activities that ensures that the software is traceable to the customer requirements.
- V&V encompasses a wide array of Software Quality Assurance
- Perform Formal Technical reviews(FTR) to uncover errors during software development
- Begin testing at component level and move outward to integration of entire component based system.
- Adopt testing techniques relevant to stages of testing
- Testing can be done by software developer and independent testing group
- Testing and debugging are different activities. Debugging follows testing
- Low level tests verifies small code segments.
- High level tests validate major system functions against customer requirements

Testing Strategies for Conventional Software 1) Unit Testing
Integration Testing 3) Validation Testing and 4) System Testing

Criteria for completion of software testing

- Nobody is absolutely certain that software will not fail
- Based on statistical modeling and software reliability models
- 95 percent confidence(probability) that 1000 CPU hours of failure free operation is at least 0.995

Software Testing

- Two major categories of software testing
- ❖ Black box testing
- ❖ White box testing

Black box testing

Treats the system as black box whose behavior can be determined by studying its input and related output not concerned with the internal structure of the program

Black Box Testing

- It focuses on the functional requirements of the software i.e. it enables the sw engineer to derive a set of input conditions that fully exercise all the functional requirements for that program.
    - Concerned with functionality and implementation 1)Graph based testing method

Equivalence partitioning Graph based testing

- Draw a graph of objects and relations
- Devise test cases to uncover the graph such that each object and its relationship exercised.

Equivalence partitioning

- Divides all possible inputs into classes such that there are a finite equivalence classes.
- Equivalence class

  -- Set of objects that can be linked by relationship
- Reduces the cost of testing

- Example
- Input consists of 1 to 10
- Then classes are n<1,1<=n<=10,n>10
- Choose one valid class with value within the allowed range and two invalid classes where values are greater than maximum value and smaller than minimum value.

Boundary Value analysis

- Select input from equivalence classes such that the input lies at the edge of the equivalence classes
- Set of data lies on the edge or boundary of a class of input data or generates the data that lies at the boundary of a class of output data

Example

- If 0.0<=x<=1.0
    - Then test cases (0.0,1.0) for valid input and (-0.1 and 1.1) for invalid input

Orthogonal array Testing

- To problems in which input domain is relatively small but too large for exhaustive testing

White Box testing

- Also called glass box testing
- Involves knowing the internal working of a program
- Guarantees that all independent paths will be exercised at least once.
- Exercises all logical decisions on their true and false sides
- Executes all loops
- Exercises all data structures for their validity
- White box testing techniques

1. Basis path testing
2. Control structure testing

Basis path testing

- Proposed by Tom McCabe
- Defines a basic set of execution paths based on logical complexity of a procedural design
- Guarantees to execute every statement in the program at least once
- Steps of Basis Path Testing
- Draw the flow graph from flow chart of the program
- Calculate the cyclamate complexity of the resultant flow graph
- Prepare test cases that will force execution of each path
- Three methods to compute Cyclomatic complexity number
- V(G)=E-N+2(E is number of edges, N is number of nodes
- V(G)=Number of regions
- V(G)= Number of predicates +1
- Control Structure testing
- Basis path testing is simple and effective
- It is not sufficient in itself
- Control structure broadens the basic test coverage and improves the quality of white box

testing

- Condition Testing
- Data flow Testing
- Loop Testing
  •

Condition Testing

- --Exercise the logical conditions contained in a program module
- --Focuses on testing each condition in the program to ensure that it does contain errors
- --Simple condition
- E1<relation operator>E2
- --Compound condition
- simple condition<Boolean operator>simple condition

Data flow Testing

- Selects test paths according to the locations of definitions and use of variables in a program
- Aims to ensure that the definitions of variables and subsequent use is tested
- First construct a definition-use graph from the control flow of a program

Loop Testing

- Focuses on the validity of loop constructs
- Four categories can be defined

1. Simple loops
2. Nested loops
3. Concatenated loops
4. Unstructured loops Testing of simple loops
   -- N is the maximum number of allowable passes through the loop

   Skip the loop entirely
   Only one pass through the loop Two passes through the loop
   m passes through the loop where m>N N-1,N,N+1 passes the loop
   Nested Loops

1. Start at the innermost loop. Set all other loops to maximum values
2. Conduct simple loop test for the innermost loop while holding the outer loops at their minimum iteration parameter.
3. Work outward conducting tests for the next loop but keeping all other loops at minimum. Concatenated loops

- Follow the approach defined for simple loops, if each of the loop is independent of other.
- If the loops are not independent, then follow the approach for the nested loops Unstructured Loops
- Redesign the program to avoid unstructured loops Validation Testing
- It succeeds when the software functions in a manner that can be reasonably expected by the customer.

  1) Validation Test Criteria 2)Configuration Review 3)Alpha And Beta Testing System Testing

- Its primary purpose is to test the complete software. 1)Recovery Testing
  2) Security Testing 3Stress Testing and 4)Performance Testing The Art of Debugging
- Debugging occurs as a consequences of successful testing.

- Debugging Strategies 1)Brute Force Method. 2)Back Tracking 3)Cause Elimination and 4)Automated debugging
- Brute force

  -- Most common and least efficient

  -- Applied when all else fails

  -- Memory dumps are taken

  -- Tries to find the cause from the load of information
- Back tracking

  -- Common debugging approach

  -- Useful for small programs

  -- Beginning at the system where the symptom has been uncovered, the source code traced backward until the site of the cause is found.

- Cause Elimination

  -- Based on the concept of Binary partitioning

  -- A list of all possible causes is developed and tests are conducted to eliminate each

Software Quality and Maintenance

- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- Factors that affect software quality can be categorized in two broad groups:

1. Factors that can be directly measured (e.g. defects uncovered during testing)
2. Factors that can be measured only indirectly (e.g. usability or maintainability)

- McCall's quality factors

1. Product operation

a. Correctness
b. Reliability
c. Efficiency
d. Integrity
e. Usability

2. Product Revision

a. Maintainability
b. Flexibility
c. Testability

3. Product Transition

a. Portability
b. Reusability

c. Interoperability ISO 9126 Quality Factors 1.Functionality

2. Reliability 3.Usability 4.Efficiency 5.Maintainability 6.Portability

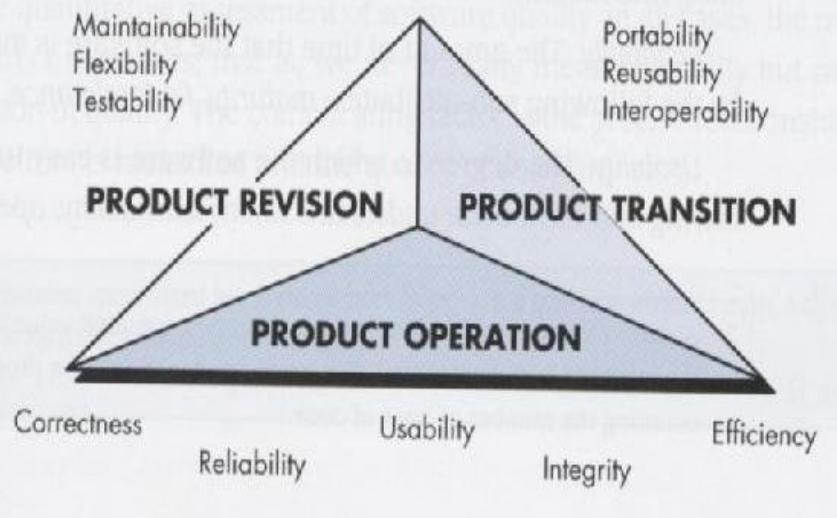


Figure 5: Software Maintenance

Product metrics

- Product metrics for computer software helps us to assess quality.
- Measure
  -- Provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process
- Metric(IEEE 93 definition)
  -- A quantitative measure of the degree to which a system, component or process possess a given attribute
- Indicator
  -- A metric or a combination of metrics that provide insight into the software process, a software project or a product itself

Product Metrics for analysis, Design, Test and maintenance

- **Product metrics for the Analysis model**

❖ Function point Metric

- First proposed by Albrecht
- Measures the functionality delivered by the system
- FP computed from the following parameters

1) Number of external inputs(EIS)
2) Number external outputs(EOS)
3) Number of external Inquiries(EQS)
4) Number of Internal Logical Files(ILF)
5) Number of external interface files(EIFS)

Each parameter is classified as simple, average or complex and weights are assigned as follows

Information Domain	Count	Simple	avg	Complex
EIS	3	4	6	
EOS	4	5	7	
EQS	3	4	6	
ILFS	7	10	15	
EIFS	5	7	10	

FP=Count Total *[0.65+0.01*E(Fi)] Metrics for Design Model

- DSQI(Design Structure Quality Index)
- US air force has designed the DSQI
- Compute s1 to s7 from data and architectural design
- S1:Total number of modules
- S2:Number of modules whose correct function depends on the data input
- S3:Number of modules whose function depends on prior processing
- S4:Number of data base items
- S5:Number of unique database items
- S6: Number of database segments
- S7:Number of modules with single entry and exit
- Calculate D1 to D6 from s1 to s7 as follows:
- D1=1 if standard design is followed otherwise D1=0
- D2(module independence)=(1-(s2/s1))
- D3(module not depending on prior processing)=(1-(s3/s1))
- D4(Data base size)=(1-(s5/s4))
- D5(Database compartmentalization)=(1-(s6/s4)
- D6(Module entry/exit characteristics)=(1-(s7/s1))
- DSQI=sigma of WiDi
- i=1 to 6,Wi is weight assigned to Di
- If sigma of wi is 1 then all weights are equal to 0.167
- DSQI of present design be compared with past DSQI. If DSQI is significantly lower than the average, further design work and review are indicated
- METRIC FOR SOURCE CODE
- HSS(Halstead Software science)
- Primitive measure that may be derived after the code is generated or estimated once design is complete
- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrence.
- Overall program length N can be computed:
- $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$
- $V = N \log_2 (n_1 + n_2)$ METRIC FOR TESTING
- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.

- N_2 = the total number of operand occurrence.
- Program Level and Effort
- PL = 1/[(n1 / 2) x (N_2 / n2 l)]
- e = V/PL
-

METRICS FOR MAINTENANCE

- M_t = the number of modules in the current release
- F_c = the number of modules in the current release that have been changed
- F_a = the number of modules in the current release that have been added.
- F_d = the number of modules from the preceding release that were deleted in the current release
- The Software Maturity Index, SMI, is defined as:
- SMI = [M_t – (F_c + F_a + F_d)/ M_t]

1) METRICS FOR SOFTWARE QUALITY

The overriding goal of software engineering is to produce a high-quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal, software engineers must apply effective methods coupled with modern tools within the context of a mature software process.

2.1 Measuring Quality

The measures of software quality are correctness, maintainability, integrity, and usability. These measures will provide useful indicators for the project team.

- **Correctness.** Correctness is the degree to which the software performs its required function. The most common measure for correctness is defects per KLOC, where a defect is defined as a verified lack of conformance to requirements.
- **Maintainability.** Maintainability is the ease with which a program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the customer desires a change in requirements. A simple time-oriented metric is ***mean-time-tochange*** (MTTC), the time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all users.
- **Integrity.** Attacks can be made on all three components of software: programs, data, and documents.

  To measure integrity, two additional attributes must be defined: threat and security. *Threat* is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. *Security* is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

  Integrity = $\sum$ [1 –(threat × □(1 – security))]
- **Usability:** Usability is an attempt to quantify user-friendliness and can be measured in terms of four characteristics:

2.2 Defect Removal Efficiency

A quality metric that provides benefit at both the project and process level is defect removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality

assurance and control activities as they are applied throughout all process framework activities.

When considered for a project as a whole, DRE is defined in the following manner: DRE = $E/ (E + D)$

Where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.

Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as

$DRE_i = E_i/ (E_i+E_{i+1})$

E_i is the number of errors found during software engineering activity i and

E_{i+1} is the number of errors found during software engineering activity $i+1$ that are traceable to errors that were not discovered in software engineering activity i.

A quality objective for a software team (or an individual software engineer) is to achieve DRE that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.

Software Maintenance

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes .A common perception of maintenance is that it merely involves fixing defects. However, one study indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions. This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system. Software maintenance is needed to correct error enhance feature, portability to new platform.

Software maintenance task:

There are basically three types of software maintenance. These are:

• Corrective:

Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.

• Adaptive:

A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

• Perfective:

A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

UNIT III -SUPPORTING QUALITY ACTIVITIES

Metrics, Reviews –SCM – Software quality assurance and risk management

1. What is Software Testing Metric?

Software Testing Metric is be defined as a quantitative measure that helps to estimate the progress, quality, and health of a software testing effort. A Metric defines in quantitative terms the degree to which a system, system component, or process possesses a given attribute.

The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.

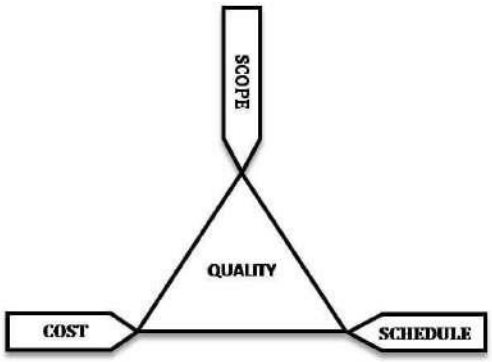


Figure 1. Software Quality

Software testing metrics - Improves the efficiency and effectiveness of a software testing process.

Software testing metrics or software test measurement is the quantitative indication of extent, capacity, dimension, amount or size of some attribute of a process or product.

2. Why Test Metrics are Important?

- "We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.
- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required

- Take decision or process or technology change

3.Software metrics can be classified into three categories –

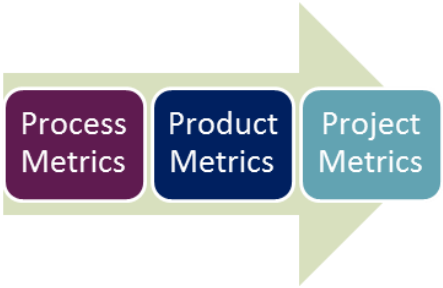


Figure2. Software Metric

Product metrics − Describes the characteristics of the product such as size, complexity, design features, performance, and quality level.

Process metrics − These characteristics can be used to improve the development and maintenance activities of the software.

Project metrics − This metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Some metrics belong to multiple categories. For example, the in-process quality metrics of a project are both process metrics and project metrics.

4.Software quality metrics

are a subset of software metrics that focus on the quality aspects of the product, process, and project. These are more closely associated with process and product metrics than with project metrics.

Software quality metrics can be further divided into three categories −

I. Product quality metrics

II. In-process quality metrics

III. Maintenance quality metrics

I. Product Quality Metrics

This metrics include the following −

- Mean Time to Failure
- Defect Density
- Customer Problems
- Customer Satisfaction

Mean Time to Failure

It is the time between failures. This metric is mostly used with safety critical systems such as the airline traffic control systems, avionics, and weapons.

Defect Density

It measures the defects relative to the software size expressed as lines of code or function point, etc. i.e., it measures code quality per unit. This metric is used in many commercial software systems.

Customer Problems

It measures the problems that customers encounter when using the product. It contains the customer's perspective towards the problem space of the software, which includes the non-defect oriented problems together with the defect problems.

The problems metric is usually expressed in terms of **Problems per User-Month (PUM)**.

PUM = Total Problems that customers reported (true defect and non-defect oriented

problems) for a time period + Total number of license months of the software during the period

Where,Number of license-month of the software = Number of install license of the software ×Number of months in the calculation period

PUM is usually calculated for each month after the software is released to the market, and also for monthly averages by year.

Customer Satisfaction

Customer satisfaction is often measured by customer survey data through the five-point scale –

Very satisfied

Satisfied

Neutral

Dissatisfied

Very dissatisfied

Satisfaction with the overall quality of the product and its specific dimensions is usually obtained through various methods of customer surveys. Based on the five-point-scale data, several metrics with slight variations can be constructed and used, depending on the purpose of analysis. For example −

Percent of completely satisfied customers

Percent of satisfied customers

Percent of dis-satisfied customers

Percent of non-satisfied customers

Usually, this percent satisfaction is used.

II. In-process Quality Metrics

In-process quality metrics deals with the tracking of defect arrival during formal machine testing for some organizations. This metric includes −

- Defect density during machine testing
- Defect arrival pattern during machine testing
- Phase-based defect removal pattern
- Defect removal effectiveness

Defect density during machine testing

Defect rate during formal machine testing (testing after code is integrated into the system library) is correlated with the defect rate in the field. Higher defect rates found during testing is an indicator that the software has experienced higher error injection during its development process, unless the higher testing defect rate is due to an extraordinary testing effort.

This simple metric of defects per KLOC or function point is a good indicator of quality, while the software is still being tested. It is especially useful to monitor subsequent releases of a product in the same development organization.

Defect arrival pattern during machine testing

The overall defect density during testing will provide only the summary of the defects. The pattern of defect arrivals gives more information about different quality levels in the field. It includes the following −

The defect arrivals or defects reported during the testing phase by time interval (e.g., week). Here all of which will not be valid defects.

The pattern of valid defect arrivals when problem determination is done on the reported problems. This is the true defect pattern.

The pattern of defect backlog overtime. This metric is needed because development organizations cannot investigate and fix all the reported problems immediately. This is a workload statement as well as a quality statement. If the defect backlog is large at the end of the development cycle and a lot of fixes have yet to be integrated into the system, the stability of the system (hence its quality) will be affected. Retesting (regression test) is needed to ensure that targeted product quality levels are reached.

Phase-based defect removal pattern

This is an extension of the defect density metric during testing. In addition to testing, it tracks

the defects at all phases of the development cycle, including the design reviews, code inspections, and formal verifications before testing.

Because a large percentage of programming defects is related to design problems, conducting formal reviews or functional verifications to enhance the defect removal capability of the process at the front-end reduces error in the software. The pattern of phase-based defect removal reflects the overall defect removal ability of the development process.

With regard to the metrics for the design and coding phases, in addition to defect rates, many development organizations use metrics such as inspection coverage and inspection effort for in-process quality management.

Defect removal effectiveness

It can be defined as follows −

DRE=Defect removed during a development phase Defects latent in the product×100%

This metric can be calculated for the entire development process, for the front-end before code integration and for each phase. It is called **early defect removal** when used for the front-end and **phase effectiveness** for specific phases. The higher the value of the metric, the more effective the development process and the fewer the defects passed to the next phase or to the field. This metric is a key concept of the defect removal model for software development.

III. Maintenance Quality Metrics

Although much cannot be done to alter the quality of the product during this phase, following are the fixes that can be carried out to eliminate the defects as soon as possible with excellent fix quality.

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog and backlog management index

Fix backlog is related to the rate of defect arrivals and the rate at which fixes for reported problems become available. It is a simple count of reported problems that remain at the end of each month or each week. Using it in the format of a trend chart, this metric can provide meaningful information for managing the maintenance process.

Backlog Management Index (BMI) is used to manage the backlog of open and unresolved problems.

BMI=(Numberofproblemsclosedduringthemonth/Numberofproblemsarrivedduringthemonth)

×100%

If BMI is larger than 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

Fix response time and fix responsiveness

The fix response time metric is usually calculated as the mean time of all problems from open to close. Short fix response time leads to customer satisfaction.

The important elements of fix responsiveness are customer expectations, the agreed-to fix time, and the ability to meet one's commitment to the customer.

Fix Quality

Fix quality or the number of defective fixes is another important quality metric for the maintenance phase. A fix is defective if it did not fix the reported problem, or if it fixed the original problem but injected a new defect. For mission-critical software, defective fixes are detrimental to customer satisfaction. The metric of percent defective fixes is the percentage of all fixes in a time interval that is defective.

A defective fix can be recorded in two ways: Record it in the month it was discovered or record it in the month the fix was delivered. The first is a customer measure; the second is a process measure. The difference between the two dates is the latent period of the defective fix.

Usually the longer the latency, the more will be the customers that get affected. If the number of defects is large, then the small value of the percentage metric will show an optimistic picture. The quality goal for the maintenance process, of course, is zero defective fixes without delinquency.

5. Manual Test Metrics

In Software Engineering, Manual test metrics are classified into two classes

Base Metrics

Calculated Metrics

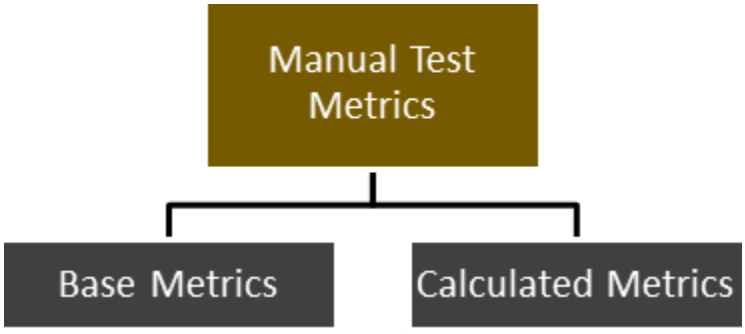


Figure 3. Metrics types

Base metrics is the raw data collected by Test Analyst during the test case development and

execution (**# of test cases executed, # of test cases**). While calculated metrics are derived from the data collected in base metrics. Calculated metrics is usually followed by the test manager for test reporting purpose (**% Complete, % Test Coverage**).

Depending on the project or business model some of the important metrics are

Test case execution productivity metrics

Test case preparation productivity metrics

Defect metrics

Defects by priority

Defects by severity

Defect slippage ratio

6. Test Metrics Life Cycle

Different stages of Metrics life cycle	**Steps during each stage**
Analysis	Identification of the Metrics Define the identified QA Metrics
Communicate	Explain the need for metric to stakeholder and testing team Educate the testing team about the data points to need to be captured processing the metric
Evaluation	Capture and verify the data Calculating the metrics value using the data captured
Report	Develop the report with an effective conclusion Distribute the report to the stakeholder and respective representative Take feedback from stakeholder

How to calculate Test Metric

Sr#	**Steps to test metrics**	**Example**

1	Identify the key software testing processes to be measured	Testing progress tracking process
2	In this Step, the tester uses the data as a baseline to define the metrics	The number of test cases planned to be executed per
3	Determination of the information to be followed, a frequency of tracking and the person responsible	The actual test execution per day will be captured by test manager at the end of the day
4	Effective calculation, management, and interpretation of the defined metrics	The actual test cases executed per day
5	Identify the areas of improvement depending on the interpretation of defined metrics	The Test Case execution falls below the goal set, need to investigate the reason and suggest improvement measures

Example of Test Metric

To understand how to calculate the test metrics, we will see an example of a percentage test case executed.

To obtain the execution status of the test cases in percentage, we use the formula.

Percentage test cases executed= (No of test cases executed/ Total no of test cases written) X 100

Likewise, you can calculate for other parameters like **test cases not executed, test cases passed, test cases failed, test cases blocked, etc.**

Test Metrics Glossary

Rework Effort Ratio = (Actual rework efforts spent in that phase/ total actual efforts spent in that phase) X 100

Requirement Creep = (Total number of requirements added/No of initial requirements)X100

Schedule Variance = (Actual efforts – estimated efforts) / Estimated Efforts) X 100

Cost of finding a defect in testing = (Total effort spent on testing/ defects found in testing)

Schedule slippage = (Actual end date – Estimated end date) / (Planned End Date – Planned Start Date) X 100

P**assed Test Cases Percentage** = (Number of Passed Tests/Total number of tests executed) X 100

Failed Test Cases Percentage = (Number of Failed Tests/Total number of tests executed) X 100

Blocked Test Cases Percentage = (Number of Blocked Tests/Total number of tests executed) X 100

Fixed Defects Percentage = (Defects Fixed/Defects Reported) X 100

Accepted Defects Percentage = (Defects Accepted as Valid by Dev Team /Total Defects Reported) X 100

Defects Deferred Percentage = (Defects deferred for future releases /Total Defects Reported) X 100

Critical Defects Percentage = (Critical Defects / Total Defects Reported) X 100

Average time for a development team to repair defects = (Total time taken for bugfixes/Number of bugs)

Number of tests run per time period = Number of tests run/Total time

Test design efficiency = Number of tests designed /Total time

Test review efficiency = Number of tests reviewed /Total time

Bug find rote or Number of defects per test hour = Total number of defects/Total number of test hours

7.Type of Software Testing Metrics

Based on the types of testing performed, following are the types of software testing metrics: -

1. Manual Testing Metrics
2. Performance Testing Metrics
3. Automation Testing Metrics

Following figure shows different software testing metrics.

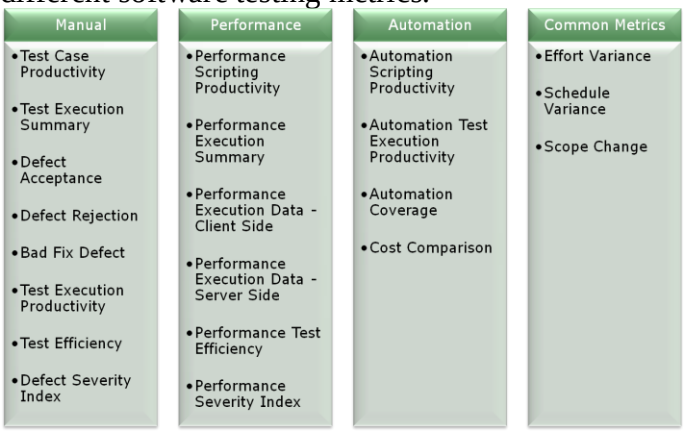


Figure 4. Software Testing Metrics

Let's have a look at each of them.

Manual Testing Metrics

Test Case Productivity (TCP)

This metric gives the test case writing productivity based on which one can have a conclusive remark.

$$\text{(check the ppt) Test Case Productivity} = \left\lceil \frac{\text{Total Raw Test Steps}}{\text{Efforts (hours)}} \right\rceil_{\text{Step(s)/hour}}$$

Example

Test Case Name	Raw Steps
XYZ_1	30
XYZ_2	32
XYZ_3	40
XYZ_4	36
XYZ_5	45
Total Raw Steps	**183**

Efforts took for writing 183 steps is 8 hours.
TCP=183/8=22.8
Test case productivity = 23 steps/hour

One can compare the Test case productivity value with the previous release(s) and draw the most effective conclusion from it.

TC Productivity Trend

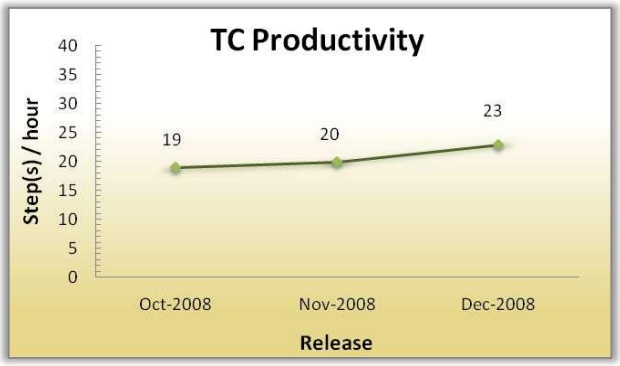


Figure 5. TC Productivity Trend

Test Execution Summary

This metric gives classification of the test cases with respect to status along with reason, if available, for various test cases. It gives the statical view of the release. One can collect the data for the number of test case executed with following status: -

- Pass.
- Fail and reason for failure.
- Unable to Test with reason. Some of the reasons for this status are time crunch, postponed defect, setup issue, out of scope.

Summary Trend

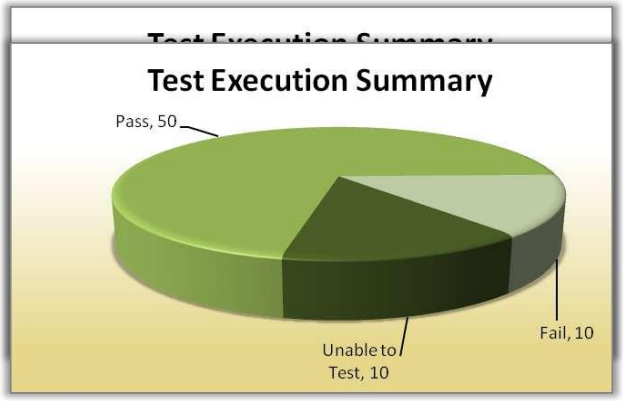


Figure 6. Summary Trend

One can also show the same trend for the classification of reasons for various unable to test and fail test cases.

Defect Acceptance (DA)

This metric determine the number of valid defects that testing team has identified during execution.

$$Defect\ Acceptance = \left\lceil \frac{Number\ of\ Valid\ Defects}{Total\ Number\ of\ Defects} * 100 \right\rceil \%$$

The value of this metric can be compared with previous release for getting better picture

Defect Acceptance Trend

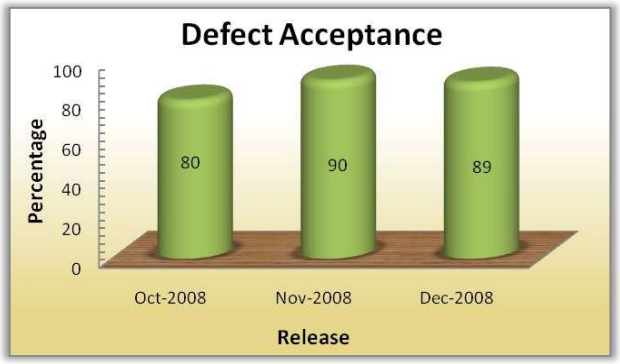


Figure 7. Defect Acceptance Trend

Defect Rejection (DR)

This metric determine the number of defects rejected during execution.

$$Defect\ Rejection = \left\lceil \frac{Number\ of\ Defect(s)\ Rejected}{Total\ Number\ of\ Defects} * 100 \right\rceil \%$$

$$\lfloor \qquad \rfloor$$

It gives the percentage of the invalid defect the testing team has opened and one can control, whenever required.

Defect Rejection Trend

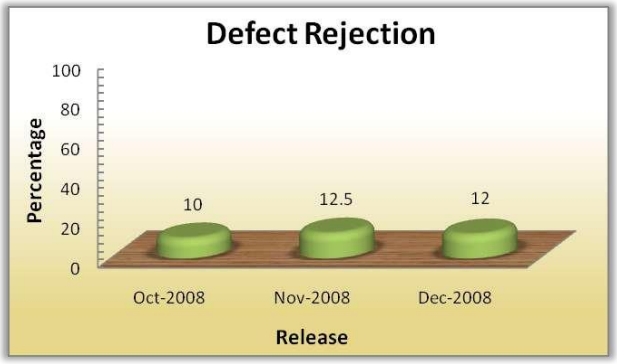


Figure 8. Defect Rejection Trend

Bad Fix Defect (B)

Defect whose resolution give rise to new defect(s) are bad fix defect.

This metric determine the effectiveness of defect resolution process.

$$Bad\ Fix\ Defect = \left\lceil \frac{Number\ of\ of\ Bad\ Fix\ Defect(s)}{Total\ Number\ of\ Valid\ Defects} * 100 \right\rceil \%$$

It gives the percentage of the bad defect resolution which needs to be controlled.

Bad Fix Defect Trend

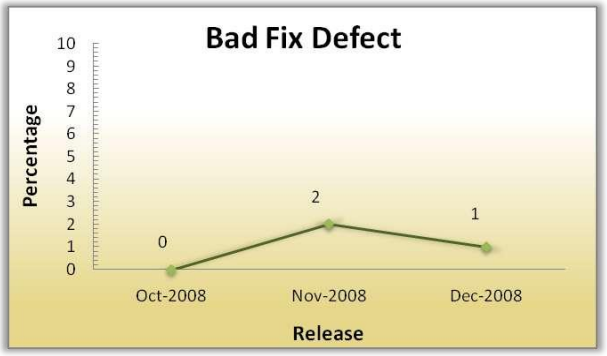


Figure 9. Bad fix

Test Execution Productivity (TEP)

This metric gives the test cases execution productivity which on further analysis can give conclusive result.

$$Test\ Execution\ Productivity = \left\lceil \frac{Total\ No.\ of\ TC\ executed\ (Te)}{Execution\ Efforts\ (hours)} * 8 \right\rceil Execution(s)/Day$$

Where Te is calculated as,

$$Te = Base\ Test\ Case + ((T(0.33) * 0.33) + (T(0.66) * 0.66) + (T(1) * 1))$$

Where,

Base Test Case = No. of TC executed atleast once.

T (1) = No. of TC Retested with 71% to 100% of Total TC steps

T (0.66) = No. of TC Retested with 41% to 70% of Total TC steps

T (0.33) = No. of TC Retested with 1% to 40% of Total TC steps

Table 1. Example

TC Name	Base Run Effort (hr)	Re-Run1 Status	Re-Run1 Effort (hr)	Re-Run2 Status	Re-Run2 Effort (hr)	Re-Run3 Status	Re-Run3 Effort (hr)
XYZ_1	2	T(0.66)	1	T(0.66)	0.45	T(1)	2
XYZ_2	1.3	T(0.33)	0.3	T(1)	2		
XYZ_3	2.3	T(1)	1.2				
XYZ_4	2	T(1)	2				
XYZ_5	2.15						

In above example,

Base Test Case	5
T(1)	4
T(0.66)	2
T(0.33)	1
Total Efforts(hr)	19.7

Te = 5 + ((1*4) + (2*0.66) + (1*0.33))) = 5 + 5.65 = 10.65

Test Execution Productivity = (10.65/19.7) * 8 = 4.3 Execution/day

One can compare the productivity with previous release and can have an effective conclusion.

Test Execution Productivity Trend

Figure 10. TEP

Test Efficiency (TE)

This metric determine the efficiency of the testing team in identifying the defects.
It also indicated the defects missed out during testing phase which migrated to the next phase.

$$Test\ Efficiency = \left\lceil \frac{DT}{DT + DU} * 100 \right\rceil \%$$

Where,

DT = Number of valid defects identified during testing.

DU = Number of valid defects identified by user after release of application. In other words, post-testing defect

Test Efficiency Trend

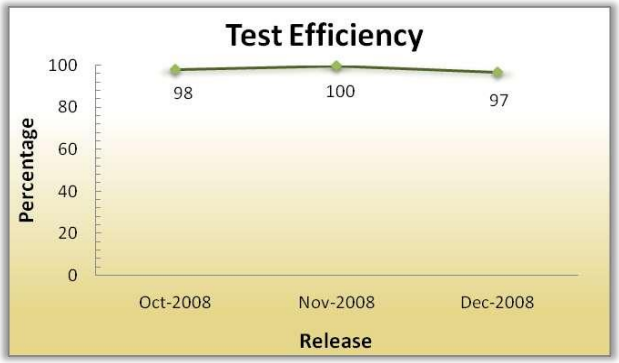


Figure 11. TE

Defect Severity Index (DSI)

This metric determine the quality of the product under test and at the time of release, based on which one can take decision for releasing of the product i.e. it indicates the product quality.

$$Defect\ Severity\ Index = \left\lceil \frac{\sum(Severity\ Index * No.\ of\ Valid\ Defect(s)\ for\ this\ severity}{Total\ Number\ of\ Valid\ Defects} \right\rceil$$

One can divide the Defect Severity Index in two parts: -

1. **DSI for All Status defect(s):** - This value gives the product quality under test.

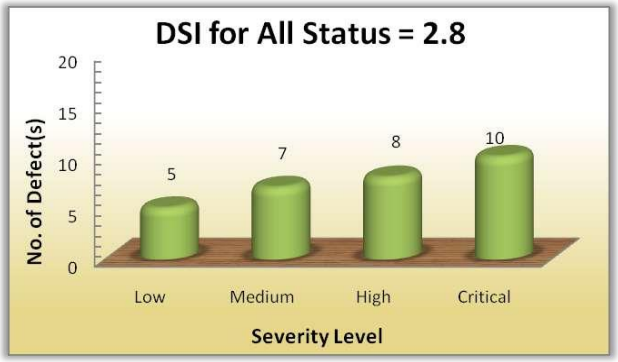
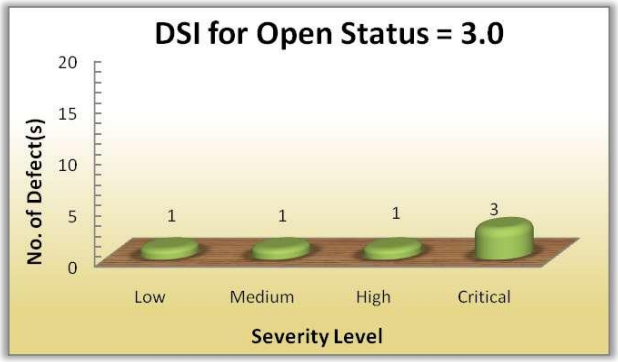


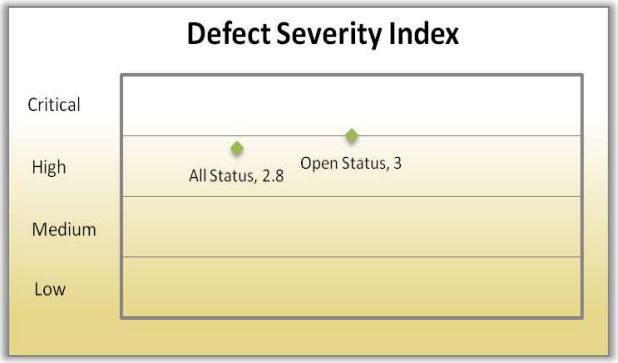
Figure 12. DSI

2. **DSI for Open Status defect(s):** - This value gives the product quality at the time of release. For calculation of DSI for this, only open status defect(s) must be considered.

$$DSI(Open) = \left\lceil \frac{\sum(Severity\ Index^{*}\ No.\ of\ Open\ Valid\ Defect(s)\ for\ this\ severity)}{Total\ Number\ of\ Open\ Valid\ Defects} \right\rceil$$



Defect Severity Index Trend



From the graph it is clear that

- Quality of product under test i.e. DSI – All Status = 2.8 (High Severity)
- Quality of product at the time of release i.e. DSI – Open Status = 3.0 (High Severity)

Performance Testing Metrics

Performance Scripting Productivity (PSP)

This metric gives the scripting productivity for performance test script and have trend over a period of time.

$$Performance\ Scripting\ Productivity = \left\lceil \frac{\sum Operations\ Performed}{Efforts\ (hours)} \right\rceil Operation(s)/hour$$

Where Operations performed is: -

1. No. of Click(s) i.e. click(s) on which data is refreshed.
2. No. of Input parameter
3. No. of Correlation parameter

Above evaluation process does include logic embedded into the script which is rarely used.

Example

Operation Performed	**Total**
No. of clicks	10
No. of Input Parameter	5
No. of Correlation Parameter	5
Total Operation Performed	**20**

Efforts took for scripting = 10 hours.

Performance scripting productivity =20/10=2 operations/hour

Performance Scripting Productivity Trend

Figure 13. PSP

Performance Execution Summary

This metric gives classification with respect to number of test conducted along with status (Pass/Fail), for various types of performance testing.

Some of the types of performance testing: -

1. Peak Volume Test.
2. Endurance/Soak Test.
3. Breakpoint/Stress Test.
4. Failover Test

Summary Trend

Performance Execution Data - Client Side

This metric gives the detail information of Client side data for execution.
Following are some of the data points of this metric -

1. Running Users
2. Response Time
3. Hits per Second
4. Throughput
5. Total Transaction per second
6. Time to first byte
7. Error per second

Performance Execution Data - Server Side

This metric gives the detail information of Server side date for execution.
Following are some of the data points of this metric -

1. CPU Utilization
2. Memory Utilization
3. HEAP Memory Utilization
4. Database connections per second

Performance Test Efficiency (PTE)

This metric determine the quality of the Performance testing team in meeting the requirements which can be used as an input for further improvisation, if required.

$$Performance\ Test\ Efficiency = \left\lceil \frac{(Req\ met\ during\ PT) - (Req\ not\ met\ after\ Signoff\ of\ PT)}{Req\ met\ during\ PT} * 100 \right\rceil \%$$

To evaluate this one need to collect data point during the performance testing and after the signoff of the performance testing.
Some of the requirements of Performance testing are: -

1. Average response time.
2. Transaction per Second.
3. Application must be able to handle predefined max user load.
4. Server Stability.

Example

Consider during the performance testing above mentioned requirements were met.

Requirement met during PT = 4

In production, average response time is greater than expected, then

Requirement not met after Signoff of PT = 1

PTE = (4 / (4+1)) * 100 = 80%

Performance Testing Efficiency is 80%

Performance Test Efficiency Trend

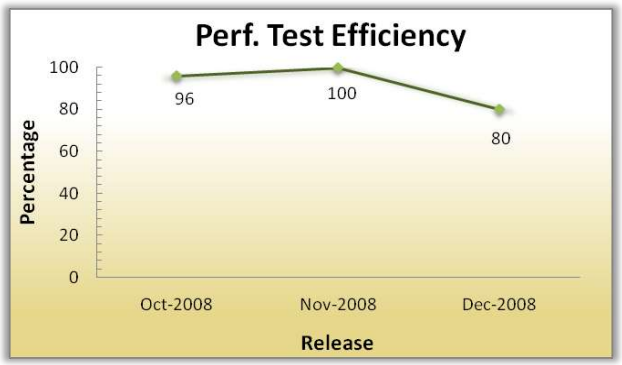


Figure 14.TE

Performance Severity Index (PSI)

This metric determine the product quality based performance criteria on which one can take decision for releasing of the product to next phase i.e. it indicates quality of product under test with respect to performance.

$$Performance\ Severity\ Index = \left\lceil \frac{\sum(Severity\ Index * No.\ of\ Req.\ not\ met\ for\ this\ severity)}{Total\ No.\ of\ Req.\ not\ met} \right\rceil$$

If requirement is not met, one can assign the severity for the requirement so that decision can be taken for the product release with respect to performance.

Example

Consider, Average response time is important requirement which has not met, then tester can open defect with Severity as Critical.

Then Performance Severity Index = (4 * 1) / 1 = 4 (Critical)

Performance Severity Trend

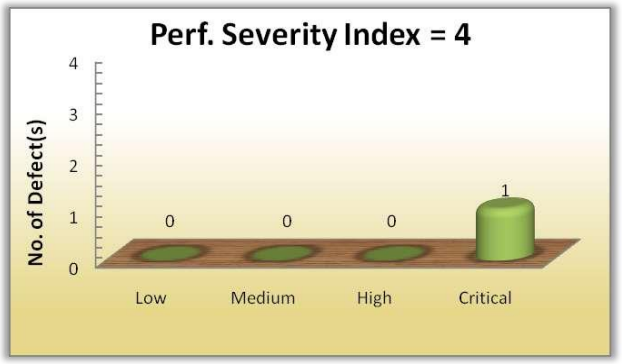


Figure 15. SI

Automation Testing Metrics Automation

Scripting Productivity (ASP)

This metric gives the scripting productivity for automation test script based on which one can analyze and draw most effective conclusion from the same.

$$Automation\ Scripting\ Productivity = \left\lceil \frac{\sum Operations\ Performed}{Efforts\ (hours)} \right\rceil\ Operation(s)/hour$$

Where Operations performed is: -

1. No. of Click(s) i.e. click(s) on which data is refreshed.
2. No. of Input parameter
3. No. of Checkpoint added

Above process does include logic embedded into the script which is rarely used.

Example

Operation Performed	Total
No. of clicks	10
No. of Input Parameter	5
No. of Checkpoint added	10
Total Operation Performed	**25**

Efforts took for scripting = 10 hours.

ASP=25/10=2.5

Automation scripting productivity = 2.5 operations/hour

Automation Scripting Productivity Trend

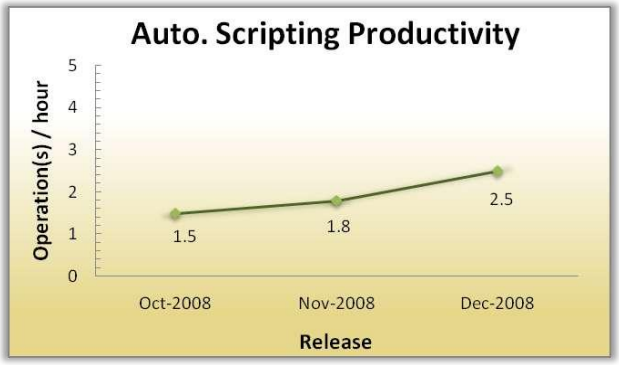


Figure 16. SP

Automation Test Execution Productivity (AEP)

This metric gives the automated test case execution productivity.

$$Auto.\ Execution\ Productivity = \left\lceil \frac{Total\ No.\ of\ Automated\ TC\ executed\ (ATe)}{Execution\ Efforts\ (hours)} * 8 \right\rceil Execution(s)/Day$$

Where ATe is calculated as,

$$ATe = Base\ Test\ Case + ((T\ (0.33) * 0.33) + (T\ (0.66) * 0.66) + (T\ (1) * 1))$$

Evaluation process is similar to Manual Test Execution Productivity.

Automation Coverage

This metric gives the percentage of manual test cases automated.

$$Automation\ Coverage = \left\lceil \frac{Total\ No.\ of\ TC\ Automated}{Total\ No.\ of\ Manual\ TC} * 100 \right\rceil \%$$

Example

If there are 100 Manual test cases and one has automated 60 test cases then

Automation Coverage = 60%

Cost Comparison

This metrics gives the cost comparison between manual testing and automation testing. This metrics is used to have conclusive ROI (return on investment).

Manual Cost is evaluated as: -

Cost (M) = Execution Efforts (hours) * Billing Rate

Automation cost is evaluated as: -

Cost (A) = Tool Purchased Cost (One time investment) + Maintenance Cost
+ Script Development Cost
+ (Execution Efforts (hrs) * Billing Rate)

If Script is re-used the script development cost will be the script update cost.

Using this metric one can have an effective conclusion with respect to the currency which plays a vital role in IT industry.

Common Metrics for all types of testing

Effort Variance (EV)

This metric gives the variance in the estimated effort.

$$Effort\ Variance = \left[\frac{Actual\ Effort - Estimated\ Effort}{Estimated\ Effort} * 100 \right] \%$$

Effort Variance Trend

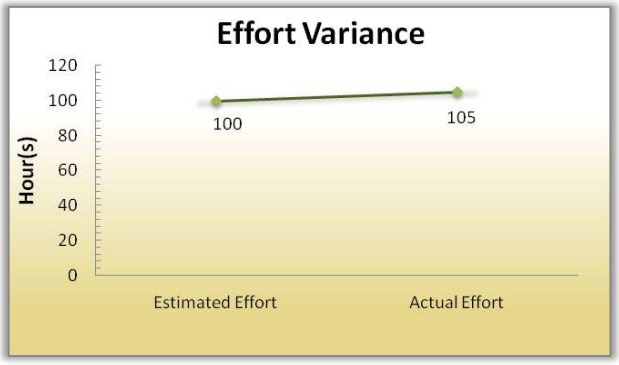


Figure 17. EV

Schedule Variance (SV)

This metric gives the variance in the estimated schedule i.e. number of days.

$$Schedule\ Variance = \left[\frac{Actual\ No.\ of\ Days - Estimated\ No.\ of\ Days}{Estimated\ No.\ of\ Days} * 100 \right] \%$$

Schedule Variance Trend

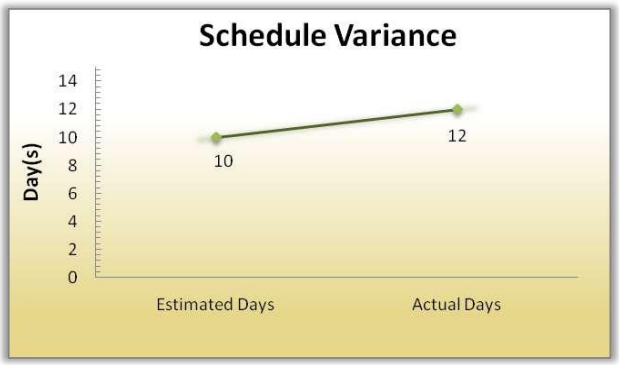


Figure 18. schedule variance

Scope Change (SC)

This metric indicates how stable the scope of testing is.

$$Scope\ Change = \left\lceil \frac{Total\ Scope - Previous\ Scope}{Previous\ Scope} * 100 \right\rceil \%$$

Where,

Total Scope = Previous Scope + New Scope, if Scope increases

Total Scope = Previous Scope - New Scope, if Scope decreases

Scope Change Trend for one release

Figure 19. scope change

8. Software Metrics Methodology

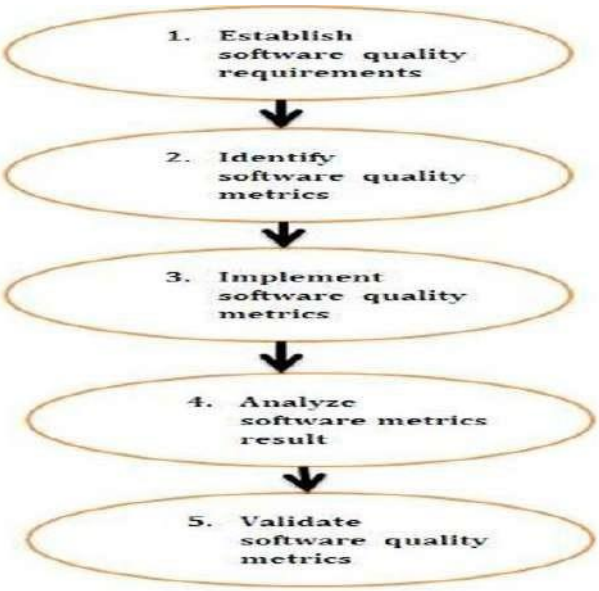
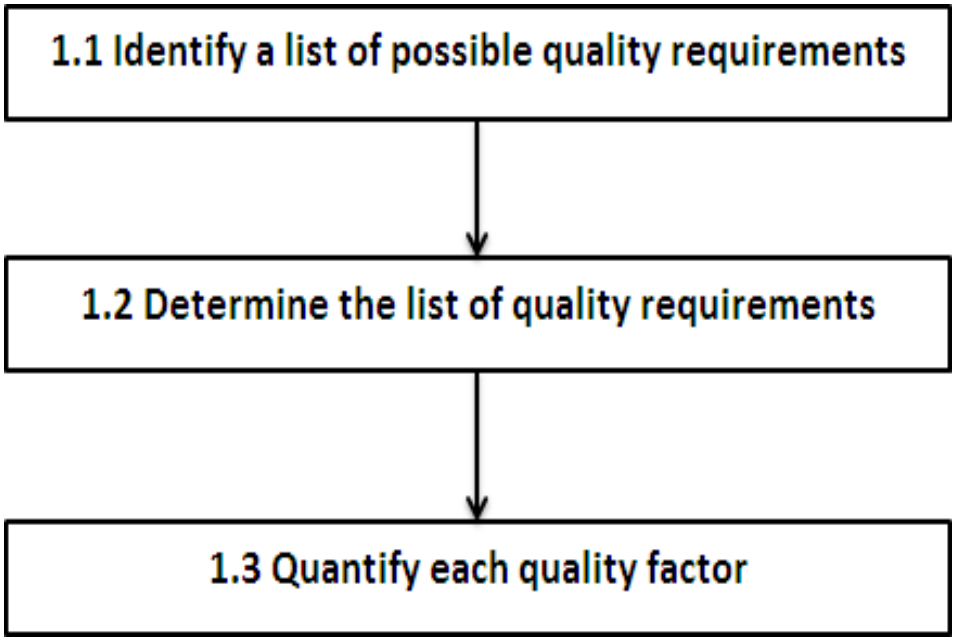


Figure 19. software Metrics Methodology

1. **Establish software quality requirements**



Identify a list of possible quality requirements

Use,

- organizational experience
- required standards
- regulations
- laws

consider,

- contractual requirements
- cost or schedule constraints
- warranties
- customer metrics requirements
- organizational self-interest

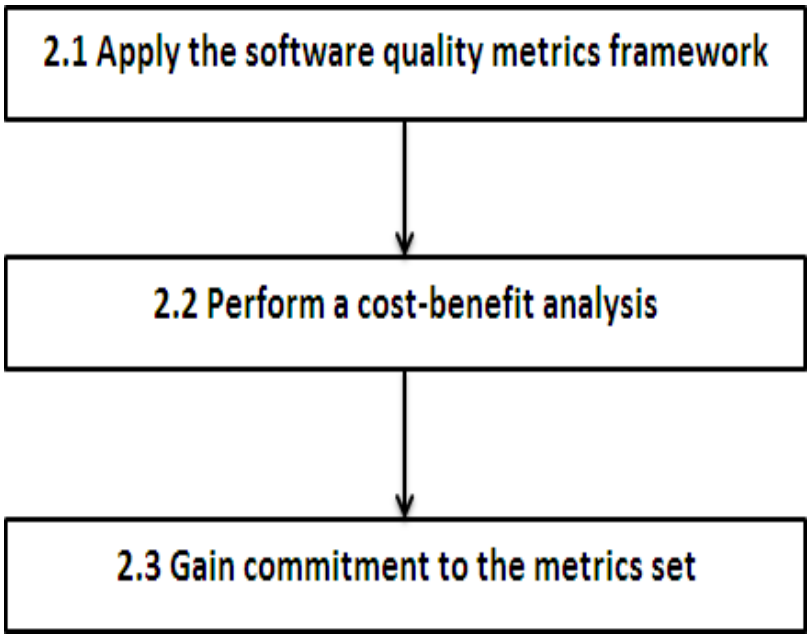
Determine the list of quality requirements:

- Survey all involved parties
- Create the list of quality requirements

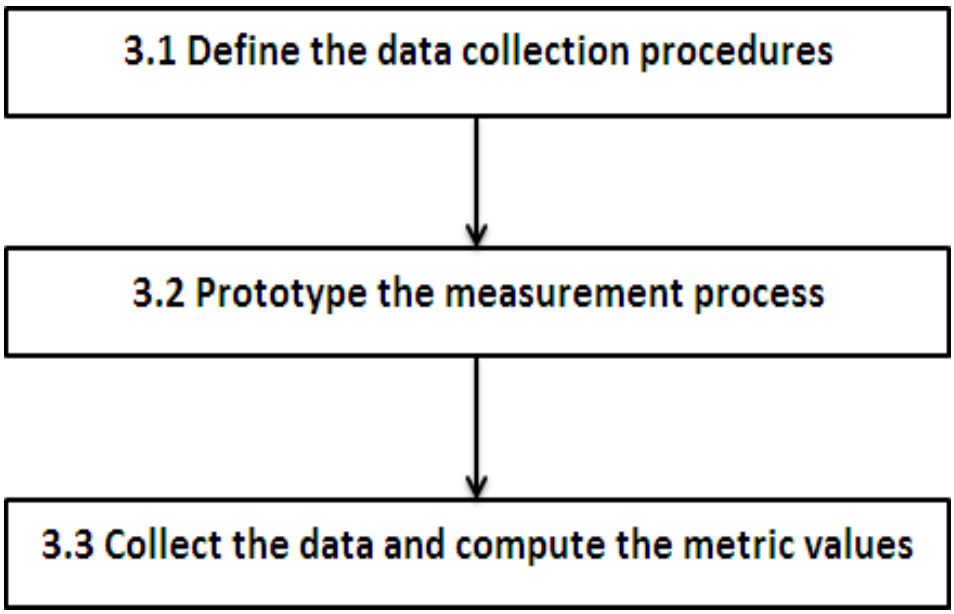
Quantify each quality factor:

- For each quality factor, assign one or more direct metrics to represent the quality factor
- Assign direct metric values to serve as quantitative requirements for that quality factor

2. IDENTIFY SOFTWARE QUALITY METRICS



3. IMPLEMENT SOFTWARE QUALITY METRICS



- Identify tools

- Describe data storage procedures
- Establish a traceability matrix
- Identify the organizational entities
- Participate in data collection
- Responsible for monitoring data collection
- Describe the training and experience required for data collection
- Training process for personnel involved

Table 2. data item description

Item	Description
Name	Name given to the data item.
Metrics	Metrics that are associated with the data item.
Definition	Straightforward description of the data item.
Source	Location of where the data item originates.
Collector	Entity responsible for collecting the data.
Timing	Time(s) in life cycle at which the data item is to be collected. (Some data items are collected more than once.)
Procedures	Methodology (e.g., automated or manual) used to collect the data.
Storage	Location of where the data are stored.
Representation	Manner in which the data are represented, e.g., precision and format (Boolean, dimensionless, etc.).
Sample	Method used to select the data to be collected and the percentage of the available data that is to be collected.
Verification	Manner in which the collected data are to be checked for errors.
Alternatives	Methods that may be used to collect the data other than the preferred method.
Integrity	Person(s) or organization(s) authorized to alter the data item and under what conditions.

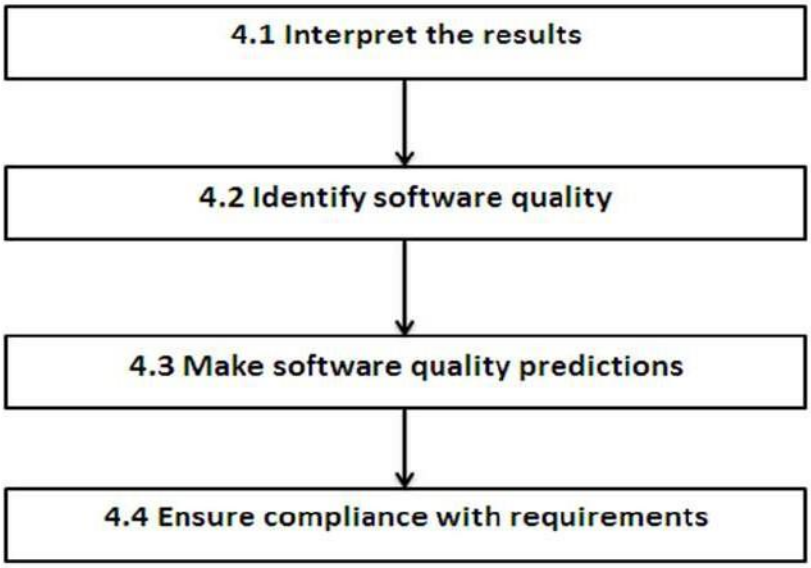
Define data collection procedures

3.2 prototype the measurement process

- Test the data collection and metrics computation procedures on selected software that will act as a prototype
- Select samples that are similar to the project(s) on which the metrics will be used
- Examine the cost of the measurement process for the prototype to verify or improve the cost analysis
- Results collected from the prototype to improve the metric descriptions and descriptions of data items.

- Using the formats in Table, **collect and store data** in the project metrics database at the appropriate time in the life cycle
- Check the data for **accuracy** and proper unit of measure
- **Monitor** the data collection
- Check for **uniformity** of data if more than one person is collecting it
- **Compute** the metric values from the collected data

4. analyse the software metric result



Interpret the results

- The purpose of metrics validation is to identify both product and process metrics that can predict specified quality factor values, which are quantitative representations of quality requirements
- Metrics shall indicate whether quality requirements have been achieved or are likely to be achieved in the future
- For the purpose of assessing whether a metric is valid
- The following thresholds shall be designated:

V- square of the linear correlation coefficient

B-rank correlation coefficient

A-prediction error @-confidence level P-success rate

- Correlation

- Tracking
- Consistency
- Predictability
- Discriminative power
- Reliability

5. VALIDATE THE SOFTARE QUALITY

Identify the quality factors sample

 A sample of quality factors shall be drawn from the metrics database

Identify the metrics sample

 A sample from the same domain (e.g., same software components), as used in 5.3.1, shall be drawn from the metrics database

Perform a statistical analysis

 The analysis described in 5.2 shall be performed

 Before a metric is used to evaluate the quality of a product or process, it shall be validated against the criteria described in

If a metric does not pass all of the validity tests, it shall only be used according to the criteria prescribed by those tests

Document the results

 Documented results shall include the direct metric, predictive metric, validation criteria, and numerical results, as a minimum

Revalidate the metrics

A validated metric may not necessarily be valid in other environments or future applications. Therefore, a predictive metric shall be revalidated before it is used for another environment or application

Evaluate the stability of the environment

Metrics validation shall be undertaken in a stable development environment (i.e., where the design language, implementation language, or project development tools do not change over the life of the project in which validation is performed)

9. Software quality indicator

A Software Quality Indicator is used to calculate and to provide an indication of the quality of the system by assessing system characteristics. The software quality indicators address management concerns. It is also used for decision making by decision maker. It includes a measure of the reliability of the code.

An indicator usually compares a metric with a baseline or expected result.

It acts as a set of tools to improve the management capabilities of personnel responsible for monitoring

Software quality indicators extract from requirements are flexibility and adaptability.

Following quality indicators can use during the software testing & development life cycle.

1] Progress:-
It Measures the amount of work accomplished by the developer in each phase

2] Stability:-
Assesses whether the products of each phase are sufficiently stable to allow the next phase to proceed

3] Process compliance:-
It compliance with the development procedures approved at the beginning of the project to the running development process

4] Quality Evaluation efforts:-
It evaluates percentage of the developer's effort that is being spent on internal quality evaluation activities and time required to deal.

5] Test coverage:-
It measures the amount of the software system / functionality covered by the developer's testing process.

6] Defect detection efficiency:-
Measures how many of the defects detectable discovered during that phase.

7] Defect removal rate:-
Total number of defects detected and resolved over a period of time

8] Defect density:-
Detects defect-prone components of the system

9]Defectageprofile:-
It measures the number of defects that have remained unresolved for a long period of time.

10]Complexity:-
It measures the complexity of the code. It counts the total path, branch, coverage to calculate the complexity

10. Fundamentals of Measurement Theory

It is undisputed that measurement is crucial to the progress of all sciences. Scientific progress is made through observations and generalizations based on data and measurements, the derivation of theories as a result and in turn the confirmation or refutation of theories via hypothesis testing based on further empirical data. As an example, consider the proposition "the more rigorously the front end of the software development process is executed, the better the quality at the back end." To confirm or software development process" and distinguish the process step and activities of the front end from those of the back end. Assume that after the requirements gathering process, our development process consists of the following phases:

- Design
- Design reviews and inspections
- Code
- Code inspection
- Debug and development tests
- Integration of components and modules to form the product
- Formal machine testing
- Early customer programs

Integration is the development phase during which various parts and components are integrated to form one complete software product. Usually after integration the product is under formal change control. Specifically, after integration every change of the software must have a specific reason (e.g., to fix a bug uncovered during testing) and must be documented and tracked. Therefore, we may want to use integration as the cutoff point: The design, coding, debugging, and integration phase are classified as the front end of the development process and the formal machine testing and early customer trials constitute the back end.

We then define rigorous implementation both in the general sense and in specific terms as they relate to the front end of the development process. Assuming the development process has been formally documented, we may define rigorous implementation as total adherence to the process: whatever is described in the process documentation that needs to be executed, we execute. However, this general on is not sufficient for our purpose, which is to gather data to test our propositions. We need to specify the indicator(s) of the definition and to make it (them) operational. For example, suppose the process documentation says all designs and code should be inspected. One operational definition of rigorous implementation may be inspection coverage expressed in terms of the percentage of the estimated lines of code (LOC) or of thee function [points (FP) that are actually inspected. Another indicator of good

reviews and inspections could be the scoring of ach inspections by the inspectors at the end of the inspection, based on a set of criteria. We may want to operationally use a five – point Likert scale to denote the degree of effectiveness (e.g., 5 = very effective, 4 = effective, 3 = somewhat effective,2 = not effective, 1 = poor inspection). There may also be other indicators.

In addition to design, design reviews, code implementation, and code inspections, development testing is part of our definition of the front end of the development process. We also need to operationally define "rigorous execution"of this test. Two indicators that could be used are the percent coverage in terms of instructions executed (as measured by some test coverage measurement tools) and the defect rate expressed in terms of number of defects removed per thousand lines of source code (KLOC) or per function point. Likewise, we need to operationally define "quality at the back end" and decide which measurement indicators to us. For the sake of simplicity let us use defects found per KLOC (or defects per function point) during formal machine testing as the indicator of back end quality. From these metrics, we can formulate several testable hypotheses such as the following:

• For software projects, the higher the percentage of the design and code that are inspected, the lower the defect rat at the later phase of formal machine testing.

• The more effective the design reviews and the code inspections as scored by the inspection team, the lower the defect rate at the later phase of formal machine testing.

• The more through the development testing (in terms of test coverage) before integration, the lower the defect rate at the formal machine testing phase.

With the hypotheses formulated, we can set out to gather data and test the hypotheses. We also need to determine the unit of analysis for our measurement and data. In this case, it could be at the project level or at the component level of a large project. If we are able to collect a number of data points that from a reasonable sample size (e.g., 35 projects or components), we can perform statistical analysis to test the hypotheses. We can classify projects or components into several groups according to the independent variable of each hypothesis, and then compare the outcome of the dependent variable (defect rate during formal machine testing) across the groups. We can conduct simple correlation analysis, or we can perform more sophisticated statistical analyses. If the hypotheses are substantiated by the data, we confirm the proposition. If they are rejected, we refute the proposition. If we have doubts or unanswered questions during the process (e.g., are our indicators valid? Are our data reliable? Are there other variables we need to control when we conduct the analysis for hypothesis testing?, then perhaps more research is needed. However, if the hypothesis (e s) or the proposition is confirmed, we can use the knowledge thus gained and act accordingly to improve our software development quality.

11. LEVEL OF MEASUREMENT

We have seen that from theory to empirical hypothesis and from theoretically defined concepts to operational definitions, the process is by no means direct .As the example illustrates, when we operationalize a definition and derive measurement indicators, we must consider the scale of measurement. For instance, to measure the quality of software inspection we may use a five-point scale to score the inspection effectiveness or we may use percentage to indicate the inspection coverage. For some cases, more than one measurement scale is applicable; for others, the nature of the concept and the resultant operational definition can b measured only with a certain scale. In this section, we briefly discuss the four levels of measurement: normal scale, ordinal scale, interval scale, and ratio scale.

Nominal Scale

The most simple operation in science and the lowest level of measurement is classification. In classifying we attempt to sort elements into categories with respect to a certain attribute. For example, if the attribute of interest is religion, we may classify the subjects of the study into Catholics, Protestants, Jews, Buddhists, and so on. If we classify software products by the development process models through which the products were developed, then we may have categories such as waterfall development process and other. In a normal scale, the two key requirements for the categories are jointly exhaustive and mutually exclusive. Mutually exclusive mans a subject can be classified into one and only one category. Jointly exhaustive means that all categories together should cover all possible categories of the attribute. If the attribute has more categories than we are interested in, an "other" category is needed to make the categories jointly exhaustive.

In a normal scale, the names of the categories and their sequence bear no assumptions about relationships among categories. For instance, we place the waterfall development process in front of spiral development process, but we do not imply that one is "better than" or "greater than" the other. As long as the requirements of mutually exclusive and jointly exhaustive are met, we may want to compare the values of interested attributes such as defect rate, cycle time, and requirements defects across the different categories of software products.

Ordinal Scale

Ordinal scale refers to the measurement operations through which the subjects can be compared in order. For example, we may classify families according to socioeconomic status: upper class, middle class, and lower class. We may classify software development projects according to the SEI maturity levels or according to a process rigor scale: totally adheres to process, somewhat adheres to process, does not adhere to process. Our earlier example of inspection effectiveness scoring is an ordinal scale.

The ordinal measurement scale is at a higher level than the nominal scale in the measurement

hierarchy. Through it we are able not only to group subjects into categories, but also to order the categories. An ordinal scale is asymmetric in the sense that if A>B is true then B>A is false. It has the transitivity property in that if A>B and B>C, A>C.

We must recognize that an ordinal scale offers no information about the magnitude of the differences between elements. For instance, for the process rigor scale we know only that "totally adheres to process" is better than "somewhat adheres to process" in terms of the quality outcome of the software product, and "somewhat adheres to process" is better than "dos not adheres to process." However, we cannot say that the difference between the former pair of categories is the same as that between the letter pair. In customer satisfaction surveys of software products, the five-point Likert scale is often used with 1=completely dissatisfied, 2= somewhat dissatisfied, 3= neutral, 4= satisfied, and 5= completely satisfied. We know only 5>4, 4>3, and 5>2, and so forth, but we cannot say how much greater 5 is than4. Nor can we say that the difference between categories 5and 4 is equal to that between categories 3 and 2, Indeed, to move customers from satisfied (4) to very satisfied (5) versus from dissatisfied (2) to neutral (3) , to neutral (3), may require very different actions and types of improvements.

Therefore, when we translate order relations into mathematical operations, we cannot use operations such as addition, subtraction, multiplication, and division. We can use "greater than" and "less than". However, in real-world application for some specific types of ordinal scales (such as the Likert five-point, seven-point, or ten-point scales), the assumption of equal distance is often made and operations such as averaging are applied to these scales. In such cases, we should be aware that the measurement assumption is deviated, and then use extreme caution when interpreting the results of data analysis.

Interval and Radio Scales

An interval scale indicates the extract differences between measurement points. The mathematical operations of addition and substraction can be applied to interval scale data .For instance, assuming products A, B, and C are developed in the same language, if the defect rate of software product A is 5 defects per KLOC and product B's rate is 3.5 defects per KLOC, then we can say product A's defect level is 1.5 defects per KLOC higher than product B's defect level. An interval scale of measurement requires a well-defined unit of measurement that can be agreed on as a common standard and that is repeatable. Given a unit of measurement, it is possible to say that the difference between two scores is 15 units or that one difference is the same as a second. Assuming product C's defect rate is 2 defects per KLOC, we can thus say the difference in defect rate between products A and B is the same as that between B and C.

When an absolute or no arbitrary zero point can be located on an interval scale, it becomes a ratio scale. Ratio scale is the highest level of measurement and all mathematical operations can be applied to it, including division and multiplication. For example, we can say that

product A's defect rate is twice as much as product C's because when the defect rate is zero, that means not a single defects exists in the product. Had the zero point been arbitrary, the statement would have been illegitimate. A good example of an interval scale with an arbitrary zero point is the traditional temperature measurement (Fahrenheit and centigrade scales).Thus we say that the difference between the average summer temperature (80 degree F) and the average winter temperature (16 degree F is 64 degree F, but we do not say that 80 degree F is five times as hot as 16 degree F Fahrenheit and centigrade temperature scales are interval, not ratio, scales. For this reason, scientists developed the absolute temperature scale (a ratio scale) for use in scientific activities.

Except for a few notable examples, for all practical purposes almost all interval measurement scales are also ratio scales. When the size of the unit is established, it is usually possible to conceive of a zero unit.

For internal and ratio scales, the measurement can be expressed in both integer and no integer data. Integer data are usually given in terms of frequency counts (e.g., the number of defects customers will encounter for a software product over a specified time length).

We should note that the measurement scales are hierarchical. Each higher level scale possesses all properties of the lower ones. The higher the level of measurement, the more powerful analysis can be applied to the data. Therefore, in our operationalization process we should devise metrics that can take advantage of the highest level of measurement allowed by the nature of the concept and its definition. A higher-level measurement can always make various types of comparisons if the scale is in terms of actual defect rate. However, if the scale is in terms of excellent, good, average, worse than average, and poor, as compared to an industrial standard, then our ability to perform additional analysis of the data is limited.

12. SOME BASIC MEASURES

Regardless of the measurement scale, when the data are gathered we need to analyze them to extract meaningful information. Various measures and statistics are available for summarizing the raw data and for making comparisons across groups. In this section we discuss some basic measures such as ratio, proportion, percentage, and rate, which are frequently used in our daily lives as well as in various activities associated with software development and software quality. These basic measures, while seemingly easy, are often misused. There are also numerous sophisticated statistical techniques and methodologies that can be employed in data analysis. However, such topics are not within the scope of this discussion.

Ratio

A ratio results from dividing one quality by another. The numerator and denominator are from two distinct populations and are mutually exclusive. For example, in demography, sex ratio is defined as:

Number of males
X 100 %
Number of females

If the ratio is less than 100, there are more females than males: otherwise there are more males than females.

Ratios are also used in software metrics. The most often used, perhaps, is the ratio of number of people in an independent test organization to the number of those in the development group. The test/ development head-count ratio could range from 1:1 to 1:10 depending on the management approach to the software development process. For the large-ratio (e.g., 1:10) organizations, the development group usually is responsible for the complete development (including extensive development tests) of the product, and the test group conducts system-level testing in terms of customer environment verifications. For the small-ratio organizations, the independent group takes the major responsibility for testing (after debugging and code integration) and quality assurance.

Proportion

Proportion is different from ratio in that the numerator in a proportion is a part of the denominator:

$P = a/a + b$

Proportion also differs from ratio in that the ratio is best used for two groups, whereas proportion is used for multiple categories (or populations) of one group. In other words, the denominator in the preceding formula can be more than just a+b. If

$a + b + c + d + e + N$

Then we have

$a/N + b/N + c/N + d/N + e/N = 1$

When the numerator and the denominator are integers and represent counts of certain events, then p is also referred to as a relative frequency. For example, the following gives the proportion of satisfied customers of the total customer set:

Number of satisfied customers

Total number of customers of a software product

The numerator and the denominator in a proportion need not be integers. Thy can be

frequency counts as well as measurement units on a continuous scale (e.g., height in inches, weight in pounds).When the measurement unit is not integer, proportions are called fractions.

Percentage

A proportion or a fraction becomes a percentage when it is expressed in terms of per hundred units (the denominator is normalized to 100). The word percent mans per hundred. A proportion p is therefore equal to 100%; percent (100/7%).

Percentages are frequently used to report results, and as such are frequently misused. First, because percentages represent relative frequencies, it is important that enough contextual information be given, specially the total number of cases, so that the readers can interpret the information correctly. Jones (1992) observes that many reports and presentations in the software industry are careless in using percentages and ratios. He cites the example:

Requirements bugs were 15% of the total, design bugs were 25% of the total, coding bugs were 50% of the total, and other bugs made up 10% of the total.

Had the results been stated as follows, it would have been much more informative:

The project consists of 8 thousand lines of code (KLOC). During its development a total of 200 defects were detected and removed, giving a defect removal rate of 25 defects per KLOC. Of the 200 defects, requirements bugs constituted 15%, design bugs 25%, coding bugs 50%, and other bugs made up 10%.

A second important rule of thumb is that the total number of cases must be sufficiently large enough to use percentages. Percentages computed from a small total are not stable: they also convey an impression that a large number of cases are involved. Some writers recommend that the minimum number of cases for which percentages should b calculated is 50. We recommend that, depending on the number of categories, the minimum number be 30, the smallest sample size required for parametric statistics. If the number of cases is too small, then absolute numbers, instead of percentages, should be used. For instance,

Of the total 20 defects for the entire project of 2 KLOC, there were 3 requirements bugs, 5 design bugs, 10 coding bugs, and 2 others.

When results in percentages appear in table format, usually both the percentages and actual numbers are shown when there is only one variable. When there are more than two groups, such as the example, it is better just to show the percentages and the total number of cases (N) for each group. With percentages and N known, one can always reconstruct the frequency distributions. The total of 100.0% should always be shown so that it is clear how the percentages are computed. In a two-way table, the direction in which the percentages are computed depends on the purpose of the comparison.

Software Engineering | Software Review

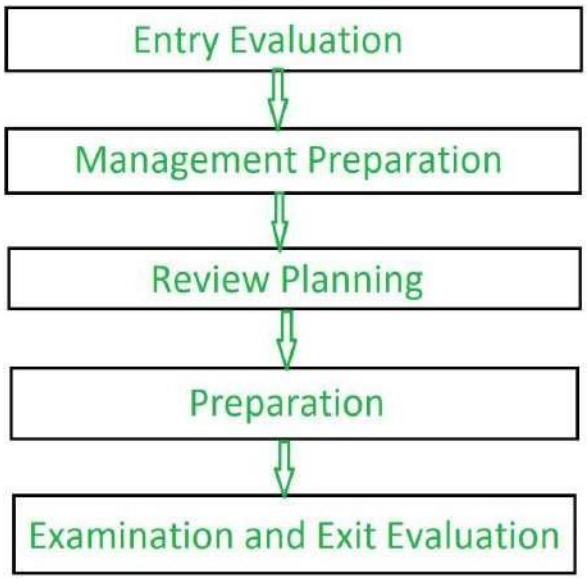
Software Review is systematic inspection of a software by one or more individuals who work together to find and resolve errors and defects in the software during the early stages of Software Development Life Cycle (SDLC). Software review is an essential part of Software Development Life Cycle (SDLC) that helps software engineers in validating the quality, functionality and other vital features and components of the software. It is a whole process that includes testing the software product and it makes sure that it meets the requirements stated by the client.

Usually performed manually, software review is used to verify various documents like requirements, system designs, codes, test plans and test cases.

Objectives of Software Review:
The objective of software review is:

1. To improve the productivity of the development team.
2. To make the testing process time and cost effective.
3. To make the final software with fewer defects.
4. To eliminate the inadequacies.

Process of Software Review:

5. 

Types of Software Reviews:

There are mainly 3 types of software reviews:

1. **Software Peer Review:**

   Peer review is the process of assessing the technical content and quality of the product and it is usually conducted by the author of the work product along with some other developers.

   Peer review is performed in order to examine or resolve the defects in the software, whose quality is also checked by other members of the team.

   Peer Review has following types:

(i) Code Review:

Computer source code is examined in a systematic way.

- **(ii) Pair Programming:**

  It is a code review where two developers develop code together at the same platform.

- **(iii) Walkthrough:**

  Members of the development team is guided bu author and other

- interested parties and the participants ask questions and make comments about defects.

- **(iv) Technical Review:**

  A team of highly qualified individuals examines the software product for its client's use and identifies technical defects from specifications and standards.

- **(v) Inspection:**

  In inspection the reviewers follow a well-defined process to find defects.

2. **Software Management Review:**

   Software Management Review evaluates the work status. In this section decisions regarding downstream activities are taken.

3. **Software Audit Review:**

   Software Audit Review is a type of external review in which one or more critics, who are not a part of the development team, organize an independent inspection of the

software product and its processes to assess their compliance with stated specifications and standards. This is done by managerial level people.

Advantages of Software Review:

1. Defects can be identified earlier stage of development (especially in formal review).
2. Earlier inspection also reduces the maintenance cost of software.
3. It can be used to train technical authors.
4. It can be used to remove process inadequacies that encourage defects.

SCM -Software configuration management

Whenever a software is build, there is always scope for improvement and those improvements brings changes in picture. Changes may be required to modify or update any existing solution or to create a new solution for a problem. Requirements keeps on changing on daily basis and so we need to keep on upgrading our systems based on the current requirements and needs to meet desired outputs. Changes should be analyzed before they are made to the existing system, recorded before they are implemented, reported to have details of before and after, and controlled in a manner that will improve quality and reduce error.

This is where the need of System Configuration Management comes.

System Configuration Management (SCM) is an arrangement of exercises which controls change by recognizing the items for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made. It is essential to control the changes in light of the fact that if the changes are not checked legitimately then they may wind up undermining a well-run programming. In this way, SCM is a fundamental piece of all project management activities.

Processes involved in SCM –

Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

1. **Identification and Establishment –** Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually

consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development). Establishing relationship among items, creating a mechanism to manage multiple level of control and procedure for change management system.

2. **Version control –** Creating versions/specifications of the existing product to build new products from the help of SCM system. A description of version is given below:

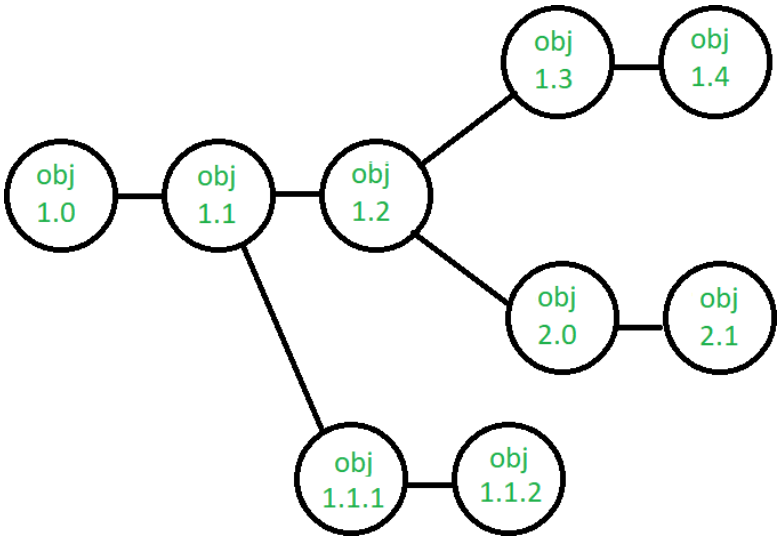


Figure. 19. Version control

Suppose after some changes, the version of configuration object changes from 1.0 to 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The development of object 1.0 continues through 1.3 and 1.4, but finally, a noteworthy change to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.

3. **Change control –** Controlling changes to Configuration items (CI). The change control process is explained in Figure below:

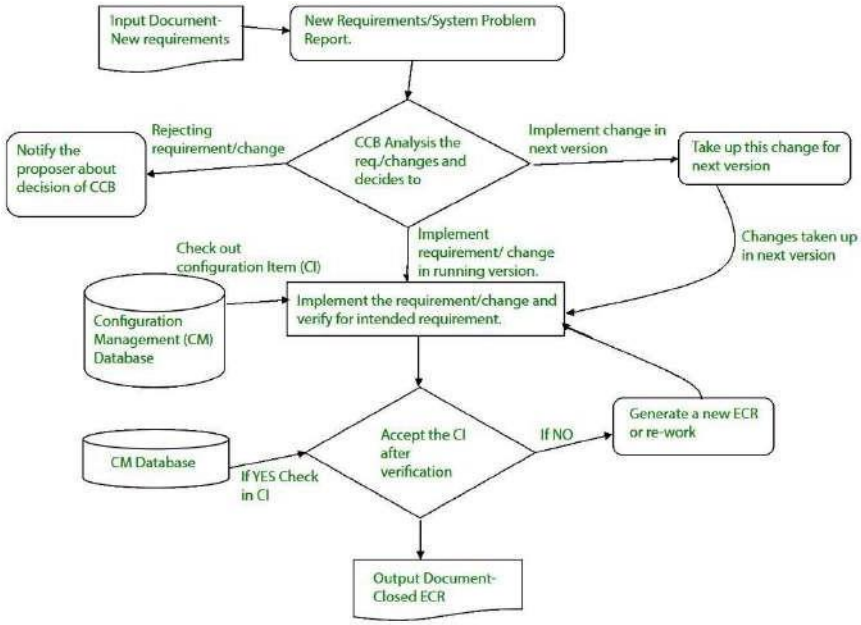


Figure 20.change control

A change request (CR) is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a change report, which is used by a change control board (CCB) —a person or group who makes a final decision on the status and priority of the change. An engineering change Request (ECR) is generated for each approved change.

Also CCB notifies the developer in case the change is rejected with proper reason. The ECR describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is "checked out" of the project database, the change is made, and then the object is tested again. The object is then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.

4. **Configuration auditing –** A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness and consistency of items in the SCM system and track action items from the audit to closure.
5. **Reporting –** Providing accurate status and current configuration data to developers, tester, end users, customers and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guide etc .

SCM Tools –

Different tools are available in market for SCM like: CFEngine, Bcfg2 server, Vagrant, SmartFrog, CLEAR CASETOOL (CC), SaltStack, CLEAR QUEST TOOL, Puppet, SVN-Subversion, Perforce, TortoiseSVN, IBM Rational team concert, IBM Configuration management version management, Razor, Ansible, etc. There are many more in the list.

It is recommended that before selecting any configuration management tool, have a proper understanding of the features and select the tool which best suits your project needs and be clear with the benefits and drawbacks of each before you choose one to use.

What is Risk?

"Tomorrow problems are today's risk." Hence, a clear definition of a "risk" is a problem that could cause some loss or threaten the progress of the project, but which has not happened yet.

These potential issues might harm cost, schedule or technical success of the project and the quality of our software device, or project team morale.

Risk Management is the system of identifying addressing and eliminating these problems before they can damage the project.

We need to differentiate risks, as potential issues, from the current problems of the project.

Different methods are required to address these two kinds of issues.

For example, staff storage, because we have not been able to select people with the right technical skills is a current problem, but the threat of our technical persons being hired away by the competition is a risk.

Risk Management

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Other risk categories

1. **1. Known risks:** Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (e.g., unrealistic delivery date)
2. **2. Predictable risks:** Those risks that are hypothesized from previous project experience (e.g., past turnover)
3. **3. Unpredictable risks:** Those risks that can and do occur, but are extremely tough to identify in advance.

Principle of Risk Management

1. **Global Perspective:** In this, we review the bigger system description, design, and implementation. We look at the chance and the impact the risk is going to have.

2. **Take a forward-looking view:** Consider the threat which may appear in the future and create future plans for directing the next events.
3. **Open Communication:** This is to allow the free flow of communications between the client and the team members so that they have certainty about the risks.
4. **Integrated management:** In this method risk management is made an integral part of project management.
5. **Continuous process:** In this phase, the risks are tracked continuously throughout the risk management paradigm.

Unit – 4

SOFTWARE QUALITY ENGINEERING TOOLS AND TECHNIQUES

Seven basic Quality tools – Checklist – Pareto diagram – Cause and effect diagram – Run chart – Histogram – Control chart– Scatter diagram – Poka Yoke – Statistical process control – Failure Mode and Effect Analysis – Quality Function deployment – Continuous improvement tools – Case study.

Seven basic tools of quality

The seven basic tools of quality are a designation given to a fixed set of graphical techniques identified as being most helpful in troubleshooting issues related to quality. They are called basic because they are suitable for people with little formal training in statistics and because they can be used to solve the vast majority of quality-related issues.

The seven tools are:

- Stratification (alternatively, flow chart or run chart)
- Histogram
- Check sheet (tally sheet)
- Cause-and-effect diagram (also known as the "fishbone diagram" or Ishikawa diagram)
- Pareto chart (80-20 rule)
- Scatter diagram (Shewhart chart)
- Control chart

1. Stratification

- Stratification analysis is a quality assurance tool used to sort data, objects, and people into separate and distinct groups.
- Separating your data using stratification can help you determine its meaning, revealing patterns that might not otherwise be visible when it's been lumped together.
- Whether you're looking at equipment, products, shifts, materials, or even days of the week, stratification analysis lets you make sense of your data before, during, and after its collection.
- To get the most out of the stratification process, consider which information about your data's sources may affect the end results of your data analysis.
- Make sure to set up your data collection so that that information is included.

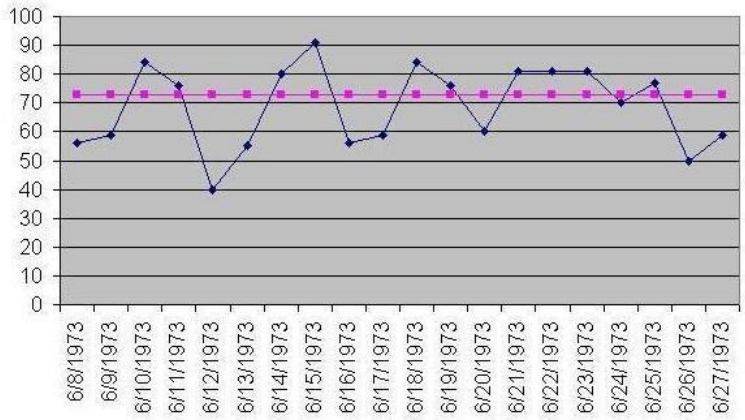


Figure 1. info collected

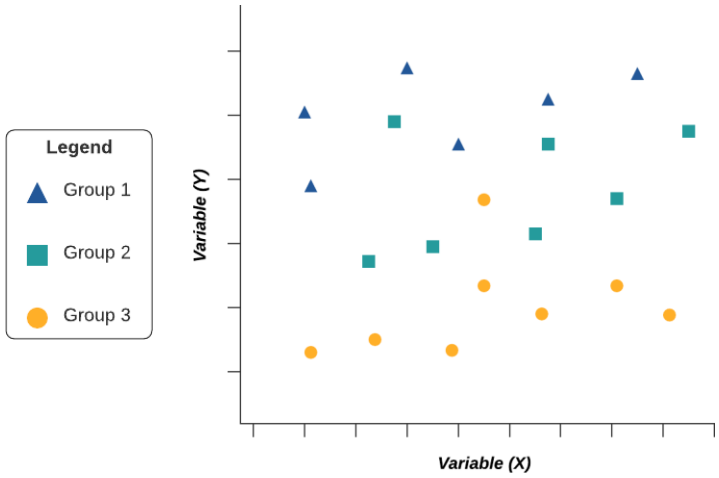


Figure 2. info ploted

- Stratification is a method of dividing data into sub–categories and classify data based on group, division, class or levels that helps in deriving meaningful information to understand an existing problem.
- The very purpose of Stratification is to divide the data and conquer the meaning full Information to solve a problem.
- **Un–stratified data:** (An employee reached late to office on following dates)

5-Jan, 12-Jan,13-Jan, 19-Jan, 21-Jan, 26-Jan,27-Jan

- **Stratified data:** (Same data classified by day of the week)

Day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Frequency - Late in Office	4	2	1	0	0	0	0

Table1: Frequency – Late in office day wise

Advantages

The reasons to use stratified sampling rather than simple random sampling include

- If measurements within strata have lower standard deviation (as compared to the overall standard deviation in the population), stratification gives smaller error in estimation.
- For many applications, measurements become more manageable and/or cheaper when the population is grouped into strata.
- When it is desirable to have estimates of population parameters for groups within the population - stratified sampling verifies we have enough samples from the strata of interest.

If the population density varies greatly within a region, stratified sampling will ensure that estimates can be made with equal accuracy in different parts of the region, and that comparisons of sub-regions can be made with equal statistical power.

For example, in Ontario a survey taken throughout the province might use a larger sampling fraction in the less populated north, since the disparity in population between north and south is so great that a

sampling fraction based on the provincial sample as a whole might result in the collection of only a handful of data from the north.

Disadvantages

- Stratified sampling is not useful when the population cannot be exhaustively partitioned into disjoint subgroups.
- It would be a misapplication of the technique to make subgroups' sample sizes proportional to the amount of data available from the subgroups, rather than scaling sample sizes to subgroup sizes (or to their variances, if known to vary significantly—e.g. by means of an F Test).
- Data representing each subgroup are taken to be of equal importance if suspected variation among them warrants stratified sampling.
- If subgroup variances differ significantly and the data needs to be stratified by variance, it is not possible to simultaneously make each subgroup sample size proportional to subgroup size within the total population.
- For an efficient way to partition sampling resources among groups that vary in their means, variance and costs, see "optimum allocation".
- The problem of stratified sampling in the case of unknown class priors (ratio of subpopulations in the entire population) can have deleterious effect on the performance of any analysis on the dataset, e.g. classification.
- In that regard, minimax sampling ratio can be used to make the dataset robust with respect to uncertainty in the underlying data generating process.
- Combining sub-strata to ensure adequate numbers can lead to Simpson's paradox, where trends that actually exist in different groups of data disappear or even reverse when the groups are combined.

2. Histogram

- Quality professionals are often tasked with analyzing and interpreting the behavior of different groups of data in an effort to manage quality.
- This is where quality control tools like the histogram come into play.
- The histogram can help you represent frequency distribution of data clearly and concisely amongst different groups of a sample, allowing you to quickly and easily identify areas of improvement within your processes.
- With a structure similar to a bar graph, each bar within a histogram represents a group, while the height of the bar represents the frequency of data within that group.
- Histograms are particularly helpful when breaking down the frequency of your data into categories such as age, days of the week, physical measurements, or any other category that can be listed in chronological or numerical order.

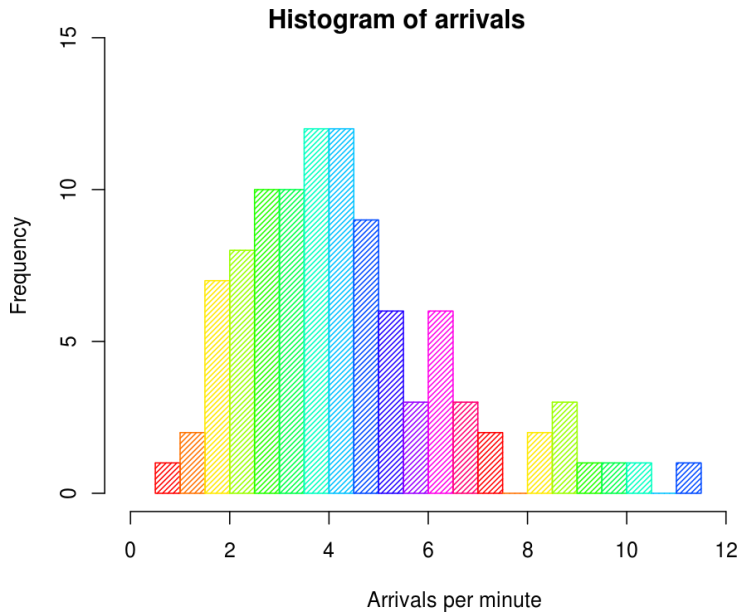


Figure 3. Histogram

- Histogram introduced by Karl Pearson is a bar graph representing the frequency distribution on each bar.
- The very purpose of Histogram is to study the density of data in any given distribution and understand the factors or data that repeat more often.
- Histogram helps in prioritizing factors and identify which are the areas that needs utmost attention immediately.

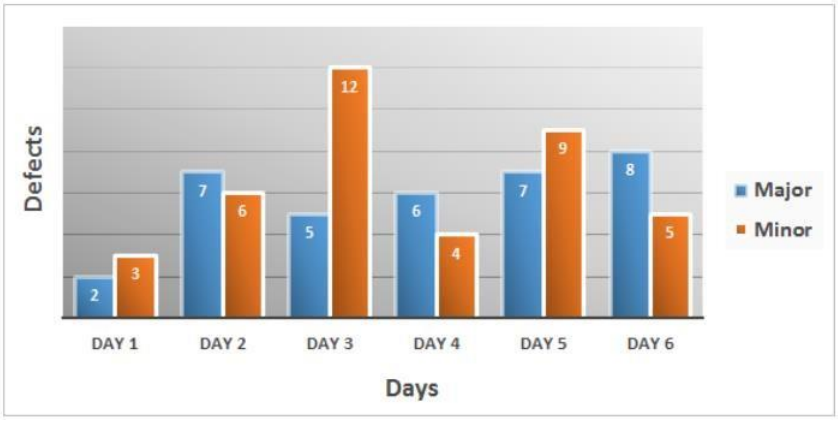


Fig 4. Chart - Histogram: Defects Day wise

3. Check sheet (or tally sheet)

- Check sheets can be used to collect quantitative or qualitative data.
- When used to collect quantitative data, they can be called a tally sheet.
- A check sheet collects data in the form of check or tally marks that indicate how many times a particular value has occurred, allowing you to quickly zero in on defects or errors within your process or product, defect patterns, and even causes of specific defects.
- With its simple setup and easy-to-read graphics, check sheets make it easy to record preliminary frequency distribution data when measuring out processes.

- This particular graphic can be used as a preliminary data collection tool when creating histograms, bar graphs, and other quality tools.

Table1. Monthly assembly check

Motor Assembly Check Sheet

Defect Types/ Event Occurrence	Dates							TOTAL
	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	
Supplied parts rusted		𝍸	𝍸	𝍸	||			20
Misaligned weld			|||			||		5
Improper test procedure								0
Wrong part issued		|		||				3
Film on parts								0
Voids in casting				𝍸	||			6
Incorrect dimensions						||		2
Adhesive failure								0
Masking insufficient					|			1
Spray failure			𝍸					5
TOTAL		10	13	10	5	4		

- A check sheet can be metrics, structured table or form for collecting data and analyzing them.
- When the information collected is quantitative in nature, the check sheet can also be called as tally sheet.
- The very purpose of checklist is to list down the important checkpoints or events in a tabular/metrics format and keep on updating or marking the status on their occurrence which helps in understanding the progress, defect patterns and even causes for defects.

Defect Types ? (Major/ Minor)	Defects in Supplied Items							Total Count
	Sun	Mon	Tue	Wed	Thu	Fri	Sat	
Rusted Items		𝍸	𝍸		𝍸	𝍸		9
Items with Scratch	𝍸							1
Dirty		𝍸		𝍸		𝍸		6
Broken/ Cracks			𝍸				𝍸	3
Main Body Dent					𝍸			3
Missing Components		𝍸		𝍸			𝍸	5
Labelling Error					𝍸	𝍸		4
Damage in Packaging			𝍸					2
Wrong Item Issued					𝍸		𝍸	3
Film on Parts			𝍸					4
Voids in Casting	𝍸					𝍸	𝍸	4
Incorrect Dimensions			𝍸	𝍸	𝍸			5
Failed to pass the quality test		𝍸				𝍸		3
Total Count	2	9	12	6	10	8	5	52

Table3: Check Sheet: Defect types with their occurrence on day of the week

Name of Data Recorder: Lester B. Rapp

Location: Rochester, New York

Data Collection Dates: 1/17 - 1/23

Defect	Monday	Tuesday	Wednesday	Thursday	Friday	Total
Wrong item in the pick location	𝍸	𝍸	𝍸	𝍸	𝍸	17
Item was picked incorrectly	𝍸	𝍸	𝍸	𝍸	𝍸	19
Two customers received each other's items	𝍸	𝍸		𝍸		4
Product was damaged	𝍸	𝍸	𝍸	𝍸	𝍸	7
Total	13	8	8	12	6	47

4. Cause-and-effect diagram (also known as a fishbone or Ishikawa diagram)

- Introduced by Kaoru Ishikawa, the fishbone diagram helps users identify the various factors (or causes) leading to an effect, usually depicted as a problem to be solved.
- Named for its resemblance to a fishbone, this quality management tool works by defining a quality-related problem on the right-hand side of the diagram, with individual root causes and sub causes branching off to its left.
- A fishbone diagram's causes and subcauses are usually grouped into six main groups, including measurements, materials, personnel, environment, methods, and machines.
- These categories can help you identify the probable source of your problem while keeping your diagram structured and orderly.

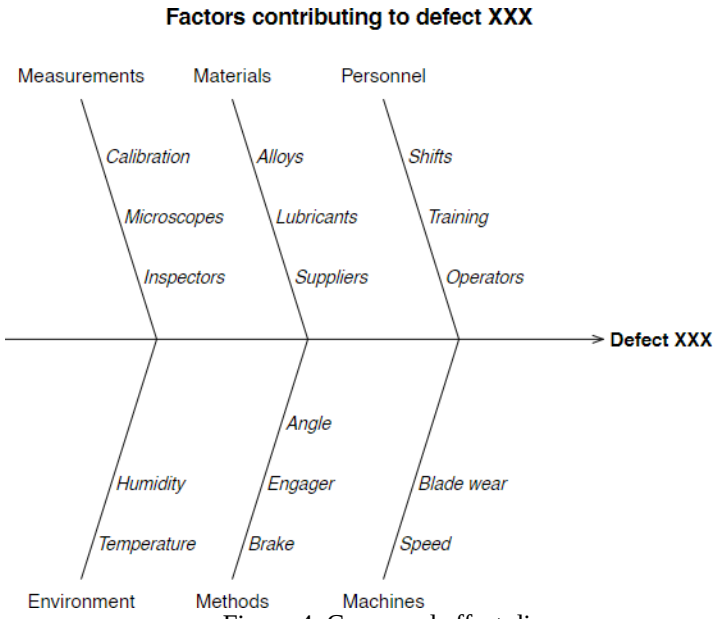
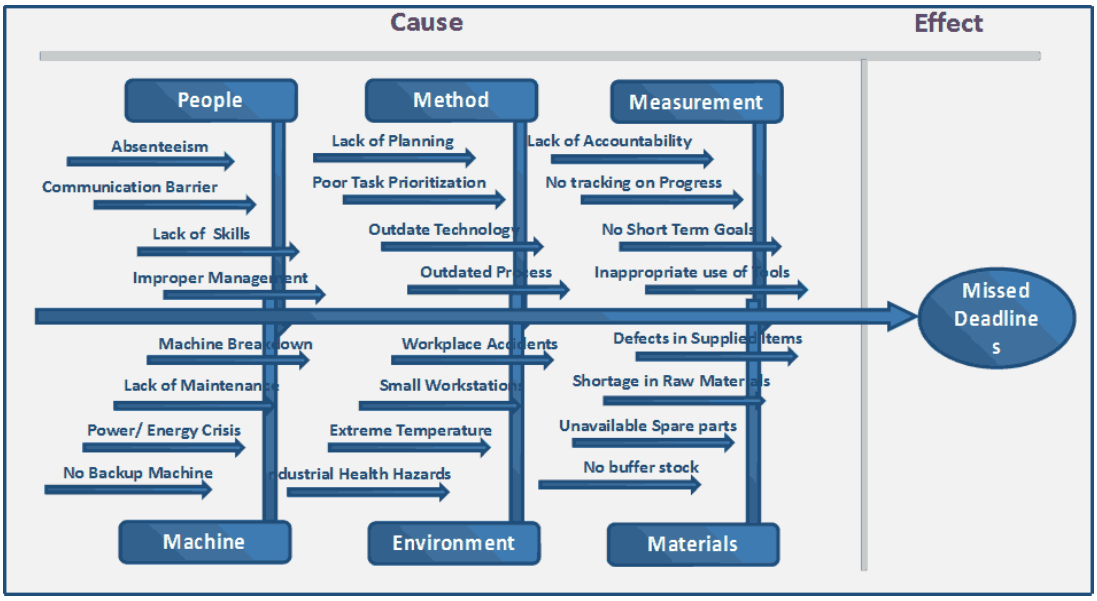
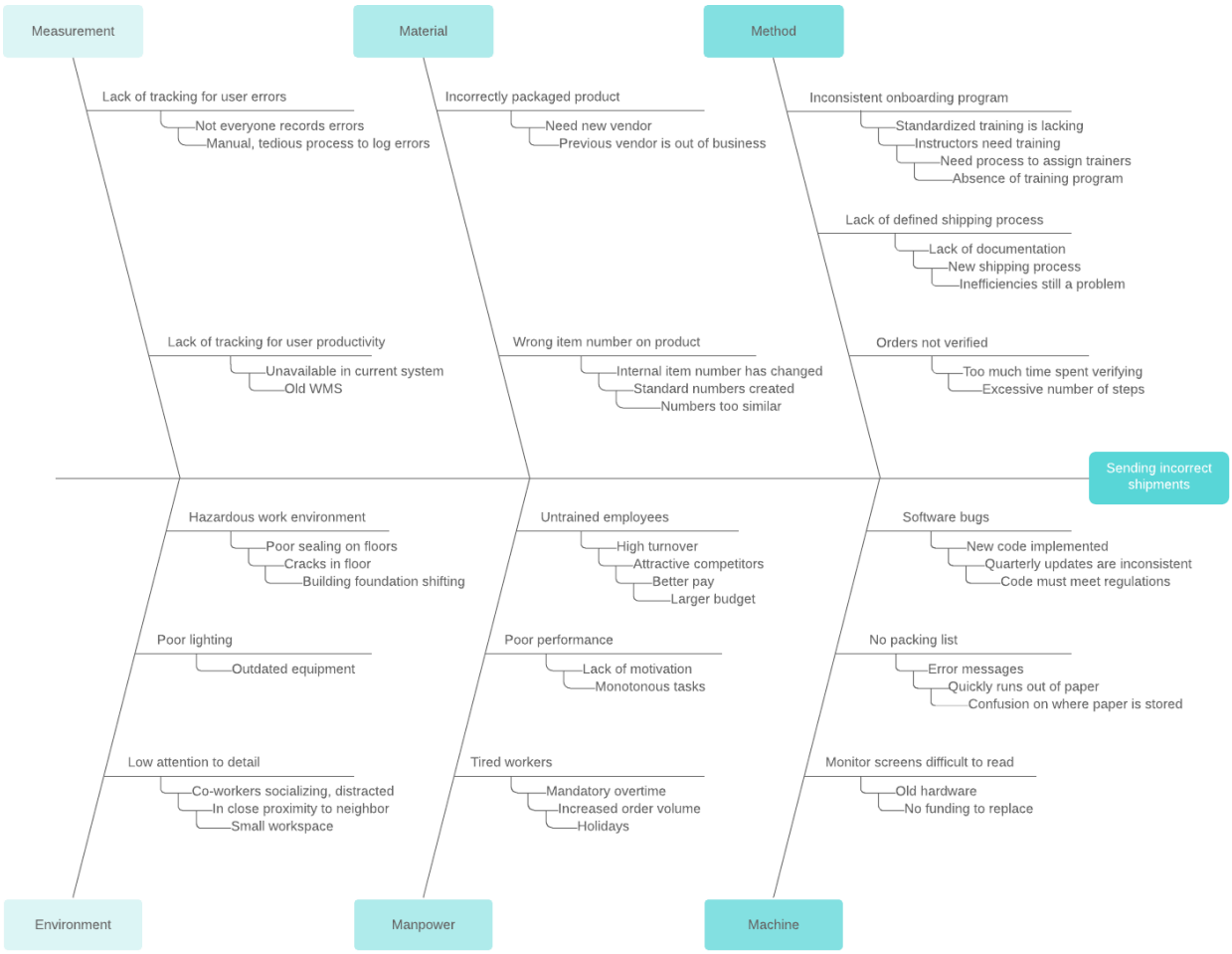


Figure 4. Cause and effect diagram

- Cause–and–effect diagram introduced by Kaoru Ishikawa helps in identifying the various causes (or factors) leading to an effect (or problem) and also helps in deriving meaningful relationship between them.
- The very purpose of this diagram is to identify all root causes behind a problem.
- Once a quality related problem is defined, the factors leading to the causal of the problem are identified.
- We further keep identifying the sub factors leading to the causal of identified factors till we are able to identify the root cause of the problem.
- As a result, we get a diagram with branches and sub branches of causal factors resembling to a fish bone diagram.
- In manufacturing industry, to identify the source of variation the causes are usually grouped into below major categories:
- People, Methods, Machines, Material, Measurements and Environment

Figure5 : Fishbone Diagram: Missed deadline in manufacturing of





Advantages

- Highly visual brainstorming tool which can spark further examples of root causes
- Quickly identify if the root cause is found multiple times in the same or different causal tree
- Allows one to see all causes simultaneously
- Good visualization for presenting issues to stakeholders

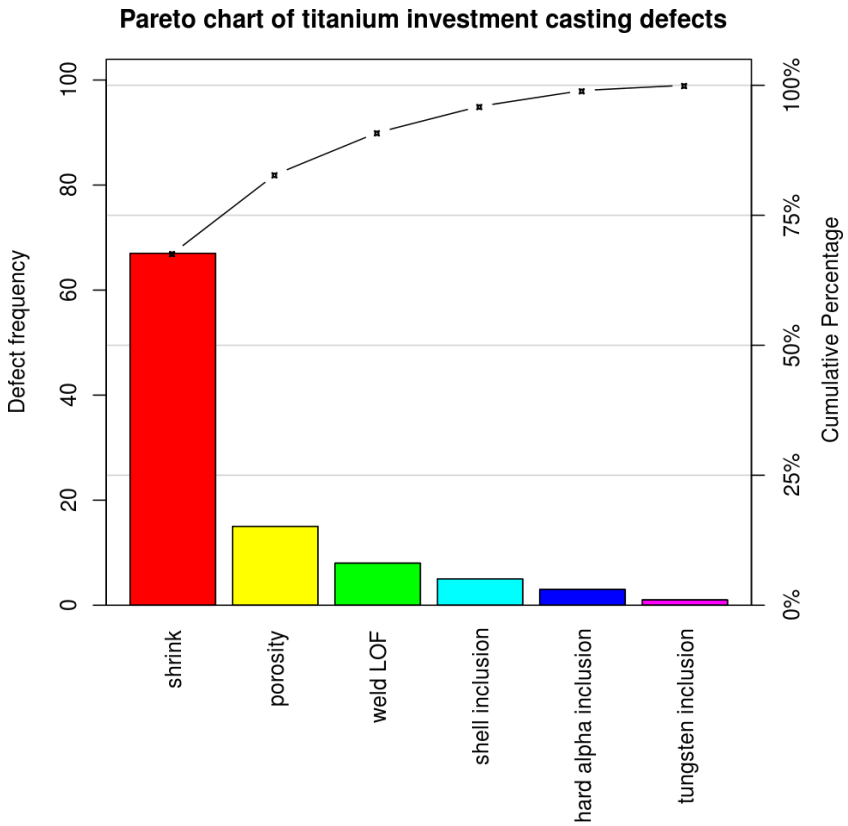
Disadvantages

- Complex defects might yield a lot of causes which might become visually cluttering
- Interrelationships between causes are not easily identifiable

5. Pareto chart (80-20 rule)

- As a quality control tool, the Pareto chart operates according to the 80-20 rule.
- This rule assumes that in any process, 80% of a process's or system's problems are caused by 20% of major factors, often referred to as the "vital few."
- The remaining 20% of problems are caused by 80% of minor factors.
- A combination of a bar and line graph, the Pareto chart depicts individual values in descending order using bars, while the cumulative total is represented by the line.
- The goal of the Pareto chart is to highlight the relative importance of a variety of parameters, allowing you to identify and focus your efforts on the factors with the biggest impact on a specific part of a process or system.

Figure 6. Pareto chart



- Pareto chart is named after Vilfredo Pareto. Pareto chart revolves around the concept of 80-20 rule which underlines that in any process, 80% of problem or failure is just caused by 20% of few major factors which are often referred as Vital Few, whereas remaining 20% of problem or failure is caused by 80% of many minor factors which are also referred as Trivial Many.
- The very purpose of Pareto Chart is to highlight the most important factors that is the reason for major cause of problem or failure.
- Pareto chart is having bars graphs and line graphs where individual factors are represented by a bar graph in descending order of their impact and the cumulative total is shown by a line graph.
- Pareto charts help experts in following ways:
    - Distinguish between vital few and trivial many.
    - Displays relative importance of causes of a problem.
    - Helps to focus on causes that will have the greatest impact when solved.

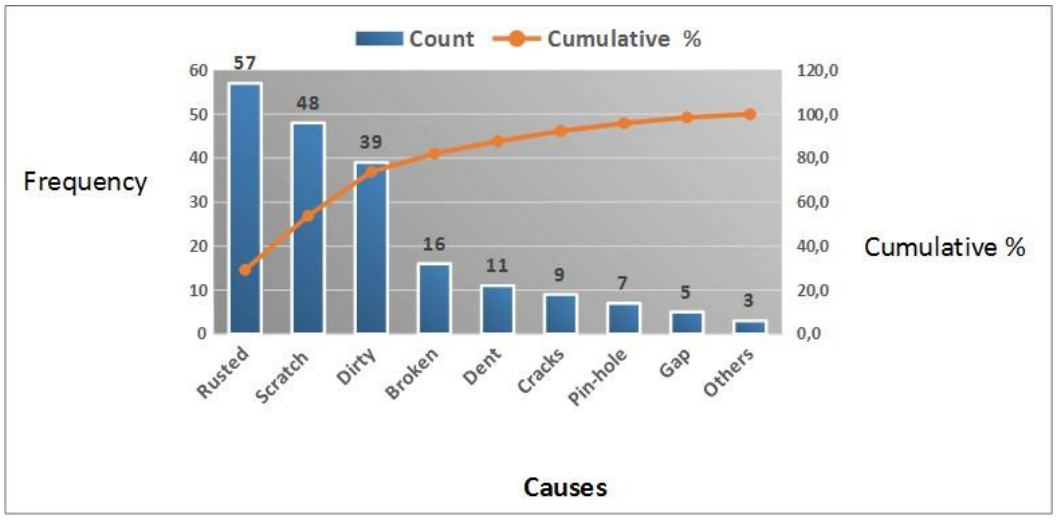


Figure 7. Chart: Pareto Chart: Causes for defects in supplied items

6. Scatter diagram

- Out of the seven quality tools, the scatter diagram is most useful in depicting the relationship between two variables, which is ideal for quality assurance professionals trying to identify cause and effect relationships.
- With dependent values on the diagram's Y-axis and independent values on the X-axis, each dot represents a common intersection point.
- When joined, these dots can highlight the relationship between the two variables.
- The stronger the correlation in your diagram, the stronger the relationship between variables.
- Scatter diagrams can prove useful as a quality control tool when used to define relationships between quality defects and possible causes such as environment, activity, personnel, and other variables.
- Once the relationship between a particular defect and its cause has been established, you can implement focused solutions with (hopefully) better outcomes.

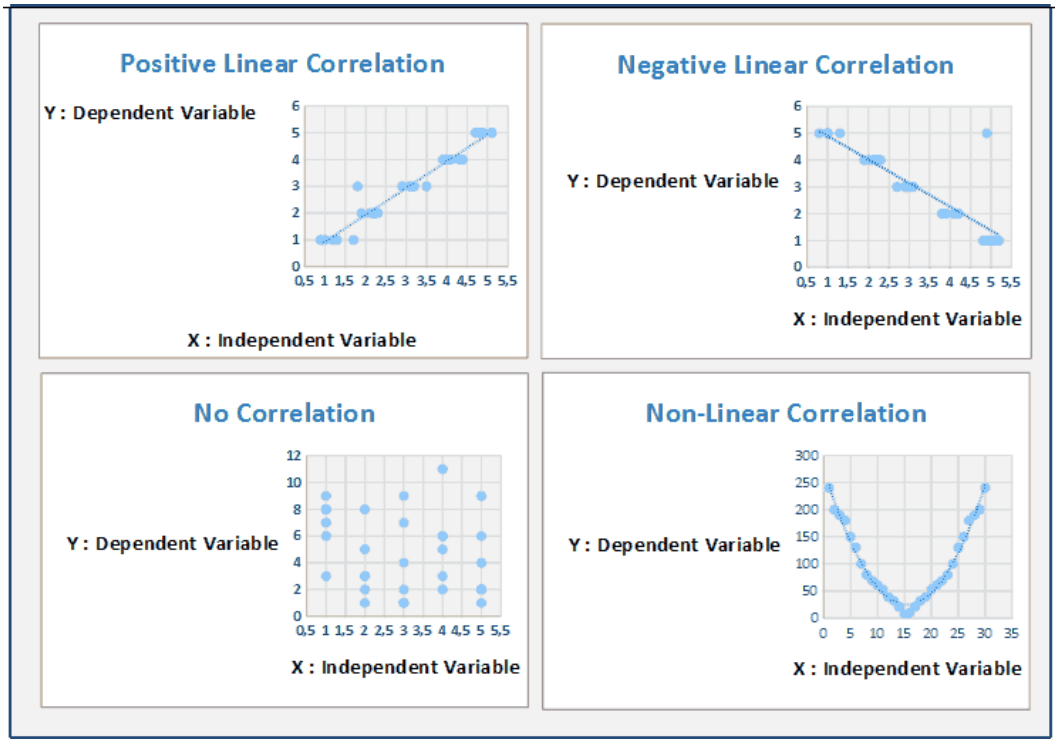
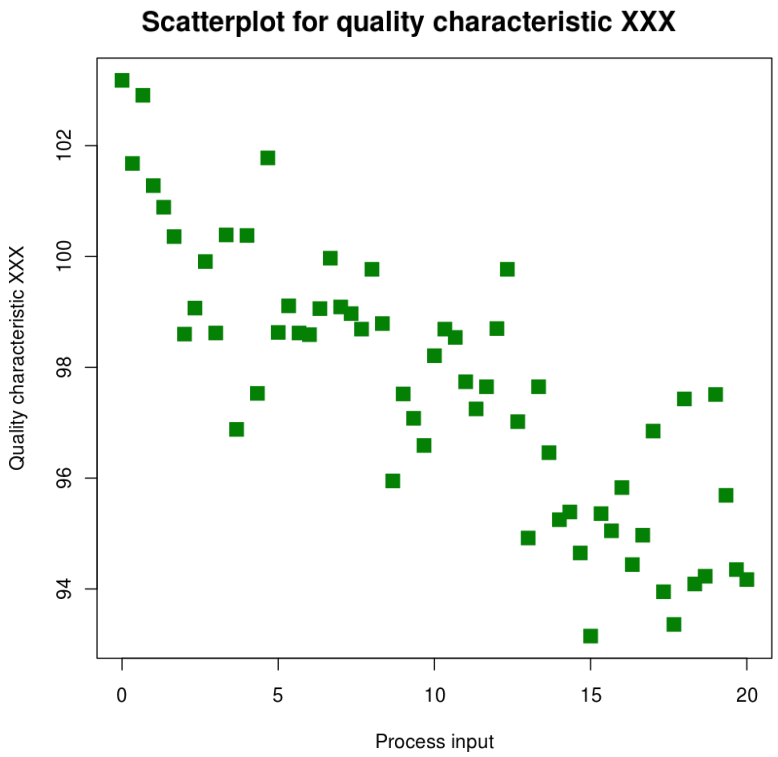


Figure 8: Scatter Diagram: Types of correlation in Scatter Plot



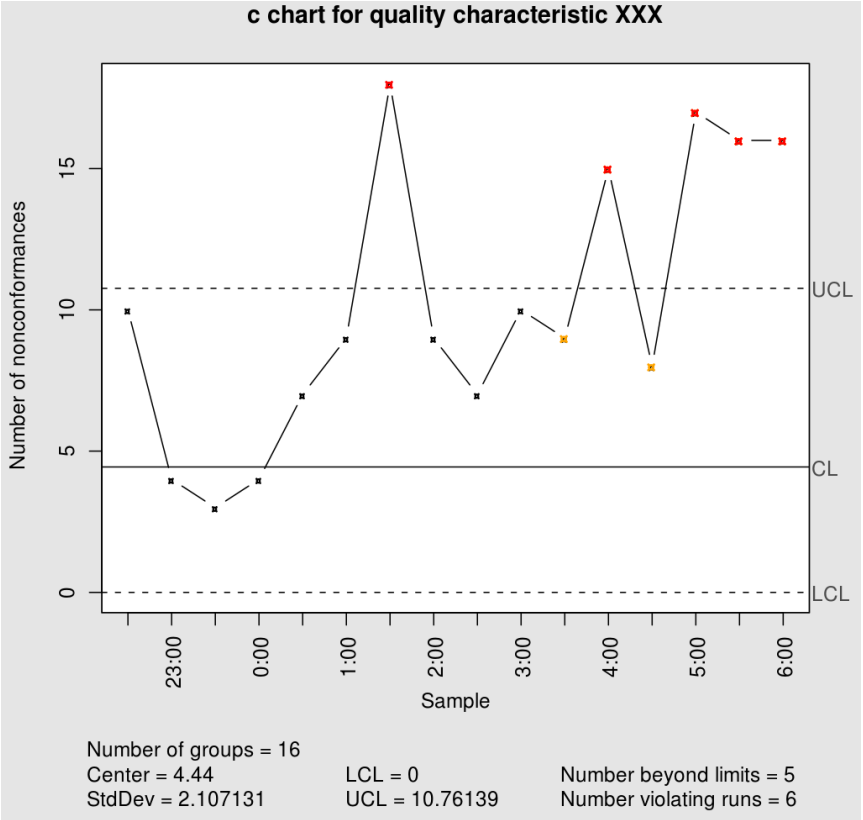
- Scatter diagram or scatter plot is basically a statistical tool that depicts dependent variables on Y – Axis and Independent Variable on X – axis plotted as dots on their common intersection

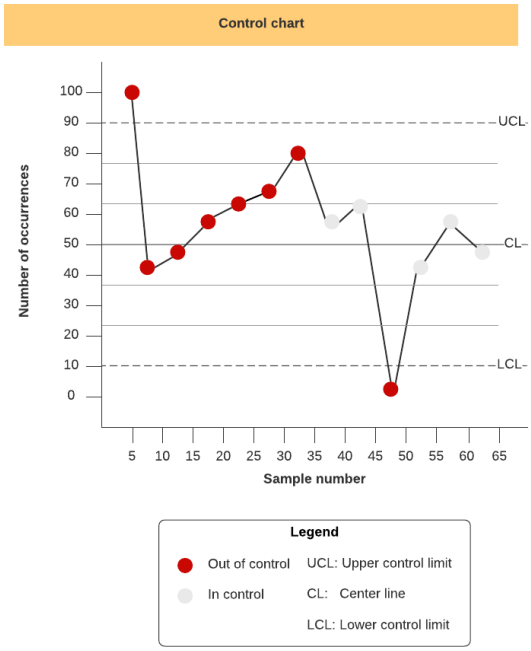
points. Joining these dots can highlight any existing relationship among these variables or an equation in format Y = F(X) + C, where is C is an arbitrary constant.

- Very purpose of scatter Diagram is to establish a relationship between problem (overall effect) and causes that are affecting.
- The relationship can be linear, curvilinear, exponential, logarithmic, quadratic, polynomial etc. Stronger the correlation, stronger the relationship will hold true. The variables can be positively or negatively related defined by the slope of equation derived from the scatter diagram.

7. Control chart (also called a Shewhart chart)

- Out of the seven quality tools, the scatter diagram is most useful in depicting the relationship between two variables, which is ideal for quality assurance professionals trying to identify cause and effect relationships.
- With dependent values on the diagram's Y-axis and independent values on the X-axis, each dot represents a common intersection point.
- When joined, these dots can highlight the relationship between the two variables.
- The stronger the correlation in your diagram, the stronger the relationship between variables.
- Scatter diagrams can prove useful as a quality control tool when used to define relationships between quality defects and possible causes such as environment, activity, personnel, and other variables.
- Once the relationship between a particular defect and its cause has been established, you can implement focused solutions with (hopefully) better outcomes.





Action plan

Action	Owner	Start date	End date	Required resources	Status
Analyze chart	Sam	3/15	3/20	Control chart, process analyst	In progress
Investigate the cause for the out-of-control metric	Mark	3/15	3/22	Documentation of expected process, process manger	In progress
Determine if changes need to be made to the process	Mark	3/16	3/22	Process manager	Waiting

Figure 9. Control chart

- Control chart is also called as Shewhart Chart named after Walter A.
- Shewhart is basically a statistical chart which helps in determining if an industrial process is within control and capable to meet the customer defined specification limits.
- The very purpose of control chart is to determine if the process is stable and capable within current conditions.
- In Control Chart, data are plotted against time in X-axis.
- Control chart will always have a central line (average or mean), an upper line for the upper control limit and a lower line for the lower control limit.
- These lines are determined from historical data.
- By comparing current data to these lines, experts can draw conclusions about whether the process variation is consistent (in control, affected by common causes of variation) or is unpredictable (out of control, affected by special causes of variation).
- It helps in differentiating common causes from special cause of variation.
- Control charts are very popular and vastly used in Quality Control Techniques, Six Sigma (Control Phase) and also plays an important role in defining process capability and variations in productions.
- This tool also helps in identifying how well any manufacturing process is in line with respect to customer’s expectation.
- Control chart helps in predicting process performance, understand the various production patterns and study how a process changes or shifts from normally specified control limits over a period of time.

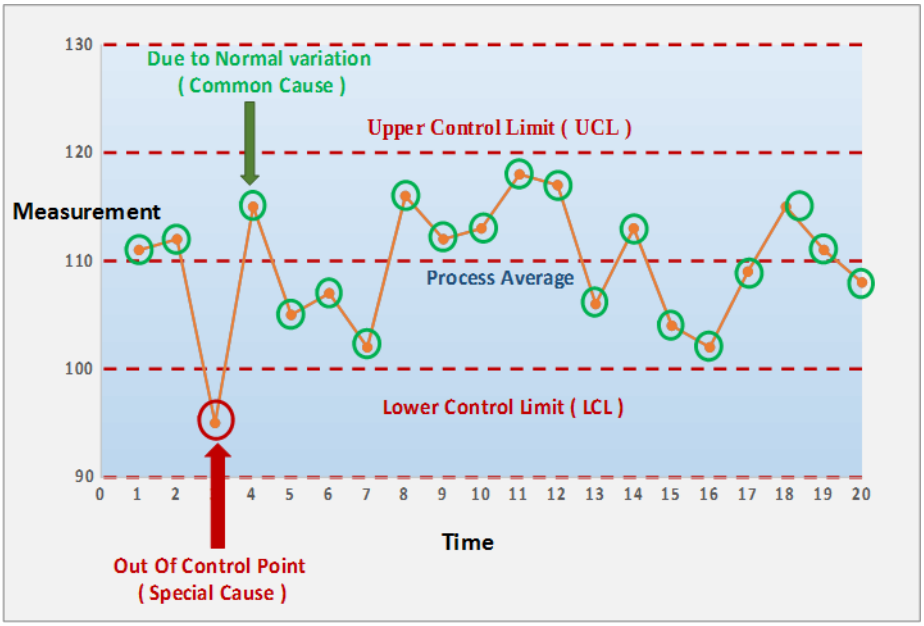
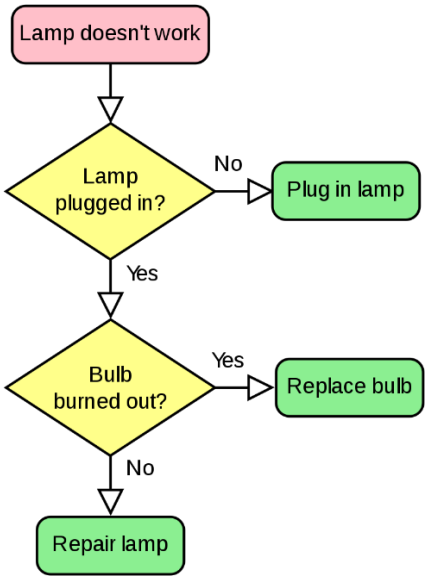


Figure10: Process Control Chart

Additional: Flowcharts

- Some sources will swap out stratification to instead include flowcharts as one of the seven basic QC tools.
- Flowcharts are most commonly used to document organizational structures and process flows, making them ideal for identifying bottlenecks and unnecessary steps within your process or system.
- Mapping out your current process can help you to more effectively pinpoint which activities are completed when and by whom, how processes flow from one department or task to another, and which steps can be eliminated to streamline your process.



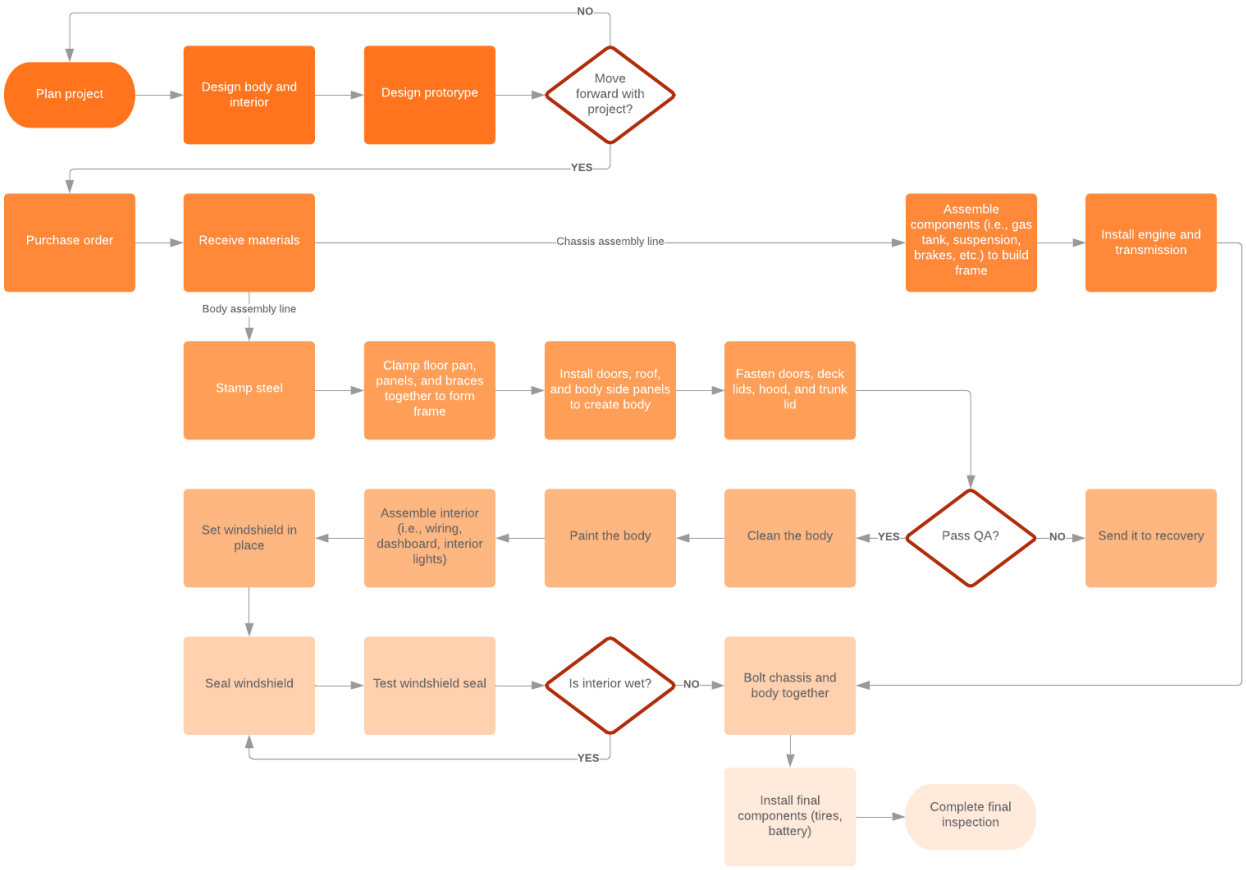
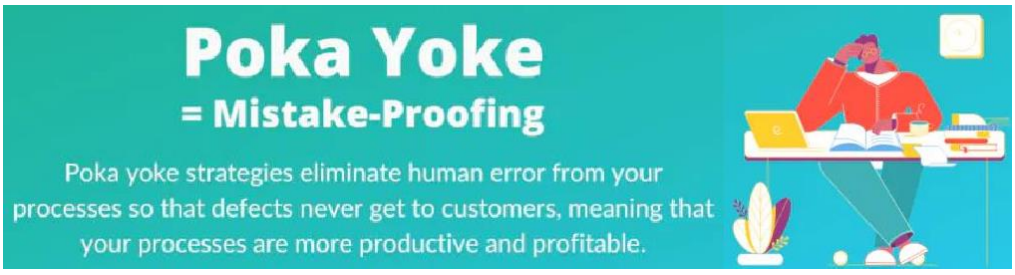


Figure 11. flowchart

Poka-yoke

What is Poka-yoke?

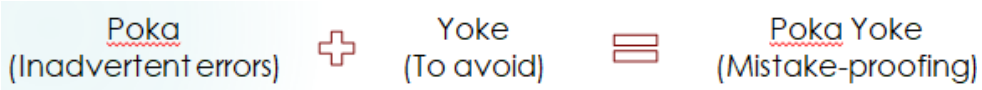
- Poka-yoke (ポカヨケ, [poka joke]) is a Japanese term that means "mistake-proofing" or "inadvertent error prevention".
- A poka-yoke is any mechanism in a process that helps an equipment operator avoid (yokeru) mistakes (poka) and defects by preventing, correcting, or drawing attention to human errors as they occur.
- The concept was formalized, and the term adopted, by Shigeo Shingo as part of the Toyota Production System.



Background

- Concept of poka-yoke is widely associated with Japanese quality leader Shigeo Shingo.

- Mr. Shingo recognized that human error does not necessarily create resulting defects.
- The success of poka-yoke is to provide some intervention device or procedure to catch the mistake before it is translated into nonconforming product.



Poka-Yoke Approach

Poka yoke is often divided into two approaches to prevent or detect defects:

  - The Control Approach
  - The Warning Approach
- The Control Approach:
  - where a process stops when a defect occurs and doesn't resume until there is a corrective action.
  - For example, elevator doors use sensors to prevent the door from closing if there is something in the way.
  - The elevator doors cannot close until the obstruction is removed, and the elevator remains stationary.

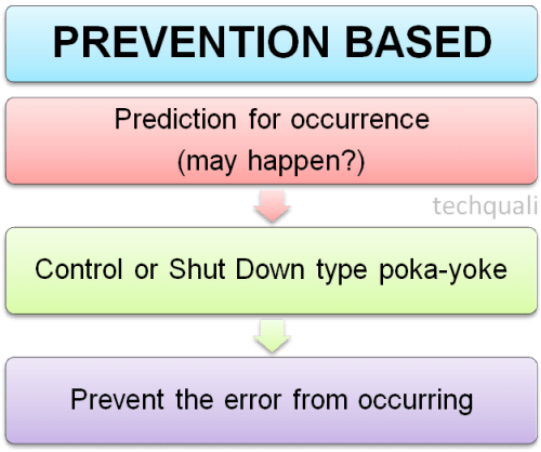


Figure 12. controlled approach

- The Warning Approach:
  - where you are given a warning that a defect has occurred, so you can fix it.
  - For example, many cars will emit beeping noises if you have not fastened your seatbelt or shut your door properly.
  - These noises will not stop until you have taken corrective action.

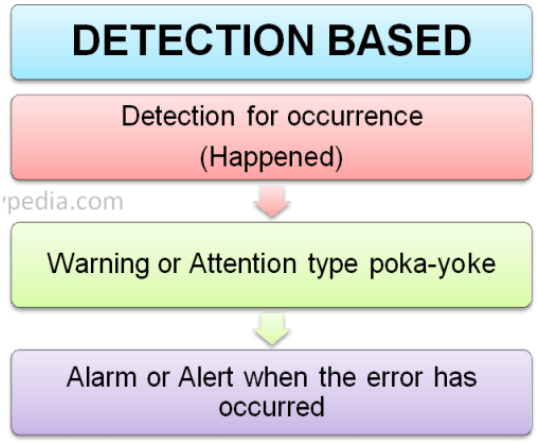


Figure 13. controlled approach

Poka-Yoke Types

- Japanese industrial engineer Shigeo Shingo, who formalized the concept of poka-yoke at Toyota identified 3 types of poka-yoke to detect and avoid errors during mass production.
- The 3 types include the Contact Method, Fixed Value Method and the Motion-Step Method.
    - Contact Method
    - Fixed Value Method (or constant number)
    - Motion Step Method (or sequence)

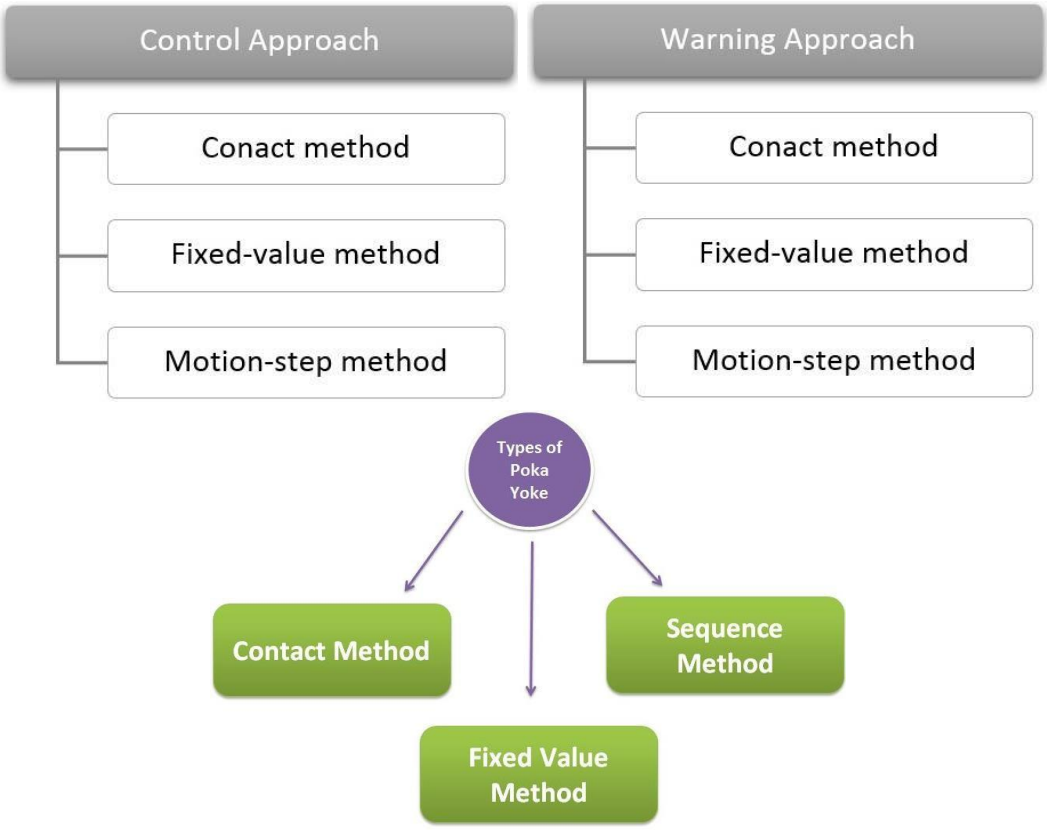


Figure 14. Poka yoke approach

Poka-Yoke Types: Contact Method

- A contact method functions by detecting whether a sensing device makes contact with a part or object within the process.
- Cylinder present Missing cylinder; piston fully extended alarm sounds An example of a physical contact method is limit switches that are pressed when cylinders are driven into a piston.
- The switches are connected to pistons that hold the part in place.
    - In this example, a cylinder is missing and the part is not released to the next process.
    - Cannot proceed to next step.
    - Contact Method using limit switches identifies missing cylinder.
- In simple words, the contact method identifies product defects by testing the product's shape, size, color, or other physical attributes.
- This is called the Contact method in poka-yoke, where product defects can be tested via the product's physical attributes such as its color, size, weight or shape.

Example: The Contact Method can be found virtually all around us.

- Many products use different shapes, sizes, and colors for plugs to ensure that the wrong type of wire is not inserted into the device.
- Be it a desktop computer, laptop, smartphone, or LED TV; there are different sockets where only a specific type of wire can be inserted.
- For example, you cannot insert the power cord or adapter to the headphone jack or the USB cable in a port not meant for this purpose.

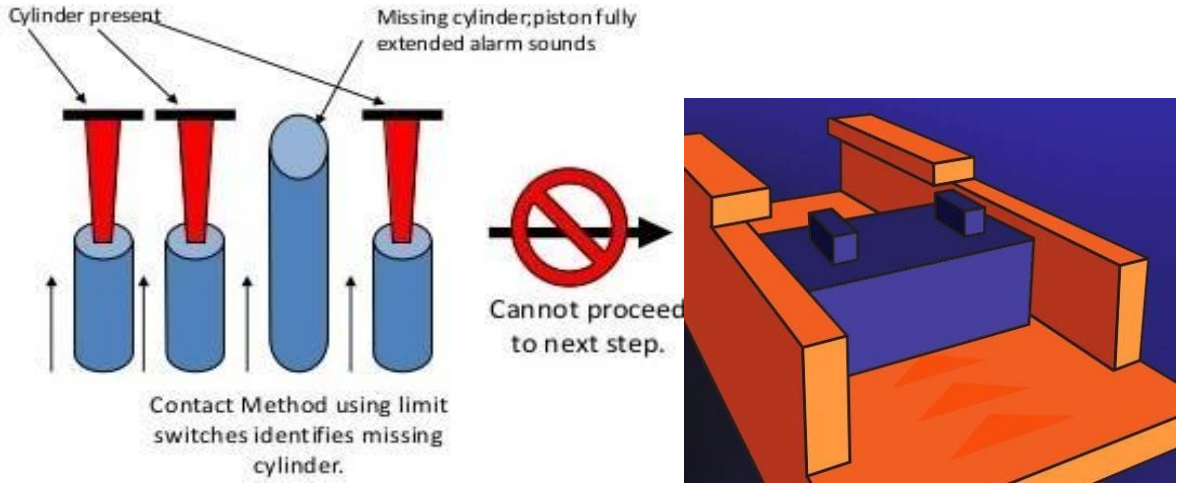


Figure 15. Switchless identifies

Poka-Yoke Types: Fixed Value Method (or constant number)

- This method for identifying an error uses a constant number to alert the operator if a particular criterion for the number of predetermined movements isn't met.
- Example: If an operator is tightening bolts with a wrench which has been dipped in paint, it would be easy to recognize which bolts have not been tightened by identifying the bolts which do not have paint on them.

- Another example is given by Dvorak, an operator charged with tightening six bolts and using a poka-yoke technique of dipping a wrench in diluted paint.
- The operator then can easily see where the untightened bolts are (untightened bolts will be lacking paint).

Figure 16. Motion approach

Poka-Yoke Types: Motion Step Method

- The Motion Step Method is a sequential mechanism for identifying errors.
- The method checks if the required sequence for a process has been followed.
- This method prevents the operator from accidently doing something they shouldn't, such as performing a step out of sequence resulting in a major issue.
    - Example: Many products have a system by which one cannot avoid a required step.
    - Many software applications have a mechanism by which until a process completes, the next one cannot begin.
    - Similarly, an operator might use a checklist to ensure that each process has been completed.
    - Proximity sensors or alarms, in the same way, assure that a device is properly assembled, or components removed.
    - Any error during the process might result in an alert.

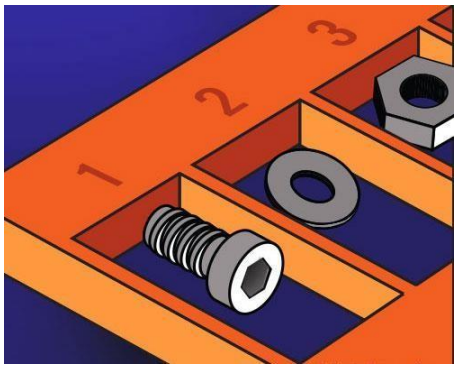


Figure 17. Motion approach

Poka-Yoke Example Automotive industry:

- Poka-yoke devices go beyond automatic braking systems when it comes to vehicles.
- Radar and video sensors constantly monitor the distance and speed of surrounding cars and objects, helping prevent collisions in the event of a sudden stop.

- These sensors can fall under the control and warning approaches (discussed earlier), depending on whether they actually stop the vehicle or just alert the driver to impending danger.

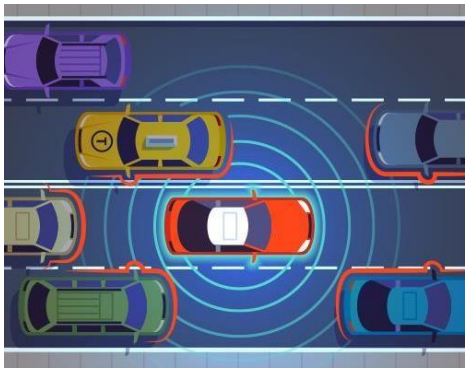


Figure 18. Motion approach

- Lane-keeping assist uses the control approach by automatically sensing and returning the car to the correct lane when the driver is unresponsive.
- Likewise, electronic stability control helps reduce accidents during a skid due to over-correcting; the condition being controlled in this instance is a wet or icy road.
- When the car is introduced to this condition, the system automatically kicks in to prevent an unwanted result.
- Even more poka-yoke devices can be found within the automotive industry, including blind-spot sensors and adaptive headlights.
- Also, the automatic transmission in most cars requires the vehicle to be in the "park" or "neutral" position before it can be started

Poka-Yoke Example – Around the house:

- Many things used on a daily basis have built-in poka-yoke devices or sensors that make them safer without much thought being given to them.
- Microwaves, washing machines, dryers, dishwashers and other household appliances might not start until a door is shut or may use sensors to know when to stop drying clothes or filling with water.
- Electrical plugs have an earth pin that is longer than the other pins, so they can only be inserted one way, which is a classic contact method of poka-yoke.
- Other examples include child-proof tops of pill bottles that prevent accidental ingestion, elevator doors that have sensors causing them to open when there is an obstruction, and lawnmowers with a bar on the handle that, when released, shuts off the mower to prevent accidents.

Poka-Yoke Example – Manufacturing industry:

- In lean manufacturing, the goal is to produce defect-free products the first time.
- Mistake-proofing is a built-in quality-assurance technique to ensure this happens.
- Common examples of poka-yoke devices in manufacturing include magnets in a food-processing plant to detect and remove metal pieces before packaging, interlock switches that can identify the position of a machine's guard and switch off the machine when the guard is lifted, safety mats near dangerous machinery that automatically trigger a machine shutdown when someone steps on them, personal protective equipment like gloves that are in eye-catching colors for the food industry in case they fall into the food, and standardized containers at

workstations that contain exact quantities of material.

- The list can be fairly extensive. These are just a few examples of how poka-yoke devices work, not only in manufacturing but also in daily life.

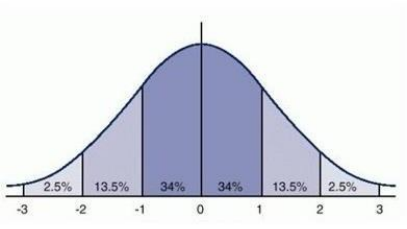
Where to Implement Poka-Yoke

- Once you've decided on which process could benefit from a poka-yoke device, there are a few areas to consider placing the device or method within the workflow. The location of the device within the process workflow can affect its level of effectiveness.
- The best location for a poka-yoke device is before an operator or the machine starts a task. In other words, it's best to prevent any errors by not allowing the process to even begin until a certain set of conditions are met.
- The second-best location for a poka-yoke device is during a particular step within a process. This way, an operator or machine receives an alert in the event of an error and stops the part or process from moving down the line.
- Finally, the third-best location for a poka-yoke device is downstream after the process step is finished, typically at an inspection station. At this point, you're assuming errors or defects may have occurred and are dependent on detection controls to find the defects before sending the process down the line.

Benefits of Poka-Yoke

- Aside from the obvious goals of eliminating human and mechanical errors and not accepting, allowing or passing along defects within a manufacturing setting, poka-yoke can yield other benefits that are not quite as evident.
    - Operators will need less training because processes containing poka-yoke devices automatically correct any deviation from what is required.
    - An increase in safety is an added benefit where workers are exposed to hazardous materials or dangerous machinery such as when working with high-inertia machines or petrochemicals.
    - Fewer quality checks through sampling and inspection are needed thanks to the mistake-proofing aspects of poka-yoke. While inspections still have their place, with poka-yoke, eliminating mistakes is built into processes through the prediction or detection of errors or defects.
    - Work becomes less repetitive for operators, as it helps them work through processes error-free the first time, which prevents them from having to go back and do it again correctly.
    - The quality of processes is improved, resulting in higher quality products. Improved processes also begin to produce effective teams that work in a coordinated effort to deliver error-free products the first time.

Statistical Processing Control



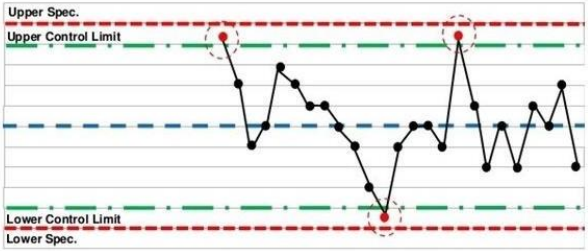


Figure 19. Statistical approach

- Is the application of Statistical Methods to monitor and control a process to ensure that it operates at its full potential to produce conforming product.

OR

- Is an analytical decision making tool which allows you to see when a process is working correctly and when it is not.

- Variation is present in any process, deciding when the variation is natural and when it needs correction is the key to quality control.

Statistical Processing Control – History

- Was Pioneered By Walter .A. Shewhart In The Early 1920s.

- W. Edwards Deming Later Applied SPC Methods In The US During World war II, Successfully Improved Quality In The Manufacture Of Munitions And Other Strategically Important Products.

- Deming introduced SPC Methods to Japanese Industry After The War Had Ended.

- Resulted high quality of Japanese products.

- Shewhart Created The Basis For The Control Chart And The Concept Of A State Of Statistical Control By Carefully Designed Experiments.

- Concluded That While Every Process Displays Variation, Some Processes Display Controlled Variation That Is Natural To The Process (Common Causes Of Variation), While Others Display Uncontrolled Variation That Is Not Present In The Process Causal System At All Times (Special Causes Of Variation).

- In 1988, The Software Engineering Institute Introduced The Notion That SPC Can Be Usefully Applied To Non-manufacturing Processes

Traditional Methods Vs Statistical Processing Control

- The quality of the finished article was traditionally achieved through post-manufacturing inspection of the product; accepting or rejecting each article (or samples from a production lot) based on how well it met its design specifications

- SPC uses Statistical tools to observe the performance of the production process in order to predict significant deviations that may later result in rejected product.

Types of Variations

Two kinds of variation occur in all manufacturing processes

1. Natural or Common Cause Variation
   consists of the variation inherent in the process as it is designed.
   may include variations in temperature, properties of raw materials, strength of an electrical current etc.

2. Special Cause Variation or Assignable-cause Variation
   With sufficient investigation, a specific cause, such as abnormal raw material or incorrect set-up parameters, can be found for special cause variations.

'In Control' and 'Out of Control'

❖**Process is said to be 'in control' and stable**

If common cause is the only type of variation that exists in the process

It is also predictable within set limits i.e. the probability of any future outcome falling within the limits can be stated approximately.

❖**Process is said to be 'out of control' and unstable**

Special cause variation exists within the process

Statistical Processing Control

Statistical process control -broadly broken down into 3 sets of activities

1. Understanding the process
2. Understanding the causes of variation
3. Elimination of the sources of special cause variation.

Understanding the process

- Process is typically mapped out and the process is monitored using control charts.

Understanding the causes of variation

- Control charts are used to identify variation that may be due to special causes, and to free the user from concern over variation due to common causes.
- It is a continuous, ongoing activity.
- When a process is stable and does not trigger any of the detection rules for a control chart, a process capability analysis may also be performed to predict the ability of the current process to produce conforming product in the future.

Statistical Processing Control – Understanding

•When excessive variation is identified by the control chart detection rules, or the process capability is found lacking, additional effort is exerted to determine causes of that variance.

• The tools used include

- **Ishikawa diagrams**
- **Designed experiments**
- **Pareto charts**

•Designed experiments are critical -only means of objectively quantifying the relative importance of the many potential causes of variation.

Elimination of the Sources of Special cause Variation

•Once the causes of variation have been quantified, effort is spent in eliminating those causes that are both statistically and practically significant.

•includes development of standard work, error-proofing and training.

•Additional process changes may be required to reduce variation or align the process with the desired target, especially if there is a problem with process capability.

Advantage of SPC

• Reduces waste

• Lead to a reduction in the time required to produce the product or service from end to end

due to a diminished likelihood that the final product will have to be reworked, identify bottlenecks, wait times, and other sources of delays within the process.

•A distinct advantage over other quality methods, such as inspection - its emphasis on early detection and prevention of problems

•Cost reduction

•Customer satisfaction

SPC Charts

•One method of identifying the type of variation present.

•Statistical Process Control (SPC) Charts are essentially:

- ❖Simple graphical tools that enable process performance monitoring.
- ❖ Designed to identify which type of variation exists within the process.
- ❖ Designed to highlight areas that may require further investigation.
- ❖Easy to construct and interpret.

•2 most popular SPC tools

- ❖ **Run Chart**
- ❖**Control Chart**

•SPC charts can be applied to both dynamic processes and static processes.

Dynamic Processes

- A process that is observed across time is known as a dynamic process.
- An SPC chart for a dynamic process - 'time-series' or a 'longitudinal' SPC chart.

Static Processes

- A process that is observed at a particular point in time is known as a static process.
- An SPC chart for a static process is often referred to as a 'cross sectional' SPC chart.

Control Charts

- Show the variation in a measurement during the time period that the process is observed.

- Monitor processes to show how the process is performing and how the process and capabilities are affected by changes to the process. This information is then used to make quality improvements.

- A time ordered sequence of data, with a centre line calculated by the mean.

- Used to determine the capability of the process.

- Help to identify special or assignable causes for factors that impede peak performance.

Control Charts – Four Key Features

1) **Data Points:**
   Either averages of subgroup measurements or individual measurements plotted on the x/y axis and joined by a line. Time is always on the x-axis.

2) **The Average or Center Line**
   The average or mean of the data points and is drawn across the middle section of the graph, usually as a heavy or solid line.

3) **The Upper Control Limit (UCL)**
   Drawn above the centerline and annotated as "UCL". This is often called the "+ 3 sigma" line.

4) **The Lower Control Limit (LCL)**
   Drawn below the centerline and annotated as "LCL". This is called the "- 3 sigma" line.

Control Charts

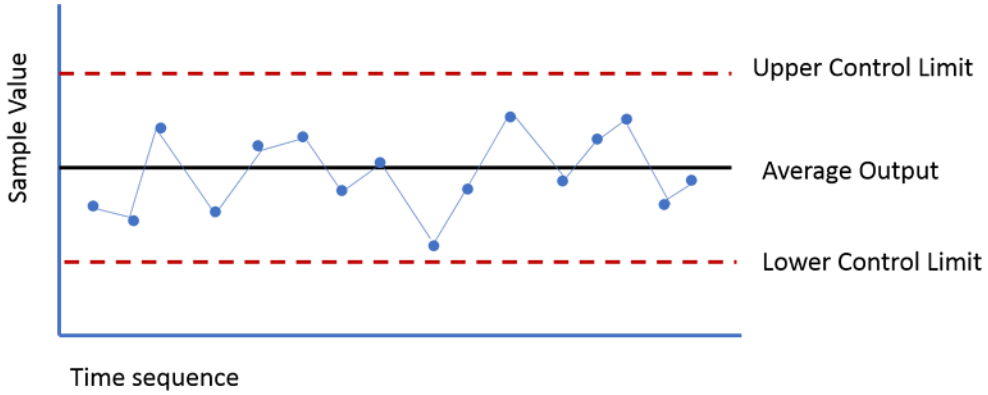


Figure 20. control charts

➢Control limits define the zone where the observed data for a stable and consistent process occurs virtually all of the time (99.7%).

➢Any fluctuations within these limits come from common causes inherent to the system, such as choice of equipment, scheduled maintenance or the precision of the operation that results from the design.

➢An outcome beyond the control limits results from a *special cause.*

➢The automatic control limits have been set at 3-sigma limits.

•The area between each control limit and the centerline is divided into thirds.

1) Zone A - "1-sigma zone“
2) Zone B - "2-sigma zone"
3) Zone C - " 3-sigma zone “

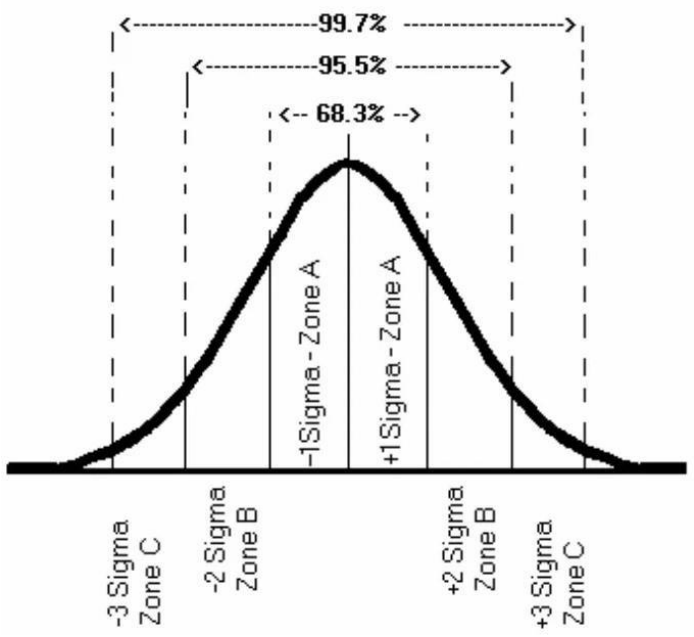


Figure 20. Statistical approach

➢Control limits define the zone where the observed data for a stable and consistent process occurs virtually all of the time (99.7%).

➢Any fluctuations within these limits come from common causes inherent to the system, such as choice of equipment, scheduled maintenance or the precision of the operation that results from the design.

➢An outcome beyond the control limits results from a *special cause.*

➢The automatic control limits have been set at 3-sigma limits.

Failure Mode and Effect Analysis (FMEA)

Figure 20. FMEA

Learning Objectives

- To understand the use of Failure Modes Effect Analysis (FMEA)
- To learn the steps to developing FMEAs
- To summarize the different types of FMEAs
- To learn how to link the FMEA to other Process tools

Benefits

- Allows us to identify areas of our process that most impact our customers
- Helps us identify how our process is most likely to fail
- Points to process failures that are most difficult to detect

Application Examples

- Manufacturing: A manager is responsible for moving a manufacturing operation to a new facility. He/she wants to be sure the move goes as smoothly as possible and that there are no surprises.
- Design: A design engineer wants to think of all the possible ways a product being designed could fail so that robustness can be built into the product.
- Software: A software engineer wants to think of possible problems a software product could fail when scaled up to large databases. This is a core issue for the Internet.

What Is A Failure Mode?

A Failure Mode is:

- The way in which the component, subassembly, product, input, or process could fail to perform its intended function
    - Failure modes may be the result of upstream operations or may cause downstream operations to fail
- Things that could go wrong

FMEA

- Why
    - Methodology that facilitates process improvement
    - Identifies and eliminates concerns early in the development of a process or design
    - Improve internal and external customer satisfaction
    - Focuses on prevention
    - FMEA may be a customer requirement (likely contractual)
    - FMEA may be required by an applicable Quality Management System Standard (possibly ISO)
- A structured approach to:
    - Identifying the ways in which a product or process can fail
    - Estimating risk associated with specific causes
    - Prioritizing the actions that should be taken to reduce risk
    - Evaluating design validation plan (design FMEA) or current control plan (process FMEA)

When to Conduct an FMEA

- Early in the process improvement investigation
- When new systems, products, and processes are being designed
- When existing designs or processes are being changed
- When carry-over designs are used in new applications
- After system, product, or process functions are defined, but before specific hardware is selected or released to manufacturing

History of FMEA

- First used in the 1960's in the Aerospace industry during the Apollo missions
- In 1974, the Navy developed MIL-STD-1629 regarding the use of FMEA
- In the late 1970's, the automotive industry was driven by liability costs to use FMEA
- Later, the automotive industry saw the advantages of using this tool to reduce risks related to poor quality

The FMEA Form

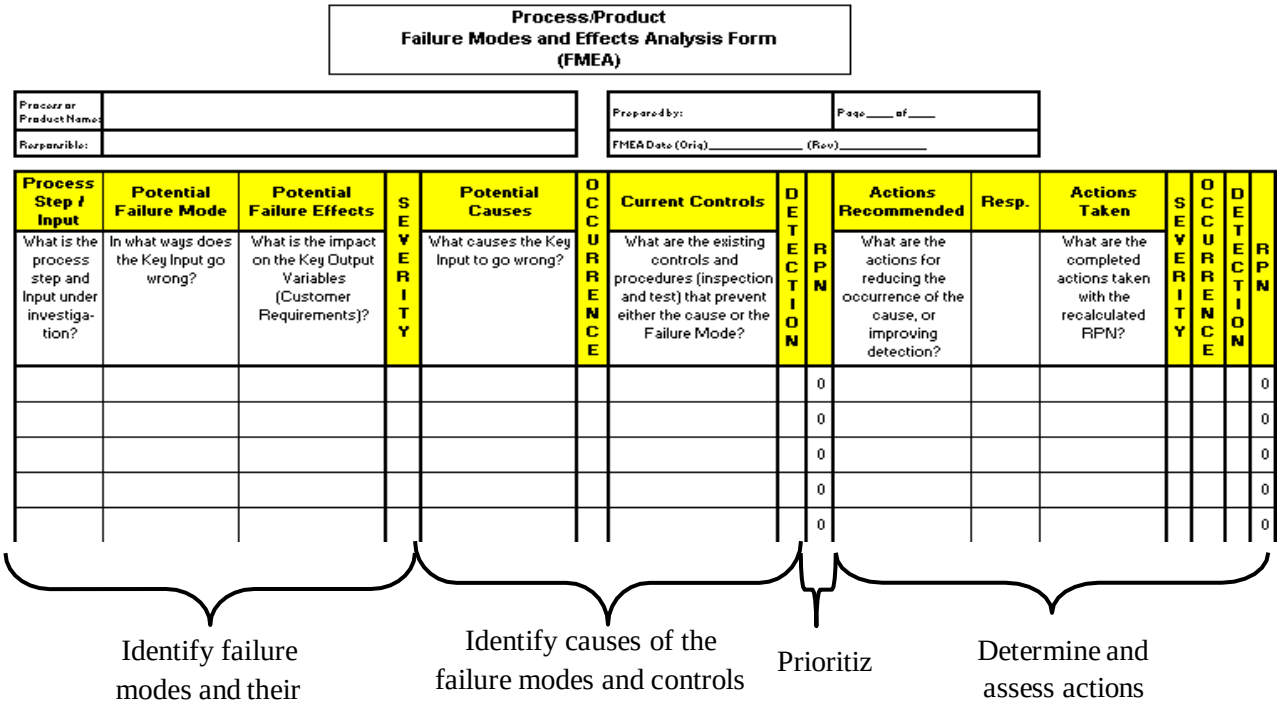


Figure 21. FMEA form

Types of FMEAs

- Design
  - Analyzes product design before release to production, with a focus on product function
  - Analyzes systems and subsystems in early concept and design stages
- Process
  - Used to analyze manufacturing and assembly processes after they are implemented

FMEA: A Team Tool

- A team approach is necessary.
- Team should be led by the Process Owner who is the responsible manufacturing engineer or technical person, or other similar individual familiar with FMEA.
- The following should be considered for team members:

– Design Engineers
– Process Engineers
– Materials Suppliers
– Customers
– Operators
– Reliability
– Suppliers

FMEA Procedure

- For each process input (start with high value inputs), determine the ways in which the input can go wrong (failure mode)
- For each failure mode, determine effects
    - Select a severity level for each effect
- Identify potential causes of each failure mode
    - Select an occurrence level for each cause
- List current controls for each cause
    - Select a detection level for each cause
- Calculate the Risk Priority Number (RPN)
- Develop recommended actions, assign responsible persons, and take actions
    - Give priority to high RPNs
    - MUST look at severities rated a 10
- Assign the predicted severity, occurrence, and detection levels and compare RPNs

FMEA Inputs and Outputs

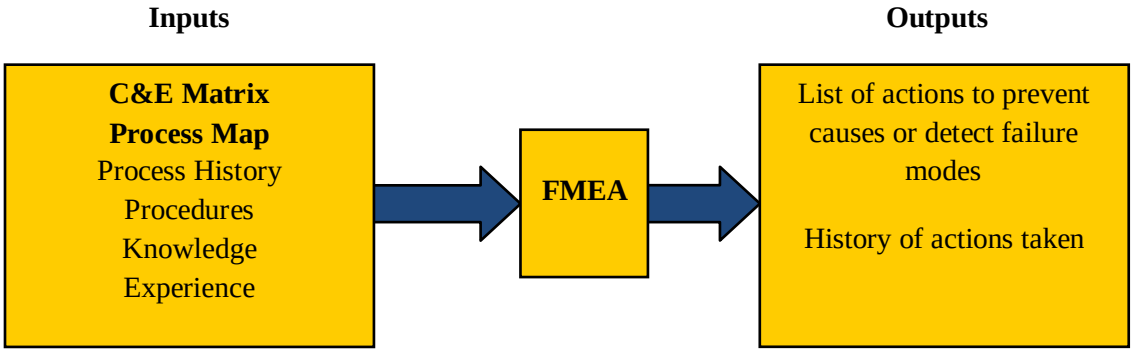


Figure 21. FMEA I/O

Severity, Occurrence and Detection

- Severity
    - Importance of the effect on customer requirements
- Occurrence
    - Frequency with which a given cause occurs and creates failure modes (obtain from past data if possible)
- Detection

Rating Scales

- The ability of the current control scheme to detect (then prevent) a given cause (may be difficult to estimate early in process operations).
    - There are a wide variety of scoring “anchors”, both quantitative or qualitative
    - Two types of scales are 1-5 or 1-10
    - The 1-5 scale makes it easier for the teams to decide on scores
    - The 1-10 scale may allow for better precision in estimates and a wide variation in scores (most common)
    - Severity
        - 1 = Not Severe, 10 = Very Severe
    - Occurrence
        - 1 = Not Likely, 10 = Very Likely

- Detection
    - 1 = Easy to Detect, 10 = Not easy to Detect

Risk Priority Number (RPN)

- RPN is the product of the severity, occurrence, and detection scores.

Severity **x** Occurrence **x** Detection = **RPN**

FMEA

- An FMEA:
    - Identifies the ways in which a product or process can fail
    - Estimates the risk associated with specific causes
    - Prioritizes the actions that should be taken to reduce risk
- FMEA is a team tool
- There are two different types of FMEAs:
    - Design
    - Process
- Inputs to the FMEA include several other Process tools such as C&E Matrix and Process Map.

Quality Function Deployment

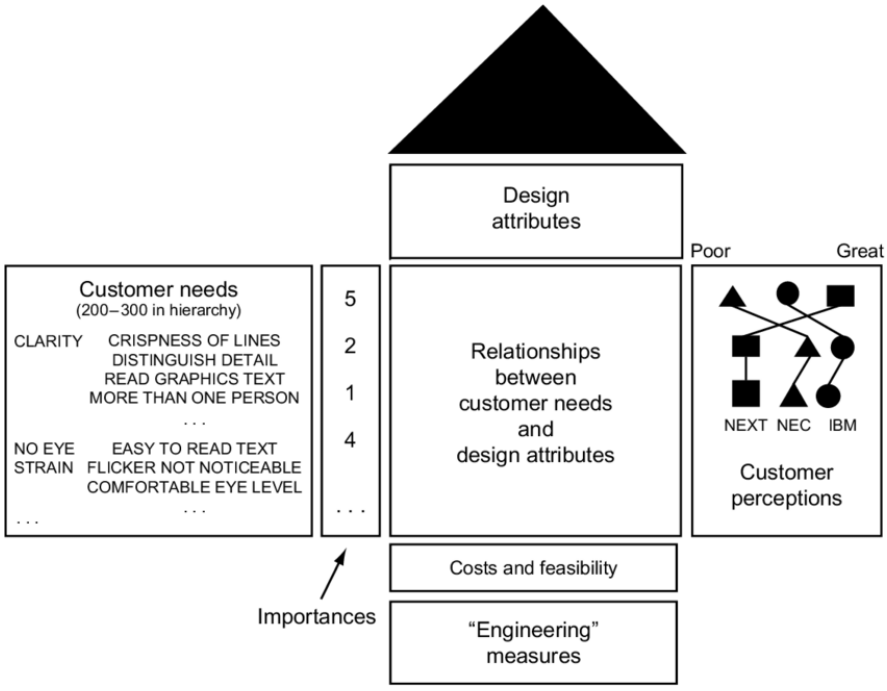


Figure 22. QFD

Overview of QFD

- The History of QFD.
- What is QFD?
- Why use QFD?
- Characteristics of QFD?

History of QFD

- 1960's, Yoji Akao conceptualized QFD.
- Statistical Quality Control, SQC, was the central quality control activity after WWII.
- SQC became Total Quality Control, TQC.
- QFD was derived from TQC.

First Application of QFD

- 1966, Bridgestone Tire Corp first used a process assurance table.
- 1972, the process assurance table was retooled by Akao to include QFD process.
- 1972, Kobe Shipyards (of Mitsubishi Heavy Industry) began a QFD Oil Tanker project.
- 1978, Kobe Shipyards published their quality chart for the tanker.

QFD in Software Engineering

- The QFD Research Group was seeking research relating to QFD in Software Engineering since 1987.
- A new style of QFD, Software QFD (SQFD), has emerged.
- DEC, AT&T, HP, IBM and Texas Instruments have all published information relating to SQFD (Haag, 1996).

Additional Techniques

- There are many techniques which are a style of QFD or are used to enhance QFD.
- These include: TRIZ, conjoint analysis, the seven product planning tools, Taguchi methods, Kano model, SQFD, DQFD, Gemba, Kaizen, Comprehensive QFD, QFD (N), QFD (B).

What is QFD?

- Quality Function Deployment, QFD, is a quality technique which evaluates the ideas of key stakeholders to produce a product which better addresses the customers needs.
- Customer requirements are gathered into a visual document which is evaluated and remodeled during construction so the important requirements stand out as the end result.

The QFD Paradigm

- QFD provides the opportunity to make sure you have a good product before you try to design and implement it.
- It is about planning and problem prevention, not problem solving (Eureka, 1988).
- QFD provides a systematic approach to identify which requirements are a priority for whom, when to implement them, and why.

High-Level QFD

- Requirements are initially elicited using other RE techniques (interviewing, brain-storming, focus-groups, etc).
- QFD involves the refinement of requirements using matrices and charts based on group decided priorities.
- There are 4 Phases of QFD. Each Phase requires internal iteration before proceeding to the next. Once at a Phase you do not go back.

What Does QFD Require?

- QFD requires time, effort, and patience.
- QFD requires access to stakeholder groups.
- The benefits of QFD are not realized immediately. Usually not until later in the project or the next project.
- QFD requires full management support. Priorities for the QFD process cannot change if benefits are to be realized.

Why use QFD?

- The QFD process leads participants to a common understanding of project direction and goals.
- QFD forces organizations to interact across their functional boundaries (Hales, 1995).
- QFD reduces design changes (Mazur, 2000).

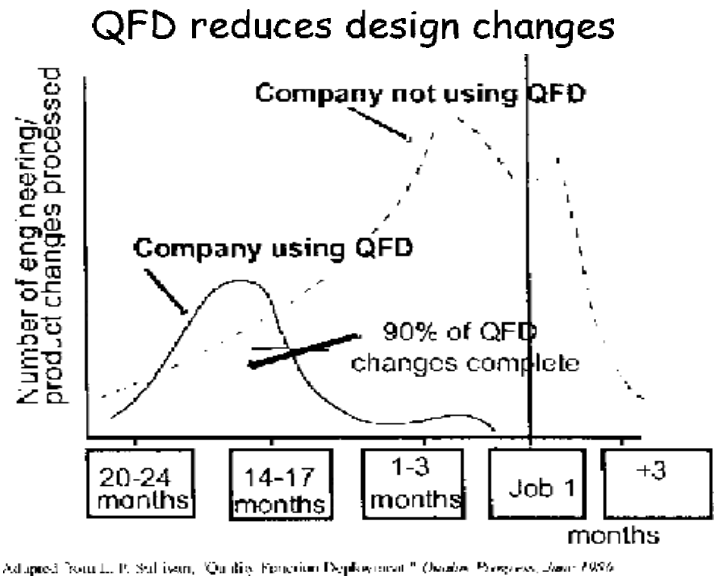


Figure 22. artifacts

QFD Artifacts

- Prioritized list of customers and competitors.
- Prioritized list of customer requirements.
- Prioritized list of how to satisfy the requirements.
- A list of design tradeoffs and an indication of how to compromise and weigh them.
- A realistic set of target values to ensure satisfaction.

What about Cost?

- Cost reduction is not mentioned as a 'Why to use QFD'.
- Initial costs will be as high or a little higher compared with traditional techniques.
- You are seeking long term savings in that product or the products that follow.

Characteristics of QFD

- 4 Main Phases to QFD
    - Product Planning including the 'House of Quality' (Requirements Engineering Life Cycle)

- Product Design (Design Life Cycle)
- Process Planning (Implementation Life Cycle)
- Process Control (Testing Life Cycle)

Ideas for Continuous Improvement

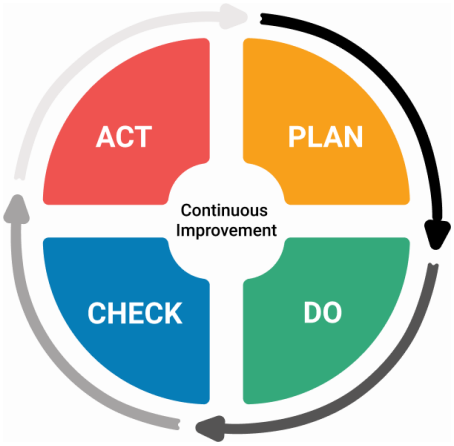


Figure 23. conteneous improvement

Ideas for Continuous Improvement

- "A stitch in time saves nine", goes the old adage. The same holds true in the case of software development life cycle. The earlier you detect and fix bugs, the more you save on costs and time. And continuous process improvement in software testing is exactly that stitch.

- The best way to ensure high-quality software is to implement effective and timely QA testing best practices that offer robust tools and methodologies to build flawless products.

Software Testing As A Continuous Improvement Process

- Software life cycle testing essentially means that testing occurs parallelly with the development cycle and is a continuous process. It is important to start the software testing process early in the application lifecycle, and it should be integrated into application development itself.

- To be able to do the same, there needs to be continuous effort and commitment on the part of the development organization, along with consistent communication with the quality assurance team.

PDCA Cycle for Continuous Improvement in Software Testing

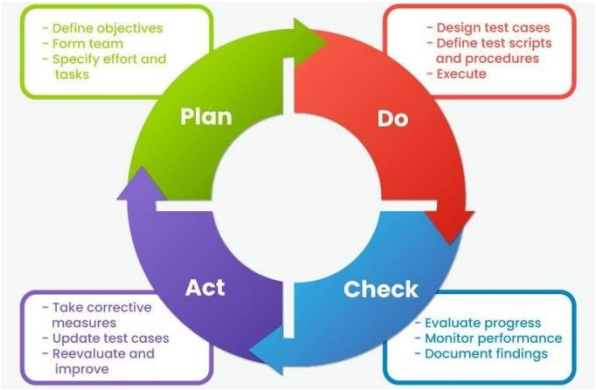


Figure 24. PDCA Cycle for Continuous Improvement in Software Testing

Plan

- One of the top approaches in software testing best practices is PDCA – plan, do, check, and act – an effective control mechanism used to control, govern, supervise, regulate, and restrain a system.
- Here is how the PDCA approach works in the context of continuous process improvement in software testing –
    - Plan, Do, Check, Act

Do

- In this step of the software testing improvement process, test objectives are defined clearly, including what is to be accomplished as a result of testing.
- While the testing criteria ensure that the software performs as per the specifications, objectives help to ensure that all stakeholders contribute to the definition of the test criteria in order to maximize quality.

- This stage in continuous process improvement in software testing describes how to design and execute the tests that are included in the test plan.
- The test design typically includes test procedures and scripts, test cases, expected results, test logs, and more.
- The more comprehensive a test plan is, the simpler the test design will be.

Check

- The Check step of the continuous improvement process primarily includes a thorough evaluation of how the testing process is progressing.
- At this stage, it is important to base decisions on accurate and timely data such as the workload effort, number and types of defects, and the schedule status.

Act

- The Act step of the continuous improvement process includes outlining clear measures for appropriate actions related to work that was not performed as per the plan.
- Once done, this analysis is used back into the plan by updating the test cases, test scripts, and reevaluating the overall process and tech details of testing.

6 Key Tips for Continuous Improvement in Software Testing

- Similar to any other business investment, quality assurance, or QA improvement ideas must bring value to the enterprise.
- This value expected from the quality assurance process is to make the software processes much more efficient while ensuring that the end-product meets customers' needs.
- When translated into measurable objectives such as flawless design and coding, elimination of defects early on, and ensuring efficient discovery, it can lead to better software processes and a value-driven final product.
- To achieve this objective, businesses need to improve their processes to install quality assurance activities at every stage of the software life cycle.
- Here are some of the software testing best practices that can help you achieve your goal of smarter and effective testing-
    - Devising A Plan and Defining Strategy
    - Scenario Analysis
    - Test Data Identification
    - Automated Testing
    - Pick the Right QA Tools
    - Robust Communication Between Test Teams

Figure 25. STLC

1. Devising A Plan and Defining Strategy

- Effective planning entails the creation of quality management and test plans for a project.
- Before you start investing time, resources, and money into the project, it's recommended to check whether the plan has covered all the basics and is feasible in terms of timeline and resources.
- **Quality management plan** – defines a clear and acceptable level of product quality and describes how the project will achieve the said level.
- The main components of a quality management plan are –
    - Key project deliverables and processes for satisfactory quality levels
    - Quality standards and tools
    - Quality control and assurance activities
    - Quality roles and responsibilities
    - Planning for quality control reporting and assurance problems
- Test strategy – The outline of a good strategy includes a detailed introduction, the overall plan, and testing requirements.
- The main components of a test strategy include –
    - Test objectives and scope of testing
    - Industry standards
    - Budget limitations
    - Different testing measurement and metrics
    - Configuration management
    - Deadlines and test execution schedule
    - Risk identification requirements

2. Scenario Analysis

- Irrespective of how comprehensive a test plan is, problems are inevitable, which would escape from one test phase to the next. Post-project & in-process escape analysis, therefore, is critical for driving the test improvements.
- While there can be instances where the testing team is required to directly start test execution, it is always better to create a high-level scenario during the early stages of requirement study and ensure that it is reviewed on a consistent basis.
- There are multiple benefits that this kind of reviews can bring including –
    - Providing indications on the understanding of the tester
    - Conformance on coverage

3. Test Data Identification

- When we design test scenarios or test cases, we create various types of tests, including negative and positive cases. To be able to execute the planned tests, we require different types of data that need testing using simple parameters. But there are several instances where the same data needs to be generated from a different source and requires transformation before it reaches the destination system or flows into multiple systems.
- It is, therefore, always a great practice to start with identifying the data sets early on during the test design phase instead of waiting until the test execution phase starts.
- At this stage, you need to look for the answers to some of the important questions such as –

- Which test phase should have removed the defect in a logical way?
- Is there any multi-threaded test that is missing from the system verification plan?
- Is there any performance problem missed?
- Have you overlooked any simple function verification test?

4. Automated Testing

- Continuous testing and process improvement typically follows the test early and test often approach. Automated testing is a great idea to get quick feedback on application quality.
- It is, however, important to keep in mind that identifying the scope of test automation doesn't always have to be a different exercise and can easily be identified during the manual test execution cycle by identifying the most painful areas and determining how those can be automated.
- Some of the points to take care of during automated testing include –
    - Clearly knowing when to automate tests and when to not
    - Automating new functionality during the development process
    - Test automation should include inputs from both developers and testers

5. Pick the Right QA Tools

- It is important for testers to pick the right testing tools based on the testing requirement and purpose.
- Some of the most widely used tools are Jenkins, Selenium, GitHub, New Relic, etc.
- Best QA improvement ideas mainly include planning the entire procedure for QA automated testing, picking up the right tools, integrating QA with other functions, creating a robust testing work environment, and performing continuous testing.

6. Robust Communication Between Test Teams

- Continuous improvement is always a byproduct of continuous communication.
- In software testing best practices particularly, it is a great strategy to consider frequent communication between teams whose activities overlap during an active product development cycle.
- This helps to ensure that they are actively communicating observations, concerns, & solutions to one another.

Benefits Of Test Process Improvement

- An increasing number of organizations are realizing the fact that improving the test process is critical for ensuring the quality of the software and overall business processes and multiple other benefits it offers.
- Some of these are listed below –

Figure 26. Test process improvement

Early and accurate feedback to stakeholders

- Deployment of continuous testing ensures early feedback to the development team about various types of issues the code may cause to existing features.
- Further test process improvement provides frequent, actionable feedback at multiple development stages to expedite the release of software applications into production with a much lesser number of defects.
- Another benefit of this early feedback is in analyzing business risk coverage to achieve a faster time to market.

Reduces the cost of defects

- The process of test process improvement plays a crucial role in ensuring error-free outputs.
- Continuous testing ensures a quicker turnaround time when it comes to the identification and elimination of the expected code errors early in the development lifecycle.
- The result is a substantial reduction in the overall cost of resolving defects.

Speeds up release cycles

- Test process improvement and automated testing equip organizations to better respond to frequent market changes. With continuous testing and test automation, organizations also get the advantage of quickly developed and frequently released updates.

- Automated testing allows testing of the developed code (existing & new) rigorously and constantly. It also focuses on rapid error resolution to ensure clean code delivery and better integrations to speed up the launch of the application on a regular basis.
    - Among some of the other advantages of test process improvement include –
    - Improved overall software quality
    - Increased efficiency and effectiveness of test activities
    - Reduced downtime
    - Testing aligned with main organizational priorities
    - Leads to more efficient and effective business operations
    - Long-term cost reduction in testing
    - Reduced errors and enhanced compliance

Bottom Line

- The continuous process improvement in software testing not only ensures higher product quality but also optimizes business processes.
- However, in practice, it is often quite challenging to define the steps needed to implement QA improvement ideas.
- Organizations need to thoroughly review their testing process, and be proactive and forward-thinking in their approach.
- The need is to have a well-defined standard for testing or a continuous improvement program that is constantly evolving to meet both the customers' and the organization's business needs.
- Get in touch with our QA experts to implement software testing best practices. Our collaborative and methodical approach can help you reduce testing time, run timely test cycles, elevate your product quality, and save resources.
- Having a robust quality assurance process in place for all stages of the software life cycle is the key to efficient systems, significant savings, and a much higher ROI.

Unit -5
QUALITY ASSURANCE MODELS

Software Quality Standards, ISO 9000 series – CMM, CMMI – P-CMM – Six Sigma – Malcolm Baldrige Quality - Case study

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- State what is meant by ISO 9000 certification.
- Identify the different industries to which the different types of ISO 9000 quality standards can be applied.
- Differentiate between the characteristics of software products and other type of products that make managing a software development effort difficult.
- Identify the reasons why obtaining ISO 9000 certification is beneficial to a software development organization.
- Explain the main requirements that a software development organization must satisfy for getting ISO 9001 certification.
- Identify the salient features of ISO 9001 certification.
- Identify the shortcomings of ISO 9000 certification.

ISO 9000 certification

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 seriesof standards in 1987. ISO certification serves as a reference for contract betweenindependent 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 ofguidelines 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 mostsoftware 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 designsfrom 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 andtesting 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 toestimate how much more time will it take.

1

- 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 developmentorganization 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 torepeatable 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 andrecommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

Summary of ISO 9001 certification

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 capabilityfor 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 configurationmanagement tools is necessary.

Purchasing (4.6)

Purchasing material, including bought-in software must be checked forconfor

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 effectiveas 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.

SEI CAPABILITY MATURITY MODEL

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI CMM can be used two ways: capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work. Therefore, the results of software process capability assessment can be used to select a contractor. On the other hand, software process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

SEI CMM classifies software development industries into the following five maturity levels. Thedifferent levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system starting from scratch.

Level 1: Initial - A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depends on individual efforts and heroics. When engineers leave, the successors have great difficulty in understanding the process followed and the work completed. Since formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

Level 2: Repeatable - At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Please remember that opportunity to repeat a process exists only when a company produces a family of products.

Level 3: Defined - At this level the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured. ISO 9000 aims at achieving this level.
Level 4: Managed - At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met. Various toolslike Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing - At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it was found that the code reviews were notvery effective and a large number of errors were detected only during the unit testing, then the process may be fine-tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best software engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the

organization.

Except for SEI CMM level 1, each maturity level is characterized by several Key Process Areas (KPAs) that includes the areas an organization should focus to improve its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.CMM Level	Focus	Key Process Ares
1. Initial	Competent people	
2. Repeatable	Project management	Software project planning Software configuration management
3. Defined	Definition of processes	Process definition Training program Peer reviews
4. Managed	Product and process quality	Quantitative process metrics Software quality management
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change management

Key process areas (KPA) of a software organization

The focus of each SEI CMM level and the corresponding key processareas

SEI CMM provides a list of key areas on which to focus to take an organization from one levelof maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such thatone stage enhances the capability already built up. For example, it considers that trying to implement a defined process (SEI CMM level 3) before a repeatable process (SEI CMM level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures. ISO 9000 certification vs. SEI/CMM

For quality appraisal of a software development organization, the characteristics of ISO 9000 certification and the SEI CMM differ in some respects. The differences are as follows:

- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organization in official documents, communication with external parties, and the tender quotations. However,

SEI CMM assessment is purely for internal use.

SEI CMM was developed specifically for software industry and therefore addresses manyissues which are specific to software industry alone.

SEI CMM goes beyond quality assurance and prepares an organization to ultimately achieve Total Quality Management (TQM). In fact, ISO 9001 aims at level 3 of SEI CMM model.

SEI CMM model provides a list of key process areas (KPAs) on which an organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement.

Applicability of SEI CMM to organizations

Highly systematic and measured approach to software development suits large organizations dealing with negotiated software, safety-critical software, etc. For those large organizations, SEI CMM model is perfectly applicable. But small organizations typically handle applications such as Internet, e-commerce, and are without an established product range, revenue base, and experience on past projects, etc. For such organizations, a CMM-based appraisal is probably excessive. These organizations need to operate more efficiently at the lower levels of maturity. For example, they need to practice effective project management, reviews, configuration management, etc.

Capability Maturity Model Integration (CMMI) is a successor of CMM and is a more evolved model that incorporates best components of individual disciplines of CMM like Software CMM, Systems Engineering CMM, People CMM, etc. Since CMM is a reference model of matured practices in a specific discipline, soit becomes difficult to integrate these disciplines as per the requirements. This is why CMMI is used as it allows the integration of multiple disciplines as and when needed.

Objectives of CMMI :

. Fulfilling customer needs and expectations.
. Value creation for investors/stockholders.
. Market growth is increased.
. Improved quality of products and services.
. Enhanced reputation in Industry

CMMI Representation – Staged and Continuous :

A representation allows an organization to pursue a different set of improvement objectives. There are two representations for CMMI :

- Staged Representation :

- uses a pre-defined set of process areas to define improvement path.
    - provides a sequence of improvements, where each part in the sequenceserves as a foundation for the next.
    - an improved path is defined by maturity level.
    - maturity level describes the maturity of processes in organization.
    - Staged CMMI representation allows comparison between differentorganizations for multiple maturity levels.
- Continuous Representation :
    - allows selection of specific process areas.
    - uses capability levels that measures improvement of an individual processarea.
    - Continuous CMMI representation allows comparison between differentorganizations on a process-area-by-process-area basis.
    - allows organizations to select processes which require more improvement.
    - In this representation, order of improvement of various processes can be selected which allows the organizations to meet their objectives and eliminate risks.

CMMI Model – Maturity Levels :

In CMMI with staged representation, there are five maturity levels described as follows :

1. Maturity level 1 : Initial
    - processes are poorly managed or controlled.
    - unpredictable outcomes of processes involved.
    - ad hoc and chaotic approach used.
    - No KPAs (Key Process Areas) defined.
    - Lowest quality and highest risk.
2. Maturity level 2 : Managed
    - requirements are managed.
    - processes are planned and controlled.
    - projects are managed and implemented according to their documented plans.
    - This risk involved is lower than Initial level, but still exists.
    - Quality is better than Initial level.
3. Maturity level 3 : Defined
    - processes are well characterized and described using standards, properprocedures, and methods, tools, etc.
    - Medium quality and medium risk involved.
    - Focus is process standardization.
4. Maturity level 4 : Quantitatively managed
    - quantitative objectives for process performance and quality are set.
    - quantitative objectives are based on customer requirements, organization needs,etc.
    - process performance measures are analyzed quantitatively.
    - higher quality of processes is achieved.
    - lower risk
5. Maturity level 5 : Optimizing

- continuous improvement in processes and their performance.
    - improvement has to be both incremental and innovative.

- highest quality of processes.
- lowest risk in processes and their performanceCMMI Model – Capability Levels

A capability level includes relevant specific and generic practices for a specific process area that can improve the organization's processes associated with that process area. For CMMI models with continuous representation, there are six capability levels as describedbelow :

1. Capability level 0 : Incomplete
   - incomplete process – partially or not performed.
   - one or more specific goals of process area are not met.
   - No generic goals are specified for this level.
   - this capability level is same as maturity level 1.
2. Capability level 1 : Performed
   - process performance may not be stable.
   - objectives of quality, cost and schedule may not be met.
   - a capability level 1 process is expected to perform all specific and genericpractices for this level.
   - only a start-step for process improvement.
3. Capability level 2 : Managed
   - process is planned, monitored and controlled.
   - managing the process by ensuring that objectives are achieved.
   - objectives are both model and other including cost, quality, schedule.
   - actively managing processing with the help of metrics.
4. Capability level 3 : Defined
   - a defined process is managed and meets the organization's set of guidelines andstandards.
   - focus is process standardization.
5. Capability level 4 : Quantitatively Managed
   - process is controlled using statistical and quantitative techniques.
   - process performance and quality is understood in statistical terms and metrics.
   - quantitative objectives for process quality and performance are established.
6. Capability level 5 : Optimizing
   - focuses on continually improving process performance.
   - performance is improved in both ways – incremental and innovation.
   - emphasizes on studying the performance results across the organization to ensurethat common causes or issues are identified and fixed.
   - Six Sigma

- The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry.•
- Six Sigma at many organizations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and product to service.
-
- The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects

per million opportunities. A Six Sigma defect is defined as any system behavior that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator.

- The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies: DMAIC and DMADV. The Six Sigma DMAIC process (defines, measure, analyze, improve, control) is an improvement system for existing processes failing below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six
  Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts. Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

The Malcolm Baldrige National Quality Award

The Malcolm Baldrige National Quality Award (MBNQA) is an award intended to raise awareness of quality management and formally recognize U.S. companies with successful quality management systems. Although the award is only given to companies based in the U.S., it is recognized internationally. The MBNQA is named after Malcolm Baldrige, who was the U.S. Secretary of Commerce during the Ronald Reagan administration from 1981 to 1987.

Malcolm Baldrige Award Ceremony

Breaking Down the Malcolm Baldrige Award

The Malcolm Baldrige National Quality Award was developed in the late 1980s by the Department of Commerce to facilitate competition among U.S. companies. The award is the highest level of recognition for performance excellence that a U.S. company can receive.

Up to 18 awards are given annually across six categories:

1. Manufacturing
2. Service
3. Small Business
4. Education
5. Healthcare
6. Nonprofit

Recipients of the MBNQA must share information about their company performance and their practices with other companies, but they are not required to share proprietary information. The information is shared through two annual conferences:

1. Quest for Excellence Conference
2. Baldrige Fall Conference

Purpose of the Malcolm Baldrige Award

The main purposes of the Baldrige Award are to:

- Raise awareness about the importance of performance excellence.
- Recognize companies that show performance excellence and pass on this information toother organizations to tailor it for their own needs.
- Motivate U.S. companies and organizations to improve their quality standards and strive forexcellence.
- Help companies and organizations embody the competitive spirit and drive the U.S.economy forward.

The Baldrige Criteria for Performance Excellence

Applicants for the Baldrige Award are judged by an independent board of examiners. The seven criteria for judging include:

1. **Leadership:** How the management team leads their organization and the community.
2. **Strategy:** How the company establishes and implements strategic decisions.
3. **Customers:** How the company builds and maintains relationships with its customers.
4. **Measurement, Analysis, and Knowledge Management:** How efficiently the organizationuses data to support processes and manage performance.
5. **Workforce:** The degree to which the company empowers and involves its workforce.
6. **Operations:** The design, management, and improvement of key processes.
7. **Results:** Examines the organization's performance and improvement in key business areas such as customer satisfaction, finances, human resources, supplier and partner performance,governance, and social responsibility.

Benefits of Applying for the Malcolm Baldrige Award

Undoubtedly, an award is not needed to prove that a company is a good organization. However, there is significant value in applying for the Baldrige Award.

The Baldrige Award is considered "the best and most cost-effective and comprehensive business health audit you can receive." Writing and submitting an application has helped companies improve their plans, key processes, and communication, and identify ideas for improvement. In addition, each applicant to the Baldrige Award receives a comprehensive feedback report that highlights their strengths and opportunities for improvement.

Past applicants or recipients of the Baldrige Award have seen improvements in:

- Revenue
- Market share
- Employee involvement
- Employee empowerment
- Cost reduction opportunities
- Return on assets and return on equity

- Product reliability
- New product sales
- Customer engagement and satisfaction

Recipients of the Malcolm Baldrige Award

Nearly 1,700 American organizations have applied for the Baldrige Award, with only 113awards being presented to 106 organizations.

The 2017 Baldrige Award recipients are:

- **Bristol Tennessee Essential Services**, Bristol, TN – Small business sector
- **Stellar Solutions**, Palo Alto, CA – Small business sector
- **City of Fort Collins**, Fort Collins, CO – Nonprofit sector
- **Castle Medical Center**, Kailua, HI – Healthcare sector
- **Southcentral Foundation,** Anchorage, AK – Healthcare sector

References

TEXT / REFERENCE BOOKS 1. Software Engineering: A Practitioners Approach, 5th Edition Roger S. Pressman McGraw – Hill International Edition, 6th Edition, 2006.
2. Ramesh Gopalswamy, Managing global Projects ; Tata McGraw Hill, 2002.
3. Norman E – Fenton and Share Lawrence P flieger, Software metrics, International Thomson Computer press , 1997.
4. Gordan Schulmeyer. G. and James .L. Mc Hanus, Total Quality management for software, International Thomson Computer press, USA , 1990.
5. Dunn Robert M., Software Quality: Concepts and Plans, Englewood clifts, Prentice Hall Inc., 1990.
6. Metrics and Models in Software Quality Engineering, Stephen, Stephen H. Kan, Pearson education, 2006, Low price edition