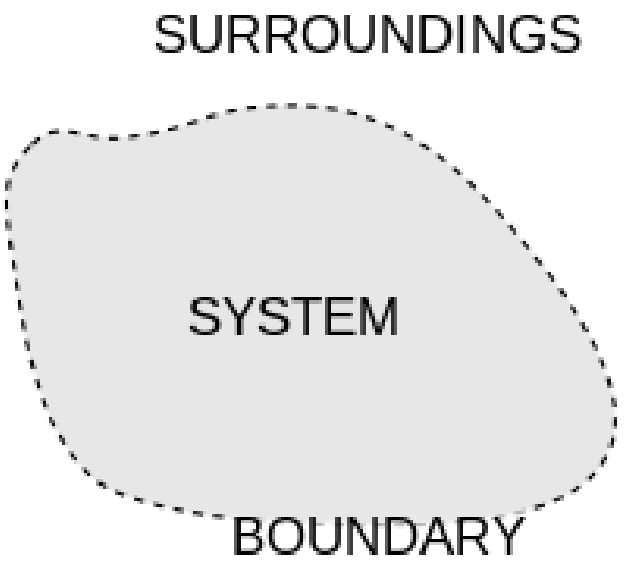
Software Engineering

Introduction

System

- A set of detailed methods, procedures and routines created to carry out a specific activity, perform a duty, or solve a problem.
- A system is a set of interacting or interdependent components forming an integrated whole.



Characteristics of a system

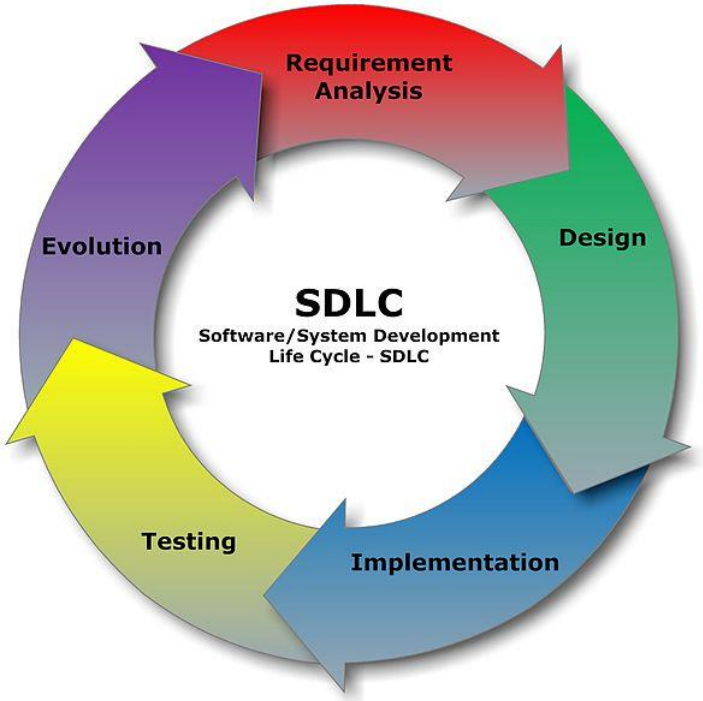
- A system has structure, it contains parts (or components) that are directly or indirectly related to each other;
- A system has behavior, it exhibits processes that fulfill its function or purpose;
- A system has interconnectivity: the parts and processes are connected by structural and/or behavioral relationships;

System development life cycle

- The systems development life cycle (SDLC), also referred to as the application development life-cycle, is a term used in systems engineering, information systems and software engineering to describe a process for planning, creating, testing, and deploying an information system.

SDLC


Analysis
Design
Planning
Implementation
Maintenance


Requirement Analysis
Design
Implementation
Testing
Evolution
SDLC
Software/System Development Life Cycle - SDLC

System development life cycle

- The systems development life-cycle concept applies to a range of hardware and software configurations, as a system can be composed of hardware only, software only, or a combination of both.

Types of systems

- Abstract and physical systems

- An abstract or conceptual system is an orderly arrangement of interdependent ideas or constructs, which may or may not have any counterpart in the real world.

- On the other hand, physical systems are generally concrete operational systems made up of people, materials, machines, energy and other physical things; Physical systems are more than conceptual constructs.

Types of systems

- Deterministic and probabilistic systems

- A deterministic system is one in which the occurrence of all events is known with certainty.

- A probabilistic system is one in which the occurrence of events cannot be perfectly predicted. Though the behavior of such a system can be described in terms of probability, a certain degree of error is always attached to the prediction of the behavior of the system.

Types of systems

- Open and closed systems

- An open system is one that interacts with its environment and thus exchanges information, material, or energy with the environment, including random and undefined inputs. Open systems are adaptive in nature, as they tend to react with the environment in such a way, so as to favor their continued existence.

- A closed system is one, which does not interact with its environment. Such systems in business world, are rare, but relatively closed systems are common. Thus, the systems that are relatively isolated from the environment but not completely closed, are termed closed system.

Types of systems

- User-machines system

- Most of the physical systems are user-machine systems It is difficult to think of a system composed only of people who do not utilize equipment of some kind to achieve their goals. In user-machine systems, both, i.e. human as well as machine perform some activities in the accomplishment of a goal

People involved in the systems development

- Users
- Managers
- System Analysts
- Programmers
- Technical Specialists
- Other stakeholders

Roles played in SDLC

- Steering Committee
  - Decision making body in a company
  - Decides how to divide resources and different projects
- Project Team
  - People who work on a specific project
  - Typically consists of systems analyst and other IT professionals

Roles played in SDLC

- Systems Analyst
  - Responsible for designing and developing the IS
  - They study user requests and generate technical specifications
- Project Management
  - Process of planning, scheduling and controlling activities during the project

Roles played in SDLC

- Project Leader
  - The person managing the budget and schedule of a project
- Project Manager
  - A person who performs the planning, scheduling and other project related activities
  - Uses various tools

Need for SE

- Software is the overriding component in terms of cost and complexity.

- Good software engineering practices and tools can therefore make a substantial difference, even to the extent that they may be the driving force of the project success.

Need for SE

- It involves the elicitation of the systems requirements, the specification of the system, its architectural and detailed design.

- In addition, the system needs to be verified and validated, a set of activities that commonly take more than 50% of all development resources.

Quality Attributes of a software product

- **Maintainability** – Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.

Quality Attributes of a software product

- **Dependability** – Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.

Quality Attributes of a software product

- **Efficiency** – Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.

Quality Attributes of a software product

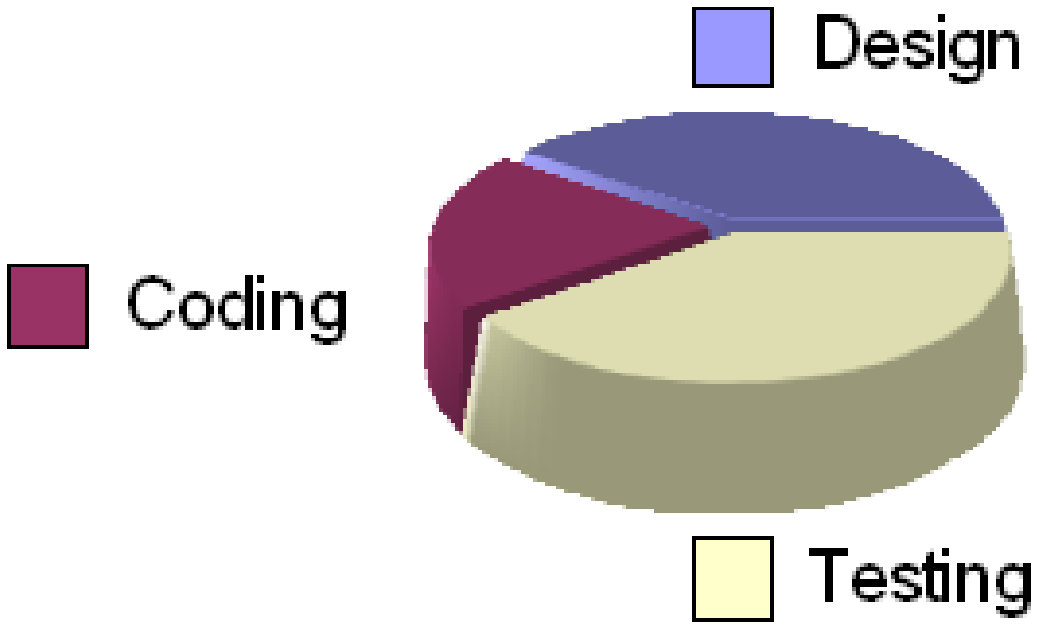
- **Usability** – Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

Benefits of SE

- Cost
  - The single greatest driving force in software engineering (and indeed almost every branch of industry today) is cost-reduction.
  - Time to market and development cost are two critical factors in deciding a product's fate.
  - Current estimates suggest that 40% of the cost in a ***successful*** software project goes into planning, 40% into testing and only 20% in actual coding.

Benefits of SE

- Cost



Benefits of SE

- Modularity/Flexibility
  - Decisions made during the design phase of a software project can often carry unforeseen hidden costs once the project is finished.
  - In an effort to reduce this problem, code is increasingly being made modular.
  - By defining a part of a program only through it's interface it is relatively easy to replace this module or component at a later date without making drastic changes to the core of the product.

Benefits of SE

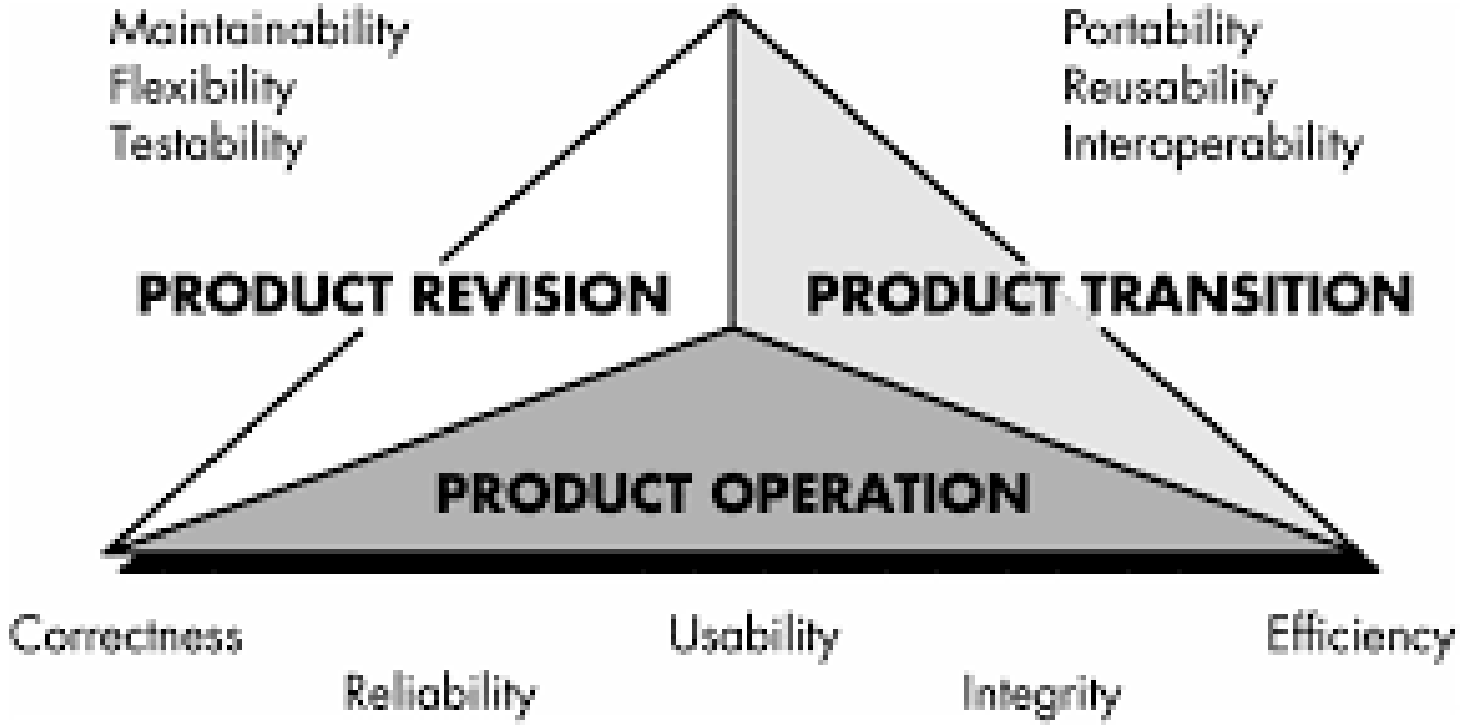
- Reliability
  - It is a relatively well known fact that the effort required to solve a problem is proportional to its complexity.
  - Indeed, the estimated cost of solving a problem increases with the square of it's complexity.
  - This means that a piece of software code which is twice as complex as another will either
    - Take four times as long to reach a given quality -or-
    - Have four times as many bugs after a given development time
- By applying the aspects of modularity, we can limit the size of any given portion of code.
- Instead of having a program with 1 million lines of code, we can now have perhaps four modules each with 250,000 lines of code.
- The total complexity has now decreased by 75%.

Objectives of SE

- Understanding user conceptual models and development of better specifications
- Improvement in design languages and reusable code
- Participatory design and interactive debugging
- Specification of interface and mockup to confirm specifications

McCall's Quality Factors

- **Correctness:** The extent to which a program satisfies its specifications and fulfills the customer's mission objectives.
- **Reliability:** The extent to which a program can be expected to perform its intended function with required precision.
- **Efficiency:** The amount of computing sources and code required by a program to perform its function.
- **Integrity:** Extent to which access to software or data by unauthorized persons can be controlled.
- **Usability:** Effort required to learn, operate, prepare input for, interpret output of a program.
- **Maintainability :** Effort required to locate and fix an error in a program.
- **Flexibility :** Effort required to modify an operational program.
- **Testability:** Effort required to test a program to ensure that it performs its intended function.
- **Portability:** Effort required to transfer the program from one hardware and / or software system environment to another.
- **Reusability:** Extent to which a program [or parts of a program] can be re-used in other applications – related to the packaging and scope of the functions that the program performs.
- **Interoperability :** Extent required to couple one system to another.


Maintainability
Flexibility
Testability
Portability
Reusability
Interoperability
PRODUCT REVISION
PRODUCT TRANSITION
PRODUCT OPERATION
Correctness
Reliability
Usability
Integrity
Efficiency

Correctness

- Concerned primarily with completeness, consistency and traceability.
- Simple correctness metrics are measured in:

  Units of total errors occurring during a given period of time divided by the total number of executable lines of source code.

  OR

  Defects per KLOC

HW: check with an example (research paper)

Reliability

- **A Formal Definition**: Reliability is the probability of failure-free operation of a system over a specified time within a specified environment for a specified purpose.
- **An Informal definition**: Reliability is a measure of how closely a system matches its stated specification.
- **Another Informal Definition**: Reliability is a measure of how well the users perceive a system provides the required services.

Fault and Failure

- A **failure** corresponds to unexpected run-time behavior observed by a user of the software.

- A **fault** is a static software characteristic which causes a failure to occur.

Improving Reliability

- **Primary objective**: Remove faults with the most serious consequences.

- **Secondary objective**: Remove faults that are encountered most often by users.

90-10 Rule

- 90% of the total execution time is spent in executing only 10% of the instructions in the program.
- This 10% part is the core of the program.

Effect of 90-10 Rule on Reliability

- Removing 60% defects from least used parts would lead to only about 3% improvement to product reliability.
- Reliability improvements from correction of a single error depends on whether the error belongs to the core or the non-core part of the program.

Reliability Metrics - 1

- **Probability Of Failure On Demand (POFOD)**:
  - Likelihood that system will fail when a request is made.
  - E.g., POFOD of 0.001 means that 1 in 1000 requests may result in failure.
  - Relevant for safety-critical systems.

Reliability Metrics - 2

- **Rate Of Occurrence Of Failure (ROCOF)**:
  - Frequency of occurrence of failures.
  - E.g., ROCOF of 0.02 means 2 failures are likely in each 100 time units.
  - Relevant for transaction processing systems.

Reliability Metrics - 3

- **Mean Time To Failure (MTTF)**:
  - Measure of time between failures.
  - E.g., MTTF of 500 means an average of 500 time units passes between failures.
  - Relevant for systems with long transactions.

Reliability Metrics - 4

- **Availability**:
  - Measure of how likely a system is available for use, taking in to account repairs and other down-time.
  - E.g., Availability of .998 means that system is available 998 out of 1000 time units.
  - Relevant for continuously running systems. E.g., telephone switching systems.

Automatic Bank Teller Example

- Bank has 1000 machines; each machine in the network is used 300 times per day.
- Lifetime of software release is 2 years.
- Therefore, there are about 300,000 database transactions per day, and each machine handles about 200,000 transactions over the 2 years.

Example Reliability Specification

Failure class	Example	Reliability metric
Permanent, non-corrupting	The system fails to operate with any card; must be restarted	ROCOF =1 occ./1000 days
Transient, non-corrupting	The magnetic strip on an undamaged card cannot be read	POFOD = 1 in 1000 trans.
Transient, corrupting	A pattern of transactions across the network causes DB corruption	Should never happen

Efficiency vs. Effectiveness

- Peter Drucker defined the difference between being efficient and being effective:
  - Efficiency is the capacity to do things right.
  - Effectiveness is the capacity to do the right thing.

- A manual worker is expected to be efficient.

- A knowledge worker is expected to be effective.

Efficiency vs. Effectiveness

- In the past, programmers were more like manual workers.

- Programmers had to be efficient, and the efficiency of the programmer could be measured by the number of lines of code (LOC) that he could produce in some unit of time.

Be effective

- Today, with the advent of object-oriented programming languages, the nature of programming tasks has changed.
- For any possible problem being resolved there are certainly libraries of classes that can be easily reused.
- For many complex systems there are frameworks that already provide most of the required functionality, and that need only be extended with a relatively small development effort.

Reinventing the wheel

- In this new reality, measuring the LOC produced by the programmer is most likely the wrong thing.
- Because if he is writing too much, he is probably reusing too few.
- It is common to see less experienced programmers constantly "**reinventing the wheel**", creating again classes that are ready to use in some library.

- As Peter Drucker said:

"There is nothing so useless as doing efficiently that which should not be done at all."

- Therefore, programmers today should be expected to be effective.
- Design Patterns can be described as effective software design decisions for several abstract problems

Code integrity

- **Code integrity** is a measurement used in software testing.
- It measures the how high is the source code's quality when it is passed on to the QA, and is affected by how extensively the code was unit tested and integration tested.
- Code integrity is a combination of code coverage and software quality, and is usually achieved by unit testing your code to reach high code coverage.

Code Coverage Definition

- In computer science, **code coverage** is a measure used to describe the degree to which the source **code** of a program is executed when a particular test suite runs.

Code Coverage Analysis

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.
- Identifying redundant test cases that do not increase coverage.

Code Coverage metrics

- Statement Coverage
- Branch Coverage
- Path Coverage
- Condition Coverage

HW – example code

Measuring code integrity

1 – (Non-covered bugs) / (Total bugs)

In words:, the 100% code integrity minus the number of bugs that weren't covered by unit testing, divided by the total bugs found during the entire product cycle., including development, is the code not in integrity.

Improve code integrity by

- Unit testing the code
- Integration testing
- Assigning a code integrity manager

Advantages of working with code

- Shorter development time
- Lower development costs
- Confidence in your code's quality
- Makes the QA's work much more efficient

Why would you need to measure Usability

- Need to effectively communicate with the stakeholders of the system being evaluated.

- Need for comparing the usability of two or more products and to quantify the severity of a usability problem.

ISO 9241-11 standard defines usability

- The extent to which a product can be used by specified users to achieve specified goals with **effectiveness**, **efficiency** and **satisfaction** in a specified context of use.

usabilitygeek.com

Usability Metrics for Effectiveness

- Completion Rate
- The completion rate is calculated by assigning a binary value of ‘1’ if the test participant manages to complete a task and ‘0’ if he/she does not.

$$Effectiveness = \frac{\text{Number of tasks completed successfully}}{\text{Total number of tasks undertaken}} \times 100\%$$

Example for Completion Rate

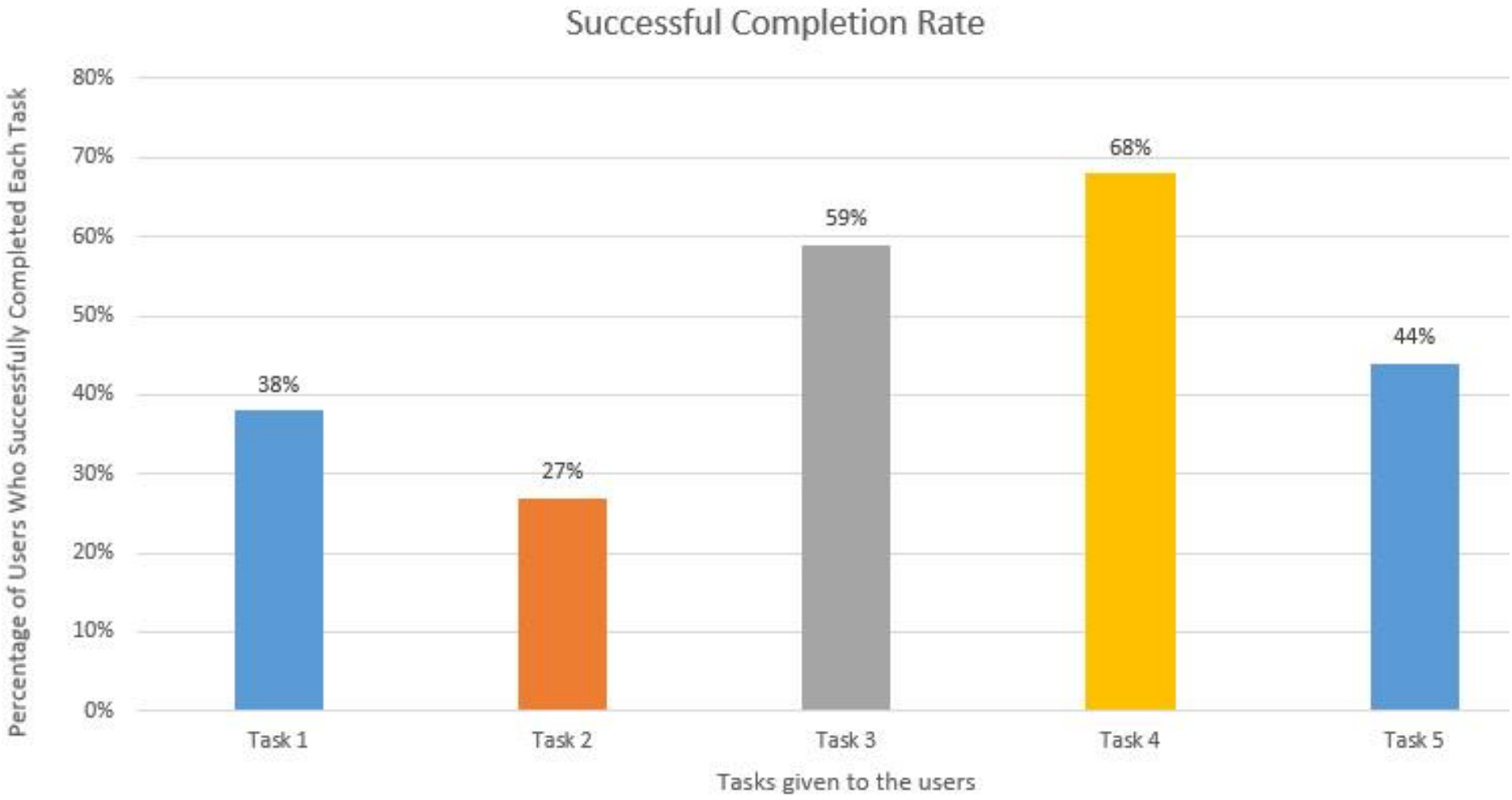
- 5 users perform a task using the same system. At the end of the test session, 3 users manage to achieve the goal of the task while the other 2 do not.

  Number of tasks completed successfully = 3
  Total number of tasks undertaken = 5

Example for Completion Rate

$$Effectiveness = \frac{3}{5} \times 100\% = 60\%$$

Example Graph



Usability Metrics for Effectiveness

- **Number of Errors**
- Another measurement involves counting the number of errors the participant makes when attempting to complete a task.
- Based on an analysis of 719 tasks performed using consumer and business software, the author concluded that the average number of **errors per task is 0.7**, with 2 out of every 3 users making an error.

Usability Metrics for Efficiency

- Efficiency is **measured in terms of task time**. that is, the time (in seconds and/or minutes) the participant takes to successfully complete a task.

Task Time = End Time – Start Time

Usability Metrics for Efficiency

- **Time-Based Efficiency**

$$Time\ Based\ Efficiency = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N}\frac{n_{ij}}{t_{ij}}}{NR}$$

where
- N = The total number of tasks (goals)
- R = The number of users
- n_{ij} = The result of task i by user j; if the user successfully completes the task, then n_{ij} = 1, if not, then n_{ij} = 0
- t_{ij} = The time spent by user j to complete task i. If the task is not successfully completed, then time is measured till the moment the user quits the task

Example - Usability Metrics for time based efficiency

- Suppose there are 4 users who use the same product to attempt to perform the same task (1 task). 3 users manage to successfully complete it – taking 1, 2 and 3 seconds respectively. The fourth user takes 6 seconds and then gives up without completing the task.

Example - Usability Metrics for time-based efficiency

N = The total number of tasks = 1

R = The number of users = 4

User 1: $N_{ij} = 1$ and $T_{ij} = 1$

User 2: $N_{ij} = 1$ and $T_{ij} = 2$

User 3: $N_{ij} = 1$ and $T_{ij} = 3$

User 4: $N_{ij} = 0$ and $T_{ij} = 6$

$$Time\ Based\ Efficiency = \frac{\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{0}{6}\right)}{1 \times 4} = 0.46\ goals/sec$$

Usability Metrics for Efficiency

- **Overall Relative Efficiency**

$$Overall\ Relative\ Efficiency = \frac{\sum_{j=1}^{R}\sum_{i=1}^{N} n_{ij}t_{ij}}{\sum_{j=1}^{R}\sum_{i=1}^{N} t_{ij}} \times 100\%$$

Example - Usability Metrics for overall relative efficiency

- Suppose there are 4 users who use the same product to attempt to perform the same task (1 task). 3 users manage to successfully complete it – taking 1, 2 and 3 seconds respectively. The fourth user takes 6 seconds and then gives up without completing the task.

Example - Usability Metrics for overall relative efficiency

N = The total number of tasks = 1

R = The number of users = 4

User 1: $N_{ij} = 1$ and $T_{ij} = 1$

User 2: $N_{ij} = 1$ and $T_{ij} = 2$

User 3: $N_{ij} = 1$ and $T_{ij} = 3$

User 4: $N_{ij} = 0$ and $T_{ij} = 6$

$$Overall\ Relative\ Efficiency = \left(\frac{((1 \times 1) + (1 \times 2) + (1 \times 3) + (0 \times 6))}{(1 + 2 + 3 + 6)}\right) \times 100 = 50\%$$

Usability Metrics for Satisfaction

- Task Level Satisfaction
- Test Level Satisfaction

Task level Satisfaction

- **After users attempt a task** (irrespective of whether they manage to achieve its goal or not), they should immediately be given a questionnaire so as to measure how difficult that task was.
- Typically consisting of up to 5 questions, these post-task questionnaires often take the form of **Likert scale** ratings and their goal is to provide insight into task difficulty as seen from the participants' perspective.

Test level satisfaction

- Test Level Satisfaction is measured by giving a formalized questionnaire to each test participant **at the end of the test session**.
- This serves to measure their impression of the overall ease of use of the system being tested.

Questions

- What are the differences between generic software product development and custom software development?
- What are the four important attributes which all software products should have? Suggest four other attributes that may sometimes be significant.

Questions

- Compare product and project
- Differentiate between validation and verification

Product

- Product is a company that releases Hardware, Middleware, Operating system, Languages, Tools etc.
- They develop the products for global clients i.e. there are no specific clients for them.
- Here the requirements are gathered from market and analyze that with some experts and start to develop the product.
- After developing the products they try to market it as a solution.
- Here the end users are more than one.
- Product development is never ending process and customization is possible.

Project

- Project is finding solution to a particular problem to a particular client.
- By using the products like Hardware, Middleware, Operating system, Languages and Tools only we will develop a project.
- Here the requirements are gathered from the client and analyze with that client and start to develop the project.
- Here the end user is one.

Verification

- Are we building the system right?
- **Verification** is the process of evaluating products of a development phase to find out whether they meet the specified requirements.
- The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.
- Following activities are involved in Verification: Reviews, Meetings and Inspections.

Validation

- Are we building the right system?
- **Validation** is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
- The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
- Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc.

Verification

- Verification is carried out by QA team to check whether implementation software is as per specification document or not.
- Execution of code is not comes under Verification.
- **Verification** process explains whether the outputs are according to inputs or not.
- Verification is carried out before the Validation.
- Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc.
- Cost of errors caught in Verification is less than errors found in Validation.
- It is basically manually checking the of documents and files like requirement specifications etc.

Validation

- Validation is carried out by testing team.
- Execution of code is comes under Validation.
- **Validation** process describes whether the software is accepted by the user or not.
- Validation activity is carried out just after the Verification.
- Following item is evaluated during Validation: Actual product or Software under test.
- Cost of errors caught in Validation is more than errors found in Verification.
- It is basically checking of developed program based on the requirement specifications documents & files.

Assignment

- What is software?
- What are the attributes of good software?
- What is software engineering?
- What are the fundamental software engineering activities?
- What is the difference between software engineering and computer science?
- What is the difference between software engineering and system engineering?

Assignment

- What are the key challenges facing software engineering?
- What are the costs of software engineering?
- What are the best software engineering techniques and methods?
- What differences has the Web made to software engineering?