MALLA REDDY ENGINEERING COLLEGE

(Autonomous)
Maisammaguda, Dhullapally, Post via Kompally, Secunderabad – 500100.

LESSON PLAN

Department: CSE **Academic Year: 2019-2020**

Year-Sec: IV–II Sem

Subject: Software Metrics **code: 50561**

Module	Lesson/Session	Topics	No. Of classes	Total No. of classes
I	Fundamentals of measurement and experimentation, Measurement, and Basics of measurement	**Fundamentals of measurement and experimentation**	1	12
		Measurement in everyday life	1	
		Measurement in software engineering,	1	
		the scope of software metrics.	1	
		B:Basics of measurement-representational theory of measurement,	2	
		Measurement and models,	2	
		Measurement scales and scale types,	2	
		Meaning filialness in measurement	2	
II	Goal-Based Framework for Software Measurement, Empirical investigation	**Goal-Based Framework for Software Measurement**	1	11
		Classifying software measures,	2	
		determining what to measure,	1	
		Applying the framework, Software measurement validation,	2	
		Software measurement validation in practice.	1	
		B:Empirical investigation-Four principles of investigation,	2	

		planning formal experiments	1	
		planning case studies.	1	
III	**Software-metrics data collection, Analyzing software-measurement data**	**A:Software-metrics data collection**-What is good data	1	11
		How to define the data,	1	
		How to collect data,	1	
		When to collect data,	1	
		How to store and extract data..	2	
		B:Analyzing software-measurement data-Introduction,	1	
		Analyzing the results of experiments,	2	
		Examples of simple analysis techniques	2	
IV	**Software-Engineering Measurement** -Structure	**Software-Engineering Measurement** -Measuring internal product attributes-	1	11
		Aspects of software size,	1	
		Length, Reuse, Functionality,	2	
		Complexity.	1	
		Structure-Types of structural measures,	1	
		Control-flow structure, Modularity and information	2	
		flow attributes, Object-oriented metrics,	1	
		Data structure,	1	
		Difficulties with general "complexity" measures.	1	
V	**Measuring External Product Attributes, Software Reliability**	**Measuring External Product Attributes**-Modeling software quality	2	11
		, Measuring aspects of quality.	1	
		Software Reliability: measurement and prediction-	1	
		Basics of reliability theory,	2	

		The software reliability problem, Parametric reliability growth models,.	1	
		Predictive accuracy,	1	
		The recalibration of software-reliability growth predictions,	1	
		The importance of the operational environment,	1	
		Wider aspects of software reliability	1	
Total No of classes Required:				**56**

Text Books:

1. Srinivasan Desikan and Gopalaswamy Ramesh,"Software Testing – Principles and Practices", Pearson education, 2006.
2. Van Nostrand Reinhold,"Software Testing Techniques", Boris Beizer, 2nd Edition, New York, 1990.

References:

3. Sams Publishing, "Software Testing", Ron Patton, Second Edition, Pearson education, 2007.
4. Renu Rajani, Pradeep Oak, "Software Testing – Effective Methods, Tools and Techniques", Tata McGraw Hill, 2004.
5. Edward Kit, "Software Testing in the Real World – Improving the Process", Pearson Education, 1995.

Outcomes:

Apply fundamental knowledge of Testing in Real time scenarios.
Test a simple application.
Understand and Applying the Techniques in Software Development Life cycle.

Module I

Fundamentals of measurement and experimentation

Measurement in everyday life

Metrics are standard (i.e commonly accepted scales) that define measurable attributes of entities their units and their scopes

Measure is a relation between an attribute and a measurement scale

Measurement in Everyday Life

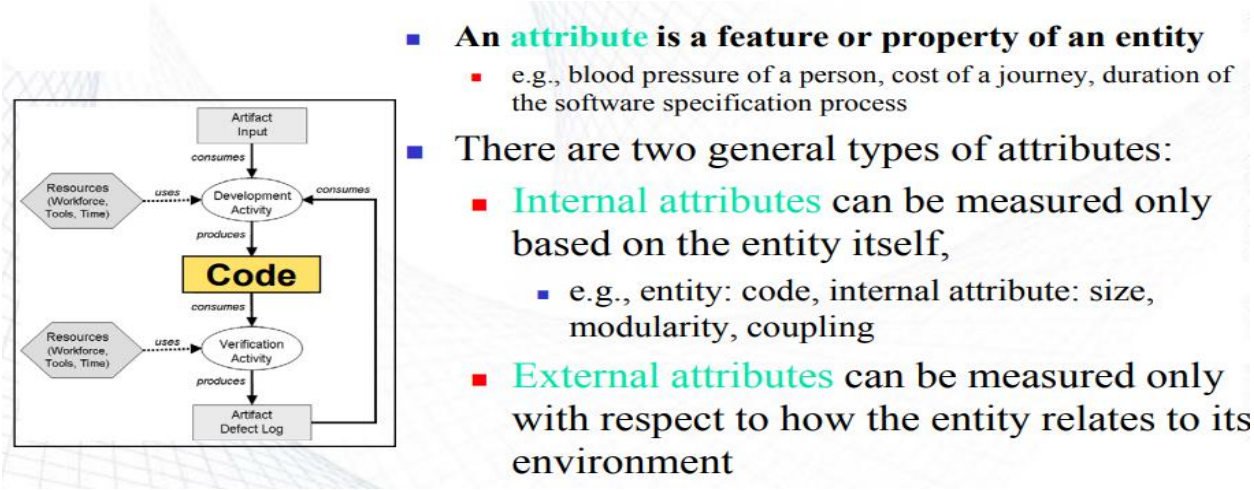
Measurement is not only used by professional technologists, but also used by all of us in everyday life. In a shop, the price acts as a measure of the value of an item. Similarly, height and size measurements will ensure whether the cloth will fit properly or not. Thus, measurement will help us compare an item with another.

The measurement takes the information about the attributes of entities. An entity is an object such as a person or an event such as a journey in the real world. An attribute is a feature or property of an entity such as the height of a person, cost of a journey, etc. In the real world, even though we are thinking of measuring the things, actually we are measuring the attributes of those things.

Attributes are mostly defined by numbers or symbols. For example, the price can be specified in number of rupees or dollars, clothing size can be specified in terms of small, medium, large.

The accuracy of a measurement depends on the measuring instrument as well as on the definition of the measurement. After obtaining the measurements, we have to analyze them and we have to derive conclusions about the entities.

Measurement is a direct quantification whereas calculation is an indirect one where we combine different measurements using some formulae.



Measurement in Everyday Life:

Measurement governs many aspects of Everyday life

-Economic indicators determine prices, pay raises

-Medical systems measurements enable diagnosis of specific illnesses

-Measurement in atmosphere systems are the basis of weather prediction

How do we use measurement in our lives

- In a shop price is a measure of the values of an item and we calculate the bill to make sure we get the correct change

-Height and size measurements ensure clothing will fit correctly

-When traveling we calculate distance choose a route, measure speed and predict when we will arrive

Measurement helps us to:

Understand our world. Interact with our surroundings and improve our lives.

What is measurement: allow to compare with other things

The process number or symbol are assigned to attributes of entities in the real world in such a way as to describe them

According to clearly define rules.

making things measurable

Ex. attributes of entity

Measurement in Software engineering

Measurement are made

-incomplete

-inconsistent

-infrequent

Most of the time

-design experiment

-what to measure

-Realistic error margin

-Information- results objectives

Measurement in Everyday Life

Software Measurement Objectives

-Assessing status

Projects

Products for a specific project or projects

-Process

Resources

Identifying trends

- need to be able to differentiate between a healthy project and one that in trouble

Determine corrective action

Measurement should indicate the appropriate corrective action if any is required.

Types of information to understand

control and improve projects

- Managers estimate cost, product, staff, customer satisfaction and improvements

Engineers: Requirements are testable, faults found, product, process goals, and future.

Measurement in Everyday life: Software metrics

-cost and effort estimation

-productivity measures and models

-Data collection

-Quality models and measures

-Reliability models

-Performance evaluation and models

-Structural and complexity metrics

-Capability maturity assessment

-Management by metrics

-Evaluation of methods and tools.

The scope of software metrics s

cost and effort estimation

Motivation accurately predict cost early in the development life cycle

-Numerous empirical cost models have been developed.

- COCOMO, COCOMO2

The scope of software metrics

Productivity models and measures

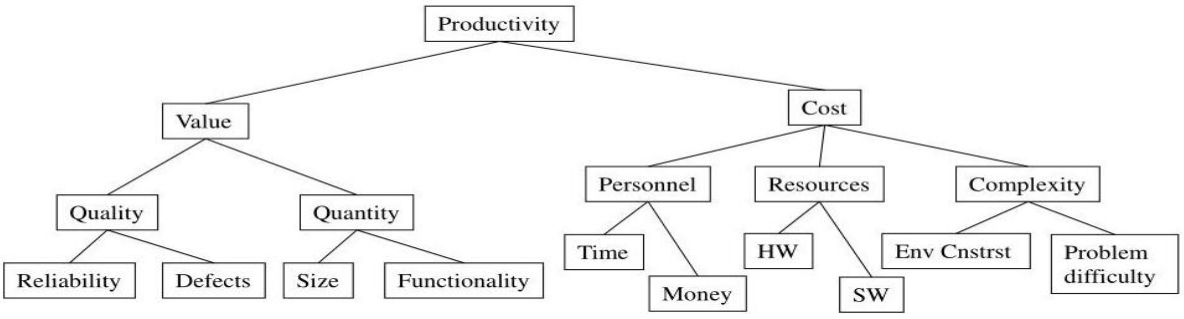
Estimate staff productivity to determine how much specified changes will cost

Naive measure: Size divided by effort, doesn't take into account things like defects, functionality, and reliability.

More comprehensive models have been developed

Measurement in Everyday Life

- The Scope of Software Metrics – some details
  - Possible productivity model



Measurement in software engineering

Measurement in Software Engineering

Software Engineering involves managing, costing, planning, modeling, analyzing, specifying, designing, implementing, testing, and maintaining software products. Hence, measurement plays a significant role in software engineering. A rigorous approach will be necessary for measuring the attributes of a software product.

For most of the development projects,

- We fail to set measurable targets for our software products
- We fail to understand and quantify the component cost of software projects
- We do not quantify or predict the quality of the products we produce

Thus, for controlling software products, measuring the attributes is necessary. Every measurement action must be motivated by a particular goal or need that is clearly defined and easily understandable. The measurement objectives must be specific, tried to what managers, developers and users need to know. Measurement is required to assess the status of the project, product, processes, and resources.

In software engineering, measurement is essential for the following three basic activities −

- To understand what is happening during development and maintenance
- To control what is happening in the project
- To improve processes and goals

- Measurement
  - is the act of obtaining a measure
- Measure
  - provides a quantitative indication of the size of some product or process attribute
- Metric
  - is a quantitative measure of the degree to which a system, component, or process possesses a given attribute

The Scope of Software Metrics

Software metrics is a standard of measure that contains many activities which involve some degree of measurement. It can be classified into three categories: product metrics, process metrics, and project metrics.

- **Product metrics** describe the characteristics of the product such as size, complexity, design features, performance, and quality level.

- **Process metrics** can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process.
- **Project 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.

Scope of Software Metrics

Software metrics contains many activities which include the following −

- Cost and effort estimation
- Productivity measures and model
- Data collection
- Quantity models and measures
- Reliability models
- Performance and evaluation models
- Structural and complexity metrics
- Capability – maturity assessment
- Management by metrics
- Evaluation of methods and tools

Software measurement is a diverse collection of these activities that range from models predicting software project costs at a specific stage to measures of program structure.

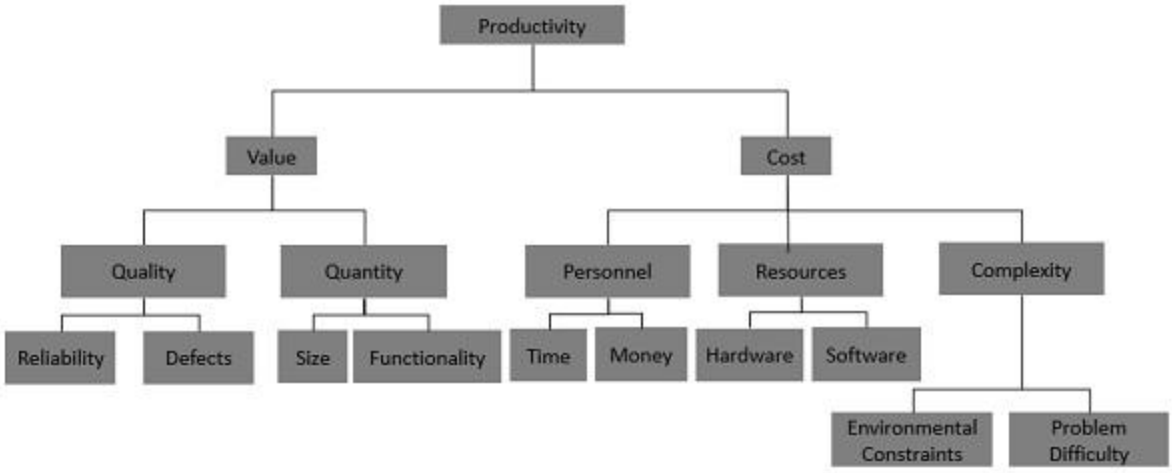
Cost and Effort Estimation

Effort is expressed as a function of one or more variables such as the size of the program, the capability of the developers and the level of reuse. Cost and effort estimation models have been proposed to predict the project cost during early phases in the software life cycle. The different models proposed are −

- Boehm's COCOMO model
- Putnam's slim model
- Albrecht's function point model

Productivity Model and Measures

Productivity can be considered as a function of the value and the cost. Each can be decomposed into different measurable size, functionality, time, money, etc. Different possible components of a productivity model can be expressed in the following diagram.



Data Collection

The quality of any measurement program is clearly dependent on careful data collection. Data collected can be distilled into simple charts and graphs so that the managers can understand the progress and problem of the development. Data collection is also essential for scientific investigation of relationships and trends.

Quality Models and Measures

Quality models have been developed for the measurement of quality of the product without which productivity is meaningless. These quality models can be combined with productivity model for measuring the correct productivity. These models are usually constructed in a tree-like fashion. The upper branches hold important high level quality factors such as reliability and usability.

The notion of divide and conquer approach has been implemented as a standard approach to measuring software quality.

Reliability Models

Most quality models include reliability as a component factor, however, the need to predict and measure reliability has led to a separate specialization in reliability modeling and prediction. The basic problem in reliability theory is to predict when a system will eventually fail.

Performance Evaluation and Models

It includes externally observable system performance characteristics such as response times and completion rates, and the internal working of the system such as the efficiency of algorithms. It is another aspect of quality.

Structural and Complexity Metrics

Here we measure the structural attributes of representations of the software, which are available in advance of execution. Then we try to establish empirically predictive theories to support quality assurance, quality control, and quality prediction.

Capability Maturity Assessment

This model can assess many different attributes of development including the use of tools, standard practices and more. It is based on the key practices that every good contractor should be using.

Management by Metrics

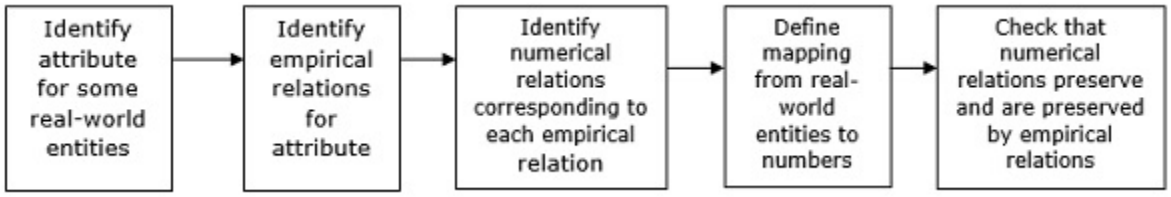
For managing the software project, measurement has a vital role. For checking whether the project is on track, users and developers can rely on the measurement-based chart and graph. The standard set of measurements and reporting methods are especially important when the software is embedded in a product where the customers are not usually well-versed in software terminology.

Evaluation of Methods and Tools

This depends on the experimental design, proper identification of factors likely to affect the outcome and appropriate measurement of factor attributes

Representation Theory of measurement

The **representational** condition asserts that a **measurement** mapping (M) must map entities into numbers, and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by numerical relations.



Representational theory of Measurement

- The data we obtain as measure should represent the attribute of the entities we observe and manipulation of data should preserve relationship that we observe among the entities
- It consists of
  - Empirical Relation
  - Rules of Mapping
  - Representation condition

Empirical relation

- We normally understand things by comparing them instead of assinging them numbers.
- Avinash **is tall** - 'is tall' is the unary relation
- Avinash is **taller than** Sushant.
  - Taller than is the binary relation

Rules of Mapping

- The real world is the domain of mapping and mathematical world is the range.
- When we map the attributes to a mathematical system, we have many choices for the mapping and the range.
  - E.g. To measure person height.

The representation condition

- The representation condition states that a measurement mapping M must map the entities in to numbers ans empirical relations into the numerical relations in such a way that the **empirical relations preserve and are preserved by the numerical relations.**
- For **taller than** in empirical relation is mapped to symbol **>** in numerical relation.

- A is taller than B iff M(A) > M(B).
- This statement implies that
  - When ever A is taller than B then M(A) must be bigger number that M(B)

Measurement and Models

Models are useful for interpreting the behavior of the numerical elements of the real-world entities as well as measuring them. To help the measurement process, the model of the mapping

should also be supplemented with a model of the mapping domain. A model should also specify how these entities are related to the attributes and how the characteristics relate.

Measurement is of two types −

- Direct measurement
- Indirect measurement

Direct Measurement

These are the measurements that can be measured without the involvement of any other entity or attribute.

The following direct measures are commonly used in software engineering.

- Length of source code by LOC
- Duration of testing purpose by elapsed time
- Number of defects discovered during the testing process by counting defects
- The time a programmer spends on a program

Indirect Measurement

These are measurements that can be measured in terms of any other entity or attribute.

The following indirect measures are commonly used in software engineering.

ProgrammerProductivity=LOCproducedPersonmonthsofeffortProgrammerProductivity=LOCproducedPersonmonthsofeffort

ModuleDefectDensity=NumberofdefectsModulesizeModuleDefectDensity=NumberofdefectsModulesize
DefectDetectionEfficiency=NumberofdefectsdetectedTotalnumberofdefectsDefectDetectionEfficiency=NumberofdefectsdetectedTotalnumberofdefects

RequirementStability=NumberofinitialrequirementsTotalnumberofrequirementsRequirementStability=NumberofinitialrequirementsTotalnumberofrequirements
TestEffectivenessRatio=NumberofitemscoveredTotalnumberofitemsTestEffectivenessRatio=NumberofitemscoveredTotalnumberofitems
Systemspoilage=EffortspentforfixingfaultsTotalprojecteffortSystemspoilage=EffortspentforfixingfaultsTotalprojecteffort

Measurement for Prediction

For allocating the appropriate resources to the project, we need to predict the effort, time, and cost for developing the project. The measurement for prediction always requires a mathematical model that relates the attributes to be predicted to some other attribute that we can measure now. Hence, a prediction system consists of a mathematical model together with a set of prediction procedures for determining the unknown parameters and interpreting the results.

Measurement scales and scale types

Measurement scales are the mappings used for representing the empirical relation system. It is mainly of 5 types −

- Nominal Scale
- Ordinal Scale
- Interval Scale
- Ratio Scale
- Absolute Scale

Nominal Scale

It places the elements in a classification scheme. The classes will not be ordered. Each and every entity should be placed in a particular class or category based on the value of the attribute.

It has two major characteristics −

- The empirical relation system consists only of different classes; there is no notion of ordering among the classes.
- Any distinct numbering or symbolic representation of the classes is an acceptable measure, but there is no notion of magnitude associated with the numbers or symbols.

Ordinal Scale

It places the elements in an ordered classification scheme. It has the following characteristics −

- The empirical relation system consists of classes that are ordered with respect to the attribute.
- Any mapping that preserves the ordering is acceptable.
- The numbers represent ranking only. Hence, addition, subtraction, and other arithmetic operations have no meaning.

Interval Scale

This scale captures the information about the size of the intervals that separate the classification. Hence, it is more powerful than the nominal scale and the ordinal scale.

It has the following characteristics −

- It preserves order like the ordinal scale.
- It preserves the differences but not the ratio.
- Addition and subtraction can be performed on this scale but not multiplication or division.

If an attribute is measurable on an interval scale, and **M** and **M'** are mappings that satisfy the representation condition, then we can always find two numbers **a** and **b** such that,

M = aM' + b

Ratio Scale

This is the most useful scale of measurement. Here, an empirical relation exists to capture ratios. It has the following characteristics −

- It is a measurement mapping that preserves ordering, the size of intervals between the entities and the ratio between the entities.
- There is a zero element, representing total lack of the attributes.
- The measurement mapping must start at zero and increase at equal intervals, known as units.
- All arithmetic operations can be applied.

Here, mapping will be of the form

M = aM'

Where **'a'** is a positive scalar.

Absolute Scale

On this scale, there will be only one possible measure for an attribute. Hence, the only possible transformation will be the identity transformation.

It has the following characteristics −

- The measurement is made by counting the number of elements in the entity set.
- The attribute always takes the form "number of occurrences of x in the entity".
- There is only one possible measurement mapping, namely the actual count.
- All arithmetic operations can be performed on the resulting count.

Empirical Investigations involve the scientific investigation of any tool, technique, or method. This investigation mainly contains the following 4 principles.

- Choosing an investigation technique
- Stating the hypothesis
- Maintaining the control over the variable
- Making the investigation meaningful

Choosing an Investigation Technique

The key components of Empirical investigation in software engineering are −

- Survey

- Case study
- Formal experiment

Survey

Survey is the retrospective study of a situation to document relationships and outcomes. It is always done after an event has occurred. For example, in software engineering, polls can be performed to determine how the users reacted to a particular method, tool, or technique to determine trends or relationships.

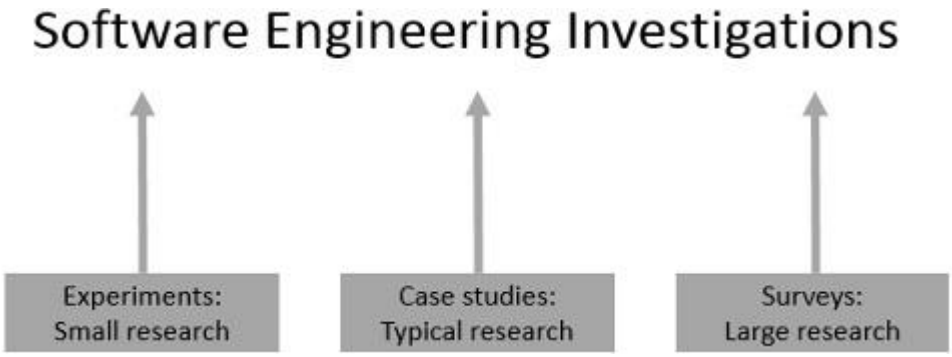
In this case, we have no control over the situation at hand. We can record a situation and compare it with a similar one.

Case Study

It is a research technique where you identify the key factors that may affect the outcome of an activity and then document the activity: its inputs, constraints, resources, and outputs.

Formal Experiment

It is a rigorous controlled investigation of an activity, where the key factors are identified and manipulated to document their effects on the outcome.



A particular investigation method can be chosen according to the following guidelines −

- If the activity has already occurred, we can perform survey or case study. If it is yet to occur, then case study or formal experiment may be chosen.
- If we have a high level of control over the variables that can affect the outcome, then we can use an experiment. If we have no control over the variable, then case study will be a preferred technique.
- If replication is not possible at higher levels, then experiment is not possible.
- If the cost of replication is low, then we can consider experiment.

Stating the Hypothesis

To boost the decision of a particular investigation technique, the goal of the research should be expressed as a hypothesis we want to test. The hypothesis is the tentative theory or supposition that the programmer thinks explains the behavior they want to explore.

Maintaining Control over Variables

After stating the hypothesis, next we have to decide the different variables that affect its truth as well as how much control we have over it. This is essential because the key discriminator between the experiment and the case studies is the degree of control over the variable that affects the behavior.

A state variable which is the factor that can characterize the project and can also influence the evaluation results is used to distinguish the control situation from the experimental one in the formal experiment. If we cannot differentiate control from experiment, case study technique will be a preferred one.

For example, if we want to determine whether a change in the programming language can affect the productivity of the project, then the language will be a state variable. Suppose we are currently using FORTRAN which we want to replace by Ada. Then FORTRAN will be the control language and Ada to be the experimental one.

Making the Investigation Meaningful

The results of an experiment are usually more generalizable than case study or survey. The results of the case study or survey can normally be applicable only to a particular organization. Following points prove the efficiency of these techniques to answer a variety of questions.

Conforming theories and conventional wisdom

Case studies or surveys can be used to conform the effectiveness and utility of the conventional wisdom and many other standards, methods, or tools in a single organization. However, formal experiment can investigate the situations in which the claims are generally true.

Exploring relationships

The relationship among various attributes of resources and software products can be suggested by a case study or survey.

For example, a survey of completed projects can reveal that a software written in a particular language has fewer faults than a software written in other languages.

Understanding and verifying these relationships is essential to the success of any future projects. Each of these relationships can be expressed as a hypothesis and a formal experiment can be designed to test the degree to which the relationships hold. Usually, the value of one particular attribute is observed by keeping other attributes constant or under control.

Evaluating the accuracy of models

Models are usually used to predict the outcome of an activity or to guide the use of a method or tool. It presents a particularly difficult problem when designing an experiment or case study, because their predictions often affect the outcome. The project managers often turn the predictions into targets for completion. This effect is common when the cost and schedule models are used.

Some models such as reliability models do not influence the outcome, since reliability measured as mean time to failure cannot be evaluated until the software is ready for use in the field.

Validating measures

There are many software measures to capture the value of an attribute. Hence, a study must be conducted to test whether a given measure reflects the changes in the attribute it is supposed to capture. Validation is performed by correlating one measure with another. A second measure which is also a direct and valid measure of the affecting factor should be used to validate. Such measures are not always available or easy to measure. Also, the measures used must conform to human notions of the factor being measured.

Meaning filialness in measurement

The framework for software measurement is based on three principles −

- Classifying the entities to be examined
- Determining relevant measurement goals
- Identifying the level of maturity that the organization has reached

Classifying the Entities to be Examined

In software engineering, mainly three classes of entities exist. They are −

- Processes
- Products
- Resources

All of these entities have internal as well as external entities.

- **Internal attributes** are those that can be measured purely in terms of the process, product, or resources itself. For example: Size, complexity, dependency among modules.
- **External attributes** are those that can be measured only with respect to its relation with the environment. For example: The total number of failures experienced by a user, the length of time it takes to search the database and retrieve information.

The different attributes that can be measured for each of the entities are as follows −

Processes

Processes are collections of software-related activities. Following are some of the internal attributes that can be measured directly for a process −

- The duration of the process or one of its activities
- The effort associated with the process or one of its activities
- The number of incidents of a specified type arising during the process or one of its activities

The different external attributes of a process are cost, controllability, effectiveness, quality and stability.

Products

Products are not only the items that the management is committed to deliver but also any artifact or document produced during the software life cycle.

The different internal product attributes are size, effort, cost, specification, length, functionality, modularity, reuse, redundancy, and syntactic correctness. Among these size, effort, and cost are relatively easy to measure than the others.

The different external product attributes are usability, integrity, efficiency, testability, reusability, portability, and interoperability. These attributes describe not only the code but also the other documents that support the development effort.

Resources

These are entities required by a process activity. It can be any input for the software production. It includes personnel, materials, tools and methods.

The different internal attributes for the resources are age, price, size, speed, memory size, temperature, etc. The different external attributes are productivity, experience, quality, usability, reliability, comfort etc.

Determining Relevant Measurement Goals

A particular measurement will be useful only if it helps to understand the process or one of its resultant products. The improvement in the process or products can be performed only when the project has clearly defined goals for processes and products. A clear understanding of goals can be used to generate suggested metrics for a given project in the context of a process maturity framework.

The Goal–Question–Metric (GQM) paradigm

The GQM approach provides a framework involving the following three steps −

- Listing the major goals of the development or maintenance project
- Deriving the questions from each goal that must be answered to determine if the goals are being met
- Decide what must be measured in order to be able to answer the questions adequately

To use GQM paradigm, first we express the overall goals of the organization. Then, we generate the questions such that the answers are known so that we can determine whether the goals are being met. Later, analyze each question in terms of what measurement we need in order to answer each question.

Typical goals are expressed in terms of productivity, quality, risk, customer satisfaction, etc. Goals and questions are to be constructed in terms of their audience.

To help generate the goals, questions, and metrics, Basili & Rombach provided a series of templates.

- **Purpose** − To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to understand, assess, manage, engineer, learn, improve, etc. **Example**: To characterize the product in order to learn it.
- **Perspective** − Examine the (cost, effectiveness, correctness, defects, changes, product measures, etc.) from the viewpoint of the developer, manager, customer, etc. **Example**: Examine the defects from the viewpoint of the customer.
- **Environment** − The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc. **Example**: The customers of this software are those who have no knowledge about the tools.

Measurement and Process Improvement

Normally measurement is useful for −

- Understanding the process and products
- Establishing a baseline
- Accessing and predicting the outcome

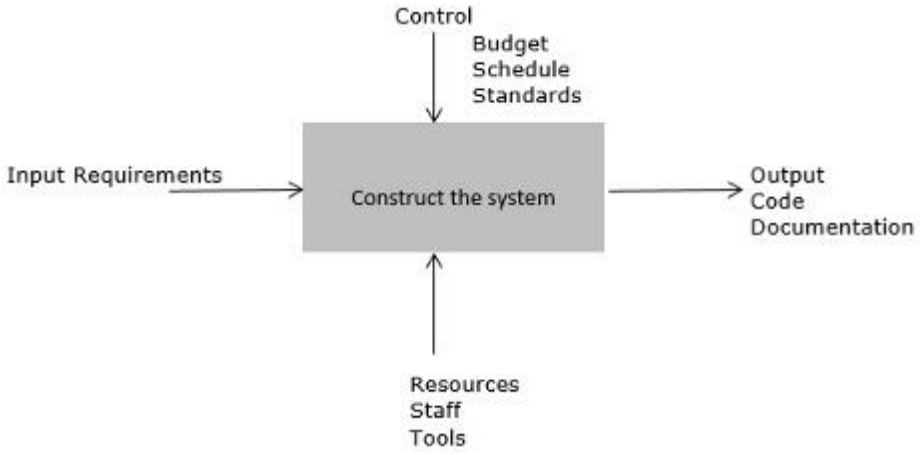
According to the maturity level of the process given by SEI, the type of measurement and the measurement program will be different. Following are the different measurement programs that can be applied at each of the maturity level.

Level 1: Ad hoc

At this level, the inputs are ill- defined, while the outputs are expected. The transition from input to output is undefined and uncontrolled. For this level of process maturity, baseline measurements are needed to provide a starting point for measuring.

Level 2: Repeatable

At this level, the inputs and outputs of the process, constraints, and resources are identifiable. A repeatable process can be described by the following diagram.

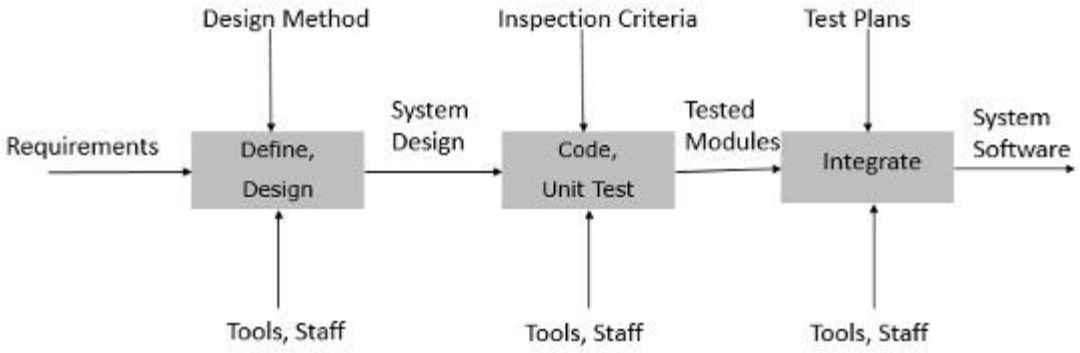


The input measures can be the size and volatility of the requirements. The output may be measured in terms of system size, the resources in terms of staff effort, and the constraints in terms of cost and schedule.

Level 3: Defined

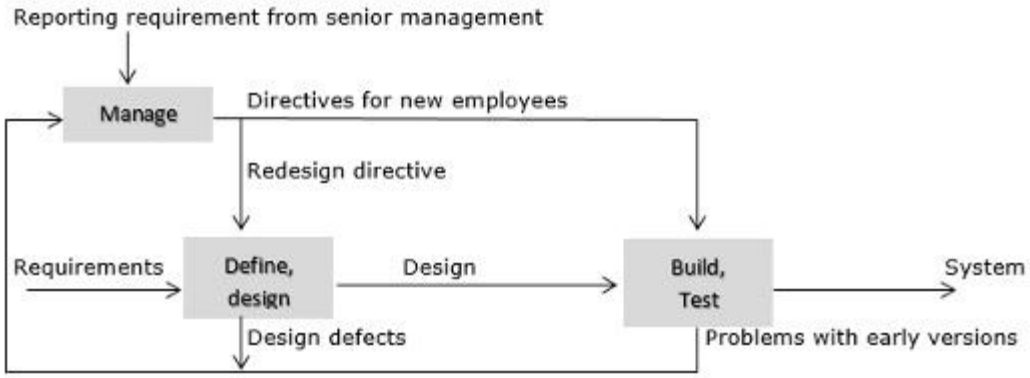
At this level, intermediate activities are defined, and their inputs and outputs are known and understood. A simple example of the defined process is described in the following figure.

The input to and the output from the intermediate activities can be examined, measured, and assessed.



Level 4: Managed

At this level, the feedback from the early project activities can be used to set priorities for the current activities and later for the project activities. We can measure the effectiveness of the process activities. The measurement reflects the characteristics of the overall process and of the interaction among and across major activities.



Level 5: Optimizing

At this level, the measures from activities are used to improve the process by removing and adding process activities and changing the process structure dynamically in response to measurement feedback. Thus, the process change can affect the organization and the project as well as the process. The process will act as sensors and monitors, and we can change the process significantly in response to warning signs.

At a given maturity level, we can collect the measurements for that level and all levels below it.

Identifying the Level of Maturity

Process maturity suggests to measure only what is visible. Thus, the combination of process maturity with GQM will provide most useful measures.

- At **level 1**, the project is likely to have ill-defined requirements. At this level, the measurement of requirement characteristics is difficult.
- At **level 2**, the requirements are well-defined and the additional information such as the type of each requirement and the number of changes to each type can be collected.
- At **level 3**, intermediate activities are defined with entry and exit criteria for each activity

The goal and question analysis will be the same, but the metric will vary with maturity. The more mature the process, the richer will be the measurements. The GQM paradigm, in concert with the process maturity, has been used as the basis for several tools that assist managers in designing measurement programs.

GQM helps to understand the need for measuring the attribute, and process maturity suggests whether we are capable of measuring it in a meaningful way. Together they provide a context for measurement.

MODULE –II

Goal-Based Framework for Software Measurement

The framework for software measurement is based on three principles −

- Classifying the entities to be examined
- Determining relevant measurement goals
- Identifying the level of maturity that the organization has reached

Classifying the Entities to be Examined

In software engineering, mainly three classes of entities exist. They are −

- Processes
- Products
- Resources

All of these entities have internal as well as external entities.

- **Internal attributes** are those that can be measured purely in terms of the process, product, or resources itself. For example: Size, complexity, dependency among modules.

- **External attributes** are those that can be measured only with respect to its relation with the environment. For example: The total number of failures experienced by a user, the length of time it takes to search the database and retrieve information.

The different attributes that can be measured for each of the entities are as follows −

Processes

Processes are collections of software-related activities. Following are some of the internal attributes that can be measured directly for a process −

- The duration of the process or one of its activities
- The effort associated with the process or one of its activities
- The number of incidents of a specified type arising during the process or one of its activities

The different external attributes of a process are cost, controllability, effectiveness, quality and stability.

Products

Products are not only the items that the management is committed to deliver but also any artifact or document produced during the software life cycle.

The different internal product attributes are size, effort, cost, specification, length, functionality, modularity, reuse, redundancy, and syntactic correctness. Among these size, effort, and cost are relatively easy to measure than the others.

The different external product attributes are usability, integrity, efficiency, testability, reusability, portability, and interoperability. These attributes describe not only the code but also the other documents that support the development effort.

Resources

These are entities required by a process activity. It can be any input for the software production. It includes personnel, materials, tools and methods.

The different internal attributes for the resources are age, price, size, speed, memory size, temperature, etc. The different external attributes are productivity, experience, quality, usability, reliability, comfort etc.

Determining Relevant Measurement Goals

A particular measurement will be useful only if it helps to understand the process or one of its resultant products. The improvement in the process or products can be performed only when the project has clearly defined goals for processes and products. A clear understanding of goals can be used to generate suggested metrics for a given project in the context of a process maturity framework.

The Goal–Question–Metric (GQM) paradigm

The GQM approach provides a framework involving the following three steps −

- Listing the major goals of the development or maintenance project
- Deriving the questions from each goal that must be answered to determine if the goals are being met
- Decide what must be measured in order to be able to answer the questions adequately

To use GQM paradigm, first we express the overall goals of the organization. Then, we generate the questions such that the answers are known so that we can determine whether the goals are being met. Later, analyze each question in terms of what measurement we need in order to answer each question.

Typical goals are expressed in terms of productivity, quality, risk, customer satisfaction, etc. Goals and questions are to be constructed in terms of their audience.

To help generate the goals, questions, and metrics, Basili & Rombach provided a series of templates.

- **Purpose** − To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to understand, assess, manage, engineer, learn, improve, etc. **Example**: To characterize the product in order to learn it.
- **Perspective** − Examine the (cost, effectiveness, correctness, defects, changes, product measures, etc.) from the viewpoint of the developer, manager, customer, etc. **Example**: Examine the defects from the viewpoint of the customer.
- **Environment** − The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc. **Example**: The customers of this software are those who have no knowledge about the tools.

Classifying software measures

The framework for software measurement is based on three principles −

- Classifying the entities to be examined
- Determining relevant measurement goals
- Identifying the level of maturity that the organization has reached

Classifying the Entities to be Examined

In software engineering, mainly three classes of entities exist. They are −

- Processes
- Products
- Resources

All of these entities have internal as well as external entities.

- **Internal attributes** are those that can be measured purely in terms of the process, product, or resources itself. For example: Size, complexity, dependency among modules.

- **External attributes** are those that can be measured only with respect to its relation with the environment. For example: The total number of failures experienced by a user, the length of time it takes to search the database and retrieve information.

The different attributes that can be measured for each of the entities are as follows −

Processes

Processes are collections of software-related activities. Following are some of the internal attributes that can be measured directly for a process −

- The duration of the process or one of its activities
- The effort associated with the process or one of its activities
- The number of incidents of a specified type arising during the process or one of its activities

The different external attributes of a process are cost, controllability, effectiveness, quality and stability.

Products

Products are not only the items that the management is committed to deliver but also any artifact or document produced during the software life cycle.

The different internal product attributes are size, effort, cost, specification, length, functionality, modularity, reuse, redundancy, and syntactic correctness. Among these size, effort, and cost are relatively easy to measure than the others.

The different external product attributes are usability, integrity, efficiency, testability, reusability, portability, and interoperability. These attributes describe not only the code but also the other documents that support the development effort.

Resources

These are entities required by a process activity. It can be any input for the software production. It includes personnel, materials, tools and methods.

The different internal attributes for the resources are age, price, size, speed, memory size, temperature, etc. The different external attributes are productivity, experience, quality, usability, reliability, comfort etc.

Determining Relevant Measurement Goals

A particular measurement will be useful only if it helps to understand the process or one of its resultant products. The improvement in the process or products can be performed only when the project has clearly defined goals for processes and products. A clear understanding of goals can be used to generate suggested metrics for a given project in the context of a process maturity framework.

The Goal–Question–Metric (GQM) paradigm

The GQM approach provides a framework involving the following three steps −

- Listing the major goals of the development or maintenance project
- Deriving the questions from each goal that must be answered to determine if the goals are being met
- Decide what must be measured in order to be able to answer the questions adequately

To use GQM paradigm, first we express the overall goals of the organization. Then, we generate the questions such that the answers are known so that we can determine whether the goals are being met. Later, analyze each question in terms of what measurement we need in order to answer each question.

Typical goals are expressed in terms of productivity, quality, risk, customer satisfaction, etc. Goals and questions are to be constructed in terms of their audience.

To help generate the goals, questions, and metrics, Basili & Rombach provided a series of templates.

- **Purpose** − To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to understand, assess, manage, engineer, learn, improve, etc. **Example**: To characterize the product in order to learn it.
- **Perspective** − Examine the (cost, effectiveness, correctness, defects, changes, product measures, etc.) from the viewpoint of the developer, manager, customer, etc. **Example**: Examine the defects from the viewpoint of the customer.
- **Environment** − The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc. **Example**: The customers of this software are those who have no knowledge about the tools.

Measurement and Process Improvement

Normally measurement is useful for −

- Understanding the process and products
- Establishing a baseline
- Accessing and predicting the outcome

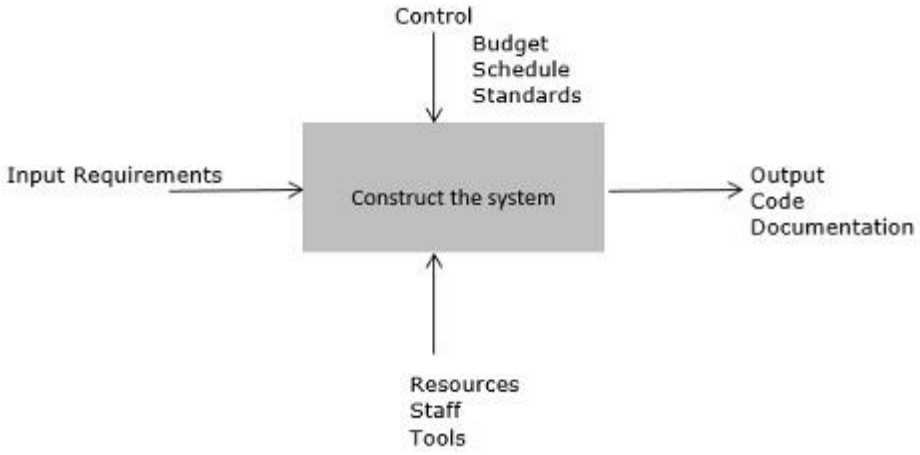
According to the maturity level of the process given by SEI, the type of measurement and the measurement program will be different. Following are the different measurement programs that can be applied at each of the maturity level.

Level 1: Ad hoc

At this level, the inputs are ill- defined, while the outputs are expected. The transition from input to output is undefined and uncontrolled. For this level of process maturity, baseline measurements are needed to provide a starting point for measuring.

Level 2: Repeatable

At this level, the inputs and outputs of the process, constraints, and resources are identifiable. A repeatable process can be described by the following diagram.

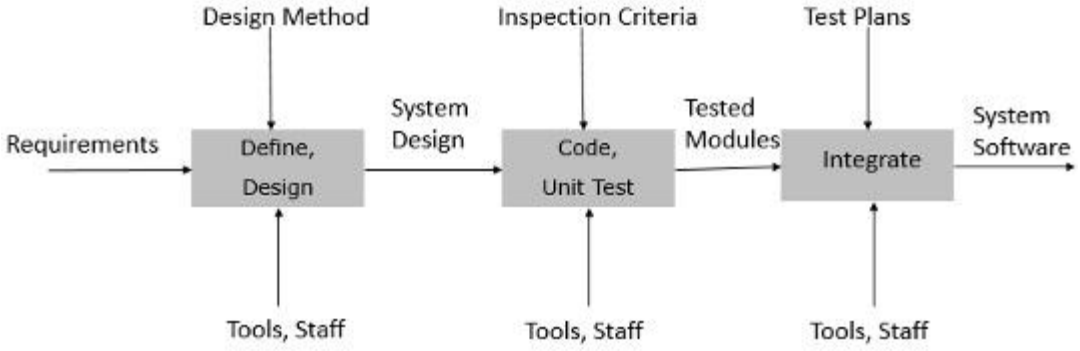


The input measures can be the size and volatility of the requirements. The output may be measured in terms of system size, the resources in terms of staff effort, and the constraints in terms of cost and schedule.

Level 3: Defined

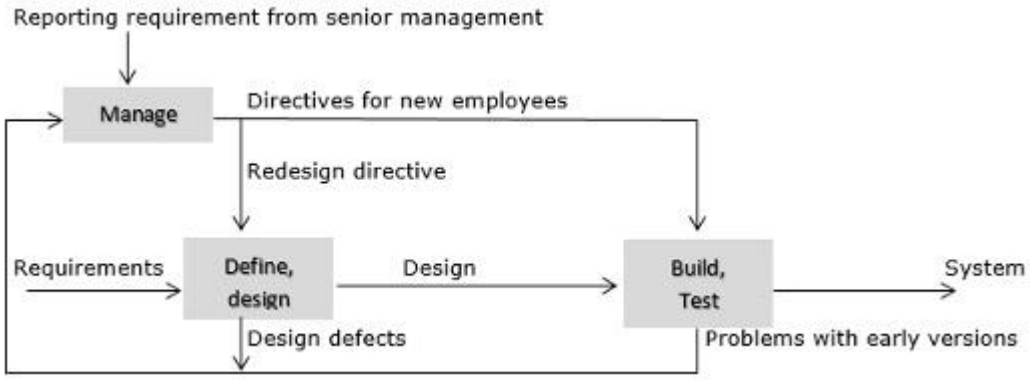
At this level, intermediate activities are defined, and their inputs and outputs are known and understood. A simple example of the defined process is described in the following figure.

The input to and the output from the intermediate activities can be examined, measured, and assessed.



Level 4: Managed

At this level, the feedback from the early project activities can be used to set priorities for the current activities and later for the project activities. We can measure the effectiveness of the process activities. The measurement reflects the characteristics of the overall process and of the interaction among and across major activities.



Level 5: Optimizing

At this level, the measures from activities are used to improve the process by removing and adding process activities and changing the process structure dynamically in response to measurement feedback. Thus, the process change can affect the organization and the project as well as the process. The process will act as sensors and monitors, and we can change the process significantly in response to warning signs.

At a given maturity level, we can collect the measurements for that level and all levels below it.

Identifying the Level of Maturity

Process maturity suggests to measure only what is visible. Thus, the combination of process maturity with GQM will provide most useful measures.

- At **level 1**, the project is likely to have ill-defined requirements. At this level, the measurement of requirement characteristics is difficult.
- At **level 2**, the requirements are well-defined and the additional information such as the type of each requirement and the number of changes to each type can be collected.
- At **level 3**, intermediate activities are defined with entry and exit criteria for each activity

The goal and question analysis will be the same, but the metric will vary with maturity. The more mature the process, the richer will be the measurements. The GQM paradigm, in concert with the process maturity, has been used as the basis for several tools that assist managers in designing measurement programs.

GQM helps to understand the need for measuring the attribute, and process maturity suggests whether we are capable of measuring it in a meaningful way. Together they provide a context for measurement.

Software metrics is a standard of measure that contains many activities which involve some degree of measurement. It can be classified into three categories: product metrics, process metrics, and project metrics.

- **Product metrics** describe the characteristics of the product such as size, complexity, design features, performance, and quality level.

- **Process metrics** can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process.
- **Project 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.

Scope of Software Metrics

Software metrics contains many activities which include the following −

- Cost and effort estimation
- Productivity measures and model
- Data collection
- Quantity models and measures
- Reliability models
- Performance and evaluation models
- Structural and complexity metrics
- Capability – maturity assessment
- Management by metrics
- Evaluation of methods and tools

Software measurement is a diverse collection of these activities that range from models predicting software project costs at a specific stage to measures of program structure.

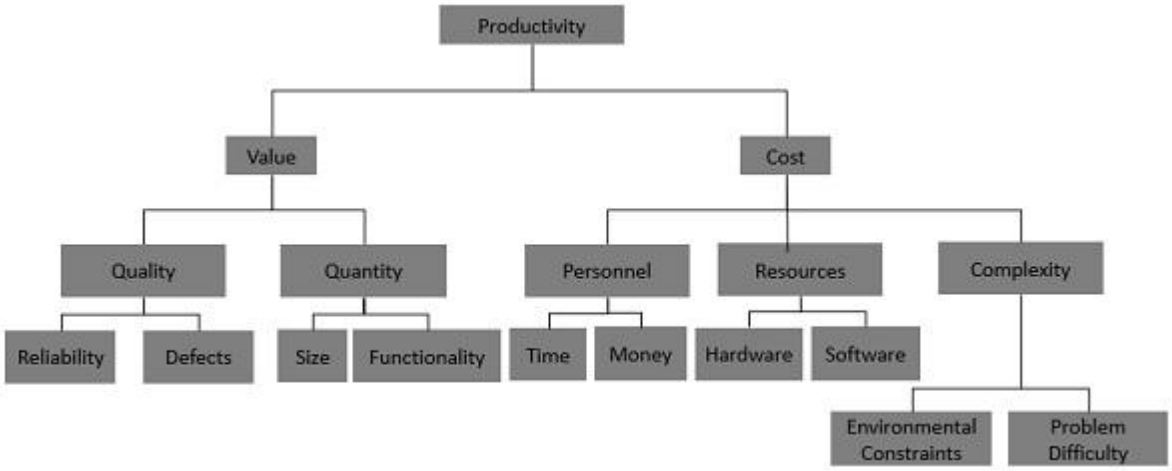
Cost and Effort Estimation

Effort is expressed as a function of one or more variables such as the size of the program, the capability of the developers and the level of reuse. Cost and effort estimation models have been proposed to predict the project cost during early phases in the software life cycle. The different models proposed are −

- Boehm's COCOMO model
- Putnam's slim model
- Albrecht's function point model

Productivity Model and Measures

Productivity can be considered as a function of the value and the cost. Each can be decomposed into different measurable size, functionality, time, money, etc. Different possible components of a productivity model can be expressed in the following diagram.



Data Collection

The quality of any measurement program is clearly dependent on careful data collection. Data collected can be distilled into simple charts and graphs so that the managers can understand the progress and problem of the development. Data collection is also essential for scientific investigation of relationships and trends.

Quality Models and Measures

Quality models have been developed for the measurement of quality of the product without which productivity is meaningless. These quality models can be combined with productivity model for measuring the correct productivity. These models are usually constructed in a tree-like fashion. The upper branches hold important high level quality factors such as reliability and usability.

The notion of divide and conquer approach has been implemented as a standard approach to measuring software quality.

Reliability Models

Most quality models include reliability as a component factor, however, the need to predict and measure reliability has led to a separate specialization in reliability modeling and prediction. The basic problem in reliability theory is to predict when a system will eventually fail.

Performance Evaluation and Models

It includes externally observable system performance characteristics such as response times and completion rates, and the internal working of the system such as the efficiency of algorithms. It is another aspect of quality.

Structural and Complexity Metrics

Here we measure the structural attributes of representations of the software, which are available in advance of execution. Then we try to establish empirically predictive theories to support quality assurance, quality control, and quality prediction.

Capability Maturity Assessment

This model can assess many different attributes of development including the use of tools, standard practices and more. It is based on the key practices that every good contractor should be using.

Management by Metrics

For managing the software project, measurement has a vital role. For checking whether the project is on track, users and developers can rely on the measurement-based chart and graph. The standard set of measurements and reporting methods are especially important when the software is embedded in a product where the customers are not usually well-versed in software terminology.

Evaluation of Methods and Tools

This depends on the experimental design, proper identification of factors likely to affect the outcome and appropriate measurement of factor attributes.

determining what to measure,

A set of metrics include the following methods:

algorithms analysis;

number of code lines;

the complexity of a **software**;

functional points analysis;

number of bugs per 1000 code lines;

level of testing;

number of classes and interfaces;

cohesion, etc.

Software Testing Metrics: What is, Types & Example

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.

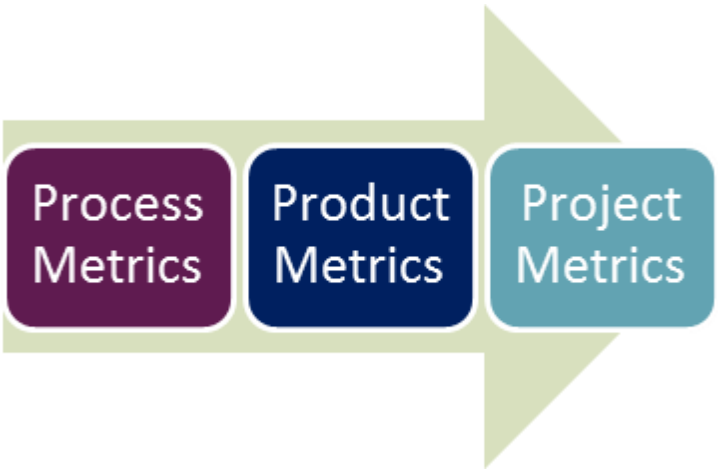
The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.

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.

Example for software test measurement: Total number of defects

Types of Test Metrics



- **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** It deals with the quality of the software product
- **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

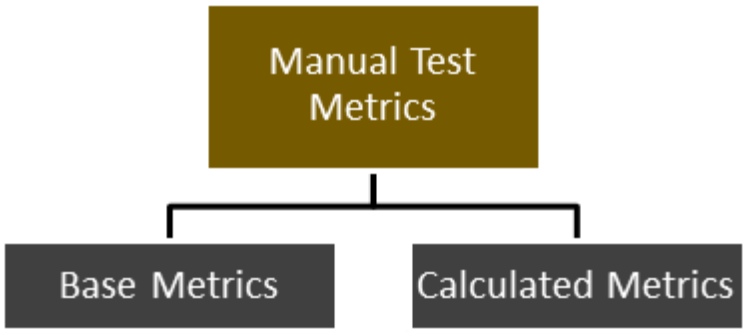
Identification of correct testing metrics is very important. Few things need to be considered before identifying the test metrics

- Fix the target audience for the metric preparation
- Define the goal for metrics
- Introduce all the relevant metrics based on project needs
- Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results in the maximum output

Manual Test Metrics

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

- **Base Metrics**
- **Calculated Metrics**



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

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 capture 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 representativ • 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 execute
3	Determination of the information to be followed, a frequency of tracking and the person responsible	• The actual test execution per day will be captu 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 s need to investigate the reason and suggest the 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

Applying the framework, Software measurement validation

Validating the measurement of software system involves two steps −

- Validating the measurement systems
- Validating the prediction systems

Validating the Measurement Systems

Measures or measurement systems are used to asses an existing entity by numerically characterizing one or more of its attributes. A measure is valid if it accurately characterizes the attribute it claims to measure.

Validating a software measurement system is the process of ensuring that the measure is a proper numerical characterization of the claimed attribute by showing that the representation condition is satisfied.

For validating a measurement system, we need both a formal model that describes entities and a numerical mapping that preserves the attribute that we are measuring. For example, if there are two programs P1 and P2, and we want to concatenate those programs, then we expect that any measure **m** of length to satisfy that,

m(P1+P2) = m(P1) + m(P2)

If a program **P1** has more length than program **P2**, then any measure **m** should also satisfy,

m(P1) > m(P2)

The length of the program can be measured by counting the lines of code. If this count satisfies the above relationships, we can say that the lines of code are a valid measure of the length.

The formal requirement for validating a measure involves demonstrating that it characterizes the stated attribute in the sense of measurement theory. Validation can be used to make sure that the measurers are defined properly and are consistent with the entity's real world behavior.

Validating the Prediction Systems

Prediction systems are used to predict some attribute of a future entity involving a mathematical model with associated prediction procedures.

Validating prediction systems in a given environment is the process of establishing the accuracy of the prediction system by empirical means, i.e. by comparing the model performance with known data in the given environment. It involves experimentation and hypothesis testing.

The degree of accuracy acceptable for validation depends upon whether the prediction system is deterministic or stochastic as well as the person doing the assessment. Some stochastic prediction systems are more stochastic than others.

Examples of stochastic prediction systems are systems such as software cost estimation, effort estimation, schedule estimation, etc. Hence, to validate a prediction system formally, we must decide how stochastic it is, then compare the performance of the prediction system with known data.

Scope of Software Metrics

Software metrics contains many activities which include the following −

- Cost and effort estimation

- Productivity measures and model
- Data collection
- Quantity models and measures
- Reliability models
- Performance and evaluation models
- Structural and complexity metrics
- Capability – maturity assessment
- Management by metrics
- Evaluation of methods and tools

Software measurement is a diverse collection of these activities that range from models predicting software project costs at a specific stage to measures of program structure.

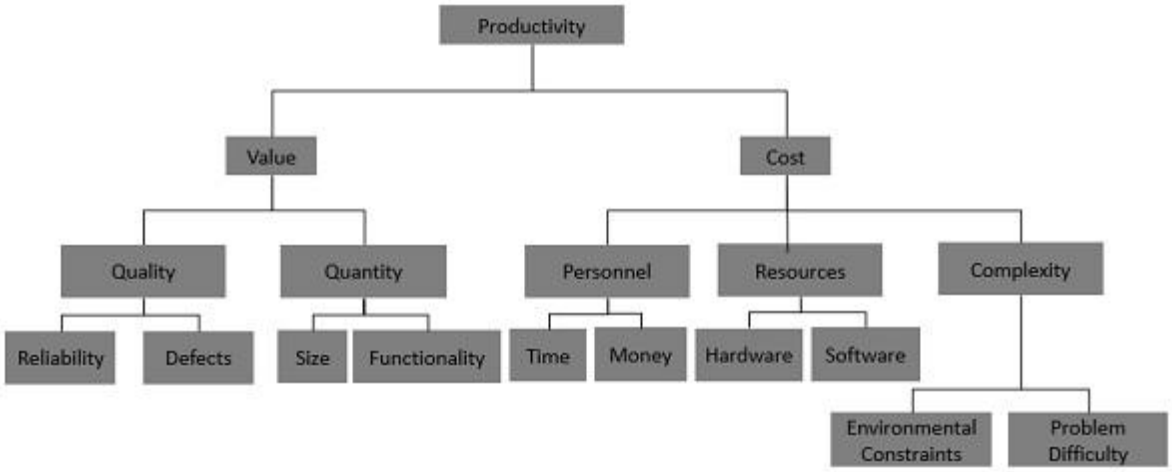
Cost and Effort Estimation

Effort is expressed as a function of one or more variables such as the size of the program, the capability of the developers and the level of reuse. Cost and effort estimation models have been proposed to predict the project cost during early phases in the software life cycle. The different models proposed are −

- Boehm's COCOMO model
- Putnam's slim model
- Albrecht's function point model

Productivity Model and Measures

Productivity can be considered as a function of the value and the cost. Each can be decomposed into different measurable size, functionality, time, money, etc. Different possible components of a productivity model can be expressed in the following diagram.



Data Collection

The quality of any measurement program is clearly dependent on careful data collection. Data collected can be distilled into simple charts and graphs so that the managers can understand the progress and problem of the development. Data collection is also essential for scientific investigation of relationships and trends.

Quality Models and Measures

Quality models have been developed for the measurement of quality of the product without which productivity is meaningless. These quality models can be combined with productivity model for measuring the correct productivity. These models are usually constructed in a tree-like fashion. The upper branches hold important high level quality factors such as reliability and usability.

The notion of divide and conquer approach has been implemented as a standard approach to measuring software quality.

Reliability Models

Most quality models include reliability as a component factor, however, the need to predict and measure reliability has led to a separate specialization in reliability modeling and prediction. The basic problem in reliability theory is to predict when a system will eventually fail.

Performance Evaluation and Models

It includes externally observable system performance characteristics such as response times and completion rates, and the internal working of the system such as the efficiency of algorithms. It is another aspect of quality.

Structural and Complexity Metrics

Here we measure the structural attributes of representations of the software, which are available in advance of execution. Then we try to establish empirically predictive theories to support quality assurance, quality control, and quality prediction.

Capability Maturity Assessment

This model can assess many different attributes of development including the use of tools, standard practices and more. It is based on the key practices that every good contractor should be using.

Management by Metrics

For managing the software project, measurement has a vital role. For checking whether the project is on track, users and developers can rely on the measurement-based chart and graph. The standard set of measurements and reporting methods are especially important when the software is embedded in a product where the customers are not usually well-versed in software terminology.

Evaluation of Methods and Tools

This depends on the experimental design, proper identification of factors likely to affect the outcome and appropriate measurement of factor attributes.

Software measurement validation in practice.

alidating the measurement of software system involves two steps −

- Validating the measurement systems
- Validating the prediction systems

Validating the Measurement Systems

Measures or measurement systems are used to asses an existing entity by numerically characterizing one or more of its attributes. A measure is valid if it accurately characterizes the attribute it claims to measure.

Validating a software measurement system is the process of ensuring that the measure is a proper numerical characterization of the claimed attribute by showing that the representation condition is satisfied.

For validating a measurement system, we need both a formal model that describes entities and a numerical mapping that preserves the attribute that we are measuring. For example, if there are two programs P1 and P2, and we want to concatenate those programs, then we expect that any measure **m** of length to satisfy that,

m(P1+P2) = m(P1) + m(P2)

If a program **P1** has more length than program **P2**, then any measure **m** should also satisfy,

m(P1) > m(P2)

The length of the program can be measured by counting the lines of code. If this count satisfies the above relationships, we can say that the lines of code are a valid measure of the length.

The formal requirement for validating a measure involves demonstrating that it characterizes the stated attribute in the sense of measurement theory. Validation can be used to make sure that the measurers are defined properly and are consistent with the entity's real world behavior.

Validating the Prediction Systems

Prediction systems are used to predict some attribute of a future entity involving a mathematical model with associated prediction procedures.

Validating prediction systems in a given environment is the process of establishing the accuracy of the prediction system by empirical means, i.e. by comparing the model performance with known data in the given environment. It involves experimentation and hypothesis testing.

The degree of accuracy acceptable for validation depends upon whether the prediction system is deterministic or stochastic as well as the person doing the assessment. Some stochastic prediction systems are more stochastic than others.

Examples of stochastic prediction systems are systems such as software cost estimation, effort estimation, schedule estimation, etc. Hence, to validate a prediction system formally, we must decide how stochastic it is, then compare the performance of the prediction system with known data.

Four principles of investigation

Empirical Investigations involve the scientific investigation of any tool, technique, or method. This investigation mainly contains the following 4 principles.

- Choosing an investigation technique
- Stating the hypothesis
- Maintaining the control over the variable
- Making the investigation meaningful

Choosing an Investigation Technique

The key components of Empirical investigation in software engineering are −

- Survey
- Case study
- Formal experiment

Survey

Survey is the retrospective study of a situation to document relationships and outcomes. It is always done after an event has occurred. For example, in software engineering, polls can be performed to determine how the users reacted to a particular method, tool, or technique to determine trends or relationships.

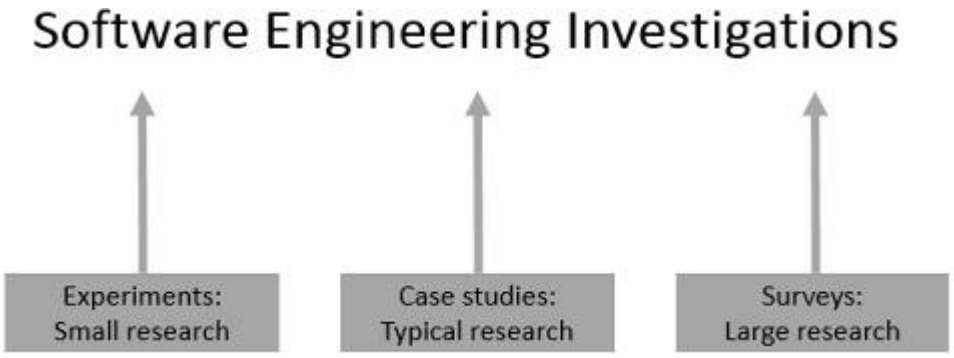
In this case, we have no control over the situation at hand. We can record a situation and compare it with a similar one.

Case Study

It is a research technique where you identify the key factors that may affect the outcome of an activity and then document the activity: its inputs, constraints, resources, and outputs.

Formal Experiment

It is a rigorous controlled investigation of an activity, where the key factors are identified and manipulated to document their effects on the outcome.



A particular investigation method can be chosen according to the following guidelines −

- If the activity has already occurred, we can perform survey or case study. If it is yet to occur, then case study or formal experiment may be chosen.
- If we have a high level of control over the variables that can affect the outcome, then we can use an experiment. If we have no control over the variable, then case study will be a preferred technique.
- If replication is not possible at higher levels, then experiment is not possible.
- If the cost of replication is low, then we can consider experiment.

Stating the Hypothesis

To boost the decision of a particular investigation technique, the goal of the research should be expressed as a hypothesis we want to test. The hypothesis is the tentative theory or supposition that the programmer thinks explains the behavior they want to explore.

Maintaining Control over Variables

After stating the hypothesis, next we have to decide the different variables that affect its truth as well as how much control we have over it. This is essential because the key discriminator between the experiment and the case studies is the degree of control over the variable that affects the behavior.

A state variable which is the factor that can characterize the project and can also influence the evaluation results is used to distinguish the control situation from the experimental one in the formal experiment. If we cannot differentiate control from experiment, case study technique will be a preferred one.

For example, if we want to determine whether a change in the programming language can affect the productivity of the project, then the language will be a state variable. Suppose we are currently using FORTRAN which we want to replace by Ada. Then FORTRAN will be the control language and Ada to be the experimental one.

Making the Investigation Meaningful

The results of an experiment are usually more generalizable than case study or survey. The results of the case study or survey can normally be applicable only to a particular organization. Following points prove the efficiency of these techniques to answer a variety of questions.

Conforming theories and conventional wisdom

Case studies or surveys can be used to conform the effectiveness and utility of the conventional wisdom and many other standards, methods, or tools in a single organization. However, formal experiment can investigate the situations in which the claims are generally true.

Exploring relationships

The relationship among various attributes of resources and software products can be suggested by a case study or survey.

For example, a survey of completed projects can reveal that a software written in a particular language has fewer faults than a software written in other languages.

Understanding and verifying these relationships is essential to the success of any future projects. Each of these relationships can be expressed as a hypothesis and a formal experiment can be designed to test the degree to which the relationships hold. Usually, the value of one particular attribute is observed by keeping other attributes constant or under control.

Evaluating the accuracy of models

Models are usually used to predict the outcome of an activity or to guide the use of a method or tool. It presents a particularly difficult problem when designing an experiment or case study, because their predictions often affect the outcome. The project managers often turn the predictions into targets for completion. This effect is common when the cost and schedule models are used.

Some models such as reliability models do not influence the outcome, since reliability measured as mean time to failure cannot be evaluated until the software is ready for use in the field.

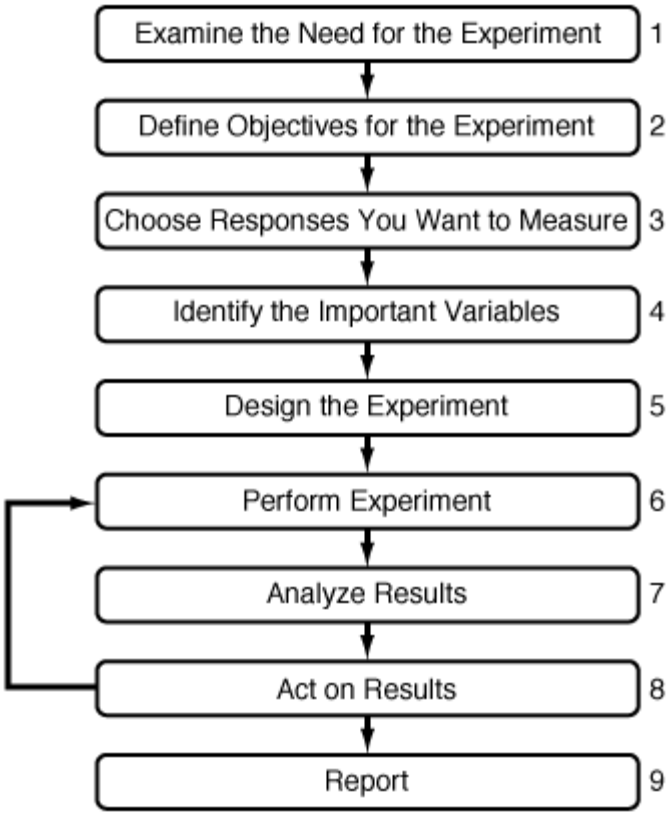
Validating measures

There are many software measures to capture the value of an attribute. Hence, a study must be conducted to test whether a given measure reflects the changes in the attribute it is supposed to capture. Validation is performed by correlating one measure with another. A second measure which is also a direct and valid measure of the affecting factor should be used to validate. Such measures are not always available or easy to measure. Also, the measures used must conform to human notions of the factor being measured.

planning formal experiments

A **formal experiment** has several fixed parts, regardless of what the **experiment** is about. First the experimenter must create a hypothesis, which is a testable theory from which

the **experiment** is derived. ... The treatment is the single variable or factor that the experimenter wishes to test



It can be classified into three categories: product **metrics**, process **metrics**, and project **metrics**. Product **metrics** describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process **metrics** can be used to improve **software** development and maintenance.

Software process and project metrics are quantitative measures that enable **software** engineers to gain insight into the efficiency of the **software process** and the **projects** conducted using the **process** framework. In **software project** management, we are primarily concerned with productivity and quality **metrics**

planning case studies.

It can be classified into three categories: product metrics, **process metrics**, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, **performance**, and quality level. **Process metrics** can be used to improve software development and maintenance.

Metrics are **an** important component of quality assurance, **management**, debugging, performance, and estimating costs, and **they**'re valuable for both developers and **development** team leaders: Managers **can** use **software metrics** to identify, prioritize, track and communicate any issues to foster better team productivity.

A **software** metric is a standard of measure of a degree to which a **software** system or process possesses some property. Even if a metric is not a measurement (**metrics** are functions, while measurements are the numbers obtained by the application of **metrics**), often the two terms are used as synonyms.

A **software metric** is a standard of measure of a degree to which a software system or process possesses some property. Even if a metric is not a measurement (metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used as synonyms. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance, testing, software debugging, software performance optimization, and optimal personnel task assignments.

software metric is a measure of software characteristics which are measurable or countable. Software metrics are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

Within the software development process, many metrics are that are all connected. Software metrics are similar to the four functions of management: Planning, Organization, Control, or Improvement.

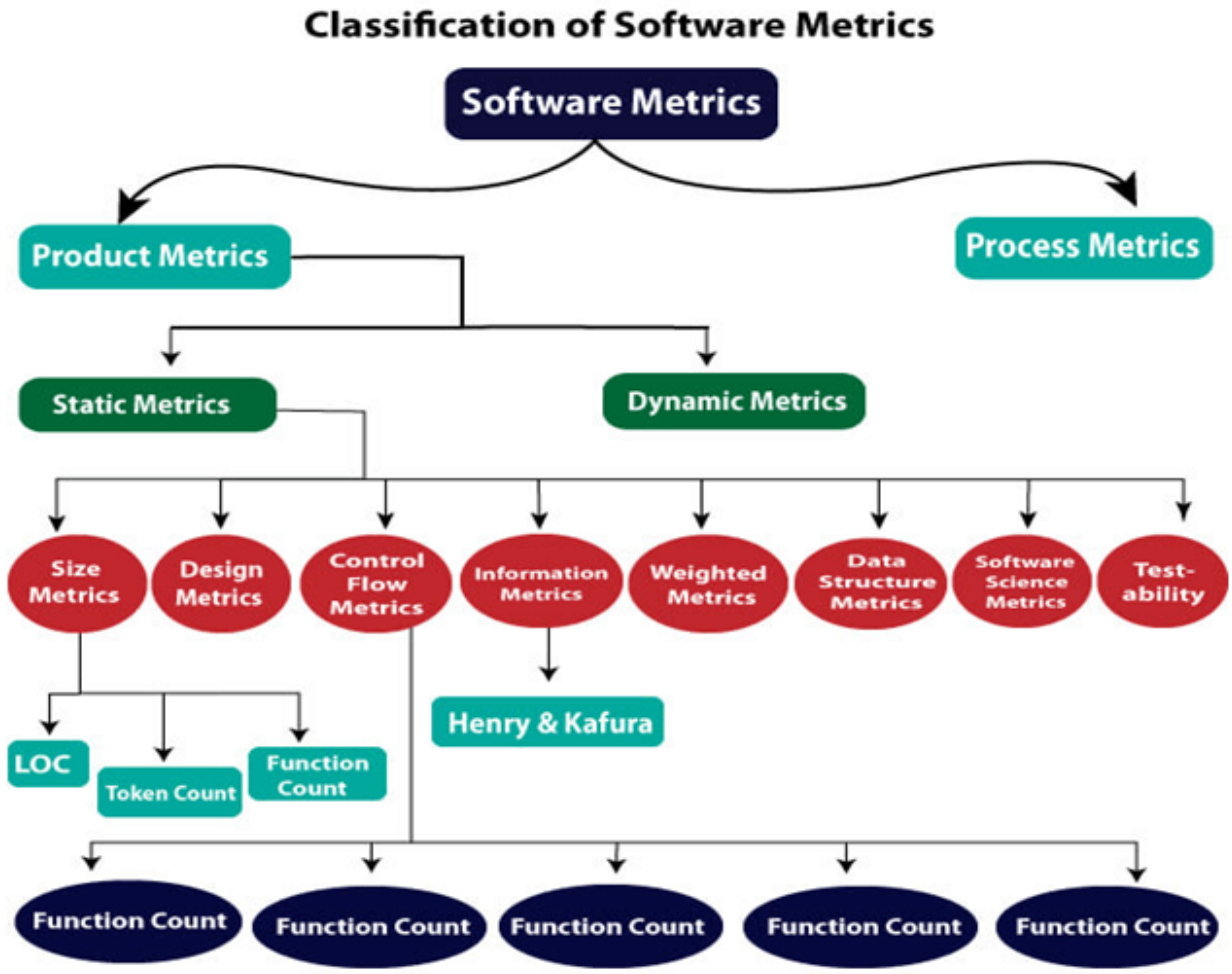
Classification of Software Metrics

1. Product Metrics: These are the measures of various characteristics of the software product. The two important software characteristics are:

1. Size and complexity of software.
2. Quality and reliability of software.

These metrics can be computed for different stages of SDLC.

2. Process Metrics: These are the measures of various characteristics of the software development process. For example, the efficiency of fault detection. They are used to measure the characteristics of methods, techniques, and tools that are used for developing software.



Types of Metrics

Internal metrics: Internal metrics are the metrics used for measuring properties that are viewed to be of greater importance to a software developer. For example, Lines of Code (LOC) measure.

External metrics: External metrics are the metrics used for measuring properties that are viewed to be of greater importance to the user, e.g., portability, reliability, functionality, usability, etc.

Hybrid metrics: Hybrid metrics are the metrics that combine product, process, and resource metrics. For example, cost per FP where FP stands for Function Point Metric.

Project metrics: Project metrics are the metrics used by the project manager to check the project's progress. Data from the past projects are used to collect various metrics, like time and cost; these estimates are used as a base of new software. Note that as the project proceeds, the project manager will check its progress from time-to-time and will compare the effort, cost, and time with the original effort, cost and time. Also understand that these metrics are used to

decrease the development costs, time efforts and risks. The project quality can also be improved. As quality improves, the number of errors and time, as well as cost required, is also reduced.

Advantage of Software Metrics

Comparative study of various design methodology of software systems.

For analysis, comparison, and critical study of different programming language concerning their characteristics.

In comparing and evaluating the capabilities and productivity of people involved in software development.

In the preparation of software quality specifications.

In the verification of compliance of software systems requirements and specifications.

In making inference about the effort to be put in the design and development of the software systems.

In getting an idea about the complexity of the code.

In taking decisions regarding further division of a complex module is to be done or not.

In guiding resource manager for their proper utilization.

In comparison and making design tradeoffs between software development and maintenance cost.

In providing feedback to software managers about the progress and quality during various phases of the software development life cycle.

In the allocation of testing resources for testing the code.

Disadvantage of Software Metrics

The application of software metrics is not always easy, and in some cases, it is difficult and costly.

The verification and justification of software metrics are based on historical/empirical data whose validity is difficult to verify.

These are useful for managing software products but not for evaluating the performance of the technical staff.

The definition and derivation of Software metrics are usually based on assuming which are not standardized and may depend upon tools available and working environment.

Most of the predictive models rely on estimates of certain variables which are often not known precisely

Small e-retailer measurement plan

Objectives	Sales and growth		Brand building		Customer loyalty		Grow and engage social community		
KPIs	Grow revenue	Grow customer base	Gain new customers	Increase reach/ visibility	Keep existing customers	Improve engagement	Grow likes/ follows	Shares/tweets/ pins/@/#	Social ripples
Measurable metrics									

MODULE –III

Data collection

- A metrics programme should be based on a set of product and process data.
- Data should be collected immediately (not after a project has finished) and, if possible, automatically.
  - Don't rely on memory
- Don't collect unnecessary data
  - The questions to be answered should be decided in advance and the required data identified.
- Tell people why the data is being collected.
  - It should not be part of personnel evaluation.

12 Steps to Useful Software Metrics

Step 1 - Identify Metrics Customers
Step 2 - Target Goals
Step 3 - Ask Questions
Step 4 - Select Metrics
Step 5 - Standardize Definitions
Step 6 - Choose a Model
Step 7 - Establish Counting Criteria
Step 8 - Decide On Decision Criteria
Step 9 - Define Reporting Mechanisms
Step 10 - Determine Additional Qualifiers
Step 11 - Collect Data
Step 12 - Consider Human Factors

Software metrics is a standard of measure that contains many activities, which involves some degree of measurement. The success in the software measurement lies in the quality of the data collected and analyzed.

What is Good Data?

The data collected can be considered as a good data, if it can produce the answers for the following questions −

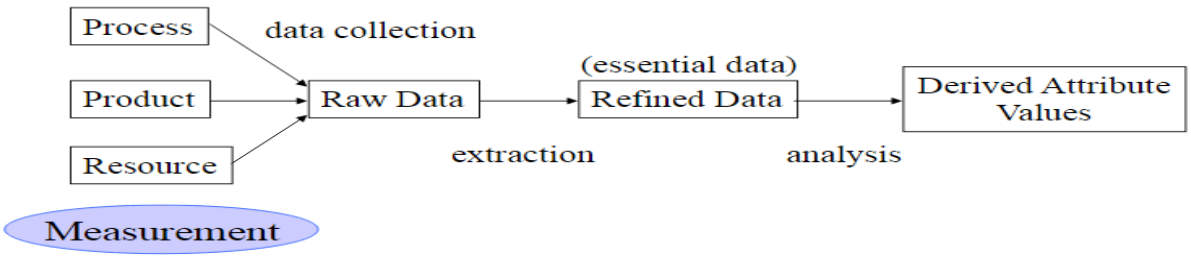
- **Are they correct?** − A data can be considered correct, if it was collected according to the exact rules of the definition of the metric.

- **Are they accurate?** − Accuracy refers to the difference between the data and the actual value.
- **Are they appropriately precise?** − Precision deals with the number of decimal places needed to express the data.
- **Are they consistent?** − Data can be considered as consistent, if it doesn't show a major difference from one measuring device to another.
- **Are they associated with a particular activity or time period?** − If the data is associated with a particular activity or time period, then it should be clearly specified in the data.
- **Can they be replicated?** − Normally, the investigations such as surveys, case studies, and experiments are frequently repeated under different circumstances. Hence, the data should also be possible to replicate easily.

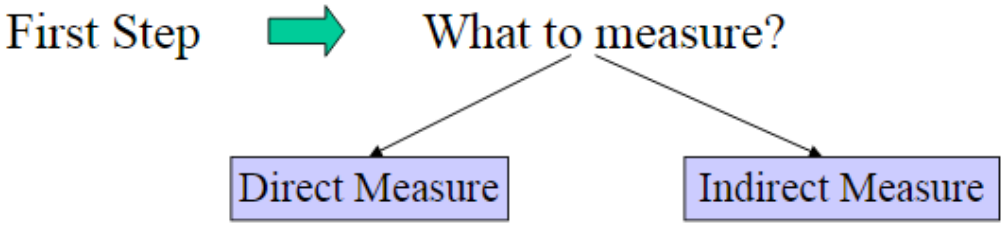
How to Define the Data?

Software Metrics - Data Collection

How to Define the Data?



How to Define the Data?



How to Define the Data?

Deciding what to measure - GQM (Goal-Question-Metric)

List the major goals of the development or maintenance project

Derive from each goal the questions that must be answered to determine if the goals are being met

Decide what must be measured in order to be able to answer the questions adequately

GQM (Goal-Question-Metric)

Goal: Evaluate effectiveness of coding standard

Questions: who is using standard?
what is coder's productivity?
What is code quality?

Metrics: Proportion of coders (多少人使用)
Experience of coders
Code size (function point, line of codes)
Errors

Software Quality

- How many problems have been found with a product
- Whether the product is ready for release to the next development stage or to the customer?
- How the current version of a product compares in quality with previous or competing versions?
- How effective are the prevention, detection, and removal processes?

Software Quality

The terminology is not uniform!!

Measure - If an organization measures its software quality in terms of faults per thousand lines of code.

Software Quality - Terminology

Human Error → Fault → Failure

IEEE standard 729

A fault occurs when a human error results in a mistake in some software product. (incorrect code as well as incorrect instructions in the user manual)

A failure is the departure of a system from its required behavior.

Software Quality

Key elements for the problem is observed

- Location - where did the problem occur
- Timing - when did it occur
- Symptom - what was observed
- End Result - which consequences resulted
- Mechanism - how did it occur
- Cause - why did it occur
- Severity - how much was the user affected
- Cost - how much did it cost

Data that is collected for measurement purpose is of two types −

- **Raw data** − Raw data results from the initial measurement of process, products, or resources. For example: Weekly timesheet of the employees in an organization.
- **Refined data** − Refined data results from extracting essential data elements from the raw data for deriving values for attributes.

Data can be defined according to the following points −

- Location
- Timing
- Symptoms
- End result
- Mechanism
- Cause
- Severity

- Cost

How to Collect Data?

- Open-Ended Surveys and Questionnaires. Opposite to closed-ended are open-ended surveys and questionnaires. ...
- 1-on-1 Interviews. One-on-one (or face-to-face) interviews are one of the most common types of data collection methods in qualitative research. ...
- Focus groups. ...
- Direct observation.

Primary **data sources** include information **collected** and processed directly by the researcher, such as observations, surveys, interviews, and focus groups. Secondary **data sources** include information retrieved through preexisting **sources**: research articles, Internet or library searches, etc.

Collection of data requires human observation and reporting. Managers, system analysts, programmers, testers, and users must record row data on forms. To collect accurate and complete data, it is important to −

- Keep procedures simple
- Avoid unnecessary recording
- Train employees in the need to record data and in the procedures to be used
- Provide the results of data capture and analysis to the original providers promptly and in a useful form that will assist them in their work
- Validate all data collected at a central collection point

Planning of data collection involves several steps −

- Decide which products to measure based on the GQM analysis
- Make sure that the product is under configuration control
- Decide exactly which attributes to measure and how indirect measures will be derived
- Once the set of metrics is clear and the set of components to be measured has been identified, devise a scheme for identifying each activity involved in the measurement process
- Establish a procedure for handling the forms, analyzing the data, and reporting the results

Data collection planning must begin when project planning begins. Actual data collection takes place during many phases of development.

For example − Some data related to project personnel can be collected at the start of the project, while other data collection such as effort begins at project starting and continues through operation and maintenance.

Software Metrics - Data Collection What is good data? Are they correct? Are they accurate? Are they appropriately precise? Are they consist? Are they associated with a particular activity or time period? Can they be replicated?

2 Software Metrics - Data Collection How to Define the Data? Process Product Resource Raw DataRefined Data Derived Attribute Values Measurement data collection extractionanalysis (essential data)

3 Software Metrics - Data Collection How to Define the Data? First StepWhat to measure? Direct MeasureIndirect Measure

4 Software Metrics - Data Collection How to Define the Data? Deciding what to measure - GQM (Goal-Question-Metric) List the major goals of the development or maintenance project Derive from each goal the questions that must be answered to determine if the goals are being met Decide what must be measured in order to be able to answer the questions adequately

5 Software Metrics - Data Collection GQM (Goal-Question-Metric) Goal: Evaluate effectiveness of coding standard Questions: who is using standard? what is coder's productivity? What is code quality? Metrics: Proportion of coders (多少人使用) Experience of coders Code size (function point, line of codes) Errors

6 Software Metrics - Data Collection Software Quality How many problems have been found with a product Whether the product is ready for release to the next development stage or to the customer? How the current version of a product compares in quality with previous or competing versions? How effective are the prevention, detection, and removal processes?

7 Software Metrics - Data Collection Software Quality The terminology is not uniform!! Measure - If an organization measures its software quality in terms of faults per thousand lines of code.

8 Software Metrics - Data Collection Software Quality - Terminology FaultFailure Human Error IEEE standard 729 A fault occurs when a human error results in a mistake in some software product. (incorrect code as well as incorrect instructions in the user manual) A failure is the departure of a system from its required behavior.

9 Software Metrics - Data Collection Software Quality - Terminology Errors - often means faults Anomalies - A class of faults that are unlikely to cause failures in themselves but may nevertheless eventually cause failures indirectly. Ex. Non-meaningful variable name Defects - normally refer collectively to faults and failures Bugs - faults occurring in the code Crashes - a special type of failure, where the system ceases to failure.

10 Software Metrics - Data Collection Software Quality - Terminology The reliability of a software system is defined in terms of failures observed during operation, rather than in terms of faults.

11 Software Metrics - Data Collection Software Quality Key elements for the problem is observed Location - where did the problem occur Timing - when did it occur Symptom - what was observed End Result - which consequences resulted Mechanism - how did it occur Cause - why did it occur Severity - how much was the user affected Cost - how much did it cost

12 Software Metrics - Data Collection Failure Report Location – installation where failure was observed Timing – CPU time, clock time or some temporal measure Symptom – Type of error message or indication of failure End Result – Description of failure, such as "operation system

crash," "services degraded," "loss of data," "wrong output," "no output" Mechanism – Cain of events, including keyboard commands and state data, leading to failure. What function the system was performed or how heavy the workload was when the failure occurred
13 Software Metrics - Data Collection Failure Report Cause – Reference the possible fault(s) leading to failure trigger: physical hardware failure operating conditions malicious(惡意的) action user error erroneous report source: unintentional(非特意的) design fault intentional design fault usability problem Severity – "Critical," "Major," "Minor" Cost – Cost to fix plus lost of potential business
14 Software Metrics - Data Collection Fault Report Location – within-system identifier (and version), such as module or document name Timing – Phases of development during which fault was created, detected, and corrected Symptom – Type of error message reported (IEEE standard on software anomalies, 1992) End Result – Failure caused by the fault Mechanism – How source was created (specification, coding, design, maintenance), detected(inspection, unit testing, system testing, integration testing), corrected(step to correct it)
15 Software Metrics - Data Collection Fault Report Cause – Type of human error that led to fault: communication: imperfect transfer of information conceptual: misunderstanding clerical: typographical or editing errors Severity – Refer to severity of resulting or potential failure Cost – Time or effort to locate and correct
16 Software Metrics - Data Collection Change Report Location – identifier of document or module changed Timing – When change was made Symptom – Type of change End Result – Success of change Mechanism – How and by whom change was performed Cause – Corrective, adaptive, preventive, or perfective Severity – Impact on rest of system Cost – Time and effort for change implementation and test
17 Software Metrics - Data Collection Refined Failure Data for Software Reliability Time Domain Data – recording the individual times at which the failure occurred (provide better accuracy in the parameter estimates) Interval Domain Data – counting the number of failures occurring over a fixed period
18 Software Metrics - Data Collection Time domain approach Failure records Actual Failure Time Between Time Failure 1 25 25 2 55 30 3 70 15 4 95 15 5 112 17
19 Software Metrics - Data Collection Interval domain approach Time Observed Number of Failures 0 1 2 2 4 3 1 4 1
20 Software Faults Software Fault Type Thayer et al. (1978) Computational errors Logical errors Input/output errors Data handling errors Operating system/system support error Configuration errors User interface errors Data base interface errors Present data base errors Global variable definition errors Operator errors Unidentified errors
21 Software Faults Fault Introduced in the Software Life Cycle Requirement and Specification Design - Function Design Logical Design Coding Testing Maintenance
22 Software Faults Error Removed in the Software Life Cycle Validation Unit Testing Integration Testing Acceptance Testing Operation and Demonstration
Fig.

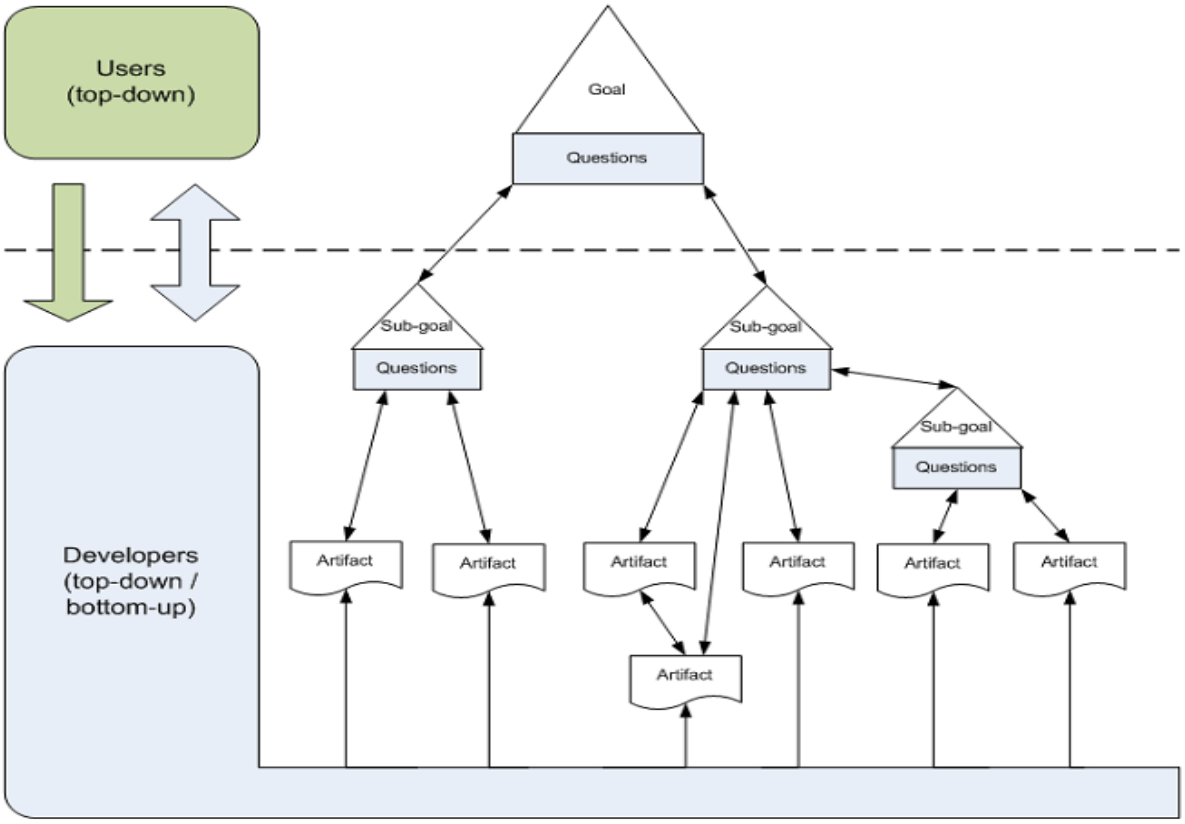


Fig. software metrics- track

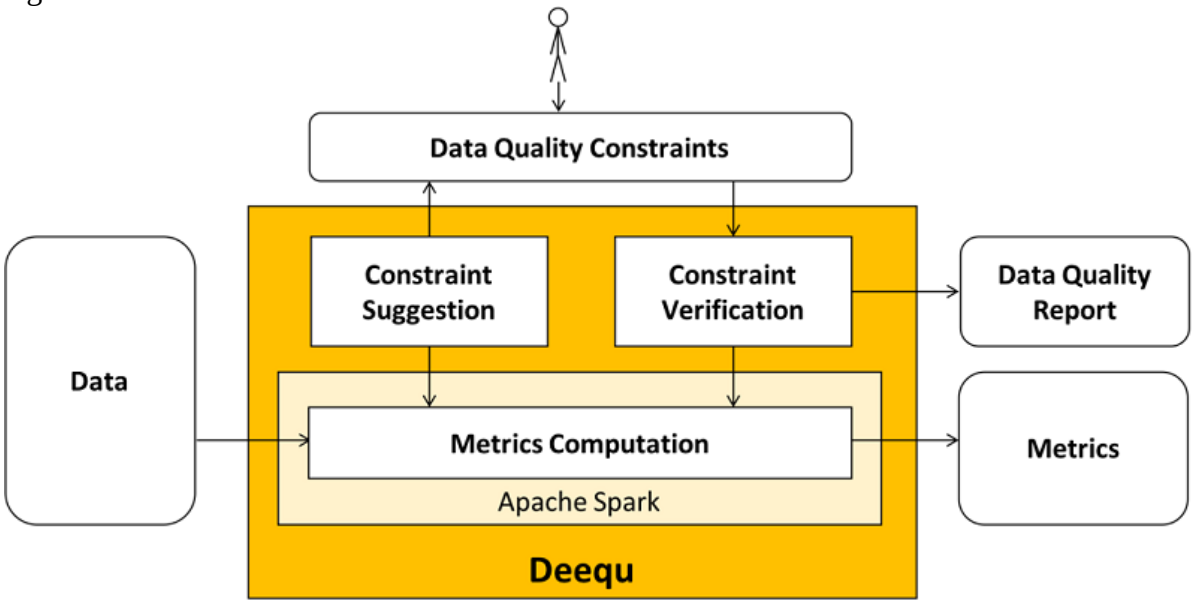
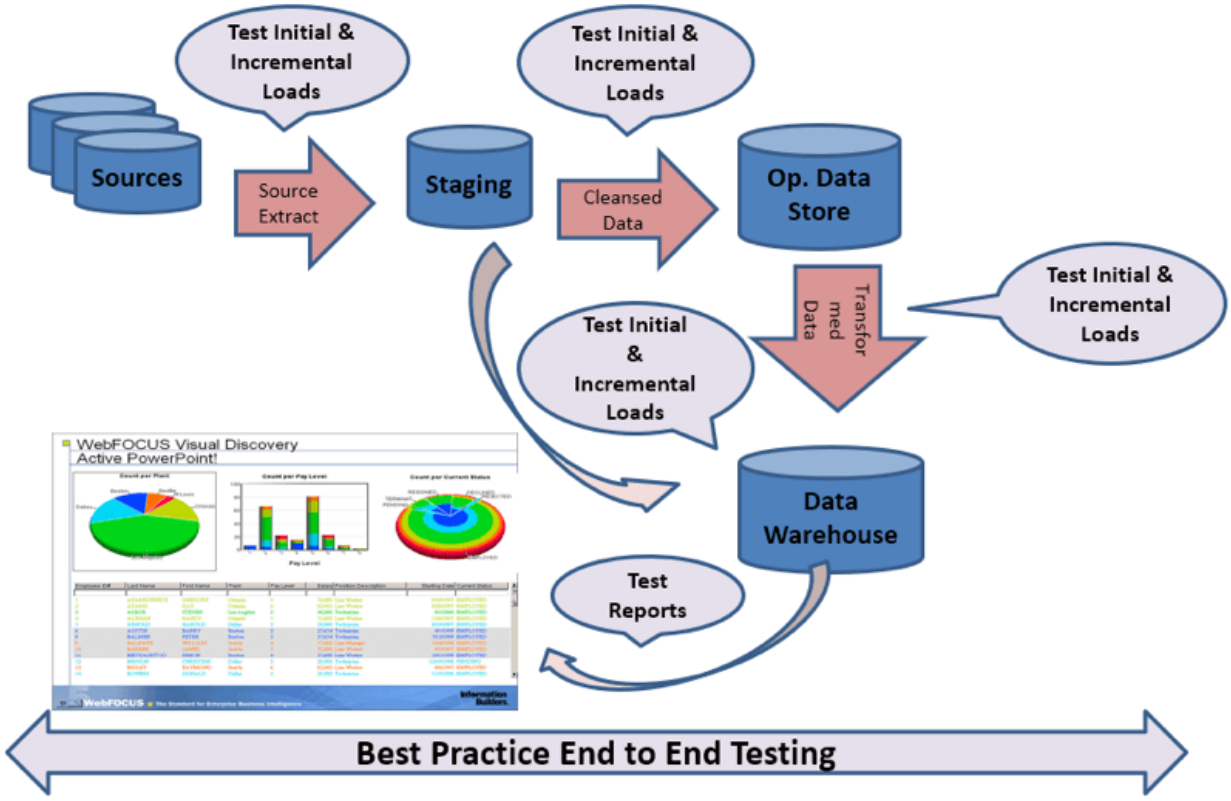


Fig. data quality



How to Store and Extract Data

In software engineering, data should be stored in a database and set up using a Database Management System (DBMS). An example of a database structure is shown in the following figure. This database will store the details of different employees working in different departments of an organization.

After collecting relevant data, we have to analyze it in an appropriate way. There are three major items to consider for choosing the analysis technique.

- The nature of data
- The purpose of the experiment
- Design considerations

Data Storage Methods

1. Floppy Disk
   - 1.44 Mb – no longer used, not enough capacity
2. Zip Disk
   - 50Gb – file is compressed and then stored
3. Hard Drive
   - 1 Terabyte – spinning disk inside your computer
4. CD-ROM
   - Compact Disk Read Only Memory 650Mb – data can only be read from not copied to
- DVD
   - Digital Versatile Disk – 4.7Gb, portable storage method used for storing larger files
6. USB Memory Stick

Analyzing the results of experiments

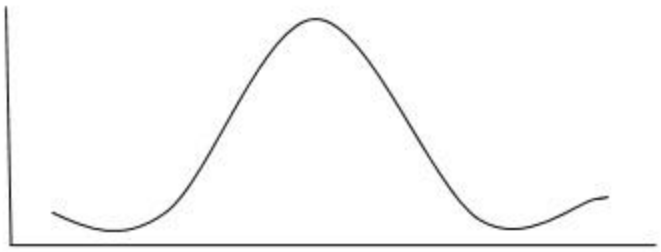
The Nature of Data

To analyze the data, we must also look at the larger population represented by the data as well as the distribution of that data.

Sampling, population, and data distribution

Sampling is the process of selecting a set of data from a large population. Sample statistics describe and summarize the measures obtained from a group of experimental subjects. Population parameters represent the values that would be obtained if all possible subjects were measured.

The population or sample can be described by the measures of central tendency such as mean, median, and mode and measures of dispersion such as variance and standard deviation. Many sets of data are distributed normally as shown in the following graph.



As shown above, data will be evenly distributed about the mean. which is the significant characteristics of a normal distribution.

Other distributions also exist where the data is skewed so that there are more data points on one side of the mean than other. For example: If most of the data is present on the left-hand side of the mean, then we can say that the distribution is skewed to the left.

The Purpose of the Experiment

Normally, experiments are conducted −

- To confirm a theory
- To explore a relationship

To achieve each of these, the objective should be expressed formally in terms of the hypothesis, and the analysis must address the hypothesis directly.

To confirm a theory

The investigation must be designed to explore the truth of a theory. The theory usually states that the use of a certain method, tool, or technique has a particular effect on the subjects, making it better in some way than another.

There are two cases of data to be considered: **normal data** and **non-normal data**.

If the data is from a normal distribution and there are two groups to compare then, the student's t test can be used for analysis. If there are more than two groups to compare, a general analysis of variance test called F-statistics can be used.

If the data is non-normal, then the data can be analyzed using Kruskal-Wallis test by ranking it.

To explore a relationship

Investigations are designed to determine the relationship among data points describing one variable or multiple variables.

There are three techniques to answer the questions about a relationship: box plots, scatter plots, and correlation analysis.

- A **box plot** can represent the summary of the range of a set of data.
- A **scatter plot** represents the relationship between two variables.
- **Correlation analysis** uses statistical methods to confirm whether there is a true relationship between two attributes.
    - For normally distributed values, use **Pearson Correlation Coefficient** to check whether or not the two variables are highly correlated.
    - For non- normal data, rank the data and use the **Spearman Rank Correlation Coefficient** as a measure of association. Another measure for non-normal data is the **Kendall robust correlation coefficient**, which investigates the relationship among pairs of data points and can identify a partial correlation.

If the ranking contains a large number of tied values, a **chi-squared test** on a contingency table can be used to test the association between the variables. Similarly, **linear regression** can be used to generate an equation to describe the relationship between the variables.

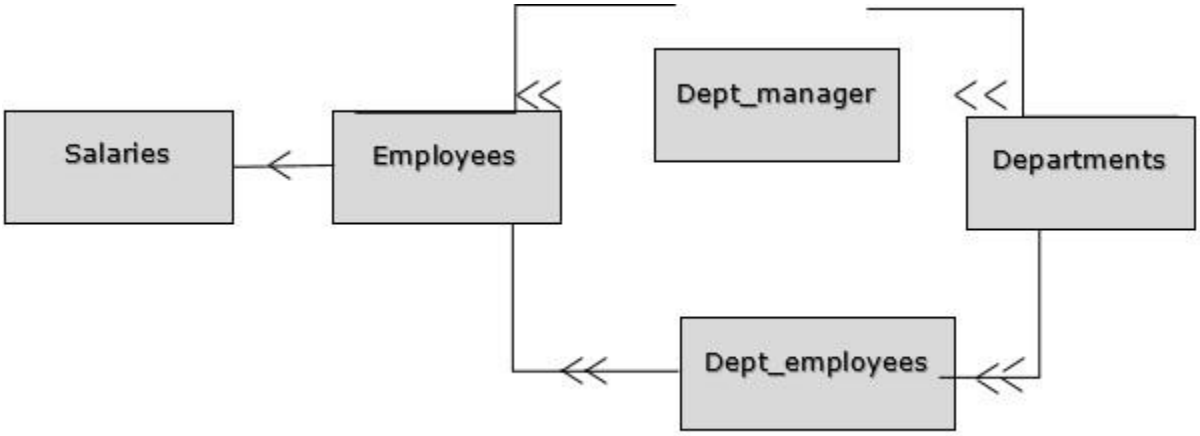
For more than two variables, **multivariate regression** can be used.

Design Considerations

The investigation's design must be considered while choosing the analysis techniques. At the same time, the complexity of analysis can influence the design chosen. Multiple groups use F-statistics rather than Student's T-test with two groups.

For complex factorial designs with more than two factors, more sophisticated test of association and significance is needed.

Statistical techniques can be used to account for the effect of one set of variables on others, or to compensate for the timing or learning effects.



In the above diagram, each box is a table in the database, and the arrow denotes the many-to-one mapping from one table to another. The mappings define the constraints that preserve the logical consistency of the data.

Once the database is designed and populated with data, we can make use of the data manipulation languages to extract the data for analysis.

Previous Page Print Page

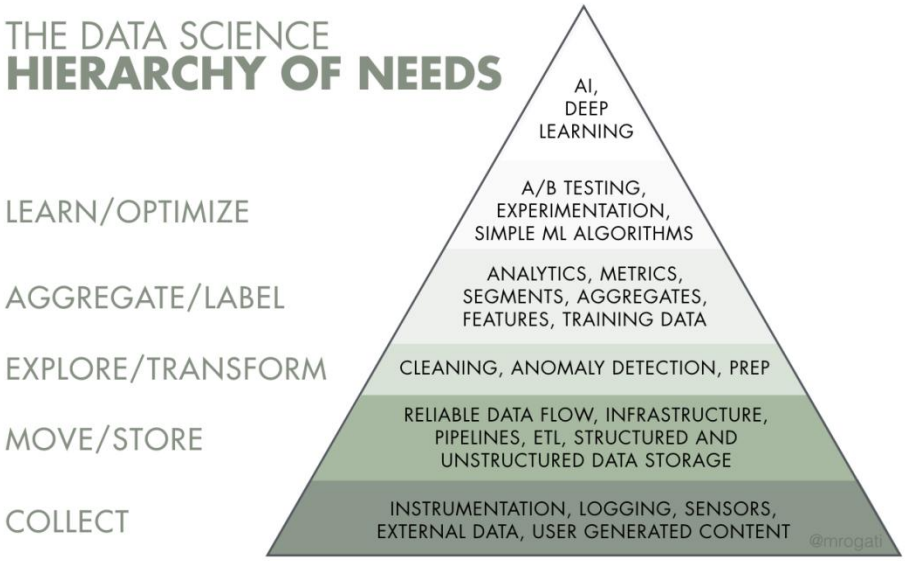


Fig. data engineering data warehouse.

1. **E measurement is a consistent but flexible process** tailored to the unique information needs and characteristics of a particular project or organization and revised as information needs change.
2. **Decision makers must understand what is being measured.** Key decision-makers must be able to connect *what is being measured* to *what they need to know* and *what decisions they need to make* as part of a closed-loop, feedback control process (Frenz et al. 2010).
3. **Measurement must be used to be effective.**

Measurement Process Overview

The measurement process as presented here consists of four activities from Practical Software and Systems Measurement (PSM) (2011) and described in (ISO/IEC/IEEE 15939; McGarry et al. 2002):

1. establish and sustain commitment
2. plan measurement
3. perform measurement
4. evaluate measurement

This approach has been the basis for establishing a common process across the software and systems engineering communities. This measurement approach has been adopted by the Capability Maturity Model Integration (CMMI) measurement and analysis process area (SEI 2006, 10), as well as by international systems and software engineering standards (ISO/IEC/IEEE 15939; ISO/IEC/IEEE 15288, 1). The International Council on Systems Engineering (INCOSE) Measurement Working Group has also adopted this measurement approach for several of their measurement assets, such as the *INCOSE SE Measurement Primer* (Frenz et al. 2010) and *Technical Measurement Guide* (Roedler and Jones 2005). This approach has provided a consistent treatment of measurement that allows the engineering community to communicate more effectively about measurement. The process is illustrated in Figure 1 from Roedler and Jones (2005) and McGarry et al. (2002).

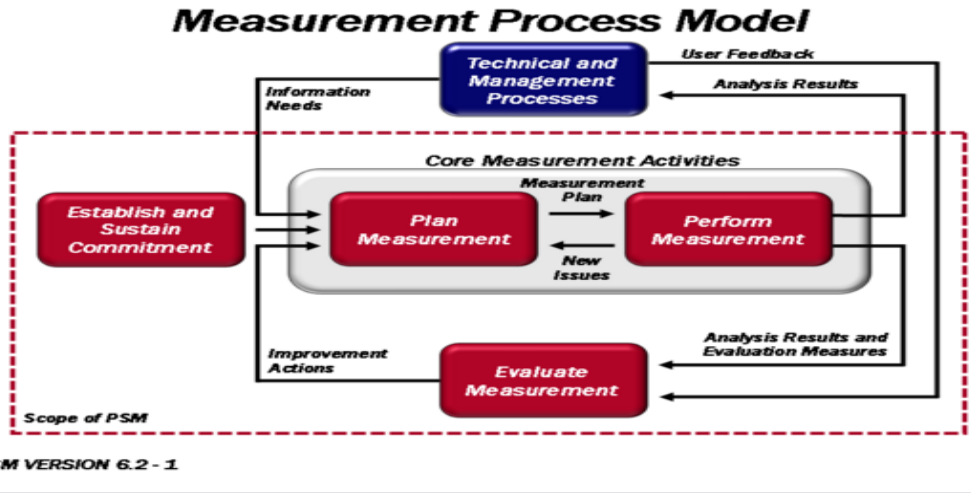


Figure 1. Four Key Measurement Process Activities (PSM 2011). Reprinted with permission of Practical Software and Systems Measurement (PSM). All other rights are reserved by the copyright owner.

Establish and Sustain Commitment

This activity focuses on establishing the resources, training, and tools to implement a measurement process and ensure that there is a management commitment to use the information that is produced. Refer to PSM (August 18, 2011) and SPC (2011) for additional detail.

Plan Measurement

This activity focuses on defining measures that provide insight into project or organization **information needs**. This includes identifying what the decision-makers need to know and when they need to know it, relaying these information needs to those entities in a manner that can be measured, and identifying, prioritizing, selecting, and specifying **measures** based on project and organization processes (Jones 2003, 15-19). This activity also identifies the reporting format, forums, and target audience for the information provided by the measures.

Here are a few widely used approaches to identify the information needs and derive associated measures, where each can be focused on identifying measures that are needed for SE management:

- The PSM approach, which uses a set of **information categories**, **measurable concepts**, and candidate measures to aid the user in determining relevant information needs and the characteristics of those needs on which to focus (PSM August 18, 2011).
- The (GQM) approach, which identifies explicit measurement goals. Each goal is decomposed into several questions that help in the selection of measures that address the question and provide insight into the goal achievement (Park, Goethert, and Florac 1996).
- Software Productivity Center's (SPC's) 8-step Metrics Program, which also includes stating the goals and defining measures needed to gain insight for achieving the goals (SPC 2011).

The following are good sources for candidate measures that address information needs and measurable concepts/questions:

- PSM Web Site (PSM 2011)
- PSM Guide, Version 4.0, Chapters 3 and 5 (PSM 2000)
- SE Leading Indicators Guide, Version 2.0, Section 3 (Roedler et al. 2010)
- Technical Measurement Guide, Version 1.0, Section 10 (Roedler and Jones 2005, 1-65)
- Safety Measurement (PSM White Paper), Version 3.0, Section 3.4 (Murdoch 2006, 60)
- Security Measurement (PSM White Paper), Version 3.0, Section 7 (Murdoch 2006, 67)
- Measuring Systems Interoperability, Section 5 and Appendix C (Kasunic and Anderson 2004)
- Measurement for Process Improvement (PSM Technical Report), version 1.0, Appendix E (Statz 2005)

The INCOSE *SE Measurement Primer* (Frenz et al. 2010) provides a list of attributes of a good measure with definitions for each **attribute**; these attributes include *relevance, completeness, timeliness, simplicity, cost effectiveness, repeatability, and accuracy.* Evaluating candidate measures against these attributes can help assure the selection of more effective measures.

The details of each measure need to be unambiguously defined and documented. Templates for the specification of measures and indicators are available on the PSM website (2011) and in Goethert and Siviy (2004).

Perform Measurement

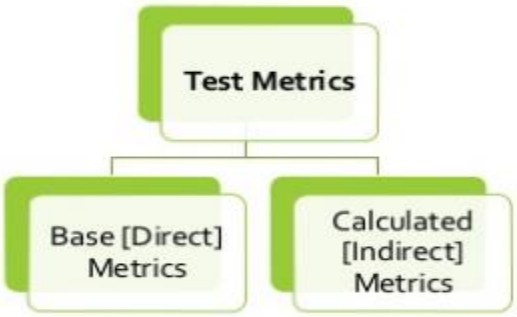
This activity focuses on the collection and preparation of measurement data, measurement analysis, and the presentation of the results to inform decision makers. The preparation of the measurement data includes verification, normalization, and aggregation of the data, as applicable. Analysis includes estimation, feasibility analysis of plans, and performance analysis of actual data against plans.

The quality of the measurement results is dependent on the collection and preparation of valid, accurate, and unbiased data. Data verification, validation, preparation, and analysis techniques

are discussed in PSM (2011) and SEI (2010). Per TL 9000, *Quality Management System Guidance, The analysis step should integrate quantitative measurement results and other qualitative project information, in order to provide managers the feedback needed for effective decision making* (QuEST Forum 2012, 5-10). This provides richer information that gives the users the broader picture and puts the information in the appropriate context.
There is a significant body of guidance available on good ways to present quantitative information. Edward Tufte has several books focused on the visualization of information, including *The Visual Display of Quantitative Information* (Tufte 2001).
Other resources that contain further information pertaining to understanding and using measurement results include

- PSM (2011)
- ISO/IEC/IEEE 15939, clauses 4.3.3 and 4.3.4
- Roedler and Jones (2005), sections 6.4, 7.2, and 7.3

Evaluate Measurement
This activity involves the analysis of information that explains the periodic evaluation and improvement of the measurement process and specific measures. One objective is to ensure that the measures continue to align with the business goals and information needs, as well as provide useful insight. This activity should also evaluate the SE measurement activities, resources, and infrastructure to make sure it supports the needs of the project and organization. Refer to PSM (2011) and *Practical Software Measurement: Objective Information for Decision Makers* (McGarry et al. 2002) for additional detail.
Software metrics used to :



ase metrics is the raw data collected by Test Analyst during the test
ıse development and execution

alculated metrics is derived from the data gathered in base metrics.

Examples of simple analysis techniques

examples of software metrics:

- Benefits of Software Metrics
- How Software Metrics Lack Clarity
- How to Track Software Metrics
- Examples of Software Metrics

Benefits of Software Metrics

The goal of tracking and analyzing software metrics is to determine the quality of the current product or process, improve that quality and predict the quality once the software development project is complete. On a more granular level, software development managers are trying to:

- Increase return on investment (ROI)

- Identify areas of improvement
- Manage workloads
- Reduce overtime
- Reduce costs

These goals can be achieved by providing information and clarity throughout the organization about complex software development projects. Metrics are an important component of quality assurance, management, debugging, performance, and estimating costs, and they're valuable for both developers and development team leaders:

- Managers can use software metrics to identify, prioritize, track and communicate any issues to foster better team productivity. This enables effective management and allows assessment and prioritization of problems within software development projects. The sooner managers can detect software problems, the easier and less-expensive the troubleshooting process.
- Software development teams can use software metrics to communicate the status of software development projects, pinpoint and address issues, and monitor, improve on, and better manage their workflow.

Software metrics offer an assessment of the impact of decisions made during software development projects. This helps managers assess and prioritize objectives and performance goals.

How Software Metrics Lack Clarity

Terms used to describe software metrics often have multiple definitions and ways to count or measure characteristics. For example, lines of code (LOC) is a common measure of software development. But there are two ways to count each line of code:

- One is to count each physical line that ends with a return. But some software developers don't accept this count because it may include lines of "dead code" or comments.
- To get around those shortfalls and others, each logical statement could be considered a line of code.

Thus, a single software package could have two very different LOC counts depending on which counting method is used. That makes it difficult to compare software simply by lines of code or any other metric without a standard definition, which is why **establishing a measurement method and consistent units of measurement** to be used throughout the life of the project is crucial.

There is also an issue with how software metrics are used. If an organization uses productivity metrics that emphasize volume of code and errors, software developers could avoid tackling tricky problems to keep their LOC up and error counts down. Software developers who write a large amount of simple code may have great productivity numbers but not great software development skills. Additionally, software metrics shouldn't be monitored simply because they're easy to obtain and display – only metrics that add value to the project and process should be tracked.

How to Track Software Metrics

Software metrics are great for management teams because they offer a quick way to track software development, set goals and measure performance. But oversimplifying software development can distract software developers from goals such as delivering useful software and increasing customer satisfaction.

Of course, none of this matters if the measurements that are used in software metrics are not collected or the data is not analyzed. The first problem is that software development teams may consider it **more important to actually do the work than to measure it**.

It becomes imperative to **make measurement easy to collect or it will not be done**. Make the software metrics work for the software development team so that it can work better. Measuring and analyzing doesn't have to be burdensome or something that gets in the way of creating code. Software metrics should have several important characteristics. They should be:

- Simple and computable
- Consistent and unambiguous (objective)
- Use consistent units of measurement
- Independent of programming languages
- Easy to calibrate and adaptable

- Easy and cost-effective to obtain
- Able to be validated for accuracy and reliability
- Relevant to the development of high-quality software products

This is why software development platforms that automatically measure and track metrics are important. But software development teams and management run the risk of having too much data and not enough emphasis on the software metrics that help deliver useful software to customers.

The technical question of how software metrics are collected, calculated and reported are not as important as deciding how to use software metrics. Patrick Kua outlines four guidelines for an appropriate use of software metrics:

1. Link software metrics to goals.

Often sets of software metrics are communicated to software development teams as goals. So the focus becomes:

- Reducing the lines of codes
- Reducing the number of bugs reported
- Increasing the number of software iterations
- Speeding up the completion of tasks

Focusing on those metrics as targets help software developers reach more important goals such as improving software usefulness and user experience.

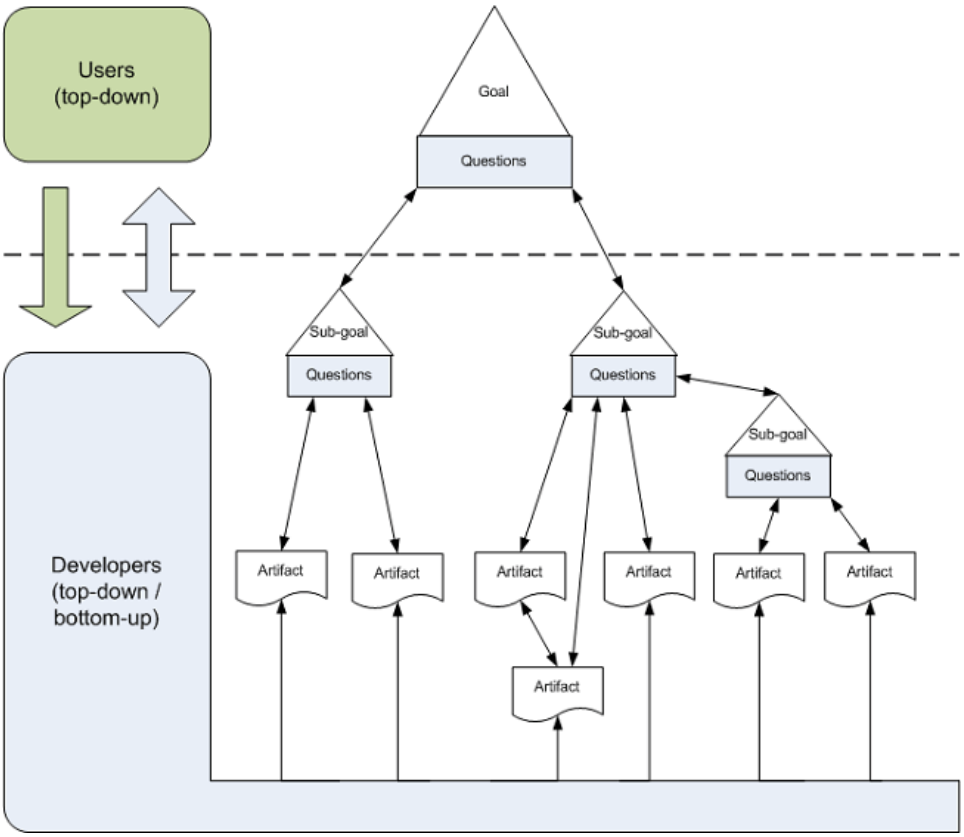


Image via Wikipedia

For example, size-based software metrics often measure lines of code to indicate coding complexity or software efficiency. In an effort to reduce the code's complexity, management may place restrictions on how many lines of code are to written to complete functions. In an effort to simplify functions, software developers could write more functions that have fewer lines of code to reach their target but do not reduce overall code complexity or improve software efficiency.

When developing goals, management needs to involve the software development teams in establishing goals, choosing software metrics that measure progress toward those goals and align metrics with those goals.

2. Track trends, not numbers.

Software metrics are very seductive to management because complex processes are represented as simple numbers. And those numbers are easy to compare to other numbers. So when a software metric target is met, it is easy to declare success. Not reaching that number lets software development teams know they need to work more on reaching that target.

These simple targets do not offer as much information on how the software metrics are trending. Any single data point is not as significant as the trend it is part of. Analysis of why the trend line

is moving in a certain direction or at what rate it is moving will say more about the process. Trends also will show what effect any process changes have on progress.

The psychological effects of observing a trend – similar to the Hawthorne Effect, or changes in behavior resulting from awareness of being observed – can be greater than focusing on a single measurement. If the target is not met, that, unfortunately, can be seen as a failure. But a trend line showing progress toward a target offers incentive and insight into how to reach that target.

3. Set shorter measurement periods.

Software development teams want to spend their time getting the work done not checking if they are reaching management established targets. So a hands-off approach might be to set the target sometime in the future and not bother the software team until it is time to tell them they succeeded or failed to reach the target.

By breaking the measurement periods into smaller time frames, the software development team can check the software metrics — and the trend line — to determine how well they are progressing.

Yes, that is an interruption, but giving software development teams more time to analyze their progress and change tactics when something is not working is very productive. The shorter periods of measurement offer more data points that can be useful in reaching goals, not just software metric targets.

4. Stop using software metrics that do not lead to change.

We all know that the process of repeating actions without change with the expectation of different results is the definition of insanity. But repeating the same work without adjustments that do not achieve goals is the definition of managing by metrics.

Why would software developers keep doing something that is not getting them closer to goals such as better software experiences? Because they are focusing on software metrics that do not measure progress toward that goal.

Some software metrics have no value when it comes to indicating software quality or team workflow. Management and software development teams need to work on software metrics that drive progress towards goals and provide verifiable, consistent indicators of progress.

Examples of Software Metrics

There is no standard or definition of software metrics that have value to software development teams. And software metrics have different value to different teams. It depends on what are the goals for the software development teams.

As a starting point, here are some software metrics that can help developers track their progress.

Agile process metrics

Agile process metrics focus on how agile teams make decisions and plan. These metrics do not describe the software, but they can be used to improve the software development process.

Lead time

Lead time quantifies how long it takes for ideas to be developed and delivered as software. Lowering lead time is a way to improve how responsive software developers are to customers.

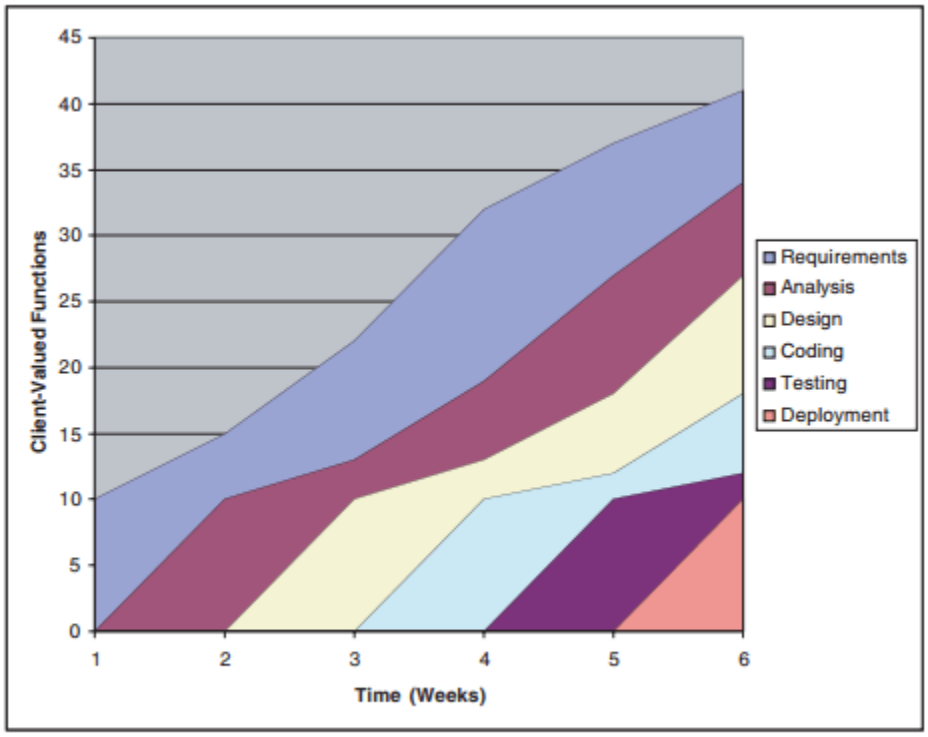


Figure 5–2
Cumulative flow of system inventory.

Screenshot via Pearsoned.co.uk

Cycle time

Cycle time describes how long it takes to change the software system and implement that change in production.

Team velocity

Team velocity measures how many software units a team completes in an iteration or sprint. This is an internal metric that should not be used to compare software development teams. The definition of deliverables changes for individual software development teams over time and the definitions are different for different teams.

Open/close rates

Open/close rates are calculated by tracking production issues reported in a specific time period. It is important to pay attention to how this software metric trends.

Production

Production metrics attempt to measure how much work is done and determine the efficiency of software development teams. The software metrics that use speed as a factor are important to managers who want software delivered as fast as possible.

Active days

Active days is a measure of how much time a software developer contributes code to the software development project. This does not include planning and administrative tasks. The purpose of this software metric is to assess the hidden costs of interruptions.

Assignment scope

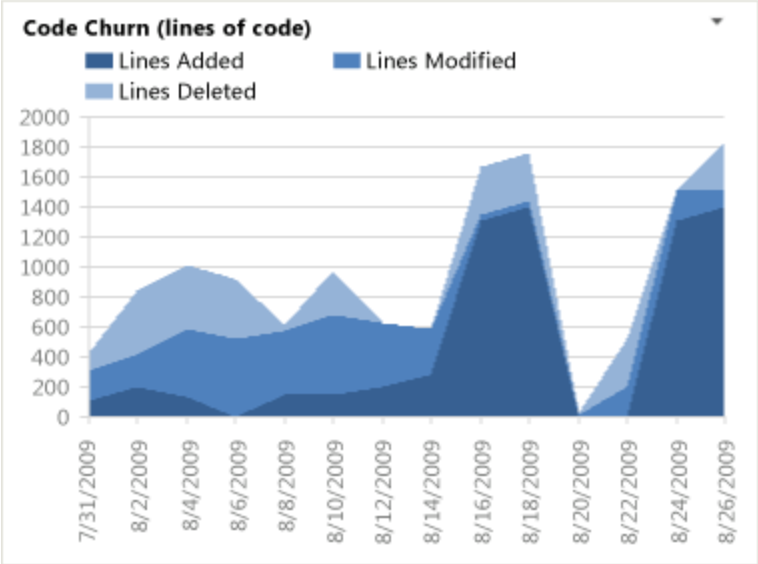
Assignment scope is the amount of code that a programmer can maintain and support in a year. This software metric can be used to plan how many people are needed to support a software system and compare teams.

Efficiency

Efficiency attempts to measure the amount of productive code contributed by a software developer. The amount of churn shows the lack of productive code. Thus a software developer with a low churn could have highly efficient code.

Code churn

Code churn represents the number of lines of code that were modified, added or deleted in a specified period of time. If code churn increases, then it could be a sign that the software development project needs attention.



Example Code Churn report, screenshot via Visual Studio

Impact

Impact measures the effect of any code change on the software development project. A code change that affects multiple files could have more impact than a code change affecting a single file.

Mean time between failures (MTBF) and mean time to recover/repair (MTTR)

Both metrics measure how the software performs in the production environment. Since software failures are almost unavoidable, these software metrics attempt to quantify how well the software recovers and preserves data.

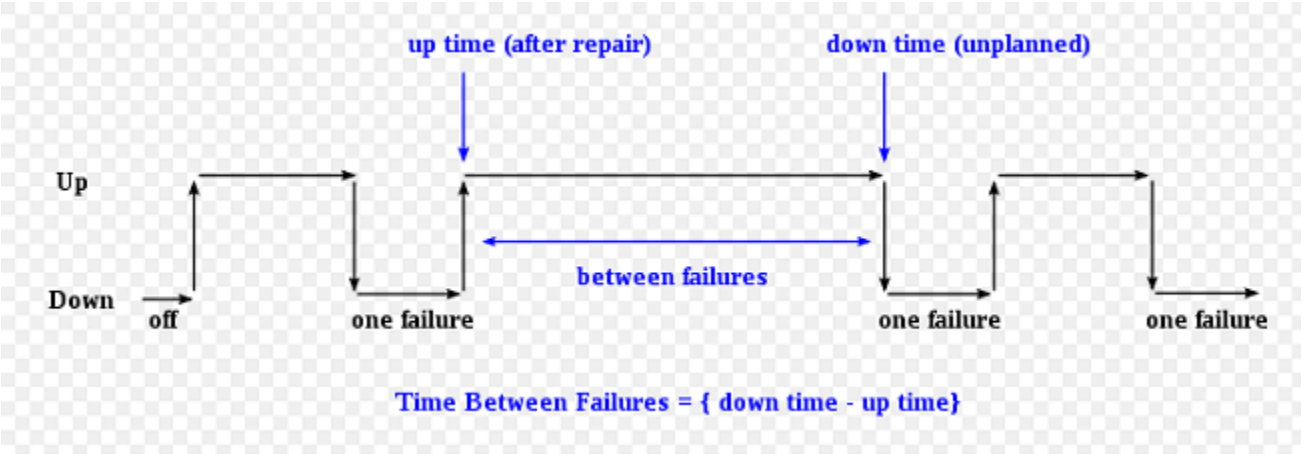


Image via Wikimedia Commons

Application crash rate (ACR)

Application crash rate is calculated by dividing how many times an application fails (F) by how many times it is used (U).

ACR = F/U

Security metrics

Security metrics reflect a measure of software quality. These metrics need to be tracked over time to show how software development teams are developing security responses.

Endpoint incidents

Endpoint incidents are how many devices have been infected by a virus in a given period of time.

Mean time to repair (MTTR)

Mean time to repair in this context measures the time from the security breach discovery to when a working remedy is deployed.

Size-oriented metrics

Size-oriented metrics focus on the size of the software and are usually expressed as kilo lines of code (KLOC). It is a fairly easy software metric to collect once decisions are made about what constitutes a line of code. Unfortunately, it is not useful for comparing software projects written in different languages. Some examples include:

- Errors per KLOC
- Defects per KLOC
- Cost per KLOC

Function-oriented metrics

Function-oriented metrics focus on how much functionality software offers. But functionality cannot be measured directly. So function-oriented software metrics rely on calculating the function point (FP) — a unit of measurement that quantifies the business functionality provided by the product. Function points are also useful for comparing software projects written in different languages.

Function points are not an easy concept to master and methods vary. This is why many software development managers and teams skip function points altogether. They do not perceive function points as worth the time.

Errors per FP or Defects per FP

These software metrics are used as indicators of an information system's quality. Software development teams can use these software metrics to reduce miscommunications and introduce new control measures.

Defect Removal Efficiency (DRE)

The Defect Removal Efficiency is used to quantify how many defects were found by the end user after product delivery (D) in relation to the errors found before product delivery (E). The formula is:

DRE = E / (E+D)

The closer to 1 DRE is, the fewer defects found after product delivery.

With dozens of potential software metrics to track, it's crucial for development teams to evaluate their needs and select metrics that are aligned with business goals, relevant to the project, and represent valid measures of progress. Monitoring the right metrics (as opposed to not monitoring metrics at all or monitoring metrics that don't really matter) can mean the difference between a highly efficient, productive team and a floundering one. The same is true of software testing: using the right tests to evaluate the right features and functions is the key to success. (Check out our guide on software testing to learn more about the various testing types.)

While the process of defining goals, selecting metrics, and implementing consistent measurement methods can be time-consuming, the productivity gains and time saved over the life of a project make it time well invested. Various software metrics are incorporated into solutions such as application performance management (APM) tools, along with data and insights on application usage, code performance, slow requests, and much more. Retrace, Stackify's APM solution, combines APM, logs, errors, monitoring, and metrics in one, providing a fully-integrated, multi-environment application performance solution to level-up your development work. Check out Stackify's interview with John Sumser with HR Examiner, and one of Forbes Magazine's 20 to Watch in Big Data, for more insights on DevOps and Big Data.

Additional Resources and Tutorials

Because there is little standardization in the field of software metrics, there are many opinions and options to learn more.

- Development Leaders Reveal the Best Metrics for Measuring Software Development Productivity
- How to Evaluate Software Quality from Source Code
- Software Metrics (Class notes from the MIT class "Software Engineering Concepts"
- Important Software Test Metrics and Measurements – Explained with Examples and Graphs
- 12 Steps to Useful Software Metrics
- Strengths and Weaknesses of Software Metrics
- Free Function Point Manual

- Development Leaders Reveal the Best Metrics for Measuring Software Development Productivity
- Guide to Advanced Empirical Software Engineering (Forrest Shull, Janice Singer, Dag I. K. Sjøberg; Springer Science & Business Media)
- Measuring Productivity Using the Infamous Lines of Code Metric

MODULE IV

Measuring internal product attributes:

Internal product attributes describe the software products in a way that is dependent only on the product itself. The major reason for measuring internal product attributes is that, it will help monitor and control the products during development.

Measuring Internal Product Attributes

The main internal product attributes include **size** and **structure**. Size can be measured statically without having to execute them. The size of the product tells us about the effort needed to create it. Similarly, the structure of the product plays an important role in designing the maintenance of the product.

Measuring the Size

Software size can be described with three attributes −

- **Length** − It is the physical size of the product.
- **Functionality** − It describes the functions supplied by the product to the user.
- **Complexity** − Complexity is of different types, such as.
    - **Problem complexity** − Measures the complexity of the underlying problem.
    - **Algorithmic complexity** − Measures the complexity of the algorithm implemented to solve the problem
    - **Structural complexity** − Measures the structure of the software used to implement the algorithm.
    - **Cognitive complexity** − Measures the effort required to understand the software.

The measurement of these three attributes can be described as follows −

Length

There are three development products whose size measurement is useful for predicting the effort needed for prediction. They are specification, design, and code.

Specification and design

These documents usually combine text, graph, and special mathematical diagrams and symbols. Specification measurement can be used to predict the length of the design, which in turn is a predictor of code length.

The diagrams in the documents have uniform syntax such as labelled digraphs, data-flow diagrams or Z schemas. Since specification and design documents consist of texts and diagrams, its length can be measured in terms of a pair of numbers representing the text length and the diagram length.

For these measurements, the atomic objects are to be defined for different types of diagrams and symbols.

The atomic objects for data flow diagrams are processes, external entities, data stores, and data flows. The atomic entities for algebraic specifications are sorts, functions, operations, and axioms. The atomic entities for Z schemas are the various lines appearing in the specification.

Code

Code can be produced in different ways such as procedural language, object orientation, and visual programming. The most commonly used traditional measure of source code program length is the Lines of code (LOC).

The total length,

$$\mathbf{LOC = NCLOC + CLOC}$$

i.e.,

$$\mathbf{LOC = Non\text{-}commented\ LOC + Commented\ LOC}$$

Apart from the line of code, other alternatives such as the size and complexity suggested by Maurice Halsted can also be used for measuring the length.

Halstead's software science attempted to capture different attributes of a program. He proposed three internal program attributes such as length, vocabulary, and volume that reflect different views of size.

He began by defining a program **P** as a collection of tokens, classified by operators or operands. The basic metrics for these tokens were,

- μ_1 = Number of unique operators
- μ_2 = Number of unique operands
- N_1 = Total Occurrences of operators
- N_2 = Number of unique operators

The length **P** can be defined as

$$N=N1+N2N=N1+N2$$

The vocabulary of **P** is

$$\mu=\mu1+\mu2\mu=\mu1+\mu2$$

The volume of program = No. of mental comparisons needed to write a program of length **N**, is

$$V=N\times\log2\mu V=N\times\log2\mu$$

The program level of a program **P** of volume **V** is,

$$L=V*VL=V*V$$

Where, $V*V*$ is the potential volume, i.e., the volume of the minimal size implementation of **P**

The inverse of level is the difficulty −

$$D=1/LD=1/L$$

According to Halstead theory, we can calculate an estimate **L** as

$$L'=1/D=2\mu1\times\mu2N2L'=1/D=2\mu1\times\mu2N2$$

Similarly, the estimated program length is, $\mu1\times\log2\mu1+\mu2\times\log2\mu2\mu1\times\log2\mu1+\mu2\times\log2\mu2$

The effort required to generate P is given by,

$$E=V/L=\mu1N2N\log2\mu2\mu2E=V/L=\mu1N2N\log2\mu2\mu2$$

Where the unit of measurement **E** is elementary mental discriminations needed to understand **P**

The other alternatives for measuring the length are −

- In terms of the number of bytes of computer storage required for the program text
- In terms of the number of characters in the program text

Object-oriented development suggests new ways to measure length. Pfleeger et al. found that a count of objects and methods led to more accurate productivity estimates than those using lines of code.

Functionality

The amount of functionality inherent in a product gives the measure of product size

Aspects of software size,

Software size is the main driver for project cost estimation

Software Size Measure	**Measurement Scope**	**Size**
Fast Function Points	Functional user requirements and configuration **size**	Gartner FFP

Story Points	Backlog items, functional and non-functional	'Effort/Complexity combination' in Story p

spects of software size

Each product of software development is a physical entity; as such, it can be described in terms of its size. Ideally, we want to define a set of attributes for software size analogous to human height and weight. That is we want them to be **fundamental**, so that each attribute captures a key aspect of software size. We suggest the following:

- **length:** physical size of the product
- **functionality:** functions supplied by the product to the user
- **complexity**
  - **problem complexity:** the complexity of the underlying problem
  - **algorithmic complexity:** efficiency of the algorithm
  - **structural complexity:** algorithm structure (Chapter 8)
  - **cognitive complexity:** understandability of software (not treated here)

When measuring length and functionality it is sometimes important to observe **reuse**.

7.2 Length, code

Some other usable length measures are **number of characters** and **bytes of computer storage**.

Maurice Halstead made an early attempt, **Software Science**, to capture notions of size and complexity. The basic metrics were

mu_1 = number of unique operators
mu_2 = number of unique operands
N_1 = total number of operators
N_2 = total number of operands

From these he then defined a number of derived measures, with **length** defined as $N = N_1 + N_2$, **vocabulary** defined as $mu = mu_1 + mu_2$, **volume** (amount of computer storage necessary for a uniform binary encoding) as $V = N \log mu$, and the rest some generally invalidated (even intuitively implausible) prediction systems.

There are several problems in defining length in visual programming and windowing environments, and also (to a lesser extent) in object orientation and fourth-generation languages. So, for instance, in object-orientation, a count of objects and methods could be more useful as measures of length for the prediction of productivity than using LOC.

7.2 Length, specifications and designs

Specification and design documents contain **text** and **diagrams**, which could make up a composite measure of length, i.e., (text length, diagram length).

7.2 Length, predicting

There are some attempts to predict length of later life-cycle documents based on length of earlier life-cycle documents but nothing really usable is in sight.

7.3 Reuse

The reuse of software (including requirements, designs, documentation, and test data and scripts as well as code) improves our productivity and quality, allowing us to concentrate on new problems, rather than to continuing solving old ones again, see Table 7.4.

Counting reused code is not simple, the **extent of reuse** plays an important role.
NASA/Goddard's Software Engineering Laboratory used the following classification

1. **Reused verbatim:** the code was reused without any changes
2. **Sightly modified:** fewer than 25% of the lines of code in the unit were modified
3. **Extensively modified:** 25% or more of the code were modified
4. **New:** none of the code come from a previously constructed unit.

Hewlett-Packard uses three levels: **reused** (unmodified), **leveraged** (modified), and **new** code.

For example of reuse, see Table 7.5 and Figure 7.3.

7.4 Functionality

Many software engineers argue that length is misleading, and that the amount of **functionality** inherent in a product is a better picture of product size. There are three interesting approaches in this direction namely

- Albrecht's **function points**
- DeMarco's **specification weight**
- the COCCOMO 2.0 approach to **object points**

All three approaches measure the functionality of specification documents, but each can also be applied to later life-cycle products to refine the size estimate and therefore the cost or productivity estimation

Software Length-reuse and functionality:

The length of the program in the terms of the total number of tokens used is

$N = N_1 + N_2$

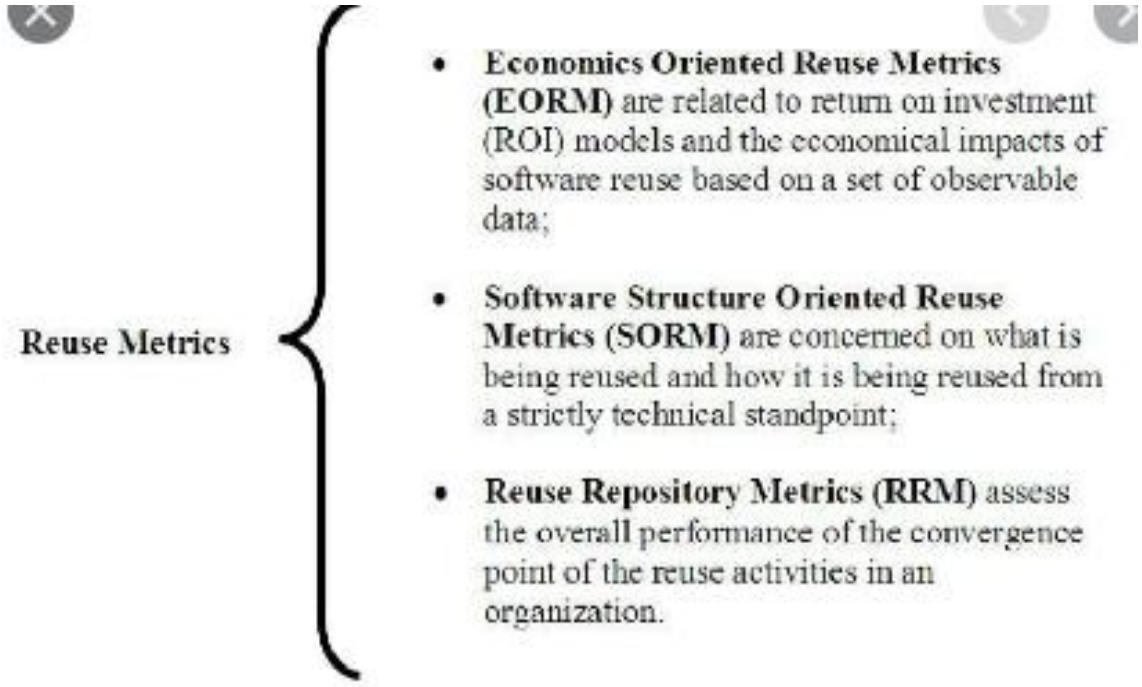
where

N : program length

N_1 : total occurrences of operators

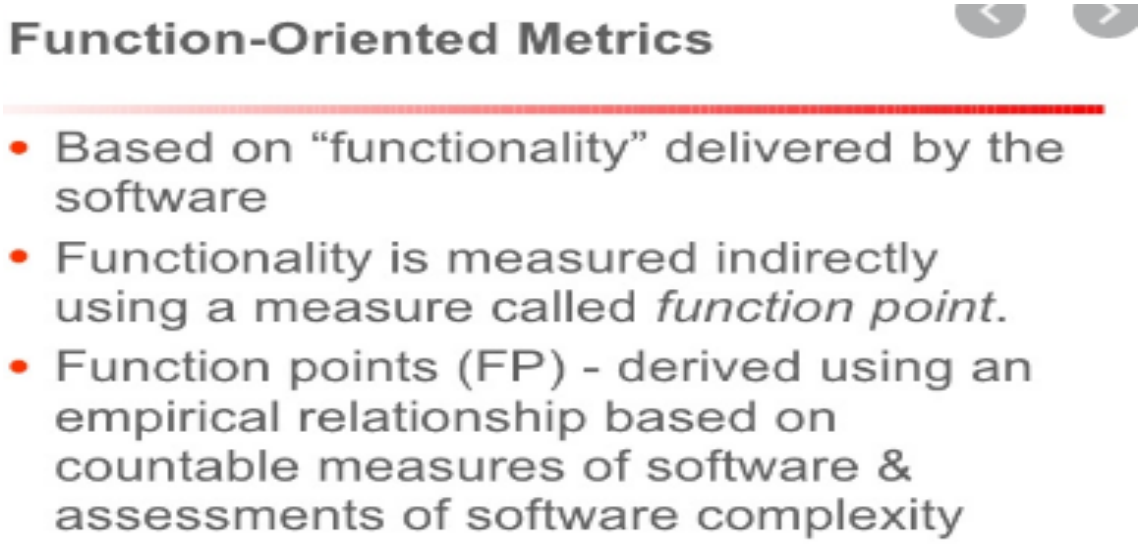
N_2 : total occurrences of operands

Software Reuse. A **definition** of **software reuse** is the process of creating **software** systems from predefined **software** components. The advantage of **software reuse**: The systematic development of **reusable** components. The systematic **reuse** of these components as building blocks to create new systems.



The purpose of most **software functions** is to transform inputs into an output or product. ... While **functions** that do not directly process data may not satisfy computer language specific or

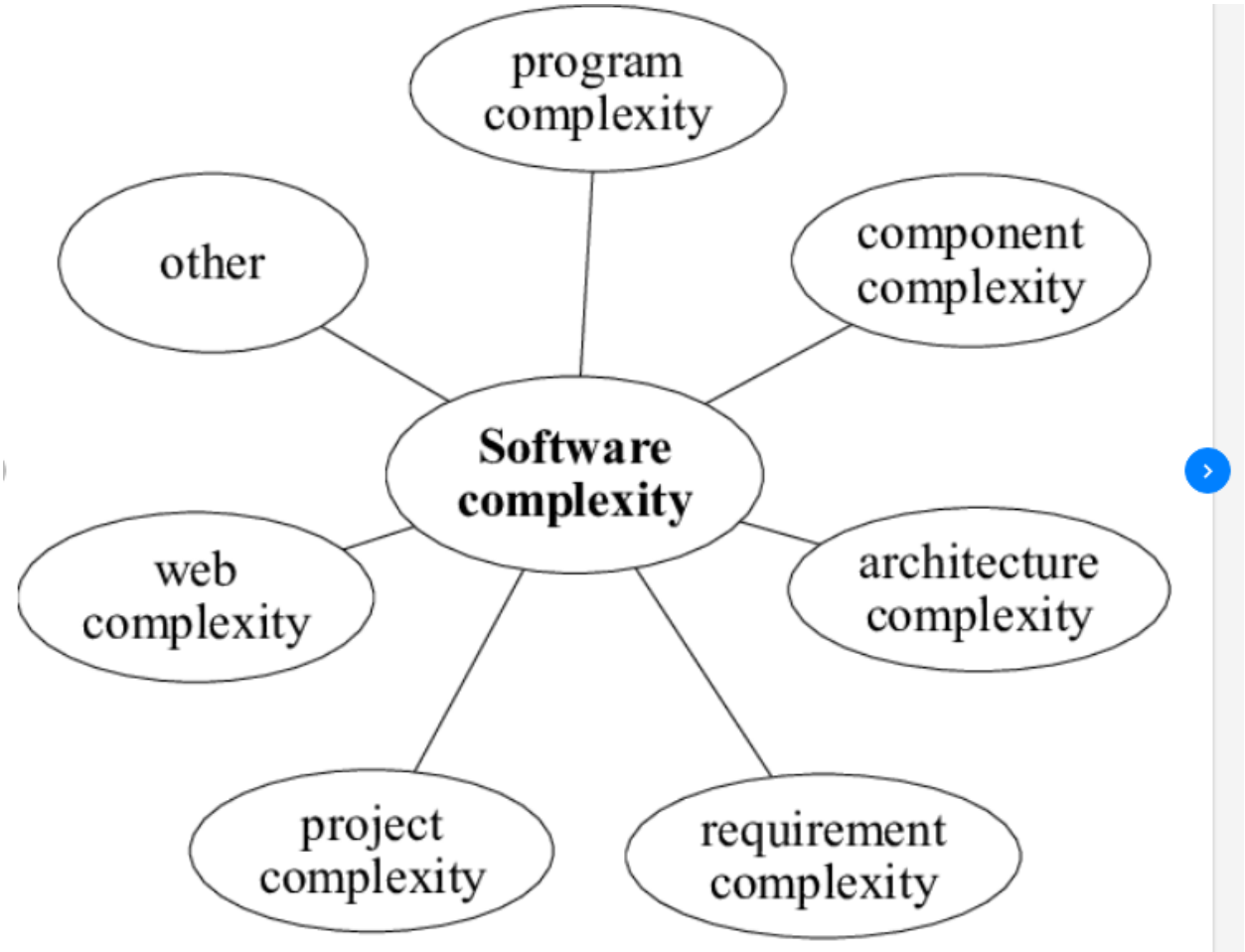
mathematical criteria, they do perform significant actions within the **software** engineering field of study



Software Complexity.

Software complexity is a way to describe a specific set of characteristics of your code. These characteristics all focus on how your code interacts with other pieces of code. The measurement of these characteristics is what determines the **complexity** of your code. It's a lot like a **software** quality grade for your code.

The **software** engineering discipline has established some common **measures of software complexity**. Perhaps the most common **measure** is the McCabe essential **complexity** metric. This is also sometimes called cyclomatic **complexity**. It is a **measure** of the depth and quantity of routines in a piece of code.



Types of structural measures

Data-flow **structure** − It is the behavior of the data as it interacts with the program. Data **structure** − It is the organization of the data elements in the form of lists, queue, stacks, or other well-defined **structures** along with algorithm for creating, modifying, or deleting them.

Measurement of structural properties of a software is important for estimating the development effort as well as for the maintenance of the product. The structure of requirements, design, and code helps understand the difficulty that arises in converting one product to another, in testing a product, or in predicting the external software attributes from early internal product measures.

Types of Structural Measures

The structure of software has three parts. They are −

- **Control-flow structure** − It is the sequence in which instructions are executed in a program.
- **Data-flow structure** − It is the behavior of the data as it interacts with the program.
- **Data structure** − It is the organization of the data elements in the form of lists, queue, stacks, or other well-defined structures along with algorithm for creating, modifying, or deleting them.

Measuring Control-Flow Structure

The control flow measures are usually modeled with directed graph, where each node or point corresponds to program statements, and each arc or directed edge indicates the flow of control from one statement to another. Theses graphs are called control-flow graph or directed graph.

If **'m'** is a structural measure defined in terms of the flow graph model, and if program **A** is structurally more complex than program **B**, then the measure **m(A)** should be greater than **m(B)**.

Measuring Data-Flow Structure

Data flow or information flow can be inter-modular (flow of information within the modules) or intra-modular (flow of information between individual modules and the rest of the system).

According to the way in which data is moving through the system, it can be classified into the following −

- **Local direct flow** − If either a module invokes a second module and passes information to it or the invoked module returns a result to the caller.
- **Local indirect flow** − If the invoked module returns information that is subsequently passed to a second invoked module.
- **Global flow** − If information flows from one module to another through a global data structure.

Information flow complexity can be expressed according to Henry and Kafura as,

$$\textbf{Information flow complexity (M) = length (M) × fan-in (M) × (fan-out (M))}^2$$

Where,

- **Fan-in (M)** − The number of local flows that terminate at M + the number of data structures from which the information is retrieved by M.
- **Fan–out (M)** − The number of local flows that emanate from M + the number of data structures that are updated by M.

Measuring Data Structure

Data structure can be both **local** and **global**.

Locally, the amount of structure in each data item will be measured. A graph-theoretic approach can be used to analyze and measure the properties of individual data structures. In that simple data types such as integers, characters, and Booleans are viewed as primes and the various operations that enable us to build more complex data structures are considered. Data structure measures can then be defined hierarchically in terms of values for the primes and values associated with the various operations.

Globally, a count of the total number of user-defined variables will be measured.

Control-flow structure, Modularity and information

Modularity and information flow attributes

So far, we have examined attributes of individual modules, i.e., **intra-modular measures**. We now look at entities being collections of modules and **inter-modular measures.**

A module is a separately compilable piece of code. Figure 8.17 contains an example of a graph describing the information flow between modules.

A **module call-graph** is a graph where the nodes are modules and an edge from node A to node B means that module A calls module B, see Figure 8.18.

A **data dependency graph** shows how data is computed based on data existing or computed earlier, see Figure 8.19.

There are several measures proposed for measuring modular structure like the **morphology measure, tree impurity, internal reuse** defined on some dependency graphs, see Figures 8.20 - 8.22.

Coupling is the degree of interdependence between modules, viewed as pairs. Coupling can be classified ordinally according to the relations as:

- **No coupling relation R_0:** Independent.
- **Data coupling relation R_1:** *x, y* communicate by parameters.
- **Stamp coupling relation R_2:** *x, y* accept the same record type as a parameter.
- **Control coupling relation R_3:** *x* passes a parameter to *y* with the intention to control its behaviour.
- **Common coupling relation R_4:** *x, y* refer to the same global data.
- **Content coupling relation R_5:** *x* refers to (branches, change data, alter statement) the inside of *y*.

Modules are said to be **loosely coupled** for R_1, R_2, and **tightly coupled** for R_4, R_5. For a coupling model graph,

Modularity and information flow attributes

The **cohesion** of a module is the extent to which its individual components are needed to perform the **same** task. An ordinal classification (Yourdon and Constantine, 1979) ordered from most desirable to least desirable is:

1. **Functional:** the module performs a single function.
2. **Sequential:** several functions in the order of specifiaction.
3. **Communicational:** several functions on the same data.
4. **Procedural:** several functions related only to a general procedure affected by the software.
5. **Temporal:** several functions that must occur within the same timespan.
6. **Logical:** several functions only related logically.
7. **Coincidental:** several unrelated functions.

One simple inter-modular measure of cohesion may be defined by:

Cohesion ratio = # functional modules / # all modules

Henry and Kafura's measure of the total level of information flow between individual modules and the rest of a system is well known, but has also been criticized as measuring both control flow and information flow (Shepperd).
We say that **local direct flow** exists if 1) a module invokes a second module and passes data to it; or 2) the invoked module returns a result to the caller.
A **local indirect flow** exists if the invoked module returns information that is subsequently passed to a second invoked module.
A **global flow** exists if information flows from one module to another via a global data structure.
The **fan-in** of a module *M* is the number of local flows that terminate at *M*, plus the number of data structures from which information is retrieved by *M*.
The **fan-out** of a module *M* is the number of local flows that emanate from *M*, plus the number of data structures that are updated by *M*.
Based on these concepts, Henry and Kafura define **information flow complexity** as (see Figure 8.24 for an example):

$$\text{i-f-c(M)} = \text{length(M)} \times (\text{fan-in(M)} \times \text{fan-out(M)})^2$$

8.3 Modularity and information flow attributes

Most data-flow testing strategies focus on the program paths, the so called ***du*-paths**, that link the **definition** (new value given) and **use** of variables. We distinguish between:

- variable uses within computations (***c*-uses**)
- variable uses within predicates (***p*-uses**)

All *du*-paths testing strategy means test cases so that every *du*-path lies on at least one program path executed by the test cases. Even if worst case analysis shows exponential complexity on the number of conditional transfers, empirical evidence suggests that *du*-path testing is feasible in practice, and that the minimal number of paths *P* required, is a very useful measure for quality-assurance purpos

Modularity and information flow attributes Functional: the module performs a single function. Sequential: several functions in the order of specifiaction. Communicational: several functions on the same data. Procedural: several functions related only to a general procedure affected by the software.

Object-oriented metrics

Chidamber and Kemerer define a number of metrics that are claimed to relate to some of the object oriented attributes coupling, cohesion, object complexity, and scope of properties:

1. Weighted methods per class: c1,...,cn
2. Depth of inheritance tree: for a class

3. Number of children: for a class
4. Coupling between object classes: for a class to other classes
5. Response for class: number of local methods + methods called from these
6. Lack of cohesion metric: number of disjoint sets of local methods

The paper of Ebert and Morschel: Metrics for quality analysis and improvement of object-oriented software lists six metrics: 1 Volume, 2 Method structure, 3 Cohesion, 4 Coupling, 5 Inheritance tree, and 6 Class organization and suggests how these should be measured. A tool, SmallMetric, helping in the measurement and analysis is demonstrated.

Data structure

In computer science, a **data structure** is a **data** organization, management, and storage format that enables efficient access and modification. More precisely, a **data structure** is a collection of **data** values, the relationships among them, and the functions or operations that can be applied to the **data**.

Data Structure Metrics

There have been few attempts to define measures of actual data items and their structure. Two examples:

Number of distinct operand = number of variables + number of unique constants + number of labels

$$D / P = \frac{\text{Database size in bytes or characteres}}{\text{Program size in delivered source instruction (DSI)}}$$

(Boehm)

Data Structure Metrics

Program	*Data Input*	*Internal Data*	*Data Output*
Payroll	Name/ Social Security No./ Pay Rate/ Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spreadsheet	Item Names/ Item amounts/ Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size/ No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig.1: Some examples of input, internal, and output data

Data Structure Metrics: Line of Code, Function Point, and Token Count are important **metrics** to find out the effort and time required to complete the project. There are some **Data Structure metrics** to compute the effort and time required to complete the project. There **metrics** are: The Amount of **Data**.

Data Structure Metrics

Essentially the need for software development and other activities are to process data. Some data is input to a system, program or module; some data may be used internally, and some data is the output from a system, program, or module.

Example:

Program	Data Input	Internal Data	Data Output
Payroll	Name/Social Security No./Pay rate/Number of hours worked	Withholding rates Overtime Factors Insurance Premium Rates	Gross Pay withholding Net Pay Pay Ledgers
Spreadsheet	Item Names/Item Amounts/Relationships among Items	Cell computations Subtotal	Spreadsheet of items and totals
Software Planner	Program Size/No of Software developer on team	Model Parameter Constants Coefficients	Est. project effort Est. project duration

That's why an important set of metrics which capture in the amount of data input, processed in an output form software. A count of this data structure is called Data Structured Metrics. In these concentrations is on variables (and given constant) within each module & ignores the input-output dependencies.

There are some Data Structure metrics to compute the effort and time required to complete the project. There metrics are

Difficulties with general "complexity" measures.

A **quality metric** is a number that represents one facet or aspect of software quality. For example, in the last lecture, *availability* was defined to be the ratio of the total time that the software was up, to the sum of the time that it was up and the sum that it was down. This is a metric that represents one aspect of the software quality factor, *reliability*.

Application to Assessment of a Quality Factor

Frequently, a software quality factor is measured (or estimated) by determining the values of several related metrics and forming a weighted sum:

$$F_q = c_1 m_1 + c_2 m_2 + ... + c_n m_n$$

where

- F_q represents a software quality factor we are interested in but that is difficult to measure directly;
- m_1, m_2, up to m_n are metrics whose values can be determined relatively easily, and that are related to the software quality factor; and
- c_1, c_2, up to c_n are regression coefficients - real numbers that are fixed from system to system (or project to project) and determine the way and extent that the value of each metric affects the software quality factor.

Values for regression coefficients are generally set (or, perhaps, estimated) by examining data from past projects and estimating the relative importance of each metric for the quality factor to be determined. The greater the amount of this past data that is available, the more accurate the regression coefficients are likely to be!

Two Types of Quality Metrics

- A *product metric* is a number derived (or extracted) directly from a document or from program code.
- A *process metric* is a number extracted from the performance of a software task. As defined above, availability is a process metric that is related to software reliability.

Uses of Metrics for SQA

Quality metrics are useful

- as a **quality assurance threshold**: Minimal (or maximal) acceptable values for metrics can be stated in requirements specifications, in order to set comprehensible and testable requirements related to quality factors (besides functional and performance requirements).

- as a **filtering mechanism**. For example, it's frequently true that sufficient resources aren't available for a thorough inspection of all the code associated with a project. It's known that code which is more *complex* will generally be more difficult to maintain and test than less *complex* code. A complexity metric can be used to identify the most highly complex code that is part of the current software project - and care can then be taken to inspect and test this code more thoroughly than would be possible if all the code received the same attention.
- to **evaluate** development methods. It is (therefore) desirable to maintain a database of statistics about past projects.
- to help **control complexity**, especially during maintenance. The complexity of code tends to increase as it is maintained and changed. It may be less expensive (in the long run) to redesign and recode a complex module, than it would be to continue to try to "patch" it.
- to provide staff with **feedback>** on the quality of their work. Once again, complexity metrics are useful for an example: If several designs are being considered then it is generally worthwhile to choose the one that is the least complex - provided that it is not disqualified for other reasons. Thus, metrics can be useful for comparing proposed solutions for problems, and choosing between or among them.

Complexity

Complexity is a measure of the resources which must be expanded in developing, maintaining, or using a software product.

A large share of the resources are used to find errors, debug, and retest; thus, an associated measure of complexity is the number of software errors.

Items to consider under resources are:

- time and memory space;
- man-hours to supervise, comprehend, design, code, test, maintain and change software; number of interfaces;
- scope of support software, e.g. assemblers, interpreters, compilers, operating systems, maintenance program editors, debuggers, and other utilities;
- the amount of reused code modules;
- travel expenses;
- secretarial and technical publications support;
- overheads relating to support of personnel and system hardware.

Consideration of storage, complexity, and processing time may or may not be considered in the conceptualization stage. For example, storage of a large database might be considered, whereas if one already existed, it could be left until system-specification.

Measures of complexity are useful to:

1. Rank competitive designs and give a "distance" between rankings.

2. Rank difficulty of various modules in order to assign personnel.
3. Judge whether subdivision of a module is necessary.
4. Measure progress and quality during development.

Complexity/Comprehension (3 types)
Program complexity can be logical, psychological or structural.

1. Logical Complexity
can make proofs of correctness difficult, long, or impossible, e.g. the increase in the number of distinct program paths.

2. Psychological Complexity
makes it difficult for people to understand software. This is usually known as comprehensibility.

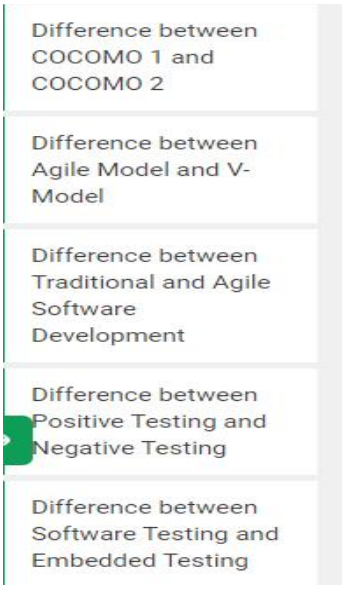
3. Structural Complexity
involves the number of modules in the program.

Logical complexity has been measure by a graph-theoretic measure.

Module 5

Modeling software quality

A **software** metric is a standard of measure of a degree to which a **software** system or process possesses some property. Even if a metric is not a measurement (**metrics** are functions, while measurements are the numbers obtained by the application of **metrics**), often the two terms are used as synonyms



The 5 levels of CMM are as follows:

Software metrics is a standard of measure that contains many activities which involve some degree of measurement. It can be classified into three categories: product metrics, process metrics, and project metrics.

- **Product metrics** describe the characteristics of the product such as size, complexity, design features, performance, and quality level.
- **Process metrics** can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process.
- **Project 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.

Measuring aspects of quality.

Most popular methods of measuring quality:

#1 – Agile **Metrics**. ...

#2 – Production **Metrics**. ...

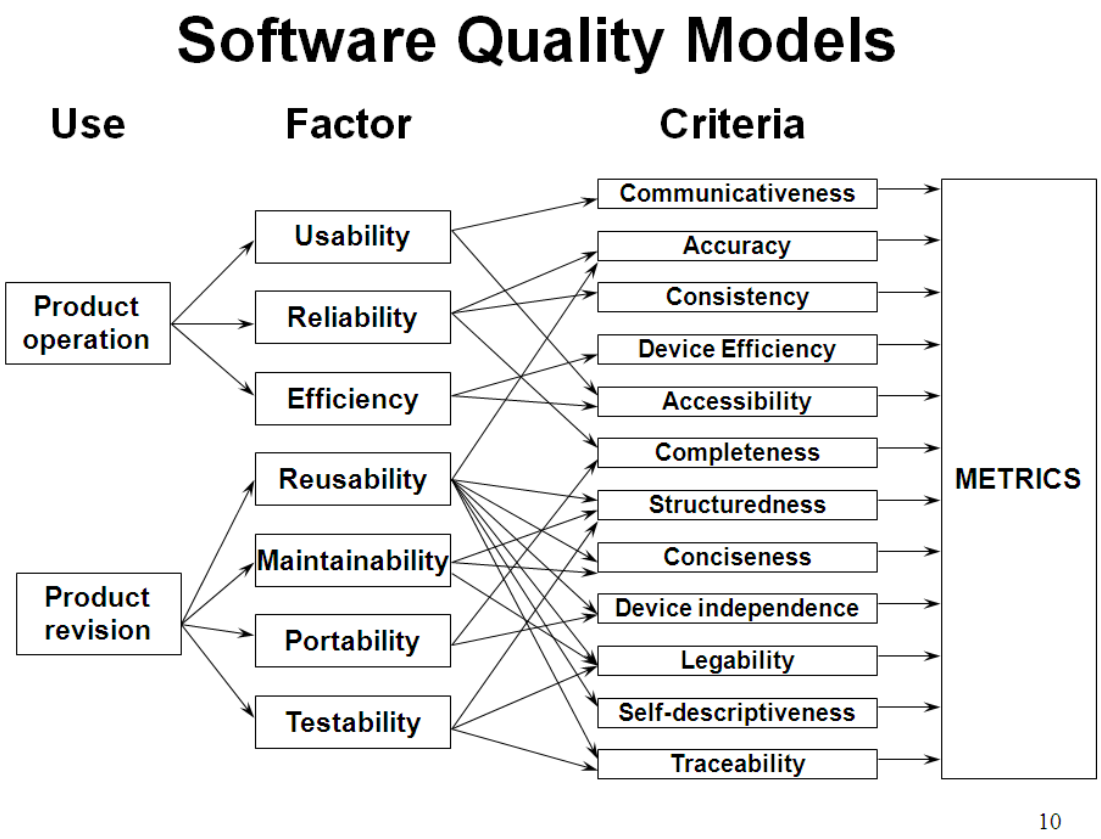
#3 – Security Responses **Metrics**. ...

#4 – Size-Oriented **Measurements**. ...

#5 – Function-Oriented Methods. ...

#6 – QA **Metrics**. ...

#7 – Customers Satisfaction.



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 −

- Product quality metrics
- In-process quality metrics
- Maintenance quality metrics

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.

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 −

$$DRE=DefectremoveddurингadevelopmentphaseDefectslatentintheproduct\times100\%DRE=Defectremovedduringadevelopmentphase Defectslatentintheproduct\times100\%$$

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.

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.

$$BMI=NumberofproblemsclosedduringthemonthNumberofproblemsarriveddurингthemonth\times100\%BMI=NumberofproblemsclosedduringthemonthNumberofproblemsarriveddurингthemonth\times100\%$$

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.

Percent delinquent fixes

It is calculated as follows −

PercentDelinquentFixes=PercentDelinquentFixes=
Numberoffixesthatexceededtheresponsetimecriteriabyceveritylevel Numberoffixesdeliveredinas pecifiedtime×100%Numberoffixesthatexceededtheresponsetimecriteriabyceveritylevel Numbero ffixesdeliveredinaspecifiedtime×100%

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.

Measurement and prediction

Software metric is a measure of software characteristics which are quantifiable or countable. Software metrics are important for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.

Software metrics is a standard of **measure** that contains many activities which involve some degree of **measurement**. It can be classified into three categories: product **metrics**, process **metrics**, and project **metrics**. ... Process **metrics** can be used to improve **software** development and maintenance.

- Software quality is a multi-dimensional notion
- Defect density is a common (but confusing) way of measuring software quality

- The notion of ‘defects’ or ‘problems’ is highly ambiguous - distinguish between faults and failures
- Removing faults may not lead to large reliability improvements
- Much data collection focuses on ‘incident types: failures, faults, and changes. There are ‘who, when, where,..’ type data to collect in each case
- System components must be identified at appropriate levels of granularity

software metrics have been used to define the complexity of the program, to estimate programming time. Extensive research has also been carried out to predict the number of defects in a module using software metrics. ... This allows the developer to run test cases in the predicted modules using test cases.

Software Metrics Product vs. process Most metrics are indirect: No way to measure property directly or Final product does not yet exist For predicting, need a model of relationship of predicted variable with other measurable variables. Three assumptions (Kitchenham)

1. We can accurately measure some property of software or process.

2. A relationship exists between what we can measure and what we want to know.

3. This relationship is understood, has been validated, and can be expressed in terms of a formula or model. Few metrics have been demonstrated to be predictable or related to product or process attributes.

Basics of reliability theory

Reliability metrics are used to quantitatively expressed the **reliability** of the **software** product. The option of which **metric** is to be used depends upon the type of system to which it applies & the requirements of the application domain.

Reliability Metrics

Reliability metrics are used to quantitatively expressed the reliability of the software product. The option of which metric is to be used depends upon the type of system to which it applies & the requirements of the application domain.

Some reliability metrics which can be used to quantify the reliability of the software product are as follows:

1. Mean Time to Failure (MTTF)

MTTF is described as the time interval between the two successive failures. An MTTF of 200 mean that one failure can be expected each 200-time units. The time units are entirely dependent on the system & it can even be stated in the number of transactions. MTTF is consistent for systems with large transactions.

For example, It is suitable for computer-aided design systems where a designer will work on a design for several hours as well as for Word-processor systems.

To measure MTTF, we can evidence the failure data for n failures. Let the failures appear at the time instants t1,t2.....tn.

MTTF can be calculated as

2. Mean Time to Repair (MTTR)

Once failure occurs, some-time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

3. Mean Time Between Failure (MTBR)

We can merge MTTF & MTTR metrics to get the MTBF metric.

$$MTBF = MTTF + MTTR$$

Thus, an MTBF of 300 denoted that once the failure appears, the next failure is expected to appear only after 300 hours. In this method, the time measurements are real-time & not the execution time as in MTTF.

4. Rate of occurrence of failure (ROCOF)

It is the number of failures appearing in a unit time interval. The number of unexpected events over a specific time of operation. ROCOF is the frequency of occurrence with which unexpected role is likely to appear. A ROCOF of 0.02 mean that two failures are likely to occur in each 100 operational time unit steps. It is also called the failure intensity metric.

5. Probability of Failure on Demand (POFOD)

POFOD is described as the probability that the system will fail when a service is requested. It is the number of system deficiency given several systems inputs.

POFOD is the possibility that the system will fail when a service request is made.

A POFOD of 0.1 means that one out of ten service requests may fail.POFOD is an essential measure for safety-critical systems. POFOD is relevant for protection systems where services are demanded occasionally.

6. Availability (AVAIL)

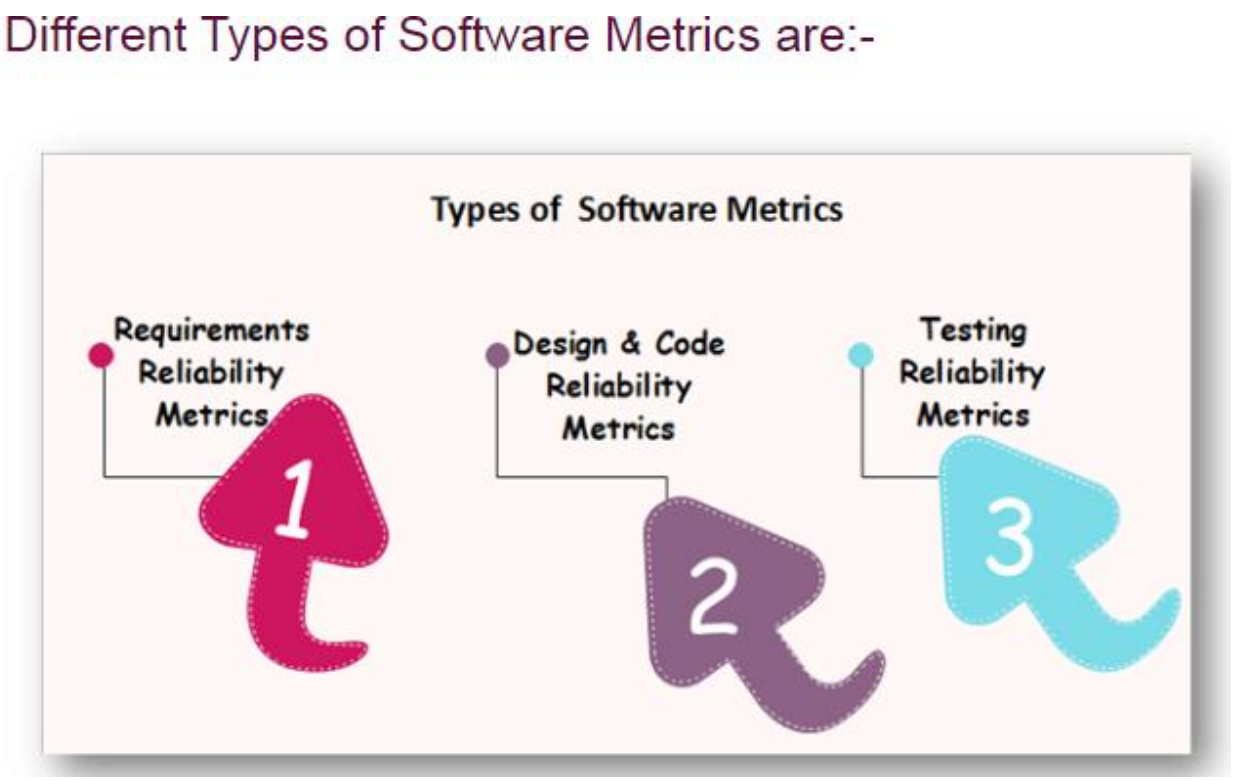
Availability is the probability that the system is applicable for use at a given time. It takes into account the repair time & the restart time for the system. An availability of 0.995 means that in every 1000 time units, the system is feasible to be available for 995 of these. The percentage of

time that a system is applicable for use, taking into account planned and unplanned downtime. If a system is down an average of four hours out of 100 hours of operation, its AVAIL is 96%.

Software Metrics for Reliability

The Metrics are used to improve the reliability of the system by identifying the areas of requirements.

Different Types of Software Metrics are:-



Requirements Reliability Metrics

Requirements denote what features the software must include. It specifies the functionality that must be contained in the software. The requirements must be written such that is no misconception between the developer & the client. The requirements must include valid structure to avoid the loss of valuable data.

The requirements should be thorough and in a detailed manner so that it is simple for the design stage. The requirements should not include inadequate data. Requirement Reliability metrics calculates the above-said quality factors of the required document.

Design and Code Reliability Metrics

The quality methods that exists in design and coding plan are complexity, size, and modularity. Complex modules are tough to understand & there is a high probability of occurring bugs. The reliability will reduce if modules have a combination of high complexity and large size or high complexity and small size. These metrics are also available to object-oriented code, but in this, additional metrics are required to evaluate the quality.

Testing Reliability Metrics

These metrics use two methods to calculate reliability.

First, it provides that the system is equipped with the tasks that are specified in the requirements. Because of this, the bugs due to the lack of functionality reduces.

The second method is calculating the code, finding the bugs & fixing them. To ensure that the system includes the functionality specified, test plans are written that include multiple test cases. Each test method is based on one system state and tests some tasks that are based on an associated set of requirements. The goals of an effective verification program is to ensure that each elements is tested, the implication being that if the system passes the test, the requirement?s functionality is contained in the delivered system.

The software reliability problem

Software

Software Reliability
Measurement and Prediction

100

- What is software reliability?
- Software reliability vs. hardware reliability

The Basic Problem of Reliability Theory

- The basic problem of reliability theory is to predict when a system will eventually fail.
- Hardware reliability concerns normally with component failures due to physical wear (e.g., corrosion, shock, over-heating, etc.).

- Such failures are probabilistic in nature, that is, we usually do not know exactly when something will fail, but we know that the product eventually will fail, so we can assign a probability that the product will fail at a particular point in time.

Software fault tolerance is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running to provide service by the specification.

Software fault tolerance is a necessary component to construct the next generation of highly available and reliable computing systems from embedded systems to data warehouse systems.

To adequately understand software fault tolerance it is important to understand the nature of the problem that software fault tolerance is supposed to solve.

Basic Reliability Theory

Probability density function $f(t)$

Distribution function $F(t) = \int_0 f(t)dt$

Reliability function $R(t) = 1 - F(t)$

Parametric reliability growth models

Reliability modeling is the process of predicting or understanding the **reliability** of a component or system prior to its implementation.

A **reliability growth model** is a **model** of how the system **reliability** changes over time during the testing process.As system failures are discovered, the underlying faults causing these failures are repaired so that the **reliability** of the system should improve during system testing and debugging.

Reliability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process.As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

Reliability growth modeling involves comparing measured reliability at a number of points of time with known functions that show possible changes in reliability. For example, an equal step function suggests that the reliability of a system increases linearly with each release. By matching observed reliability growth with one of these functions, it is possible to predict the reliability of the system at some future point in time. Reliability growth models can therefore be used to support project planning.

Examples of reliability growth models

Predictive accuracy

Predictive accuracy A. The **predictive accuracy** A describes whether the predicted values match the actual values of the target field within the incertitude due to statistical fluctuations and noise in the input data values.

The **predictive accuracy** of an ensemble tends to improve more if individual models are not only themselves accurate, but also diverse, ie, if they make different ...

Accuracy is one metric for evaluating classification **models**. Informally, **accuracy** is the fraction of predictions our **model** got right. Formally, **accuracy** has the following definition:

Accuracy = Number of correct predictions /Total number of predictions.

When **measuring** the **accuracy** of a **prediction** the magnitude of relative error (MRE) is often used, it is defined as the absolute value of the ratio of the error to the actual observed value: $\left|(\text{actual}-\textbf{predicted})/\text{actual}\right|$ or $\left|(y-\hat{y})/y\right|$. When multiplied by 100% this gives the absolute percentage error (APE

The recalibration of software-reliability growth predictions

A **reliability growth model** is a **model** of how the system **reliability** changes over time during the testing process.As system failures are discovered, the underlying faults causing these failures are repaired so that the **reliability** of the system should improve during system testing and debugging

- Failure
  - A failure is said to occur if the **observable** outcome of a **program execution** is different from the expected outcome.
- Fault
  - The adjudged cause of failure is called a fault.
  - Example: A failure may be cause by a defective block of code.
- Time
  - Time is a key concept in the formulation of reliability. If the time gap between two successive failures is short, we say that the system is less reliable.
  - Two forms of time are considered.
    - Execution time (τ)
    - Calendar time (t)

liability growth modeling

A reliability growth model is a model of how the system reliability changes over time during the testing process.As system failures are discovered, the underlying faults causing these failures are repaired so that the reliability of the system should improve during system testing and debugging. To predict reliability, the conceptual reliability growth model must then be translated into a mathematical model.

Reliability growth modeling involves comparing measured reliability at a number of points of time with known functions that show possible changes in reliability. For example, an equal step function suggests that the reliability of a system increases linearly with each release. By matching observed reliability growth with one of these functions, it is possible to predict the

reliability of the system at some future point in time. Reliability growth models can therefore be used to support project planning.

Examples of reliability growth models

I have simplified reliability growth modelling here to give you a basic understanding of the concept. If you wish to use these models, you have to go into much more depth and develop an understanding of the mathematics underlying these models and their practical problems. Littlewood and Musa (Littlewood, 1990, Abdel-Ghaly et al., 1986)(Musa, 1998) have written extensively on reliability growth models and Kan (Kan, 2003) has an excellent summary in his book. Various authors have described their practical experience of the use of reliability growth models (Ehrlich et al., 1993, Schneidewind and Keller, 1992, Sheldon et al., 1992).

Simplistically, you can predict reliability by matching the measured reliability data to a known reliability model. You then extrapolate the model to the required level of reliability and observe when the required level of reliability will be reached (Figure 1). Therefore, testing and debugging must continue until that time.

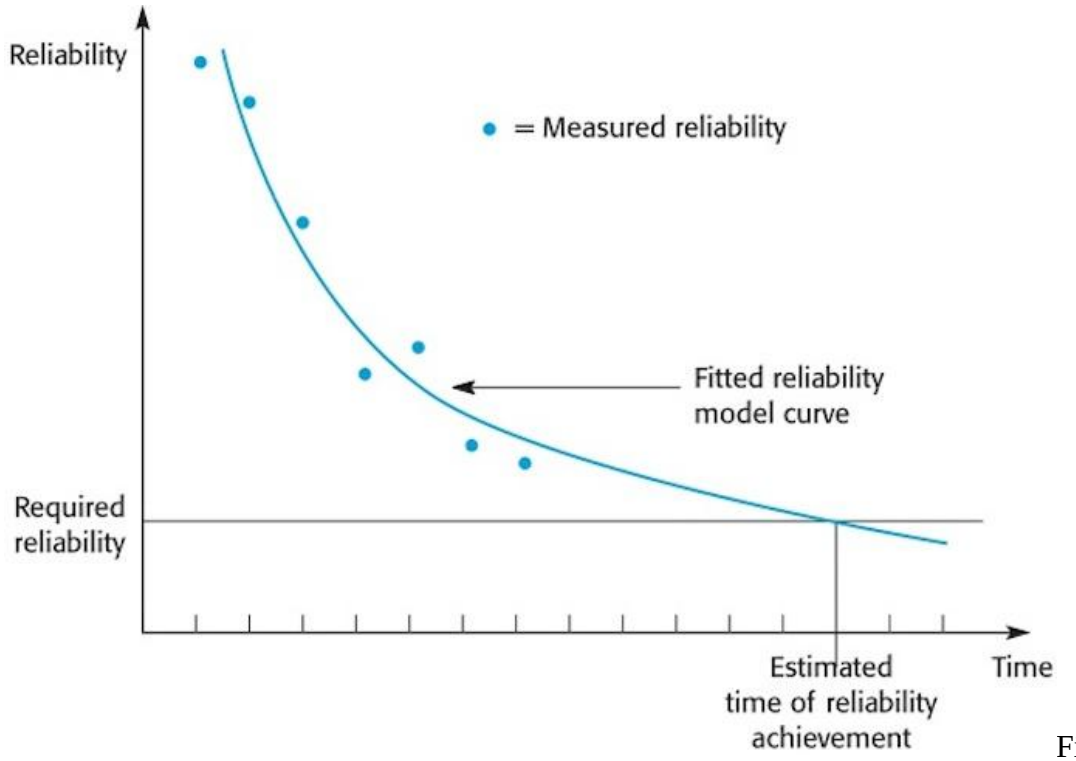


Figure 1 Reliability prediction

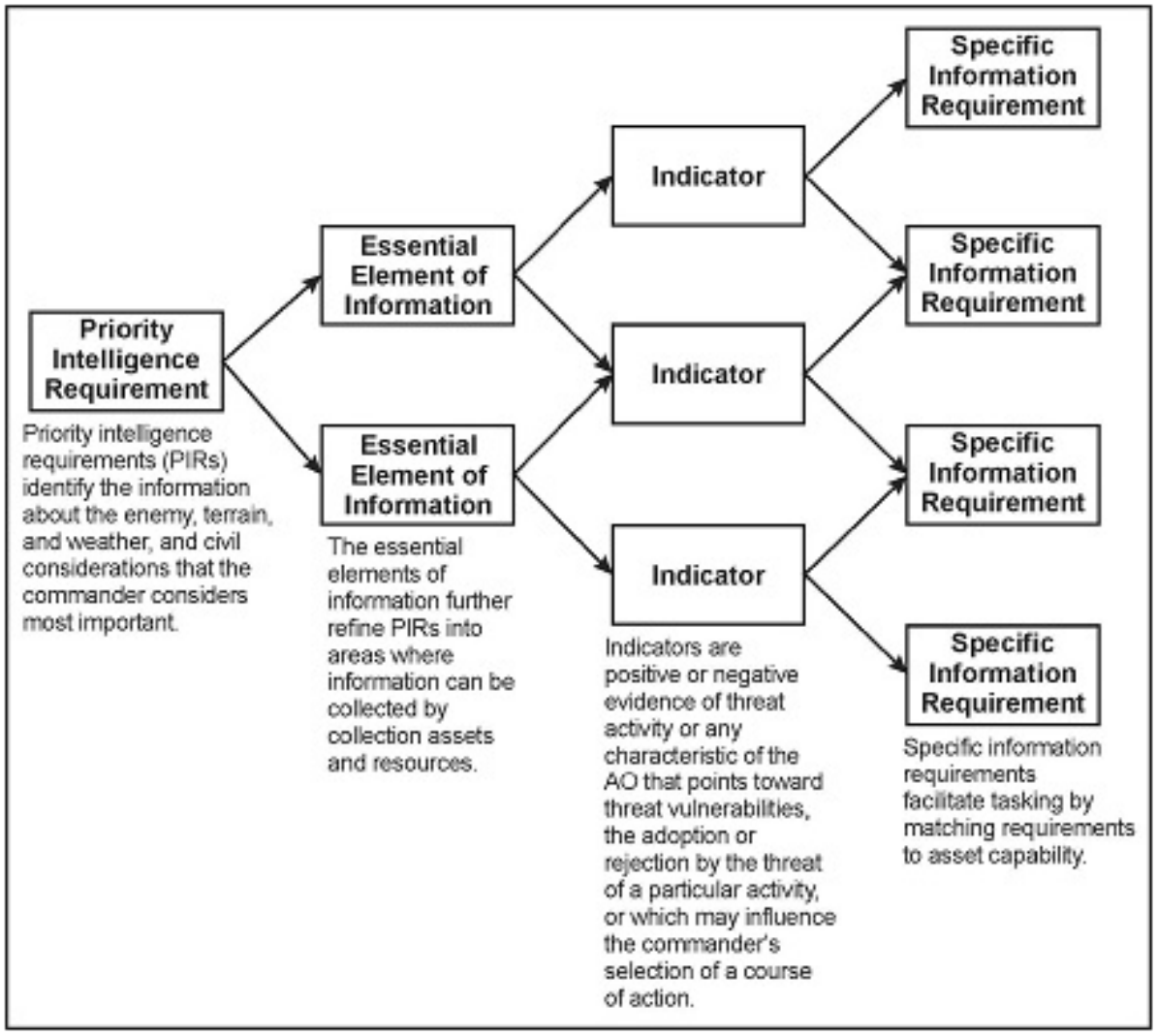
- *Planning of testing* Given the current testing schedule, you can predict when testing will be completed. If this is after the planned delivery date for the

system then you may have to deploy additional resources for testing and debugging to accelerate the rate of reliability growth.

- *Customer negotiations* Sometimes the reliability model shows that the growth of reliability is very slow and that a disproportionate amount of testing effort is required for relatively little benefit. It may be worth renegotiating the reliability requirements with the customer. Alternatively, it may be that the model predicts that the required reliability will probably never be reached. In this case, you will have to renegotiate the reliability requirements with the customer for the system.

The importance of the operational environment

The **Operational Environment** (**OE**) is a combination of conditions and variables that impact a commander's decision-making process and his/her ability to employ capabilities.



Importance of Operations Management in an Organisation

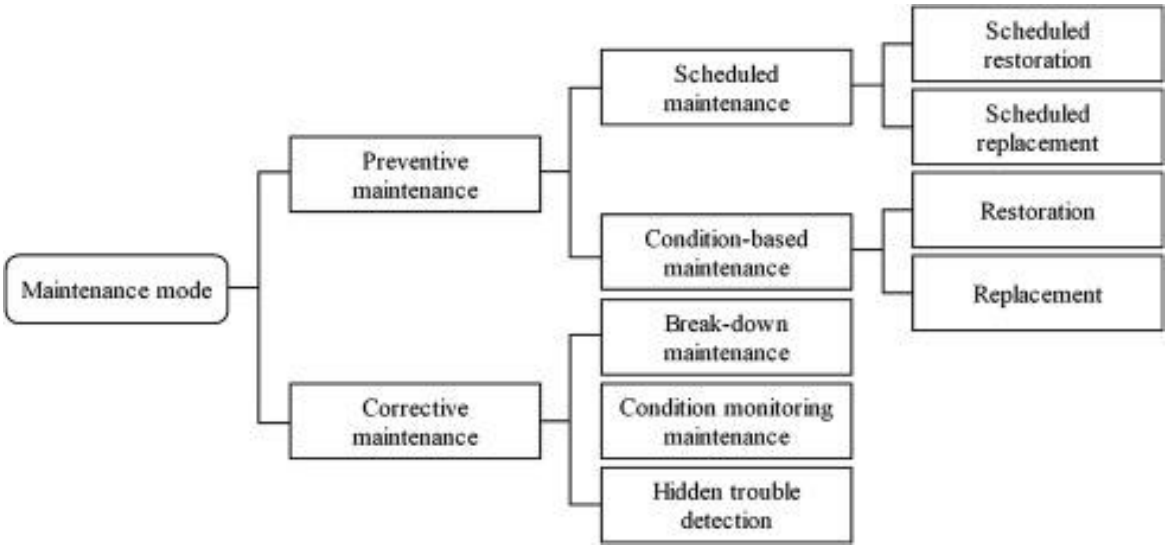
- **Helps in achievement of objectives:** Operations management has an effective role in the achievement of pre-determined objectives of an organization. It ensures that all activities are going as per plans by continuously monitoring all operations of organization.
- **Improves Employee productivity:** Operation management improves the productivity of employees. It checks and measures the performance of all people working in the organization. Operation manager trains and educate their employees for better performance.
- **Enhance Goodwill**: Operation management helps in improving the goodwill and presence of the organization. It ensures that quality products are delivered to all customers that could provide them better satisfaction and makes them happy.
- **Optimum utilization of resources**: Operation management focuses on optimum utilization of all resources of the organization. It frames proper strategies and accordingly continues all operations of the organization. Operation managers keep a check on all activities and ensure that all resources are utilized on only useful means and are not wasted.
- **Motivates Employees:** Operation management helps in motivating the employees towards their roles. Operation managers guide all peoples in performing their roles and provide them with better atmosphere. Employees are remunerated and rewarded according to their performance level.

Importance of Operations Management

Software Reliability is an important part of **software quality**. **Software reliability** is the prospect of the failure free operation of a computer program for a specified period of time in a specified environment

Software Reliability is defined as: the probability of failure-free

A Formal Definition: Reliability is the probability of failure-free operation of a system over a specified time within a specified



Software reliability is the probability of the failure free operation of a. ... The article will also provide an overview of improving **software reliability** and then ... Finally, there is a brief discussion of some **wider** issues which are not covered by a ... early life cycle **issues in** data-starved domains that lack an experience repository.

Reliability engineering is a sub-discipline of systems engineering that emphasizes dependability in the lifecycle management of a product. Reliability describes the ability of a system or component to function under stated conditions for a specified period of time.[1] Reliability is closely related to availability, which is typically described as the ability of a component or system to function at a specified moment or interval of time.

The Reliability function is theoretically defined as the probability of success {\displaystyle

({\text{Reliability}}=1-{\text{Probability of Failure}});} as, R(t), the probability of failure at time t; as a probability derived from reliability, availability, testability and maintainability. Availability, Testability, maintainability and maintenance are often defined as a part of "reliability engineering" in reliability programs. Reliability plays a key role in the cost-effectiveness of systems; for example, cars have a higher resale value when they fail less often.

Reliability and quality are closely related. Normally quality focuses on the prevention of defects during the warranty phase whereas reliability looks at preventing failures during the useful lifetime of the product or system from commissioning to decommissioning.

Reliability engineering deals with the estimation, prevention and management of high levels of "lifetime" engineering uncertainty and risks of failure. Although stochastic parameters define and affect reliability, reliability is not (solely) achieved by mathematics and statistics.[2][3] One cannot really find a root cause (needed to effectively prevent failures) by only looking at statistics. "Nearly all teaching and literature on the subject emphasize these aspects, and ignore the reality that the ranges of uncertainty involved largely invalidate quantitative methods for prediction and measurement."[4] For example, it is easy to represent "probability of failure" as a symbol or value in an equation, but it is almost impossible to predict its true magnitude in practice, which is massively multivariate, so having the equation for reliability does not begin to equal having an accurate predictive measurement of reliability.

Reliability engineering relates closely to safety engineering and to system safety, in that they use common methods for their analysis and may require input from each other. Reliability engineering focuses on costs of failure caused by system downtime, cost of spares, repair equipment, personnel, and cost of warranty claims. Safety engineering normally focuses more on preserving life and nature than on cost, and therefore deals only with particularly dangerous system-failure modes. High reliability (safety factor) levels also result from good engineering and from attention to detail, and almost never from only reactive failure management (using reliability accounting and statistics